Journal of Statistical Software
March 2015, Volume 64, Issue 11. http://www.jstatsoft.org/

BatchJobs and BatchExperiments: Abstraction Mechanisms for Using R in Batch Environments

Bernd Bischl
TU Dortmund

Michel Lang
TU Dortmund

Olaf Mersmann
TU Dortmund

Jörg Rahnenführer
TU Dortmund

Claus Weihs
TU Dortmund

Abstract

Empirical analysis of statistical algorithms often demands time-consuming experiments. We present two R packages which greatly simplify working in batch computing environments. The package **BatchJobs** implements the basic objects and procedures to control any batch cluster from within R. It is structured around cluster versions of the well-known higher order functions `Map`, `Reduce` and `Filter` from functional programming. Computations are performed asynchronously and all job states are persistently stored in a database, which can be queried at any point in time. The second package, **BatchExperiments**, is tailored for the still very general scenario of analyzing arbitrary algorithms on problem instances. It extends package **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 problems and algorithms and therefore to systematically study their influence on the results.

The packages' main features are: (a) *Convenient usage*: All relevant batch system operations are either handled internally or mapped to simple R functions. (b) *Portability*: Both packages use a clear and well-defined interface to the batch system which makes them applicable in most high-performance computing environments. (c) *Reproducibility*: Every computational part has an associated seed to ensure reproducibility even when the underlying batch system changes. (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.

Keywords: high-performance computing, R, batch computing, reproducibility.

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 the computational power of these systems is still substantial. The time spent to familiarize oneself with their ins and outs is a major hindrance to adoption. But even after overcoming this hurdle, practitioners are burdened by spartanic tooling and little or no automation to support their typical computing workflow.

In traditional high performance computing (HPC), many nodes of a cluster are often combined for hours or days at a time to solve a single problem. Statistical simulations on the other hand usually call for many repetitions of the same or very similar (often smaller) tasks. Most of the time, these tasks are "embarrassingly parallel", i.e., they do not have to communicate with each other.

To understand the intricacies of typical HPC clusters we have to examine how they are managed. Here, 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 not possible to directly start a process on a given node. Instead, we have to submit special job definition files to the batch system which in turn – based on our resource requirements – decides when and where our task is executed. All jobs are placed in queues and processed in some order to guarantee efficiency and fair share among users.

On such HPC clusters, the user has basically two different options to parallelize a collection of independent tasks. He could request a large number of nodes/CPUs for a lengthy amount of time in a single batch job and then synchronously work on them by using one of the already existing packages for explicit parallelism (see Section 2). But depending on the requested resources, this single batch job might get queued for an unacceptably long time or the approach is completely infeasible due to excessive resource requirements. Because of this disadvantage the user might resort to creating many small batch jobs, exactly one for each logical task. This is usually done by shell or R (R Core Team 2014) scripting for job definitions, job submissions, reconstruction of result objects (because individual job return values must be stored on disk) and so on. The outcome is often unportable boilerplate code. Matters get worse if errors occur and the user has to effectively "crawl" through a large number of log files to identify the potential problems.

From these considerations we can already draw some 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) Being able to conveniently track and query the computational state of each logical job would be beneficial. One obvious reason is that for very large statistical experiments we will in general not be able to submit all jobs at once because due to queue management overhead many systems impose technical limits on the number of job submissions. Also, errors can occur in all stages and some jobs might have to be resubmitted. (c) The smaller we can construct our tasks, the more efficiently the batch system can schedule them on the cluster because our resource requirements will be lower and we can exploit smaller gaps of low load in the schedule. 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 and many nodes. But this more efficient approach requires a considerable programming effort for the client programmer and makes projects less portable and maintainable.

Our packages follow this more efficient "one batch job per independent task" approach, but perform all necessary, tedious scripting internally and completely hide it from the end user. Instead we export a convenient and portable application programming interface (API) built on top of it. It does not replace the job scheduler, instead it is a front-end and abstraction layer which feeds the many tasks into the batch system as resources allow and tracks their processing.

Our R package **BatchJobs** (Bischl, Lang, and Bengtsson 2015a) implements the core infrastructure required to interact with, in principle, any cluster system. It enables a user to conveniently define batch jobs, submit them to the scheduler, query the jobs' status and collect the results. Its main interface is designed to mimic the powerful functional programming concepts of `Map`, `Reduce` and `Filter` – which are also available in R's **base** package under these names. Package **BatchJobs** transparently manages a persistent database to provide live information on the computational state and cluster status. We also offer a mechanism – coined "chunking" – to combine a number of quickly terminating tasks into a single batch job to reduce overhead. As the cluster management is clearly separated from the user, user written code must not be adjusted to work on other batch systems. Furthermore, seeding mechanisms ensure reproducibility even when facing a batch system migration. Results can be processed sequentially on a single machine or again in parallel on the batch system by mapping over them again.

