Technical Report

Computing on high performance clusters with R: Packages BatchJobs and BatchExperiments

Bernd Bischl, Michel Lang,
Olaf Mersmann,
Jörg Rahnenführer, Claus Weihs

1/2012

technische universität
dortmund

SFB 876 Verfügbarkeit von
Information durch Analyse unter
Ressourcenbeschränkung

This work was partly supported by the Research Training Group "Statistical Modelling" of the German Research Foundation (DFG), the "Graduate School of Energy Efficient Production and Logistics" and by the German Research Foundation (DFG) within the Collaborative Research Center SFB 876 "Providing Information by Resource-Constrained Analysis", project A3.

We thank the team of the LiDO HPC cluster at the TU Dortmund for their technical support while developing and using both R packages.

We also thank Heike Trautmann, Oliver Flasch and Uwe Ligges for helpful discussions and comments.

Released on: 2012-05-29.

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

Abstract

Empirical analysis of statistical algorithms often demands time-consuming experiments which are best performed on high performance computing clusters. We present two R packages which greatly simplify working in batch computing environments.

The package **BatchJobs** implements the basic objects and procedures to control a batch cluster within R. It is structured around cluster versions of the well-known higher order functions `Map`, `Reduce` and `Filter` from functional programming. An important feature is that the state of computation is persistently available in a database. The user can query the status of jobs and then continue working with a desired subset.

The second package, **BatchExperiments**, is tailored for the still very general scenario of analyzing arbitrary algorithms on problem instances. It extends **BatchJobs** by letting the user define an array of jobs of the kind "apply algorithm A to problem instance P and store results". It is possible to associate statistical designs with parameters of algorithms and problems and therefore to systematically study their influence on the results.

In general our main contributions are: (a) **Portability**: Both packages use a clear and well-defined interface to the batch system which makes them applicable in most high-performance computing environments. (b) **Reproducibility**: Every computational part has an associated seed that the user can control to ensure reproducibility even when the underlying batch system changes. (c) **Efficiency**: Efficiently use batch computing clusters completely within R. (d) **Abstraction and good software design**: The code layers for algorithms, experiment definitions and execution are cleanly separated and enable the writing of readable and maintainable code.

1 Introduction

Time-consuming computer experiments play an increasingly important role in modern statistical applications for researchers and practitioners alike. While many scientists have access to powerful cluster systems or could purchase computing resources from cloud providers such as Amazon, the effort required to harness these types of systems is still substantial. The time spent to familiarize oneself with the ins and outs of these types of systems is a major hindrance to adoption – not every statistician is a proficient computer scientist. But even after overcoming this hurdle, practitioners are burdened by spartanic tooling and little or no automation to support their typical computing workflow. There are many reasons for this lack of tooling. One is that the type of workload that statistical problems induce is sometimes atypical for "traditional" high-performance computing (HPC). In traditional HPC, many nodes of a cluster are combined for hours or days at a time to solve a single problem. Statistical simulations on the other hand often call for many repetitions of the same or very similar (often smaller) jobs. They are what is called "embarrassingly parallel". Therefore, instead of running a small number (say 10-20) of jobs, we usually produce thousands of jobs which must be processed, but that do not necessarily have to communicate with each other. Also, in the increasingly important field of large scale data analysis some methods exist that rely on bagging or data partitioning techniques. These algorithms are mostly embarrassingly parallel and they often require only moderate, simple communication in between longer intervals of independent calculations.

To understand the intricacies of typical HPC clusters we have to examine how they are managed. Here, so-called job schedulers come into play, sometimes also referred to as batch queuing systems. It is their duty to assign tasks to worker nodes, manage job submissions, handle the job accounting and perform routine housekeeping. On these systems it is therefore not possible to directly start a process on a given node. Instead, we have to submit special job definitions to the batch system which in turn – based on our resource requirements – decide when and where our task is executed. This means all jobs are placed in queues and processed in some order to guarantee efficiency and fair share among users. Because the assignment of jobs to queues and the management of the queues incurs considerable overhead, these systems can usually deal with no more than a few hundred jobs at once[1].

From these rather superficial considerations we can already draw important conclusions: (a) The user has to cope with many technicalities to operate such clusters. This includes commands on the operating system level for submission and status overview as well as file formats for job definitions. Even worse, both commands and file formats are not standardized across different batch systems. (b) For very large statistical experiments we will in general not be able to submit all jobs at once. One reason is the already mentioned technical queue limit. Also, errors can occur on all stages and some jobs might have to be resubmitted. And last but not least, after we have gained more insight into the results, we might want to add further experiments. Keeping track of the state of computation

[1]There are schedulers which will work with many more jobs, but they require considerable computing power dedicated to the scheduler. Their use is usually restricted to the fastest cluster systems with many thousands of nodes.

might become difficult. (c) Furthermore, up to a point it will hold true that the smaller we can construct our tasks, the more efficiently we can use our cluster. Many small jobs that require only a few minutes or hours will be scheduled much earlier than few large ones that require days of computation time.

We provide two R (R Development Core Team, 2012) packages to work in these environments which are applicable in many different scenarios. Our design goals are:

- Allow any batch system in principle.
- Complete control of the batch system from within R.
- Let the user concentrate on the experiments instead of cluster management code.
- Build upon `Map`, `Reduce` and `Filter` from functional programming.
- Persistent state of computation for experiments.
- Efficient use of available computing resources.
- Clean separation of cluster and submission code from experiment definitions.
- Reproducibility in distributed environments, even if the architecture changes.

The first package, **BatchJobs**, implements the core infrastructure required to interact with a cluster system. It enables the user to define jobs, submit them to the batch system, query their status and collect the results. Its interface is designed to mimic the powerful functional programming concepts of `Map`, `Reduce` and `Filter` – which are also available in R under these names. Results can be further processed in parallel by mapping over them again.

The second package, **BatchExperiments**, is tailored for the general problem of studying combinations of algorithms on problem instances. This subsumes among other things:

- benchmarking experiments where multiple algorithms are compared on multiple problem instances,
- sensitivity analysis for studying the influence of algorithm parameters on algorithm performance,
- statistical simulation studies which often comprise a combination of the previous two tasks,
- parameter tuning of algorithms,
- meta learning where one tries to predict algorithm performance based on problem characteristics.

BatchExperiments builds upon the **BatchJobs** framework by letting the user quickly define jobs of the kind "apply algorithm A to problem instance P". It is furthermore

possible to associate a statistical design with the parameters of an algorithm or problem so that jobs are automatically generated for all parameterizations.

Both packages are written in such a way that they can be used on almost any cluster. Even a loosely coupled set of nodes which are only accessible via SSH can be employed as a makeshift cluster. This is achieved by using an abstract interface to specify how the **BatchJobs** package interacts with the cluster. Several implementations of these so-called “cluster functions” are provided in the package but system administrators and users are free to implement their own versions which fit their environments. It should be noted that this is a one-time effort for a site and we have spent a considerable amount of time on providing flexible generic cluster functions which should work for a large number of installations.

The next Chapter 2 gives a brief overview of other comparable work. The following Chapters 3 and 4 introduce the packages **BatchJobs** and **BatchExperiments**. For readers with a strong focus on applied work we recommend to mainly concentrate on the chapter about **BatchExperiments**. Nevertheless, the sections from **BatchJobs** on the configuration file (3.1), on the registry (3.2), on submitting jobs (3.6), on status queries (3.7), on collecting results (3.8) and on debugging (3.11) are crucial for usage and should be read as well. Chapter 5 addresses the aspects of reproducibility in computational statistics, Chapter 6 finally provides a conclusion and an outlook.

2 Review of other relevant work

A wealth of R packages for HPC is available. As of writing, the CRAN Task View “High-Performance and Parallel Computing with R”[2] lists over 60 packages for HPC and parallel computing in R. Of these, some provide low-level parallel computing functions, others contain specialized parallel versions of certain algorithms and still others serve as bridges to other computing resources or frameworks. The most widely used low-level packages are **Rmpi** (Yu, 2010) and **nws** (Revolution Analytics and Pfizer, 2010). These packages allow the explicit parallelization of R functions by using, e. g., the Message Passing Interface (MPI) or the network spaces (NWS) abstraction. All of these packages require extensive reworking of the code that should be executed concurrently and are meant for settings where the parallel tasks need to communicate with each other. They are best used in traditional HPC settings when there is one big task that can be parallelized and there is significant communication between its subtasks to synchronize or exchange temporary results.

Since these low-level interfaces are somewhat cumbersome to use, there are a variety of wrappers for these packages which provide a more abstract interface to the underlying cluster. Examples are **snow** (Tierney *et al.*, 2011), **snowfall** (Knaus, 2010) and **foreach**[3] (Revolution Analytics, 2011). Most of these packages provide parallel versions of R’s

[2] `http://cran.r-project.org/web/views/HighPerformanceComputing.html`

[3] **foreach** is actually just a front-end which requires a back end to dispatch the work. There are many different packages on CRAN that can serve as a back end for **foreach**. They are all named **do**<**technology**>.

`apply` family of functions. This is also true for **multicore** (Urbanek, 2011) which includes a parallel version of `lapply` for multiple cores or CPUs. A notable exception to this programming style is **foreach** which mimics a parallel `for` loop. A good overview of these functions and their respective advantages and disadvantages is given in Schmidberger *et al.* (2009).

In addition to these, there is a plethora of specialized packages that implement parallel versions of statistical algorithms. They usually employ one of the above mentioned packages internally to access the parallel computing resources. A notable exception is the **SPRINT** package (Hill *et al.*, 2008) which contains only a few highly optimized routines that are explicitly tuned to run well on large HPC clusters. There are also packages to tap into the vast computing power of modern graphics processors, but these currently require considerable low-level programming skills w. r. t. the graphics processing unit and it is especially hard to map memory-intensive algorithms to these architectures.

