Dynamic Parallelization of R Functions

by Stefan Böhringer

Abstract R offers several extension packages that allow it to perform parallel computations. These operate on fixed points in the program flow and make it difficult to deal with nested parallelism and to organize parallelism in complex computations in general. In this article we discuss, first, of how to detect parallelism in functions, and second, how to minimize user intervention in that process. We present a solution that requires minimal code changes and enables to flexibly and dynamically choose the degree of parallelization in the resulting computation. An implementation is provided by the R package **parallelize.dynamic** and practical issues are discussed with the help of examples.

Introduction

The R language (Ihaka and Gentleman, 1996) can be used to program in the functional paradigm, *i.e.* return values of functions only depend on their arguments and values of variables bound at the moment of function definition. Assuming a functional R program, it follows that calls to a given set of functions are independent as long as their arguments do not involve return values of each other. This property of function calls can be exploited and several R packages allow to compute function calls in parallel, *e.g.* packages **parallel**, **Rsge** (Bode, 2012) or **foreach** (Michael et al., 2013; Revolution Analytics and Weston, 2013). A natural point in the program flow where to employ parallelization is where use of the `apply`-family of functions is made. These functions take a single function (here called the compute-function) as their first argument together with a set of values as their second argument (here called the compute-arguments) each member of which is passed to the compute-function. The calling mechanism guarantees that function calls cannot see each others return values and are thereby independent. This family includes the `apply`, `sapply`, `lapply`, and `tapply` functions called generically `Apply` in the following. Examples of packages helping to parallelize `Apply` functions include **parallel** and **Rsge** among others and we will focus on these functions in this article as well.

In these packages, a given `Apply` function is replaced by a similar function from the package that performs the same computation in a parallel way. Fixing a point of parallelism introduces some potential problems. For example, the bootstrap package **boot** (Davison and Hinkley, 1997; Canty and Ripley, 2013) allows implicit use of the **parallel** package. If bootstrap computations become nested within larger computations the parallelization option of the `boot` function potentially has to be changed to allow parallelization at a higher level once the computation scenario changes. In principle, the degree of parallelism could depend on parameter values changing between computations thereby making it difficult to choose an optimal code point at which to parallelize. Another shortcoming of existing solutions is that only a single `Apply` function gets parallelized thereby ignoring parallelism that spans different `Apply` calls in nested computations. The aim of this paper is to outline solutions that overcome these limitations. This implies that the parallelization process should be as transparent as possible, *i.e.* requiring as little user intervention as necessary. An ideal solution would therefore allow the user to ask for parallelization of a certain piece of code and we will try to approximate this situation. Potential benefits for the user are that less technical knowledge is required to make use of parallelization, computations can become more efficient by better control over the scaling of parallelization, and finally programs can better scale to different resources, say the local machine compared to a computer cluster.

This article is organized as follows. We first give further motivation by an example that highlights the problems this approach seeks to address. We then outline the technical strategy needed to determine the parallelism in a given function call. After that, trade-offs introduced by such a strategy are discussed. We conclude by benchmarking two examples and discussing important practical issues such as deviations of R programs from the functional programming style.

Dynamic parallelism in R functions

Let us start by looking at an example that tries to condense real-world problems in short, self-contained code which illustrates issues to be solved. Regression analyses are performed on the `iris` data set as follows.