The second package, **BatchExperiments** (Bischl, Lang, and Mersmann 2015b), is tailored for the general problem of studying combinations of algorithms on problem instances. This subsumes among other things: benchmarking experiments, sensitivity analysis, statistical simulation studies, parameter tuning or meta learning. Package **BatchExperiments** builds upon the **BatchJobs** framework by letting the user quickly define sets of problems $\mathcal{P}$ and algorithms $\mathcal{A}$. Problems and algorithms can then be connected using statistical designs to create batch jobs of the kind "apply algorithm $A \in \mathcal{A}$ to problem instance $P \in \mathcal{P}$ using parameter settings D".

Both packages are written in such a way that they can be used with almost any cluster. Even a loosely coupled set of nodes, which are only accessible via the secure shell (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 invested a considerable amount of time to provide flexible generic cluster functions which should work for a large number of installations.

The rest of the paper is organized as follows: Section 2 gives a brief overview of other comparable work. The following two sections introduce the packages **BatchJobs** and **BatchExperiments**. Section 5 addresses the aspects of reproducibility in computational statistics and Section 6 finally provides conclusions as well as an outlook. Both packages are available from the Comprehensive R Archive Network (CRAN) at `http://CRAN.R-project.org/package=BatchJobs` and `http://CRAN.R-project.org/package=BatchExperiments`. For further technical details we refer the reader to the material on the project web page at https://github.com/tudo-r/BatchJobs.

2. Review of other relevant work

A wealth of R packages for HPC are available. As of writing, the CRAN Task View "High-Performance and Parallel Computing with R" (Eddelbuettel 2015) lists 80 packages for HPC and parallel computing in R. The packages can roughly be split into three groups:

- Packages providing low level interfaces. The most widely used packages are **Rmpi** (Yu 2002, 2014) and **nws** (Revolution Analytics and Pfizer 2010). They allow R processes to communicate with each other using the respective standardized protocol interface. Their advantages shine in traditional HPC settings when there is one big task that should be parallelized and there is significant communication between its subtasks to synchronize or exchange temporary results. As they usually require inconvenient and extensive reworking of the source code and are somewhat cumbersome to use, there is a variety of more convenient high-level APIs.

- Packages providing high level interfaces, serving as bridge to other computing resources or frameworks. Arguably most popular are package **multicore** (Urbanek 2014) to utilize multiple CPU cores on a single host machine and package **snow** (Tierney, Rossini, Li, and Sevcikova 2013) which offers different back ends, e.g., local and network sockets or message passing interface (MPI), to utilize the computing power of up to a few dozen machines. Both packages provide, among others, parallel Map-like functionality – a concept well-known to experienced R programmers due to the popularity of R's apply-family of functions (e.g., `lapply`, `sapply` and `mapply`/`Map`). Most of the functionality of **multicore** and **snow** is combined in the R package **parallel** which ships with R as of version 2.14.0 as a base package and is maintained by the R Core Team. Another package worth mentioning is the **snowfall** package (Knaus 2013) which is very similar to **snow** but provides additional helper functions for common tasks in parallel computation. The package **foreach** (Kane, Emerson, and Weston 2013; Revolution Analytics 2014) mimics `for`-loop like semantic. Yet the functionality is conceptually similar to the apply-family of functions as the iterations have to be independent. There are many different packages on CRAN that can serve as a back end for **foreach**, they are all named **do**<*Technology*>. A good overview of all functions and their respective advantages and disadvantages is given in Schmidberger, Morgan, Eddelbuettel, Yu, Tierney, and Mansmann (2009).

- Packages tailored for specific tasks. In addition to the already mentioned packages, there is a plethora of packages that implement parallel versions of statistical algorithms. They usually employ one of the discussed packages internally to access the parallel computing resources. The **SPRINT** package (Hill, Hambley, Forster, Mewissen, Sloan, Scharinger, Trew, and Ghazal 2008) instead contains 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 and that all nodes must be in contact with the "master" R process during the whole execution. 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 because they would require tens to hundreds of nodes for days or maybe even weeks at a time. What we really aim for is a mechanism to run our tasks in an "asynchronous" fashion. That means, we want to define all our independent tasks a priori and then let the job scheduler deal with their execution to optimally exploit available computational resources.