All of the packages described so far are "synchronous" in that they require a running R process to orchestrate the parallel execution on multiple nodes that have been allocated using a job scheduler. While this is certainly feasible for, e. g., small benchmark studies, larger experiments tend to result in job allocations that are too large for the cluster to satisfy. What we really want is a mechanism to run our tasks in an "asynchronous" fashion. That means, we want to define all our tasks and then let the job scheduler execute these as resources become available. While this is conceptually possible with current batch systems, practice has shown that they cannot cope with the huge number of jobs produced by this approach. Even moderately large experiments can easily produce tens of thousands of tasks that need to be executed and, while each task may only run for a couple of minutes, their sheer number will in almost all situations overwhelm the job scheduler. We therefore started to develop tools that let us submit just a few jobs, wait until they were finished and then submit some more, thereby running all jobs in such a piecemeal fashion. Over time this collection of utilities grew and was then rewritten as a proper R package resulting in what is now the **BatchJobs** package.

BatchJobs is not the first package which implements this idea, there is another R package named **batch** (Hoffmann, 2011) which is similar but less fully featured. It provides facilities to submit an R script to a batch system and combine the results as data frames. Parameters have to be passed as command line arguments and error handling is less sophisticated compared to our package.

Another package under active development is **RHIPE** (Cleveland *et al.*, 2011) which integrates R into a Hadoop[4] environment. Apache's Hadoop is a derivation of Google's MapReduce framework (Dean and Ghemawat, 2008) to support running applications in distributed environments, especially for the purpose of handling very large data. The scope of the Hadoop system is however not as broad as what we are aiming for. Hadoop focuses on efficiently processing massive amounts of data distributed over many nodes by running the actual analysis code as "close" to the data as possible, ideally on the node where the data is stored. This may fit some problems naturally, but many statistical tasks do not map well to the Hadoop framework.

[4] `http://hadoop.apache.org/`

Finally it should be mentioned that there are language agnostic approaches to describe these types of workflows. An example of such a system is **makeflow** (Bui *et al.*, 2011).

3 BatchJobs

The R package **BatchJobs** provides the basic infrastructure and abstractions to work with R on any cluster or batch system. First, you create a so-called registry object which defines a directory where all relevant information, files and results of the computational jobs will be stored. We currently require the cluster to provide a shared filesystem for all computational nodes. All jobs are declared at the registry and their computational status is held in a database. Therefore, the registry bundles the access to both the database and the information stored on the filesystem. It is automatically saved and can be reused later, e. g., when you login to the system again. The registry provides a persistent description of the experimental setup and computational state.

Jobs are not submitted when they are created, instead their definition is stored in the database until they are explicitly submitted to the cluster. Every job has a unique ID that you can use as a reference. Most functions of the **BatchJobs** and **BatchExperiments** packages allow or require a vector of these IDs as an argument. Jobs communicate with the registry via the database, so their status is available to you on the master node at all times. You do not have to query the database yourself (or be aware of its existence for that matter), as it is transparently managed. While jobs are running, you can query their status, find out which ones have completed successfully and which have failed by raising an R exception. It is even possible to configure the system to send you a status email after each individual job or after the termination of the last submitted job. Internally the package **sendmailR** (Mersmann, 2011) is used for this, note that its documented restrictions therefore apply to this feature. We also provide functions to aid in debugging, as this is one of the most important and time-consuming tasks when working with parallel programs.

The usual tasks to perform with both packages are summarized in Figure 1. All these functions will be explained in more detail in the current and in the following chapter.

3.1 Configuration file

After installing the package you should set up a short configuration file. The configuration is a concise description of your computing environment and personal settings. The file itself is an R script that is sourced when **BatchJobs** is loaded. To explain its makeup, let us look at an exemplary configuration:

```
cluster.functions <- makeClusterFunctionsTorque("~/torque.tmpl")
mail.start <- "first"
mail.done <- "last"
mail.error <- "all"
mail.from <- "<bischl@lidong1.itmc.tu-dortmund.de>"
```

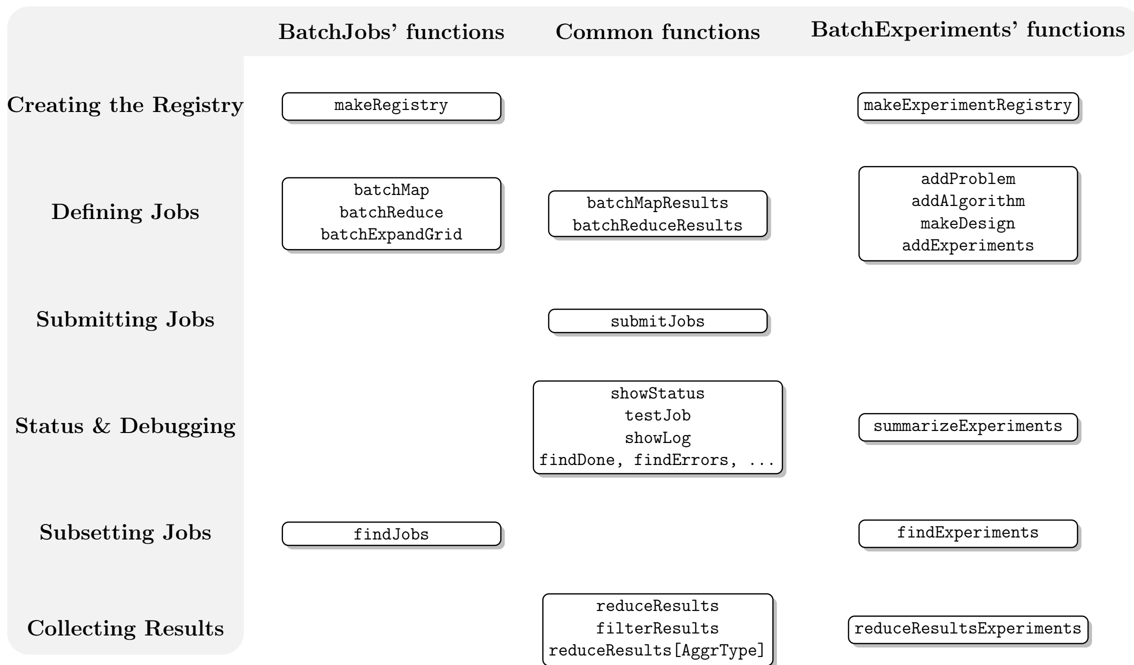

Figure 1: Overview of the most important functions grouped by package and task. Common functions are functions from **BatchJobs** that are also useful when working with **BatchExperiments**. `reduceResults[AggrType]` is a convience wrapper around `reduceResults` for aggregation into the types `Vector`, `Matrix`, `DataFrame` or `List`.

```
mail.to <- "<bischl@statistik.tu-dortmund.de>"
mail.control <- list(smtpServer="mail.statistik.tu-dortmund.de")
```

The first line specifies the batch system you are using. Here, we use a TORQUE-based system. TORQUE[5] (Terascale Open-Source Resource and QUEue Manager) is a distributed resource manager that allows to handle batch jobs on distributed compute nodes. Jobs are scheduled by writing short Portable Batch System (PBS) scripts on such clusters, and we have specified the path to a template PBS file here. Working on another architecture simply means exchanging this single line in your configuration. Further details regarding this topic are discussed in Section 3.12.

The following lines concern the status mails that are potentially sent when important events occur in a job's life cycle. Any of the options `mail.start`, `mail.done` and `mail.error` can be set to either `"none"`, `"first"`, `"last"`, `"first+last"` or `"all"`, defining for which jobs status mails are generated at their state transitions, see Section 3.7 for a more detailed explanation. The remaining options set the return and recipient address fields for the status mails and the options passed to **sendmailR**. For the latter, setting the SMTP server should be sufficient in most cases.

The configuration file must be named `.BatchJobs.R` and placed at one of three possible locations. If multiple configuration files are found, the more user-specific settings will overwrite the more general settings in the following order: package directory $\prec$ user home

[5] `http://www.adaptivecomputing.com/products/torque.php`

directory ≺ working directory. The configuration file in the package directory allows site administrators to define reasonable global settings for all users. Placing a configuration file in your home directory should suffice for most users, but for some scenarios it might be necessary to define project specific configurations in the working directory of your current R session. The default settings are to run jobs sequentially in the same R session and never to send any status mails.

3.2 Job registry

The registry is the central object for any **BatchJobs** project. It contains all meta-data about jobs. To create a new registry, run:

```
reg <- makeRegistry(id="my_reg", seed=123, file.dir="my_job_dir")

Creating dir: my_job_dir
Saving registry: /home/lang/my_job_dir/registry.RData
```

The most important arguments are:

`id`: Name of your registry. Displayed in mails and cluster queue.

`file.dir`: This is the directory where the registry, the database, R scripts, log files and all results are stored. If not defined, it defaults to the subdirectory `<id>_files` in your current R working directory. **BatchJobs** will not overwrite an existing directory.

`work.dir`: This directory will be used as working directory when your jobs get executed. Useful if you want to source other R files or load data in your jobs.

`packages`: These R packages are loaded for each job before it is executed.

`seed`: This seed will be used for the first job and is incremented for each subsequent job. Useful to generate reproducible results. Defaults to a random positive integer if not set. See Chapter 5 for further details on the seeding mechanism.

`sharding`: With sharding set to `TRUE`, the job and result files will be distributed over several subdirectories. This avoids problems with some (network) filesystems that perform poorly when there are thousands of files in a single directory. Defaults to `TRUE`.