Example 1

```
Lapply <- lapply
Sapply <- sapply
```

```
library(sets)
data(iris)
d <- iris; response <- 'Species'; R <- .01; Nl <- 1e1; Nu <- 1e5

vars <- setdiff(names(d), response)
responseLevels <- levels(d[[response]])

minimax <- function(v, min = -Inf, max = Inf)
  ifelse(v < min, min, ifelse(v > max, max, v))
N <- function(p, r = R)
  (2 * qnorm(p, lower.tail = FALSE)/r)^2 * (1 - p)/p
analysis <- function(data, vars) {
  f1 <- as.formula(sprintf('%s ~ %s', response, paste(vars, collapse = ' + ')));
  f0 <- as.formula(sprintf('%s ~ 1', response));
  a <- anova(glm(f0, data = data), glm(f1, data = data), test = 'Chisq')
  p.value <- a[['Pr(>Chi)']][[2]]
}
permute <- function(data, vars, f, ..., M) {
  ps <- Sapply(0:M, function(i, data, vars, f, ...) {
    if (i > 0) data[, vars] <- data[sample(nrow(data)), vars];
    f(data, vars, ...)
  }, data, vars, f, ...)
  p.data <- ps[1]
  ps <- ps[-1]
  list(p.raw = p.data, p.emp = mean(ps[order(ps)] < p.data))
}
subsetRegression <- function() {
  r <- Lapply(responseLevels, function(l) {
    subsets <- as.list(set_symdiff(2^as.set(vars), 2^set()))
    r1 <- Sapply(subsets, function(subset) {
      d[[response]] = d[[response]] == l
      p.value <- analysis(d, unlist(subset))
      unlist(permute(d, unlist(subset), analysis,
        M = as.integer(minimax(N(p.value), Nl, Nu))))
    })
    output <- data.frame(subset = sapply(subsets, function(s)
      paste(s, collapse = '+')), t(r1))
  })
  names(r) <- responseLevels
  r
}
print(subsetRegression())
```

Variable `Species` is dichotomized for all of its levels and a subset analysis is performed by regressing these outcomes on all possible subsets of the other variables (function `analysis`). Also a permutation based P-value is computed (function `permute`) and the number of iterations depends on the raw P-value (p_{raw}) from the original logistic regression. Here, the number of iterations is chosen to control the length of the confidence interval for the permutation P-value (ci_l, ci_u) so that $(ci_u - ci_l)/p_{\text{raw}} < r$ (in probability), where r is a chosen constant (function `N`). To ensure robustness, the resulting number is constrained within an integer interval (function `minimax`).

Analyzing computational aspects of this code, we first note that our most global models are represented by response levels, in this case three, constituting a low level of parallelization. Second, the subset models vary in size, in this case by a factor of four. Third, the parallelism of permutation computations is data dependent and cannot be determined beforehand. It is thus not straightforward to choose a good point at which to parallelize. Finally, observe that we have copied the symbols `sapply` and `lapply` to upper-cased symbols and used them in places where parallelization is desirable. The `sapply` used for computing the variable `output` has not been marked in this way as it constitutes a trivial computation. The remainder of the article is concerned with achieving the goals stated above for a program for which desirable parallelization has been marked as in the code above.

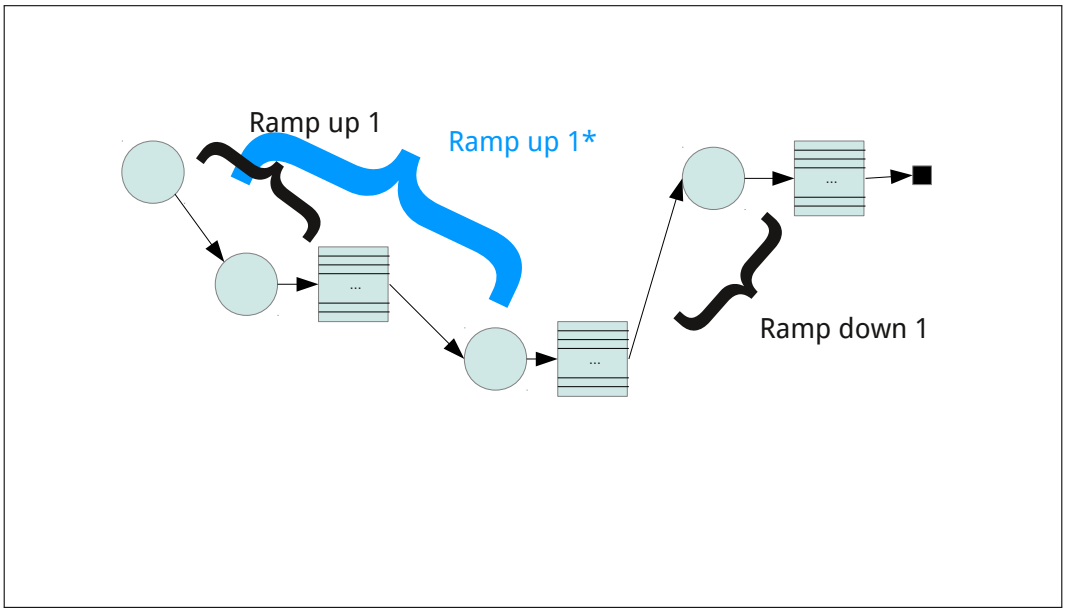

Figure 1: Abstraction of program flow with the following symbol semantics. Circles: entry points into or return points from functions; downward arrows: function calls; tables: `Apply` function calls; upward arrows: return path from functions; square: end of computation. Further explanation in text.