Package **BatchJobs** is not intended to replace packages such as **parallel** or **snowfall**, but complements them by providing infrastructure and tools for large scale tasks which cannot or should not be run synchronously. The mentioned packages for explicit parallelism are better suited when one wants to implement a parallel version of an algorithm or parallelize over a (large) data set. They can even be fruitfully combined with package **BatchJobs**, when one wants to run an existing algorithm multiple times, which already allows explicit parallelization. In this case multiple **BatchJobs** jobs can be spawned, where each individual job allocates more than one core and internally uses explicit parallelization via one of the mentioned packages.

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, Guha, Hafen, Li, Rounds, Xi, and Xia 2011) which integrates R into a **Hadoop** (Apache Software Foundation 2014) 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.

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, Yu, Thrasher, Carmichael, Lanc, Donnelly, and Thain 2011).

3. The package 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 file system for all computational nodes. All jobs are declared at the registry and their computational status is held in a database. You do not have to query the database yourself (or be aware of its existence for that matter), as it is transparently managed. Therefore, the registry bundles the access to both the database and the information stored on the file system. The registry is automatically saved to its associated project directory when it is created or its state is altered. It can be reused later, e.g., when you login to the system again, by calling the function `loadRegistry(file.dir)`.

Jobs are not submitted when they are created, instead their definitions are stored in the database until they are explicitly sent 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. During job submission short R scripts and cluster job files are created, which are submitted to the batch system via operating system (OS) commands (e.g., by an internal call to `qsub`). Jobs report their status to the database and write results to disk as `.RData` files. Therefore, the status of each job is available to you on the master node at all times.

Let us begin by looking at a first instructive toy example. Assume we want to optimize a certain objective function using simulated annealing, as provided by the R function `optim`. Since we suspect that the function might be multimodal, we want to perform multiple optimization runs from 10 random start points. These 10 runs shall now be performed in parallel.

```
R> library("BatchJobs")
R> library("soobench")
R> reg <- makeRegistry(id = "optimexample",
+    file.dir = "optimexample-files", packages = "soobench")
R> starts <- replicate(10, runif(5, min = -5, max = 5), simplify = FALSE)
R> myoptim <- function(start)
+    optim(par = start, fn = rosenbrock_function(5), method = "SANN")
R> batchMap(reg, myoptim, starts)
R> submitJobs(reg)
R> waitForJobs(reg)
R> reduceResultsVector(reg, fun = function(job, res) res[["value"]])
```

In the above code snippet, we have told our registry to store all relevant files in the directory `optimexample-files`, which will be created in the current directory. We have also made sure that the package **soobench** (Mersmann and Bischl 2012), which provides our objective function (in this case the Rosenbrock function in 5 dimensions) will be loaded on every slave. We then create 10 random start points and map the `optim` function over this list. After we have submitted our jobs (and they have terminated), we collect all results in a numeric vector by iterating over all stored result objects and picking out the final objective value of each run.

Figure 1 provides a visual overview of all common tasks which we can perform using the packages. All functions mentioned in the diagram will be explained in more detail in this and the following section.

3.1. Functional programming idioms for parallelization

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). Package **BatchJobs** provides a parallelized version of `Map` to the end user, which is in line with many other R packages for explicit parallelism (see Section 2). The most important difference is that – as we work under the control of a scheduler – the corresponding jobs are not executed at once but only defined and stored. Their execution happens asynchronously after arbitrary subsets of jobs have been submitted for execution.