`multiple.result.files`: If this is set to `TRUE` and your jobs return a list, each list item will be saved in a separate file. This enables partial loading of results when further processing them. This is useful when you store large objects at the end of your jobs that are rarely required in subsequent analysis. All functions in **BatchJobs** as well as in **BatchExperiments** that operate on results allow the automatic reconstruction of such lists.

The registry is stored in the defined `file.dir` and its database is continually updated when jobs are registered. Together both objects provide a persistent state of the computation. In a later session you can use the function `loadRegistry(file.dir)` and resume working.

3.3 Filesystem layout

The `file.dir` defined in your registry contains all files belonging to this registry that are created during the lifetime of your project. In general, the user does not need to worry about the technical details of how information is stored in this file hierarchy since everything is accessible through utility functions. However, for completeness' sake, we will explain its structure in the following paragraphs. The following elements are always present in the main directory:

`registry.RData`: The stored registry.

`BatchJobs.db`: An SQLite database which contains all transactional information about jobs and their (computational) state. See Section 3.4. Handled transparently.

`conf.RData`: The user configuration of **BatchJobs**. Will be loaded on the slave nodes. If the user changes his configuration during his session, the most recent configuration will always be stored here.

`functions`: Functions subdirectory. Contains mapped functions as `.RData` files. The file is named with a `digest` hash. This directory mainly exists for later extensibility, as we currently always map only one single function over a list in **BatchJobs**.

`jobs`: Jobs subdirectory. It contains the individual job files explained below. If the option `sharding` in `makeRegistry` is set to `TRUE` (the default), these files are distributed over many subdirectories (shards) in order to bypass problems occuring when working with directories which contain many thousands of files.

There are three files for each job (start script, result, log) in the jobs directory, and they are named with the job ID the following way:

`<ID>.R`: A small R script which loads the registry, executes a single job and stores the result. The complete workflow resulting from its execution is explained in detail in Appendix A.

`<ID>.out`: The log file of a single job, produced by `R CMD BATCH`. Can be displayed with `showLog`.

`<ID>.RData`: The stored return value of a single job. If `multiple.result.files` is set to `TRUE` in `makeRegistry`, each element of the named list that is returned by the job will be saved as `<ID>_<list_element_name>.RData`.

3.4 Job database and message passing protocol

The job database holds two tables with all information necessary to describe a job, identify it on the batch system and query its computational status. The first table stores informations about the job's status. This includes the **BatchJobs** job ID, the seed, timestamps of events (submit, start, termination), potentially an error message and further technical information like the name of the executing cluster node, the ID of the job assigned by the batch system or the R process ID. The job ID is a primary key (automatically incremented positive integer) for this table. Hence, all job IDs are assured to be unique. A further column contains the job definition ID to relationally link to the second database table, the job definition table. This table holds informations about the filename of the function to call and its parameters.

We use RSQLite (James, 2011) as interface to the SQLite[6] database back end. SQLite calls itself a "self-contained, serverless, zero-configuration, transactional SQL database engine". This means that you do not have to install or configure a database server yourself. Also, you do not have to worry whether it is possible that your jobs are allowed to communicate with the database server when they get executed on a node because they can directly access the database file on the shared filesystem.

As the jobs are concurrently executed, we write their status into the database, making sure that the ACID (atomicity, consistency, isolation, durability) properties hold. A usual sequence of messages for a job looks like this:

1. submit message: The job gets submitted to the batch system, its ID on the batch system and timestamp are stored.
2. start message: When all resources are available, the job gets executed and the timestamp is stored.
3. done message: When the job has successfully completed, its result file is written and the timestamp is stored.
4. error message: If an error occurs, the job is terminated and the R exception message is stored.

We also experimented with other DBMS like MySQL[7] or PostgreSQL[8]. Albeit these are currently not supported, the use of a fully featured database server could increase the overall performance of all database transactions. The major disadvantage of SQLite is that it does not support row-level locking, i. e., the whole database has to be locked for a write operation. This can lead to lock congestion if very many jobs access the database in parallel, and the effect gets worse the faster the jobs terminate. As every database operation on the worker only relates to a single, unique row of the `"job status"` table, row-level locking (which is supported by both MySQL and PostgreSQL) would probably eliminate this problem completely.

[6] `http://www.sqlite.org`
[7] `http://www.mysql.com`
[8] `http://www.postgresql.org`

However, we did not observe performance problems with SQLite in our studies. Nevertheless the support of these alternative DBMS is an option for future versions of the packages.

3.5 Mapping, reducing and filtering

The higher order functions `Map`, `Reduce` and `Filter` are arguably the most important building blocks of functional programming and – as the name suggests – also appear in the famous MapReduce framework (Dean and Ghemawat, 2008). Most packages for parallelization in R provide a parallelized version of Map. This package is not different in this regard, with the exception that – as we work on a batch system – the corresponding jobs are not submitted at once but only defined. They can be submitted in arbitrary subsets.

The following overview lists the most important operations provided in **BatchJobs** for computing on the cluster and obtaining the results:

`batchMap`: Applies a function over a list or vector. A batch version of R's `Map` function which is basically the same as `lapply`. One job is one function application.

`batchExpandGrid`: Generates a cross product of parameter vectors and applies a function to each combination. A simple wrapper for `expand.grid` and `batchMap`.

`batchReduce`: Reduces (aggregates) a list or vector with a binary function. The binary function is successively called to combine the elements of the vector – think of reducing a numeric vector with the "+"-operation. In other languages this concept is also known as folding or accumulation. `batchReduce` is nothing more than a batch aware version of R's `Reduce` function. One job is one reduction of a shorter sub-list. The results can then be reduced one final time on the master node.

`batchMapResults`, `batchReduceResults`: Same as `batchMap` and `batchReduce` but operate on the results of previous operations. Will be covered in Section 3.9.

`reduceResults`: Loads and reduces your stored results on the master after jobs have terminated.

`filterResults`: Filters all results with a predicate. Essentially works like R's `Filter` function.

In the next code snippet we demonstrate the parallel calculation of the square of the first 100 natural numbers. Note that only the first 50 jobs are actually submitted to the batch system by using the IDs of the jobs.

```
xs <- 1:100
batchMap(reg, function(x) x^2, xs)
ids <- getJobIds(reg)
submitJobs(reg, ids[1:50])
```

Parallel reduction is also possible by partitioning a list or vector into blocks and reducing each block as an individual job. This means successively combining all elements of a block with a binary function to an aggregated object. In the next example we partition the vector of the first 100 natural numbers into blocks of size 10 and calculate their respective sums.

```
xs <- 1:100
r <- function(aggr, x) aggr + x
batchReduce(reg, r, xs, init=0, block.size=10)
submitJobs(reg)
sapply(getJobIds(reg), loadResult, reg=reg)
```

```
[1]  55 155 255 355 455 555 655 755 855 955
```

In the final line we apply the function `loadResult` over all job IDs of the registry to retrieve the partial sums. Further information regarding the collection of results are given in Section 3.8.

3.6 Submitting jobs

Jobs can be submitted to the batch system via the `submitJobs` function which already appeared in some of the previous examples. It takes the registry and the selected job IDs as arguments, as well as an arbitrary list of resource requirements which are to be handled by the cluster back end. Usual examples for the latter are maximal wall time and required memory. The function tries to submit as many jobs as possible. If either system limits or user limits are exhausted, the function waits until submission is possible again. By default an exponential back-off mechanism is used where the waiting time is doubled after each unsuccessful retry. But this is customizable as well, as the following example shows:

```
submitJobs(reg, resources=list(walltime=3600, memory=4*1024),
           wait=function(retries) 100, max.retries=10)
```

Here, we have passed the wall time in seconds, requested 4 gigabytes of memory per job and told the system to wait for 100 seconds in case of temporary submission errors like filled queues. After 10 unsuccessful tries the submission would finally terminate with an exception.

In some cases it can be advantageous to set the computational resources differently for subsets of the jobs. As this is more likely to happen with `BatchExperiments` where your experimental setup might contain algorithms of very different run times or problems that vastly differ in size, we demonstrate how this works in the example of Section 4.4.

3.7 Status queries and status mails

After you have submitted all or a subset of your jobs to the batch system, you can query their status by using the `showStatus` function.

```
showStatus(reg)
```

```
Status for jobs:  200
Submitted:        200 (100.00%)
Started:          200 (100.00%)
Running:          100 ( 50.00%)
Done:             100 ( 50.00%)
Errors:             0 (  0.00%)
Expired:            0 (  0.00%)
Time: min=0.00 avg=0.00 max=1.00
```

The resulting output includes the number of jobs in the registry, how many: have currently been submitted, have started to execute on the batch system, are currently running, have successfully completed, have terminated due to an R exception or have expired because they hit the wall time or required too much memory.

The user should always call this function if he is unsure about the current computational status.

The last line shows the minimal, average and maximal execution times for all selected jobs. This is useful if the user has to set wall times and does not exactly know how long each job will take or if there are vast differences in execution times. You can restrict `showStatus` to take only a subset of jobs into account by passing a vector of job IDs. See Section 4.4 for an example on how to extract specific job IDs from the registry. If errors occurred, the first error messages are displayed as well, see Section 3.11 for further remarks on debugging.

Simple helper functions that query the computational state of your jobs and which all return vectors of job IDs are:

`findSubmitted`: Jobs that have been submitted with `submitJobs`.

`findStarted`: Jobs that have already begun to execute.

`findDone`: Jobs that terminated successfully.

`findMissingResults`: Jobs where results are still missing.

`findErrors`: Jobs that terminated due to an exception.

`findOnSystem`: Jobs that currently "live" on the batch system. This includes idle, blocked, running, etc. jobs.