Dynamic analysis

Parallelism in programs can be detected by static analysis of source code (*e.g.* Cooper and Torczon, 2011) or by dynamic analysis (*e.g.* Ernst, 2003) the latter relying on execution of the code at hand and the analysis of data gleaned from such executions. Example 1 motivates the use of dynamic analysis and we discuss static analysis later. In cases where parallelism is data dependent, dynamic analysis is the only means to precisely determine the level of parallelism. On the other hand also in cases where parallelism is known or can be determined by inexpensive computations, dynamic analysis has the advantage of convenience as the user is not responsible for making decisions on parallelization.

In the following, dynamic analysis is performed on `Apply` functions marked as in Example 1. The overarching idea is to run the program but stop it in time to determine the degree of parallelism while still having spent only little computation time. Like in existing packages the assumption is made that the functional style is followed by the called functions, *i.e.* they do not exert side-effects nor depend on such.

Abstract program flow

Figure 1 depicts the program flow as seen in the parallelization process. Given code becomes a sequence of linear code (circles) leading to an `Apply` function, the `Apply`-function (table), and linear code executed thereafter (circles). This pattern repeats whenever `Apply` functions are nested. For now, we ignore the case where `Apply` functions are called sequentially as this case is not interesting for understanding the algorithm. We call code leading to a given `Apply` call the *ramp-up* and code executed after the `Apply` the *ramp-down* such that every program can be seen as an execution *ramp-up* – `Apply` – *ramp-down*. The task of dynamic analysis is to select the "best" `Apply` function, then separate execution into *ramp-up* – `Apply` – *ramp-down*, and perform the computation. Then, calls resulting from the selected `Apply` can be computed in parallel.

Algorithm

We now outline an abstract algorithm for implementing this program flow. Specific details about R-specific behavior are given in the implementation section. The problem is solved by re-executing the code several times – a choice that is justified below. The re-executions involve custom `Apply` functions which replace the original implementations with the ability to return execution to higher level `Apply` functions without executing the ramp-down, namely code following the `Apply` (escaping). The following re-executions take place:

- Probing (ramp-up): determine the level of parallelism, stop
- Freezing (ramp-up): save calls from `Apply`s for parallel execution, stop
- Recovery (ramp-down): replace calls that were parallelized with stored results, continue execution

Between the freezing and recovery steps, parallel computations are performed.

Probing

Probing potentially involves several re-executions as parallelism is determined for increasing nesting levels. For a given nesting level, probing simply stores the number of elements passed to the `Apply` calls at the specified nesting level and returns to the higher level. The sum of these elements is the level of parallelism achievable at that nesting level. If higher degree of parallelism is desired probing is repeated at a deeper nesting level.

Freezing

After a nesting level is chosen in the probing step, execution is stopped again in `Apply` calls at that nesting level. The calls that the `Apply` would generate are stored as unevaluated calls in a so-called freezer object.

Parallel execution

Parallel execution is controlled by a backend object. Similarly to the **foreach** package, several options are available to perform the actual computations (*e.g.* **snow** Tierney et al., 2013, or batch queuing systems).

Recovery

During recovery, execution is stopped at the same position as in the freezing step. These time results computed during parallel execution are retrieved and returned instead of evaluating function calls. Finally the whole computation returns with the final result.

Corner cases

If the requested level of parallelism exceeds the available parallelism, the computation will already finish in the probing step. This is because the probing level exceeds the nesting level at some point and execution will not be stopped. In this case the computation is performed linearly (actually sub-linearly because of the repeated re-executions).

When all results have been retrieved in the recovery step, the algorithm can switch back to probing and parallelize `Apply` code sequential to the first hierarchy of `Apply` calls. If no such `Apply` calls are present probing will compute the result linearly thus not incurring a performance penalty.