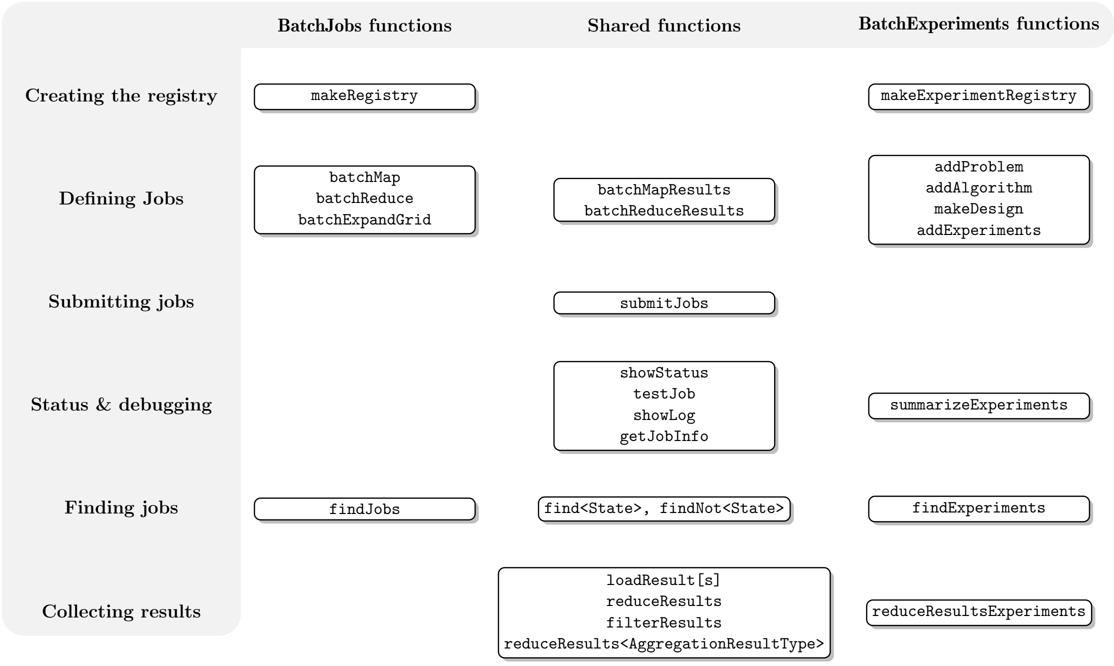

Figure 1: Overview of the most important functions grouped by package and task. Shared functions are functions from package **BatchJobs** that are also useful when working with package **BatchExperiments**.

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

`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. Note that one job is not one step of the reduction but instead a combination of several reduction steps. The results must then be reduced one final time on the master node to obtain a single result object.

`batchMapResults`, `batchReduceResults`**:** Same as `batchMap` and `batchReduce` but operate on the results of a previous registry to further transform them in parallel.

3.2. Submitting jobs

Jobs can be submitted to the batch system via the `submitJobs` function, which already appeared in the previous example. 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 wall time and required memory.

```
R> submitJobs(reg, resources = list(walltime = 3600, memory = 4 * 1024))
```

Here, we have passed the wall time in seconds and requested 4 gigabytes of memory per job. The function tries to submit as many jobs as possible. If either system limits or user limits are exhausted (e.g., due to queue limits), the function waits until submission is possible again and then continues to submit further jobs.

In some cases it may be advantageous to assign different computational resources to certain subsets of the jobs. This can be achieved by submitting only a certain subset of job IDs. As this is more likely to happen with package **BatchExperiments**, where your experimental setup might contain algorithms of very different run times or problems that vastly differ in size, we will discuss this more in-depth in Section 4.4.

3.3. Status queries

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.

```
R> 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 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 last line shows the minimum, mean and maximum run times for all selected jobs. This is useful if the user has to set wall times and is unsure how long each job will take or if there are vast differences in execution times. If errors occurred, the first error messages are displayed as well, see Section 3.6 for further remarks on debugging.

Simple helper functions that query the computational state of your jobs and which all return a vector of corresponding job IDs are: `findSubmitted`, `findOnSystem`, `findRunning`, `findDone`, `findErrors`, `findTerminated`, `findExpired`. Negated versions exist in the form `findNot<State>`.

If you want to access individual, detailed information regarding the computational state and execution of jobs, you can use `getJobInfo(reg, ids)` to obtain a `data.frame` which contains (among other things) job event and job run times, error messages, node names and batch job IDs.

3.4. Collecting results

After jobs have successfully terminated, we can load their results on the master. This can be done in a simple fashion by using either `loadResult(reg, id)` or `loadResults(reg, ids)`, which return a single result exactly in the form it was calculated during mapping or load a couple of these objects into a list.

The function `reduceResults` allows a more flexible aggregation and works very similar to R's usual `Reduce`. 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`. The argument `job` is rarely needed, but can be useful in package **BatchExperiments** when a job contains the description of an experiment, e.g., names of the current problem and algorithm and their respective parameters.

We now show a simple example of operating on results of numeric vectors, reducing them into a vector by summing them up. Additionally we will `cbind` all result vectors into a single matrix.