`findRunning`: Jobs that are currently running.

findExpired: Jobs that have probably hit the wall time or were terminated by the scheduler due to excessive memory usage and such. This is quite difficult to figure out (as we have lost control over the R process under these conditions), therefore we employ a simple heuristic here which works well most times: Started, not done, no R exception, not currently on batch system.

BatchJobs also contains a configurable status mailer that internally uses the package **sendmailR**. The mail options were already briefly introduced in Section 3.1. Mail sending is triggered at the start of the job (`mail.start`), the successful completion (`mail.done`) and the termination by an R exception (`mail.error`). For each of these events you can define for which jobs mails should be sent. You can set these options to `"none"` to receive no mails at all, `"first"` for the first job, `"last"` for the last job, `"first+last"` for both the first and last job and finally `"all"` for all jobs.

You can combine these options for fine adjustment of the mail notification mechanism – with `mail.start="first"`, `mail.done="last"` and `mail.error="all"` you will receive a mail right before the scheduler starts your first job and as soon as the last scheduled job is finished. Furthermore, all jobs with errors will trigger mails immediately. Note that when the option `mail.done="last"` triggers its status mail, not all jobs will have necessarily completed. However, this is an acceptable heuristic time point to inform the user of the nearby completion of the computation. Also, the mail of the last submitted job to the batch system includes the current output of `showStatus`, so the percentage of still running jobs is immediately available. A typical status mail includes information about the respective job, a summary of its results and a status output:

```
### Ids ########################################
# 10
###############################################

### Job Info ###################################
# BatchJobs job:
#   Job id: 10
#   Fun id: 9bfef973a2c6345763ce3bfb76394874
#   Fun formals: x
#   Pars: <unnamed>=10
#   Seed: 132
###############################################

### Results ####################################
#  num 100
###############################################
```

3.8 Collecting results

We provide the function `reduceResults` to collect the results on the master node. The function operates by default on all currently available results, but can easily be restricted

to another subset utilizing job IDs.

The passed reduce function must have the formal arguments `aggr`, `job` and `res`. The argument `aggr` contains the so far aggregated results, while `job` holds a job description object and the job result is passed as `res`. `job` is rarely needed, but can be useful in **BatchExperiments** when a job contains the description of an experiment, e. g., names of the current problem and algorithm and their respective parameters.

We show three common examples to accumulate results by an arithmetic operator, collecting them into a numeric vector and collecting them into a data frame.

```
xs <- seq(1, 5)
batchMap(reg, function(x) x^2, xs)
submitJobs(reg)
reduceResults(reg, fun=function(aggr, job, res) aggr+res)

[1] 55

reduceResults(reg, fun=function(aggr, job, res) c(aggr, res))

[1]  1  4  9 16 25

reduceResults(reg, fun=function(aggr, job, res)
rbind(aggr, data.frame(foo=res)), init = data.frame())

  foo
1   1
2   4
3   9
4  16
5  25
```

Note that in the last reduction above we have specified an initialization value. If not set, the first result object is used.

For convenience we also provide wrappers for `reduceResults` for the most common aggregation types. Here, `init` can be omitted and the provided function `fun` has only `job` and `res` as arguments. It again selects the desired elements for reduction:

- `reduceResultsVector`: Combines the respective results of `fun` with `c`.
- `reduceResultsMatrix`: Coerces the results of `fun` to a vector and combines with `rbind`/`cbind`.
- `reduceResultsDataFrame`: Coerces the results of `fun` to a data frame and combines with `rbind`.
- `reduceResultsList`: Coerces the results of `fun` to a list and combines with `c`.

If you omit the argument `fun` it defaults to `function(job, res) res`.

3.9 Transforming results

It is also possible to operate further on the stored results of a registry. This might be necessary if subsequent steps or further analysis are so time-consuming that they should be performed on the cluster as well. For this purpose **BatchJobs** provides the functions `batchMapResults` and `batchReduceResults` which behave very similar to their respective counterparts `batchMap` and `batchReduce`. These functions generate a new job on the batch system, applying a user defined function over the result files.

Let us again perform some very trivial calculations on a vector of numbers to demonstrate the principle. All performed operations should be directly understandable from the commented code:

```
# square some numbers
reg1 <- makeRegistry(id="squaring")
f <- function(x) x^2
batchMap(reg1, f, 1:10)
submitJobs(reg1)

# look at results
reduceResultsVector(reg1)

[1]   1   4   9  16  25  36  49  64  81 100

# transform the results back by calculating the square root
reg2 <- makeRegistry(id="root")
batchMapResults(reg1, reg2, fun=function(job,res) sqrt(res))
submitJobs(reg2)

# check results
reduceResultsVector(reg2)

[1]  1  2  3  4  5  6  7  8  9 10

# now reduce them in parallel by summing them
reduce <- function(aggr, job, res) aggr+res
reg3 <- makeRegistry(id="partial_sums")
batchReduceResults(reg2, reg3, fun=reduce,
  init=0, block.size=3)
submitJobs(reg3)

# check results
reduceResultsVector(reg3)

[1]  6 15 24 10
```

```
# finally reduce on the master to total sum
reduceResults(reg3, fun=reduce)

[1] 55
```

3.10 Chunking of small jobs

In a scenario with thousands of fast executing jobs, computation on classical HPC systems is problematic. The vast number of jobs puts much stress on the scheduler and the time starting R sessions on the slaves may exceed the time required to actually do a single computation.

As a resort for such scenarios we offer to block jobs together into chunks which will get executed sequentially on the slaves. If you pass `ids` to `submitJobs` as a list of IDs, then each list element defines a chunk and `submitJobs` internally combines them into one job for the batch system. You can either create such a list yourself or use the helper function `chunk`. Note that the vector `ids` get shuffled per default by `chunk` which balances differences in computation time when we execute the chunks.

In order to mitigate the already mentioned lock congestion problem with SQLite because of the many, quickly terminating jobs, we cache the write operations on the worker for some time and then efficiently flush them to the database in one go. To ensure efficiency we recommend to build chunks with an execution time of at least 10 minutes.

3.11 Debugging tools

As you can imagine, in any experiment many things can and will go wrong. The cluster might have an outage, jobs may run into resource limits or crash, subtle bugs in your code could be triggered or any other error condition might arise. In these situations it is important to quickly determine what went wrong and to recompute only the minimal number of required jobs.

While **BatchJobs** cannot handle all conceivable error conditions, it does include extensive functionality to aid in debugging problems when they arise. Large parts of your code can and should be tested independently of the batch system. For complex projects you can turn to test-driven development and use either **testthat** (Wickham, 2011b) or **RUnit** (Burger *et al.*, 2010) to define your tests, possibly by directly developing an R package. But other parts directly have to do with the execution of the experiment. Nearly everybody makes typing errors in first versions, forgets to load a required package or makes other trivial or less trivial mistakes. It is not very efficient to figure out these types of things while working live on the batch system. Therefore, before you submit anything you should use `testJob(reg, id)` with `id` as a job ID to catch errors that are easy to spot because they are raised in many or all jobs. This function runs the job in an independent R process on your local machine via `R CMD BATCH` exactly as on the slave, redirects the output of the process to your R console, loads the job result and returns it.

It is important to note that the relevant parts of your registry's file directory are copied to the system's temporary directory, so no side effect during this test can affect your current project.

When you have submitted jobs and suspect that something is going wrong, the first thing to do is to run `showStatus` to display a summary of what has already run, what needs to run and what errors were caught. Assume we run the following artificial example on our cluster:

```
flakyFunction = function(value) {
  if (value %in% 2:3) stop("Ooops.")
  value^2
}

reg <- makeRegistry(id="error")
batchMap(reg, flakyFunction, 1:4)
submitJobs(reg)
```

Two of our four jobs will fail. If we call `showStatus` after all jobs have been processed we get the following output:

```
showStatus(reg)
```

```
Status for jobs:    4
Submitted:          4 (100.00%)
Started:            4 (100.00%)
Running:            0 (  0.00%)
Done:               2 ( 50.00%)
Errors:             2 ( 50.00%)
Expired:            0 (  0.00%)
Time: min=0.00 avg=0.00 max=1.00

Showing first 2 errors:
Error in 2: Error in function (value)  : Ooops.
Error in 3: Error in function (value)  : Ooops.
```

We see that we have four jobs which have all been submitted and started. Two of these jobs completed successfully and two of them terminated with an error condition.

Finally, `showStatus` outputs the first few errors that were caught. If we want to get the IDs of all jobs that failed due to an error we can use:

```
failed <- findErrors(reg)
failed
```

```
 [1] 2 3
```

And if we want to peek into the R log file of the job to see more context for the error we can use

```
showLog(reg, failed[1])
```

which will show the log for the first failed job.

Finally we can fix our code by using `setJobFunction`. This function allows the redefinition of the mapped function for some jobs as a last measure when errors have occurred and we want to keep our already calculated results. We can then resubmit the jobs with missing results by running:

```
setJobFunction(reg, failed, fun=function(value) value^2)
submitJobs(reg, failed)
```

If unwanted jobs or jobs with programming bugs are still running on the cluster you can manually terminate them with the function `killJobs(reg, ids)`.

3.12 Supported batch systems and how to integrate a new one

BatchJobs is designed to work on any cluster using any type of batch system. To achieve this goal, all interaction with the batch system is delegated to a so-called "cluster functions" interface. Its implementation must provide the following three operations:

`submitJob`: Submit a new job to the cluster system. The name of the job, the R script to run, the path where the log file must be placed and the required resources are provided so that the function can allocate the job to an appropriate node. The function is expected to return a `SubmitJobResult` object created with the `makeSubmitJobResult` function. The result is composed of a status code (0 to indicate success or any positive integer to indicate failure), a batch job ID to identify this job on the batch system and optionally a message string to further clarify the error code.