Implementation of R package parallelize.dynamic

The parallelization implementation is split into a so-called *front-end* part which implements the algorithm described above and a *backend* part which performs parallel execution. Currently, there are backends for local execution (*local* backend) which executes linearly, parallel execution on *snow* clusters (*snow* backend), and a backend for execution on *Sun Grid Engine* or *Open Grid Scheduler* batch queuing systems (*OGSremote* backend). This is a similar approach to the **foreach** package and potentially code can be shared between the packages. Back-ends are implemented as S4-classes and we refer to the package documentation for details on how to implement new backends.

API

Dynamic analysis of parallelism is performed by making use of replacements of `Apply` functions similar to existing packages. In this implementation, replacement functions have the same name as replaced functions with an upper case first letter, referred to as `Apply` functions. The new functions have exactly the same programming interface and the same semantics (*i.e.* they compute the same

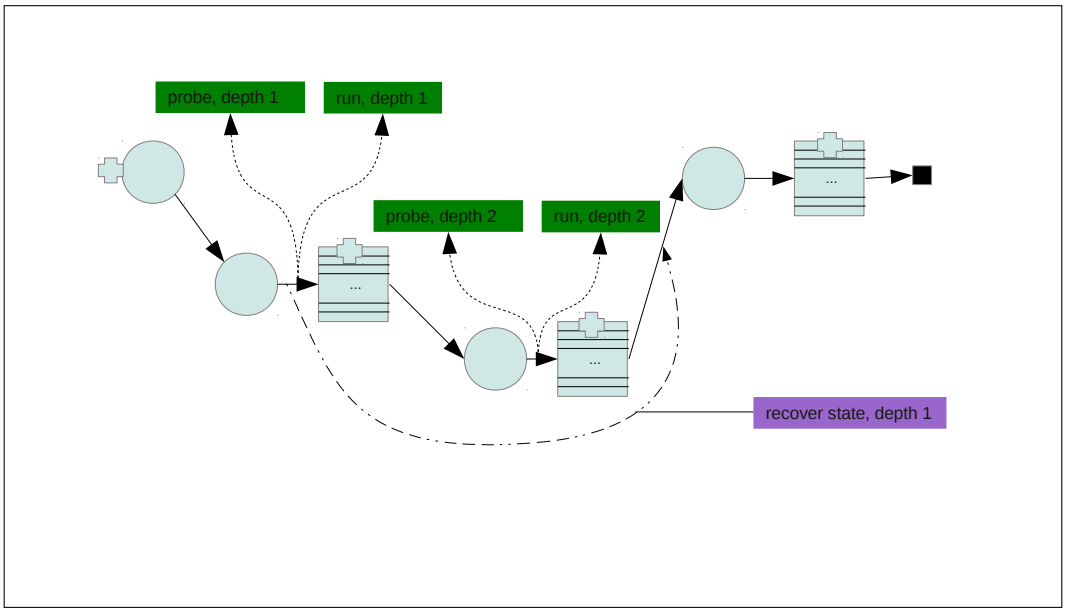

Figure 2: Exceptions used in the process of parallelization. Plus symbols indicate exception handlers and dotted lines indicate exceptions. See Figure 1 for explanation of other symbols.

result) as the replaced functions, which is a difference to similar packages. One advantage is that programs can be very quickly adapted for parallel execution and the mechanism can be turned of by re-instating the original function definitions for `Apply` functions as done in Example 1.

Global state

In order to perform probing, freezing and recovery `Apply` functions maintain a global state containing the current position in the program flow. This is defined by the nesting level and an index number counting `Apply` calls seen so far. Probing and freezing maintain additional state, respectively. During probing, the number of elements passed to the `Apply` call under investigation is stored, during freezing, unevaluated calls resulting from the parallelized `Apply` call are stored.

Escaping

Probing and freezing need the ability to skip the ramp-down as they did not compute any results yet. This capability is implemented using the R exception handling mechanism defined by the functions `try`/ `stop`. Any `Apply` that needs to escape the ramp-down issues a call to `stop` that is caught by a `try` call that was issued in a higher-level `Apply`. As this disrupts normal program flow, execution can later not be resumed at the point where `stop` was called, requiring re-execution of the whole code. Figure 2 illustrates the use of exception handling.

Freezing

The freezing mechanism is defined by a reference class (`LapplyFreezer`) which stores unevaluated calls for parallel execution and results of these calls. The base class simply stores unevaluated calls and results as R objects. Subclasses that store results on disk (`LapplyPersistentFreezer`) or defer generation of individual calls from `Apply` calls to the parallel step (`LapplyGroupingFreezer`) exist and can be paired with appropriate backends.

Lexical scoping and lazy evaluation

R allows the use of variables in functions that are not part of the argument list (unbound variables). Their values are resolved by lexical scoping, *i.e.* a hierarchy of environments is searched for the first definition of the variable. This implies that all environments including the global environment would potentially have to be available when a parallel function call is executed to guarantee resolution of variable values. As for the *snow* and *OGS* backends execution takes place in different processes

transferring these environments often constitutes unacceptable overhead. Therefore, if unbound variables are used with these backends the option `copy_environments` can be set to `TRUE` to force copying of environments. This mechanism constructs a new environment that only contains variables unbound in parallel function calls and computes their values using `get` in the correct environment. This is a recursive process that has to be repeated for functions called by compute-functions and is possibly expensive (compare Example 1). Potentially, these variables could be part of expressions that are evaluated lazily, *i.e.* values of these expressions should only be computed later when the expression is assigned to a variable. Code relying on the semantics of lazy evaluation could therefore work incorrectly.

A way to avoid copying of environments is to not use unbound variables in compute-functions. Functions called from compute-functions are allowed to contain unbound variables as long as they are bound by any calling function. The `copy_environments` option helps to minimize code changes to achieve parallelization but its use is not recommended in general (see discussion of Example 1 below).

Package and source dependencies

The `copy_environments` option can be used to ensure that function definitions are available in the parallel jobs. However, this mechanism avoids copying functions defined in packages as packages might contain initialization code containing side-effects upon which these functions could depend. Instead, required packages have to be specified either as an element in the configuration list passed to `parallelize_initialize` or as the `libraries` argument of `parallelize_initialize`. Similarly, the `sourceFiles` component of the configuration list or the `sourceFiles` argument of `parallelize_initialize` specify R scripts to be sourced prior to computing the parallel job.

Examples

Example 1 continued

We continue Example 1 by extending it for use with package **parallelize.dynamic**. Parallelization is initialized by a call to `parallelize_initialize`. The following code has to replace the call to `subsetRegression` in Example 1.

```
library(parallelize.dynamic)
Parallelize_config <- list(
  libraries = 'sets',
  backends = list(snow = list(localNodes = 8, stateDir = tempdir()))
)
parallelize_initialize(Parallelize_config,
  backend = 'snow',
  parallel_count = 32,
  copy_environments = TRUE
)
print(parallelize_call(subsetRegression()))
```

It is good practice to put the definition of `Parallelize_config` into a separate file and describe all resources available to the user there. This file can then be sourced into new scripts and the call to `parallelize_initialize` can quickly switch between available resources by specifying the appropriate backend.

Backend	#Parallel jobs	Time	Speed-up
OGSremote	3	9129 sec	1.00
OGSremote	15	6073 sec	1.50
OGSremote	50	6206 sec	1.47

Table 1: *Time* is the absolute waiting time by the user averaged across two runs. *Speed-up* is relative to the first line.

The example is benchmarked on a four-core machine (*Intel Core i7*) running the *Open Grid Scheduler* (OGS; Open Grid Scheduler Development Team, 2013). In this example, we investigate the influence of varying the number of parallel jobs generated. Results are listed in Table 1 and times include all waiting times induced by polling job-statuses and wait times induced by OGS (on average each job

waits 15 seconds before being started). The number of parallel jobs reflects parallelism in the program. For three jobs, each of the response levels is analyzed in parallel. Fifteen is the number of subsets of the covariates so that in this scenario the second level is parallelized (`Sapply` over the subsets). Fifty exceeds the number of subset-scenarios (45 subsets in total) so that the `Sapply` within `permute` is parallelized. Note, that in the last scenario execution is truly dynamic as the number of permutations depends on the generalized linear model (glm) computed on each subset. This implies that this glm is repeatedly computed during the re-executions, creating additional overhead. Choosing 15 jobs instead of three makes better use of the processing power so that speed-up is expected, a job count of 50 results in about the same wait time.

Increasing job counts beyond the number of parallel resources can increase speed if the parallel jobs differ in size and the overall computation depends on a single, long critical path. This path can potentially be shortened by splitting up the computation into more pieces. In this example, we could not benefit from such an effect. On the other hand, should we run the computation on a computer cluster with hundreds of available cores, we could easily create a similar amount of jobs to accommodate the new situation.