`killJob`: Kill a running job given its batch job ID as returned by `submitJob`. This function is expected to always succeed, i. e., all possible exception handling has to be done internally. Not being able to finally kill a job after all available measures have been taken is considered as a drastic error because this would imply that you have lost control over the batch system at this point.

`listJobs`: Return a list of all scheduled jobs. Technically this is a character vector of batch job IDs.

All high-level functionality is built upon these simple operations. In most cases you will not need to write these functions yourself. Instead you will be able to use the provided constructors of the package in your configuration file to choose an existing implementation, see section 3.1. This enables you to swap the back end on the fly if you

need to switch to a different cluster system, say scale up from a local lab cluster to a larger cluster at a remote computing facility, or if a different user wants to replicate the results, but does not have access to the same type of cluster as you. We feel that this is an important aspect in making these types of large scale experiments accessible for collaborative work and review.

The package currently provides the following implementations:

Interactive execution (`makeClusterFunctionsInteractive`): All jobs are executed sequentially in the same R session. This setting is the default and provided for small toy examples and to try out the package.

Multicore execution (`makeClusterFunctionsMulticore`): All jobs are executed in parallel on the local machine in independent R processes. The multicore cluster functions are very similar to the following SSH cluster functions. The options `ncpus`, `max.load` and `max.jobs` are available, for their description and the scheduling heuristic see below.

SSH cluster (`makeClusterFunctionsSSH`): Jobs are distributed to different (Linux) nodes using the secure shell (SSH) as the underlying communication layer. All nodes must be accessible without manually entering passwords (e. g., by ssh-agent or passwordless pubkey). This mode is suited for ad-hoc clusters of workstations when you do not have access to a true batch system. But it is somewhat fragile in production use because only rudimentary resource control can be provided. The implementation tries very hard not to overload nodes, but since others may start jobs without our knowledge on any node at any time this cannot be avoided entirely. To configure a makeshift SSH cluster, use the constructor and pass it workers constructed with `makeSSHWorker`. The latter function requires the hostname of the node, the number of available cores `ncpus` (otherwise auto-detected) and the installation directory of R (if it is not the same on the worker as on the master and not in the systems `PATH` environment variable). Resource management can be achieved by setting the `max.load` and `max.jobs` arguments. During job submission, the following rules determine whether it is currently disallowed to schedule a job to a worker:

- There is an upper limit on the number of open R processes of all users. Currently this is 3 times the number of cores of the worker.
- There are already as many "expensive" R processes running (by any user) as there are cores. Such a process is defined to have a load greater than 50%.
- The current total load of the worker already exceeds `max.load`. The default for this setting is the number of CPU cores minus one.
- No more than `max.jobs` are allowed to run on the worker in parallel for the current registry. The default for this setting is the number of cores.

TORQUE/PBS cluster (`makeClusterFunctionsTorque`): Jobs are submitted to a cluster managed by TORQUE which is used by many medium and large size computing sites. Jobs are submitted using the `qsub` command, killed using `qdel` and

queried using `qselect`. Given that these are part of the original Portable Batch System upon which TORQUE is built, any PBS derivative will likely work with this implementation.

Since each cluster is different and has different requirements for the job files, a flexible approach is used where the user supplies a job file template and **brew** (Horner, 2011) is used to turn this into a PBS job file. For details on how this is done and what variables are available in the template see the package documentation for the constructor. We have also provided a very simple and a much more complicated template file in the examples directory of the package.

Oracle/Sun Grid Engine (SGE) (`makeClusterFunctionsSGE`): Jobs are submitted to a cluster managed by Sun Grid Engine[9] in a similar fashion as for TORQUE. Jobs are submitted using the `qsub` command, killed using `qdel` and queried using `qstat`. Again, the user supplies a job file template and **brew** is used to turn this into a `.job` file.

Load Sharing Facility (`makeClusterFunctionsLSF`): The interface to an LSF[10] cluster is again very similar to the TORQUE implementation and also uses a configurable **brew** template. The respective commands for submitting, killing and status queries are `bsub`, `bkill` and `bjobs`.

While the above six cluster function implementations cover a wide variety of situations and systems, they may not work with the system available to you. In that case you will have to write code to implement the three operations described previously. But this is not hard and a one time effort. Anyone interested in writing a custom cluster function implementation is encouraged to look at the interface specification[11] and the source code of the existing implementations.

4 BatchExperiments

The package **BatchExperiments** expands **BatchJobs** with an abstraction layer for the very general task of applying a set of algorithms to a set of problems and recording some arbitrary results. Both, problems and algorithms, may be parametrized in an arbitrary way and these parameters can be varied according to a statistical design. We call a problem with a specific parameter setting a problem instance and define an experiment as an application of an algorithm (together with its parameter values) to a problem instance. As experiments might be stochastic, each one can be replicated any number of times.

In our opinion, a large number of applied statistical tasks can be mapped to this abstraction, especially in the domains of benchmarking and statistical evaluation. Many articles nowadays include simulation studies and comparisons to alternative methods. Moreover,

[9] `http://www.oracle.com/us/products/tools/index.html`
[10] `http://www.platform.com/workload-management/high-performance-computing`
[11] `http://code.google.com/p/batchjobs/wiki/ClusterFunctions`

for a lot of statistical domains large and meaningful comparison studies are still missing. Many researchers have experienced the fact that theoretical results and guarantees quite often tell only part of a method's story, as mathematical assumptions will nearly always be violated to some degree in practice. Empirical knowledge about a method's behavior and characteristics are equally important. In order to generate the data for such an analysis, we need two fundamental ingredients: First, enough computational power. This is already accessible to many scientists and the situation will further improve when we take general technological progress and specifically the rapid development in the cloud computing area into account. Secondly, a framework to succinctly define these experiments; quickly, cleanly and in a reproducible fashion.

BatchExperiments combines exactly these two aspects. It allows you to compare many candidate methods to each other, work with large problem domains instead of a few, possibly unrepresentative instances, and investigate the influence of algorithm parameters and problem characteristics on performance measures (sensitivity analysis). "Benchmarking experiments" might subsume all of this under a single term, although **BatchExperiments** does not force you to follow a specific, formal methodology in your experiments or analysis. You are free to set them up and focus on individual aspects as you like. You might even start to build large, shared, growing databases of such empirical results (see Vanschoren *et al.* (2011) for a recent example). From these, sophisticated statistical models might be derived to further enhance our understanding of statistical algorithms. Or one might be able to construct automatic algorithm selection mechanisms, see for example the area of meta-learning in machine learning or hyper-heuristics in optimization. Actually, just the organized, machine-readable collection and public accessibility of such results would be a huge step forward in many areas of statistics.

In the following sections, we will stick to a rather simple, but not unrealistic example to explain the package's functionality. We will apply two classification algorithms on the famous iris data set, vary a few of their hyperparameters and record the classification performance.

Just as in **BatchJobs**, we use a registry as the central meta-data object which records technical details and the setup of the experiments. The internals are slightly different, therefore a special experiment registry is needed. The (subtle) differences of the underlying filesystem and database structure are described in Section 4.6. An experiment registry is created using the `makeExperimentRegistry` function which has the same arguments as `makeRegistry` (see Section 3.2):

```
library(BatchExperiments)
reg <- makeExperimentRegistry(id="my_experiments")
```

Once the registry is created, you can start to add problems, algorithms and designs. Experiments can then be created from these building blocks.

4.1 Problems and Algorithms

Some problems we encounter in practice are based on a "static" data object like a matrix, data frame or a multidimensional array that always stays the same for all subsequent

experiments. Other problems (or problem parts) are of a more "dynamic" nature. This means that the problem instance (or instance part) is either created stochastically, e. g., by sampling, or depends on problem parameters. With this in mind, we opted for a unified interface which deals with both possibilities.

To illustrate the interplay of problems, algorithms and designs we provide Figure 2 as a schematic overview. All further details will be covered in this chapter. As a reminder, Figure 1 in the previous chapter may prove useful again to keep track of the general tasks to perform.

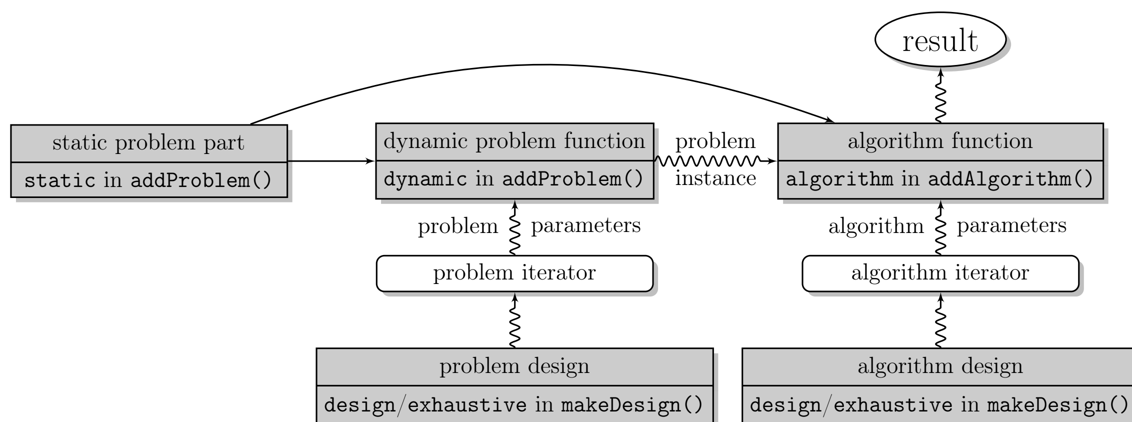

Figure 2: Relationship of **BatchExperiment** functions. Grey rectangulars require user input. White boxes represent internal functions. A straight arrow stands for direct passing of the object or function, a squiggly line denotes passing of the evaluated result.

The problem-defining function `addProblem` requires the registry `reg` and a unique problem identifier `id` as its first two arguments. The latter is used for referencing purposes. Additionally, at least one of the arguments `static` or `dynamic` must be given. The `static` part may be any R object. The `dynamic` argument on the other hand is restricted to be a function. It may have arbitrary further arguments which will later be filled in from a statistical design. If the dynamic function has `static` as named argument in its signature then the static problem part will be loaded on the node and passed to the function. If your problem does not have a dynamic part, you are of course free to omit the `dynamic` argument.

We illustrate a typical workflow now with our already mentioned toy classification example on the `iris` dataset. The `iris` data frame embodies the static part whereas resampled observation indices – used for evaluation – may be interpreted as the "dynamic" part of the problem instance. In the following snippet we use subsampling as the resampling strategy and define a function `subsample` which takes the additional argument `ratio` to define the ratio of training to test observations in the data set.

```
subsample <- function(static, ratio) {
  n <- nrow(static)
  train <- sample(n, floor(n * ratio))
  test <- setdiff(seq(n), train)
  list(test=test, train=train)
```

```
}
data(iris)
addProblem(reg, id="iris", static=iris,
           dynamic=subsample, seed=123)
```

The function addProblem files the problem (including the static part and the dynamic function) to the filesystem and the problem gets recorded in the registry. The dynamic function will be evaluated at a later stage on the workers. In this process, the static part will be loaded and passed to the dynamic function. Note that we set a special problem seed to synchronize the experiments in the sense that the same resampled training and test sets are used for the algorithm comparison, see Section 5 for detailed information on the seeding mechanism.

Algorithms are added to the registry in a similar manner. The first two arguments to addAlgorithm should again be the registry reg as well as a unique algorithm identifier id. The third argument fun has to be a function with the two optional formal arguments static and dynamic. Further arguments (e. g., hyperparameters or strategy parameters) to connect with a statistical design may be analogously defined as for the dynamic function in addProblem. When a job gets executed on the node the static part and the evaluated result of the function dynamic, if present in the function's signature, are passed to your algorithm. Appendix A gives a more in-depth description of this workflow.

For our example we define functions for a classification tree (package **rpart**, Therneau and Atkinson (2011)) and a random forest (package **randomForest**, Liaw and Wiener (2002)). Note that instead of loading these two packages via library in the wrappers we could have also used the packages option of the registry.

```
tree.wrapper <- function(static, dynamic, ...) {
  library(rpart)
  mod <- rpart(Species ~ ., data=static[dynamic$train, ], ...)
  pred <- predict(mod, newdata=static[dynamic$test, ], type="class")
  table(static$Species[dynamic$test], pred)
}
addAlgorithm(reg, id="tree", fun=tree.wrapper)

forest.wrapper <- function(static, dynamic, ...) {
  library(randomForest)
  mod <- randomForest(Species ~ ., data=static,
                      subset=dynamic$train, ...)
  pred <- predict(mod, newdata=static[dynamic$test, ])
  table(static$Species[dynamic$test], pred)
}
addAlgorithm(reg, id="forest", fun=forest.wrapper)
```

The return value of an algorithm can be any R object. The algorithm will be serialized to disk for later use. It is often reasonable to construct a named list of objects which will in many cases contain one or more performance measures. Here, we compute a confusion

matrix of the classification prediction on the test set which will later be used to calculate the misclassification rate. Note that using the "..." argument in the wrapper definitions allows us to omit design parameters in the signature. This is an advantage if we later want to extend the set of algorithm parameters in the experiment. The functions provided as `fun` in `addAlgorithm` are stored on the filesystem and the algorithms are also recorded in the registry.

4.2 Parametrization with statistical designs

The parametrization of both problems and algorithms is covered by `makeDesign`. This function takes a problem or algorithm ID as its first argument which defines the object of reference. The parameters for the respective problem or algorithm can be passed as either `design`, `exhaustive` or a combination of both. The argument `design` takes a user generated design as a data frame where parameter names correspond to column names. `exhaustive` can be used to create exhaustive grid designs, i. e., crossproducts of value vectors. `makeDesign` expects `exhaustive` to be a named list of atomic vectors or factors, the same holds for the columns of `design`. If both `design` and `exhaustive` are provided, a crossproduct of each row of `design` and each row of `exhaustive`'s grid is generated.

For our example we will try two different cross-validation ratios as problem parameters. We will also vary the complexity parameter `cp` and the parameter `minsplit` of the tree, as well as the number of trees in the random forest.

```
pars <- list(ratio=c(0.67, 0.9))
iris.design <- makeDesign("iris", exhaustive=pars)

pars <- list(minsplit=c(5, 10, 20), cp=c(0.01, 0.1))
tree.design <- makeDesign("tree", exhaustive=pars)

pars <- list(ntree=c(100, 500, 1000))
forest.design <- makeDesign("forest", exhaustive=pars)
```

By passing a pre-generated data frame to the argument `design` you are free to use any statistical design – and corresponding R package – you deem fit for your experiments. For computer experiments often space-filling designs like latin hypercube designs come to mind, but depending on your setup more classical approaches like the ones derived from A-, D-, or E-optimality and many others might also be feasible. For the generation of such designs we recommend using external packages like **lhs** (Carnell, 2012), **AlgDesign** (Wheeler, 2011) or **DiceDesign** (Franco *et al.*, 2011)[12].

It is worth mentioning that the exhaustive grid will never be expanded in memory to a matrix or data frame with all possible rows. Instead, an iterator object is used internally which traverses all defined rows. This minimizes the memory footprint on the master and allows experiments with rather large grid designs.

[12] But see `http://cran.r-project.org/web/views/ExperimentalDesign.html` for a better overview

Note that you can easily reference more complex R objects in your design by naming them and providing a "retrieval" function:

```
getMyObject <- function(name) {
  switch(name, foo=matrix(0,2,2), bar=identity)
}
algo <- function(static, dynamic, my.obj.name) {
  obj <- getMyObject(my.obj.name)
  ...
}
des <- makeDesign(id="algo",
                  exhaustive=list(my.obj.name=c("foo", "bar")))
```

4.3 Adding Experiments

In the previous sections we have shown how to register problems as well as algorithms and how to associate statistical designs with them. Now it is time to connect these parts to actually define an experiment. `addExperiments` takes the arguments `prob.designs` for the problem designs and `algo.designs` for the algorithm designs to add them to registry `reg`. You may pass any number of designs wrapped inside a list to both arguments. If you pass a single design to `addExperiments` via `prob.designs` or `algo.designs` it will automatically be wrapped in a list for you. Moreover you may pass a problem or algorithm ID as a string. In this case `addExperiments` treats the input as an unparametrized problem or algorithm. In addition you can set the integer parameter `repls` to define any number of replications for your experiments.

Suppose we want to subsample the iris dataset 100 times and apply both classifiers. To do this we combine both algorithm designs into a list and then pass this list to `addExperiments`:

```
addExperiments(reg, prob.designs=iris.design,
               algo.designs=list(tree.design, forest.design),
               repls=100)
```

Internally `addExperiments` checks problems and algorithms for their existence in the registry, generates the rows of the designs for the respective problems and algorithms and utilizes **BatchJobs** to finally create batch jobs.

To list all known problem and algorithm names, we provide the functions `getProblemIds` and `getAlgorithmIds`. Given an ID you can obtain the respective problem or algorithm by using `getProblem` or `getAlgorithm`. The function `summarizeExperiments` returns a cross table with the number of jobs which belong to specific problems (rows) and algorithms (columns):

```
summarizeExperiments(reg)
```

```
        Algorithm
Problem forest tree
   iris    600 1200
```

It is also possible to display further details like parameter settings by setting the argument `details=TRUE`. Once the experiments are added to the registry you can use `testJob` or `submitJobs` from **BatchJobs**, as described in Section 3.6, to run the jobs on your cluster.

4.4 Subsetting experiments

Before submitting all jobs to your batch system, we encourage you to test each algorithm individually. Or sometimes you want to submit only a subset of experiments because they extremely differ in runtime. Other reoccurring tasks include the investigation of algorithm specific errors, the checking or collecting of results for only a subset of experiments and so on. For all these use cases, `findExperiments` can be employed to conveniently select a particular subset of jobs. It expects a registry as its first argument and always returns the IDs of all experiments that match the given criteria. Your selection can depend on substring matches of problem or algorithm IDs using `prob.pattern` or `algo.pattern`, respectively. You can also pass R expressions which will be evaluated in your problem parameter setting (`prob.pars`) or algorithm parameter setting (`algo.pars`). The expression is expected to evaluate to a boolean value. Furthermore, you can restrict the experiments to specific replication numbers.