For the sake of demonstrating that the package can handle code with almost no modification, we allowed for unbound, global variables (*i.e.* functions use globally defined variables that are not passed as arguments). This forced us to use the `copy_environments = TRUE` option that makes the package look for such variables and include their definition into the job that is later executed. It is better practice in terms of using this package but also in terms of producing reproducible code in general to pass data and parameters needed for a computation explicitly as arguments to a function performing a specific analysis (analysis-function). We could also define all needed functions called from the analysis-function in separate files and list these under the `sourceFiles` key in the `Parallelize_config` variable. The package can then establish a valid compute environment by sourcing the specified files, loading listed libraries and transferring arguments of the analysis-function in which case we would not have to use the `copy_environments = TRUE` option. As a convenience measure the definition of the analysis-function itself is always copied by the package.

Example 2

The second example mimics the situation in Figure 1. It is more or less purely artificial and is meant to illustrate the overhead induced by the parallelization process.

```
parallel8 <- function(e) log(1:e) %*% log(1:e)
parallel2 <- function(e) rep(e, e) %*% 1:e * 1:e
parallel1 <- function(e) Lapply(rep(e, 15), parallel2)
parallel0 <- function() {
  r <- sapply(Lapply(1:50, parallel1), function(e) sum(as.vector(unlist(e))))
  r0 <- Lapply(1:49, parallel8)
  r
}

parallelize_initialize(Lapply_config, backend = 'local')
r <- parallelize(parallel0)
```

Backend	#Parallel jobs	Time
off	24	0.01 sec
local	24	0.14 sec
snow	24	19.80 sec
OGSremote	24	29.07 sec

Table 2: *Time* is the absolute waiting time by the user averaged across at least 10 runs.

Again, a call to `parallelize_initialize` defines parameters of the parallelization and determines the backend. Function `parallelize` then executes the parallelization. Arguments to `parallelize` are the function to parallelize together with arguments to be passed to that function. Results from benchmark runs are shown in Table 2 and again absolute clock times are listed, *i.e.* time measured from starting the computation until the result was printed. *snow* and *OGSremote* backends were run on an eight core machine with Sun Grid Engine 6.2 installed with default settings. `parallelize` was configured to produce 24 parallel jobs. *off* in Table 2 denotes time for running without any parallelization. This can always be achieved by calling `parallelize_setEnable(F)` before `parallelize` which

replaces the `Apply` functions by their native versions and `parallelize` by a function that directly calls its argument. The *local* backend performs parallelization but executes jobs linearly, thereby allowing to measure overhead. In this example overhead is ~ 0.13 seconds which is large in relative terms but unproblematic if the computation becomes larger. This overhead is roughly linear in the number of jobs generated. Comparing *snow* and *OGSremote* backends we can judge the setup time of these backends for their parallel jobs. It took *snow* a bit below a second and *OGSremote* a bit above a second to setup and run a job. It should be noted that the batch queuing characteristics are very influential for the *OGSremote* backend. This instance of the Sun Grid Engine was configured to run jobs immediately upon submission. This example does not transfer big data sets which would add to overhead, however, it seems plausible that an overhead of at most a couple of seconds per job makes parallelization worthwhile even for smaller computations in the range of many minutes to few hours.

Discussion & outlook

Limitations

Using the **parallelize.dynamic** package, existing code can be made to run in parallel with minimal effort. Certain workflows do not fit the computational model assumed here. Most notably the cost of the ramp-up determines the overhead generated by this package and might render a given computation unsuitable for this approach. In many cases re-factoring of the code should help to mitigate such overhead, however, this would render the point of convenience moot. It should also be pointed out that for a given computation for which time can be invested into choosing the code point at which to parallelize carefully and subsequently using packages like **parallel** or **foreach** should result in a more efficient solution.

Technical discussion

One way to reduce the cost of ramp-ups is to pull out code from nested loops and pre-compute their values, if possible. To help automate such a step, static code analysis can be used to separate computational steps from ramp-ups by analyzing code dependencies. Another option would be to extent the R language with an option to manipulate the "program counter", which would allow to resume code execution after a parallelization step in a very efficient manner. Such a change seems not straightforward but could also benefit debugging mechanisms.