To illustrate `findExperiments` we will select two experiments, one with a decision tree and the other with a random forest and the parameter `ntree=1000`. The selected experiment IDs are then passed to `testJob`.

```
id1 <- findExperiments(reg, algo.pattern="tree")[1]
id2 <- findExperiments(reg, algo.pattern="forest",
                       algo.pars=(ntree == 1000))[1]
testJob(reg, id1)
testJob(reg, id2)
```

The next example shows how to kill jobs on the cluster, assuming you have found a small bug in your experiments. For this example let us suppose that you have accidentally hardcoded the subset sampling ratio in the dynamic problem function:

```
subsample <- function(static, ratio) {
  n <- nrow(static)
  train <- sample(n, floor(n * 0.67)) # <- hardcoded ratio
  test <- setdiff(seq(n), train)
  list(test=test, train=train)
}
```

This bug renders the parameter `ratio` effectless. We want to first stop the experiments which use other ratios than 0.67 on the cluster and remove the affected experiments from the registry.

```
ids <- findExperiments(reg, prob.pattern="iris",
                       prob.pars=(ratio != 0.67))
# kill still running jobs
killJobs(reg, intersect(ids, findRunning(reg)))
# remove jobs from registry
removeExperiments(reg, ids)
```

Now we want to fix the dynamic problem function and add the experiments to the registry again. To do this we simply add all experiments again and set the option `skip.defined` in `addExperiments` to `TRUE` – this way all so far known experiments which may already have completed calculation remain unchanged. Finally we submit those jobs where recalculation is necessary:

```
# fixed dynamic problem function
subsample <- function(static, ratio) {
    n <- nrow(static)
    train <- sample(n, floor(n * ratio)) # <- fixed
    test <- setdiff(seq(n), train)
    list(test=test, train=train)
}
# patch the problem function
addProblem(reg, id="iris", static=iris,
           dynamic=subsample, overwrite=TRUE)

# re-add all experiments using skip.defined
new <- addExperiments(reg, prob.designs=iris.design,
                      algo.designs=list(tree.design, forest.design),
                      repls=100, skip.defined=TRUE)
# submit all freshly added experiments
submitJobs(reg, new)
```

Note that the jobs with the previously hardcoded ratio 0.67 are not interrupted nor need recalculation. Keep in mind that patching your problems or algorithms with the `overwrite` option should only be used when absolutely necessary. Your code gets cluttered with those patches and the experiments are harder to read, to understand and to reproduce.

4.5 Collecting results

To collect the results of your experiments you are free to use the collection functions introduced in Section 3.8. In addition to these, we provide `reduceResultsExperiments` to conveniently collect parameters and atomic (performance) values into a data frame. It works very similar to the already explained `reduceResultsDataFrame` and requires a function argument `fun` with the signature `function(job, res)`. The result of `fun` must be a named list containing desired values. `reduceResultsExperiments` converts these

lists to data frames, stacks them and prepends columns for the problem and algorithm IDs, the settings and the replication number.

During our toy analysis of the iris dataset we calculate the misclassification rate for all submitted and successfully terminated jobs (suppose only 4 already terminated):

```
reduce <- function(job, res) {
    n <- sum(res)
    list(mcr=(n-sum(diag(res)))/n)
}
res <- reduceResultsExperiments(reg, ids=findDone(reg), fun=reduce)
print(res[c(1:2, 601:602), ])
```

```
    prob ratio   algo minsplit   cp repl  mcr ntree
1   iris  0.67   tree        5 0.01    1 0.08    NA
2   iris  0.67   tree        5 0.01    2 0.06    NA
601 iris  0.67 forest       NA   NA    1 0.08   100
602 iris  0.67 forest       NA   NA    2 0.06   100
```

We could have omitted the explicit calculation of the IDs with `findDone(reg)` as this is the default behavior of the reduction / collection functions.

After the results have been collected, the data can be summarized, explored, visualized, modeled or tested with any method of your choice. An encompassing overview of these topics is of course out of scope for this paper. Nevertheless we would like to point the reader to a few packages that we have found useful for basic operations on such experimental results, namely the **plyr** package (Wickham, 2011a) which is handy for aggregating and subsetting data frames, **reshape2** (Wickham, 2007) to convert the data frame to "long" or "wide" formats, the **sqldf** (Grothendieck, 2011) package to query the data frame using SQL as the query language, **ggplot2** (Wickham, 2009) and **lattice** (Sarkar, 2008) to quickly visualize results and **benchmark** (Eugster and Leisch, 2008; Eugster *et al.*, 2008) for explorative and inferential analysis of benchmark experiments.

As a final step let us quickly peek into our complete results by calculating the mean misclassification rate for all algorithm variants. We use `ddply` from **plyr** to partition the data frame into groups w. r. t. the problem, the algorithms and their parameters.

```
vars <- setdiff(names(res), c("repl", "mcr"))
ddply(res, vars, summarise, mean.mcr=mean(mcr))
```

```
  prob ratio   algo minsplit   cp ntree   mean.mcr
1 iris  0.67 forest       NA   NA   100 0.05020000
2 iris  0.67 forest       NA   NA   500 0.04920000
3 iris  0.67 forest       NA   NA  1000 0.04920000
4 iris  0.67   tree        5 0.01    NA 0.05660000
5 iris  0.67   tree        5 0.10    NA 0.06540000
6 iris  0.67   tree       10 0.01    NA 0.05600000
```

```
7  iris  0.67   tree       10 0.10    NA 0.06540000
8  iris  0.67   tree       20 0.01    NA 0.06540000
9  iris  0.67   tree       20 0.10    NA 0.06540000
10 iris  0.90 forest       NA   NA   100 0.04866667
11 iris  0.90 forest       NA   NA   500 0.04666667
12 iris  0.90 forest       NA   NA  1000 0.04600000
13 iris  0.90   tree        5 0.01    NA 0.05266667
14 iris  0.90   tree        5 0.10    NA 0.07133333
15 iris  0.90   tree       10 0.01    NA 0.04800000
16 iris  0.90   tree       10 0.10    NA 0.07133333
17 iris  0.90   tree       20 0.01    NA 0.07000000
18 iris  0.90   tree       20 0.10    NA 0.07133333
```

4.6 Further technical differences between the packages

As already mentioned, **BatchExperiments** completely builds upon the structure and mechanisms provided by **BatchJobs**. Most of its functions can be reused here, see Figure 1. There are some technical differences, but nearly all of them are internal and transparent for the user. We have opted for these differences mainly out of run time and space efficiency considerations. Firstly, for **BatchExperiments** the two additional subdirectories `problems` and `algorithms` are created inside your `file.dir` to store your problems and algorithms. Another difference lies in the need for an additional database table to hold experiment definitions so that they can be quickly queried in the database without expanding all designs in main memory. This is especially relevant for subsetting experiments with `findExperiments` (see Section 4.4). The additional experiment definition table stores the problem and algorithm ID, their serialized parameters and the number of replications. Note that we store replicated experiments only once here, but multiple time in the job status table as the computational state of jobs for a replicated experiment can of course be different.

5 Reproducibility

Reproducibility of experiments is an important aspect of modern day computational statistics, but a somewhat disregarded topic. Even for simpler experiments that do not require hundreds of lines of code or parallelization, the current situation is still not ideal. As Hothorn and Leisch (2011) point out, the number of papers contributing both data and source code for simulation studies or analyzes is still rather limited. Even for the ones that do, reproducibility is sometimes arguable. Code often contains important details unmentioned in the respective article, and – if it is well written – constitutes a precise documentation of methods and experiments. Another current roadblock is that published code is not treated with the same kind of diligence in the review process that ensures the quality of the published article itself. Our personal opinion is that code is an integral part of the scientific text as a whole and should be treated as such: It must be published, read and criticized. Only then we can critically compare our results and build upon them. A

recent remedy is the proposal of the R^2 platform by Leisch *et al.* (2011). It aims at the publication of R packages that specifically contain supplementary code as well as input and result data for scientific articles.

For computationally expensive experiments that require parallelization or batch computation the situation is more complicated for two different reasons: First, if we want to reproduce such scientific experiments, we need to have access to sufficient computational power and the skills to manage it. Even if assuming all of this as given, a general problem still remains: As different institutions and individuals have access to or prefer different operating systems, hardware and computational management systems, everybody writes their own, sometimes extensive code for parallelizing such experiments – quite often in an ad-hoc manner. In our experience it is not a trivial undertaking to cleanly separate the code parts of the actual algorithms and experiments from their scheduling on the computational system. Even if this is done by the original authors, we would still have to rewrite the latter part for our system to perform replicated or similar experiments.

A major advantage of both **BatchJobs** and **BatchExperiments** is their independence of the underlying batch system. By using abstract experiment descriptions where the mapping to the computational jobs, their submission to the batch system and the collection of results is out-sourced, your calculations become portable. If you publish your `file.dir` everybody can reproduce the results on their own batch system by simply exchanging the cluster functions back end. Smaller up to moderately sized experiments you could even reproduce with some patience on your personal multi-core machine or some ad-hoc connection of a few Unix machines by using our SSH cluster functions if you do not have access to a high performance cluster. Moreover, due to the separation of problems, algorithms, experiments and batch system specific parts you can write clear, understandable and well structured code. On top of that, other researchers might easily expand your registry with their own problems, algorithms or evaluation methods, without touching or recomputing your results. Or you can do this yourself when you later want to extend your own study. Comparison and exchange of problems and algorithms is thereby easily achieved.

Especially in simulation studies seeding is crucial and special care has to be taken if jobs are executed in parallel. **BatchJobs** and **BatchExperiments** store a seed for each potentially stochastic part to ensure reproducibility. The registry of **BatchJobs** allows the definition an initial seed. A jobs' seeds is defined by incrementing this initial seed when job gets added to the database. Therefore each job is assigned a unique seed. For **BatchExperiments** the situation is slightly more complicated as two potentially stochastic computational parts exist: The dynamic problem generation and the subsequent algorithm application. If the problem is simply static the mechanism works exactly in **BatchJobs**. Each job has one unique seed, automatically defined by incrementing the initial seed. For dynamically generated problems we assign a second seed to this part. We differentiate between "unsynchronized" and "synchronized" problem generation. Per default, by not setting a problem seed in `addProblem`, you will get unsynchronized problem generation where the problem seed is randomly defined in `addExperiments`. For synchronized problems on the other hand the user-provided problem seed is incremented with each experiment replication and your algorithms will retrieve the same problem instances for each distinct replication. We also refer the reader to the detailed description

of the workflow on the slave in Appendix A.