All parallelization packages rely on function calls that are executed in parallel not to have side-effects themselves or to depend on such. It would be impractical to formally enforce this with the language features offered by R. Again, a language extension could enforce functional behavior efficiently, *i.e.* only the current environment (stack frame) may be manipulated by a function. For now, some care has to be taken by the user, however, this does not seem to be a big problem in practice. For most "statistical" applications such as simulations, bootstrapping, permutations or stochastic integration a reasonable implementation should naturally lead to side-effect free code.

Is a fully transparent solution possible? Replacing native `apply` functions directly with parallelization ones would have the benefit of requiring no modifications of the code at all. The current implementation would certainly suffer from too great an overhead, however, it is conceivable that profiling techniques (measuring computing time of individual function calls) could be used to gather prior knowledge on computational behavior of "typical" code allowing to exclude certain `apply` calls from the parallelization process. This seems a very challenging approach and requires extensive further efforts.

Conclusions

In the author's experience, most standard statistical workloads can easily be adapted to this parallelization approach and subsequently scale from the local machine to mid-size and big clusters without code modifications. Once standard configurations for the use of a local batch queuing system at a given site are created, this package can potentially dramatically broaden the audience that can make use of high performance computing.

As a final note, R is sometimes criticized for being an inefficient programming language which can be attributed to highly dynamic language features. The current implementation makes liberal use of many such features most notably introspection features to create unevaluated calls and perform their remote execution. The roughly 1000 lines of implementation (split roughly evenly between front-end and backend) demonstrate that these features are powerful and allow to execute a project such as this with a small code base. Also, the functional programming paradigm implemented in R allows

for a natural attacking point of parallelization. This can be exploited to gain computational speed in a highly automatized way, a mechanism that is hard to imitate in procedural languages which traditionally have stakes in high performance computing.

Summary

In many practical situations it is straightforward to parallelize R code. This article presents an implementation that reduces user intervention to a minimum and allows us to parallelize code by passing a given function call to the `parallelize_call` function. The major disadvantage of this implementation is the induced overhead which can often be reduced to a minimum. Advantages include that potentially little technical knowledge is required, computations can become more efficient by better control over the amount of parallelization, and finally that programs can be easily scaled to available resources. Future work is needed to reduce computational overhead and to complement this dynamic with a static analysis.

Bibliography

D. Bode. *Rsge: Interface to the SGE Queuing System*, 2012. URL http://CRAN.R-project.org/package=Rsge. R package version 0.6.3. [p88]

A. Canty and B. D. Ripley. *boot: Bootstrap R (S-Plus) Functions*, 2013. R package version 1.3-9. [p88]

K. Cooper and L. Torczon. *Engineering a Compiler*. Elsevier, Jan. 2011. ISBN 9780080916613. [p90]

A. C. Davison and D. V. Hinkley. *Bootstrap Methods and Their Applications*. Cambridge University Press, Cambridge, 1997. URL http://statwww.epfl.ch/davison/BMA/. ISBN 0-521-57391-2. [p88]

M. D. Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, 2003. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.182.5350&rep=rep1&type=pdf#page=25. [p90]

R. Ihaka and R. Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996. URL http://www.tandfonline.com/doi/abs/10.1080/10618600.1996.10474713. [p88]

K. Michael, J. W. Emerson, and S. Weston. Scalable strategies for computing with massive data. *Journal of Statistical Software*, 55(14):1–19, 2013. URL http://www.jstatsoft.org/v55/i14. [p88]

Open Grid Scheduler Development Team. Open Grid Scheduler: The official open source grid engine, 2013. URL http://gridscheduler.sourceforge.net/. [p93]

Revolution Analytics and S. Weston. *foreach: Foreach Looping Construct for R*, 2013. URL http://CRAN.R-project.org/package=foreach. R package version 1.4.1. [p88]

L. Tierney, A. J. Rossini, N. Li, and H. Sevcikova. *snow: Simple Network of Workstations*, 2013. URL http://CRAN.R-project.org/package=snow. R package version 0.3-13. [p91]

Stefan Böhringer
Leiden University Medical Center
Department of Medical Statistics and Bioinformatics
Postzone S-5-P, P.O.Box 9600
2300 RC Leiden
The Netherlands
correspondence at s-boehringer.org