6 Conclusion and outlook

We have presented two packages for performing statistical calculations on high performance computing clusters. **BatchJobs** is intended as a general purpose tool which is applicable in as many scenarios as possible. It also allows users to extend it to build their own special purpose parallel systems on top of it. Most people will find **BatchExperiments** more convenient for analyzing their problems and algorithms. As an abstraction for statistical experiments, **BatchExperiments** allows to write clear, understandable, easily extensible and well structured R scripts for statistical experiments which are both reproducible and portable.

The first obvious extension is to support more batch systems and schedulers. **BatchJobs** is already applicable in many common environments, but, as pointed out in Section 3.12, the cluster functions interface is general enough for further extentsions. The most important ones are probably Slurm[13], Amazon EC2[14] and standalone, multi-core Windows machines. The first requires only minor modifications to the current cluster function implementation for TORQUE, while EC2 support would likely require a custom Amazon Machine Image (AMI) and some sort of globally shared filesystem between the nodes. We will of course support anybody who would like to integrate other systems and will either include well-written code for this purpose in the package or publish it on the project's web.

Although we have not experienced any problems with SQLite up to now, for an extremely large number of jobs with short computation times it might be advantageous to have a DBMS with row-level locking. We are therefore looking into supporting MySQL and PostgreSQL as well in the foreseeable future. However, the different data back ends must be convertible into each other in order to assure portability of registries. Regarding SQL compatible database systems the latter can be easily achieved.

Another option would also be to support architectures where the computational nodes only have local filesystems. This requires an additional abstraction layer for file access to import and export data for the nodes.

Furthermore, it might also be beneficial to have a system that allows for scheduling workflows of dependent jobs to a batch system. The idea is to specify a graph of dependent computational steps where the parents of a node define its required results, possibly with the option of recalculating only the required parts if some of the input data change.

In the near future we will apply **BatchExperiments** to perform broad benchmark studies in the areas of machine learning, optimization and survival analysis. Furthermore, we are generally interested in solving computationally expensive black-box optimization algorithms and analyzing computer experiments. Often, either evolutionary algorithms, model-based approaches or a combination of both are put to work for such problems. Especially the notion of model-based optimizers originates from the desire to efficiently

[13] `https://computing.llnl.gov/linux/slurm/`
[14] `http://aws.amazon.com/ec2`

produce approximate solutions for expensive problems. We think it might be worthwhile to create an interface on top of **BatchJobs** that allows such optimizers to create cluster jobs on-the-fly, collect their results and iterate asynchronously.

A Workflow on the slave

The calculation of a job is started by an `R CMD BATCH <scriptfile>` command where the argument corresponds to an automatically generated small R script. The standard output and standard error stream of the R process are redirected to a log file for debugging purposes. The various steps of the script are going to be explained in the following:

A.1 R scriptfile

Inside the script, job specific settings as location of the registry file and job IDs are hardcoded. They have automatically been filled in during the script creation. The following steps are performed / triggered by function calls:

1. Load the **BatchJobs** package.
2. Load the registry from the hardcoded location.
3. Evaluate the hardcoded vector of job IDs. The vector contains either a single job ID or, if you have used `submitJobs` in conjunction with `chunk`, all job IDs of one chunk.
4. Load the required packages of the registry.
5. Load the stored configuration file.
6. Change working directory to the one defined in the registry.
7. Deserialize the current job definition from the DB.
8. Send `"started"`-message for current job. The current timestamp is inserted into the database.
9. Potentially send `"start"`-mail, depending on configuration.
10. Save the current state of the RNG, set the job seed.
11. Call the `applyJobFunction` function inside a `try`-block. For **BatchJobs** this is simply applying the job function to the job parameters. For **BatchExperiments** see below.
12. If an error occurred: Send `"error"`-message for current job, potentially send `"error"`-mail.

13. If success: Store result, send `"done"`-message with timestamp for current job, potentially send `"done"`-mail.

14. Reset the seed to the previous seed.

15. Reset working directory.

If you have joined your jobs into chunks the steps 7 - 13 will basically be repeated for each individual job while using a caching mechanism to efficiently flush the messages.

A.2 BatchExperiments: applyJobFunction

BatchExperiments overloads the "apply-job function", otherwise the rest of the already explained workflow stays the same and is reused. The steps are as follows:

1. Load the static problem part and dynamic problem function from the filesystem.

2. Save the current state of the RNG, set the preliminary in `addExperiments` generated problem seed.

3. If not missing, call the dynamic problem function and pass it the problem parameters from the deserialized job definition. Create a problem instance (otherwise set to `NULL`).

4. Reset the seed to the previous seed.

5. Load the algorithm function from the filesystem.

6. Apply the algorithm function to the static problem part, the problem instance and the algorithm parameters from the deserialized job definition.

7. Return the result of your algorithm function.

References

Bui, P., Yu, L., Thrasher, A., Carmichael, R., Lanc, I., Donnelly, P., and Thain, D. (2011). Scripting distributed scientific workflows using weaver. *Concurrency and Computation: Practice and Experience*.

Burger, M., Juenemann, K., and Koenig, T. (2010). *RUnit: R Unit test framework*. R package version 0.4.26.

Carnell, R. (2012). *lhs: Latin Hypercube Samples*. R package version 0.7.

Cleveland, W., Guha, S., Hafen, R., Li, J., Rounds, J., Xi, B., and Xia, J. (2011). Divide and Recombine for the Analysis of Complex Big Data. Technical report, Department of Statistics, Purdue University.

Dean, J. and Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Commun. ACM*, **51**, 107–113.

Eugster, M. J. A. and Leisch, F. (2008). Bench plot and mixed effects models: First steps toward a comprehensive benchmark analysis toolbox. In P. Brito, editor, *Compstat 2008—Proceedings in Computational Statistics*, pages 299–306. Physica Verlag, Heidelberg, Germany.

Eugster, M. J. A., Hothorn, T., and Leisch, F. (2008). Exploratory and inferential analysis of benchmark experiments. Technical Report 30, Institut fuer Statistik, Ludwig-Maximilians-Universitaet Muenchen, Germany.

Franco, J., Dupuy, D., and Roustant., O. (2011). *DiceDesign: Designs of Computer Experiments*. R package version 1.1.

Grothendieck, G. (2011). *sqldf: Perform SQL Selects on R Data Frames*. R package version 0.4-6.1.

Hill, J., Hambley, M., Forster, T., Mewissen, M., Sloan, T., Scharinger, F., Trew, A., and Ghazal, P. (2008). SPRINT: A new parallel framework for R. *BMC Bioinformatics*, **9**(1), 558+.

Hoffmann, T. J. (2011). Passing in command line arguments and parallel cluster/multicore batching in R with batch. *Journal of Statistical Software, Code Snippets*, **39**(1), 1–11.

Horner, J. (2011). *brew: Templating Framework for Report Generation*. R package version 1.0-6.

Hothorn, T. and Leisch, F. (2011). Case studies in reproducibility. *Briefings in Bioinformatics*, **12**(3), 288–300.

James, D. A. (2011). *RSQLite: SQLite interface for R*. R package version 0.11.1.

Knaus, J. (2010). *snowfall: Easier cluster computing (based on snow).* R package version 1.84.

Leisch, F., Eugster, M., and Hothorn, T. (2011). Executable papers for the r community: The r2 platform for reproducible research. *Procedia Computer Science*, **4**(0), 618 – 626.

Liaw, A. and Wiener, M. (2002). Classification and regression by randomforest. *R News*, **2**(3), 18–22.

Mersmann, O. (2011). *sendmailR: send email using R*. R package version 1.1-1.

R Development Core Team (2012). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

Revolution Analytics (2011). *foreach: Foreach looping construct for R*. R package version 1.3.2.

Revolution Analytics and Pfizer (2010). *nws: R functions for NetWorkSpaces and Sleigh*. R package version 1.7.0.1.

Sarkar, D. (2008). *Lattice: Multivariate Data Visualization with R*. Springer, New York.

Schmidberger, M., Morgan, M., Eddelbuettel, D., Yu, H., Tierney, L., and Mansmann, U. (2009). State of the art in parallel computing with r. *Journal of Statistical Software*, **31**(1), 1–27.

Therneau, T. M. and Atkinson, B. (2011). *rpart: Recursive Partitioning*. R package version 3.1-50, R port by Brian Ripley.

Tierney, L., Rossini, A. J., Li, N., and Sevcikova, H. (2011). *snow: Simple Network of Workstations*. R package version 0.3-8.

Urbanek, S. (2011). *multicore: Parallel processing of R code on machines with multiple cores or CPUs*. R package version 0.1-7.

Vanschoren, J., Blockeel, H., Pfahringer, B., and Holmes, G. (2011). Experiment databases. A new way to share, organize and learn from experiments. *Machine Learning*.

Wheeler, B. (2011). *AlgDesign: Algorithmic Experimental Design*. R package version 1.1-7.

Wickham, H. (2007). Reshaping data with the reshape package. *Journal of Statistical Software*, **21**(12), 1–20.

Wickham, H. (2009). *ggplot2: elegant graphics for data analysis*. Springer New York.

Wickham, H. (2011a). The split-apply-combine strategy for data analysis. *Journal of Statistical Software*, **40**(1), 1–29.

Wickham, H. (2011b). *testthat: Testthat code. Tools to make testing fun :)*. R package version 0.5.

Yu, H. (2010). *Rmpi: Interface (Wrapper) to MPI (Message-Passing Interface)*. R package version 0.5-8.