```
R> reg <- makeRegistry(id = "reduceexample")
R> batchMap(reg, function(x) rnorm(2), 1:3)
R> submitJobs(reg)
R> waitForJobs(reg)
R> reduceResults(reg, fun = function(aggr, job, res) aggr + res)
R> reduceResults(reg, fun = function(aggr, job, res) cbind(aggr, res))
R> reduceResultsMatrix(reg, rows = FALSE)
```

The last two lines in the above example are equivalent, the last one simply shows a more convenient way to perform the same operation. We provide the following wrappers to aggregate results into the respective R data types: `reduceResults[Vector,Matrix,DataFrame,List]`. The following binary combination functions are used in the four wrappers respectively: `c` (vector), `cbind` / `rbind` (matrix), `rbind` (data.frame) and `c` (list).

There is also a `filterResults` function which filters all results with a predicate, similar to R's `Filter` function, but instead of returning a subset of the results, it only returns the subset of jobs IDs for which the predicate returns `TRUE`.

3.5. 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 starting of R sessions on the slaves will likely introduce a considerable overhead.

In order to increase efficiency in such cases, we offer a mechanism that combines multiple jobs together into chunks which are executed sequentially on the slaves within one R instance. 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`.

In order to increase database throughput in the case of 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.

3.6. Debugging tools

In any large scale 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 package **BatchJobs** cannot handle all conceivable error conditions, it does include extensive functionality to aid in debugging. 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 2011a, 2014) or **RUnit** (Burger, Juenemann, and Koenig 2015) to define your unit tests, possibly by directly developing an R package. But other parts directly have to do with the execution of the experiment. Nearly everybody forgets to load a required package or makes other trivial or less trivial mistakes in first code versions. 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)` to catch errors that are easy to spot because they are raised in many or all jobs. This function runs the job without side effects 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.

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 the current state of the system. Suppose we run the following artificial example on our cluster:

```
R> flakeyFunction <- function(value) {
+    if (value %in% 2:3) stop("Ooops.")
+    value^2
+  }
R> reg <- makeRegistry(id = "error")
R> batchMap(reg, flakeyFunction, 1:4)
R> submitJobs(reg)
```

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

```
R> waitForJobs(reg)
R> 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.
```

If we want to get the IDs of all jobs that failed due to an error we can use `findErrors(reg)`. And if we want to peek into the R log file of a job to see more context for the error we can use `showLog(reg, id)`.

Finally we can fix our code by using `setJobFunction`. This function allows us to redefine the function that is mapped. This is meant 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:

```
R> failed <- findErrors(reg)
R> setJobFunction(reg, failed, fun = function(value) value^2)
R> submitJobs(reg, failed)
```

Please note that changing the mapped function is a dangerous operation. If the value of the "new" map function for completed jobs is different from the value of the "old" map function, the results will not be meaningful.

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)`.

3.7. Job database back end

We use **RSQLite** (Wickham, James, and Falcon 2014) as interface to SQLite (Hipp 2015). 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 file system. As the jobs are concurrently executed, we write their status into the database, making sure that the ACID (atomicity, consistency, isolation, durability) properties hold.

Although we have not experienced any problems with SQLite up to now, for an extremely large number of jobs with short computation times, lock congestion might get a serious issue to deal with. To deal with this problem, we provide for these situations the Boolean option `staged.queries` to cache all database queries sent from the nodes in files. These files are then parsed and merged with the SQLite database on the master. Further technical information regarding this topic is available on our project page.

3.8. Supported batch systems

Package **BatchJobs** is designed to work on any cluster using any type of batch system. To communicate with the cluster internally, all interactions with the batch system are delegated to a so-called "cluster functions" interface which must provide three low-level operations to submit an R script as a job, kill a job based on its ID on the batch system and list current jobs by ID on the batch system. All high-level functionality is built upon these simple operations. 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 and offer the same resource management.

SSH cluster (`makeClusterFunctionsSSH`): Jobs are distributed to different (Linux) nodes using the SSH as the underlying communication layer. All nodes must be accessible without manually entering passwords (e.g., by ssh-agent or passwordless public key). This mode is suited for ad-hoc clusters of workstations when you do not have access to a true batch system. Resource management is possible in a rudimentary fashion: The user can specify an upper limit on the total load for each worker and the maximal number of jobs that should be run in parallel for the current registry on the worker. This allows to avoid overallocating workers in general and also to leave computational power available for competing users. The user can also exploit the waiting mechanism of `submitJobs` by running this command in a terminal multiplexer (e.g., `screen` or `tmux`) to submit jobs at once when computational resources become available.

True batch cluster mode (`makeClusterFunctions[Torque,SGE,LSF,SLURM]`): Currently we support systems managed by TORQUE (Terascale Open-Source Resource and QUEue Manager; Adaptive Computing Inc. 2014), Load Sharing Facility (Platform Computing 2012), Oracle / Sun Grid Engine (Gentzsch 2001) or SLURM (Simple Linux Utility for Resource Management; Yoo, Jette, and Grondona 2003). Jobs are submitted, listed and killed with the respective command line utilities (e.g., `qsub`, `qselect` and `qdel` for TORQUE). Since each cluster is different and has different requirements for the job files, a flexible approach is used where the user refers to a template and package **brew** (Horner 2011) is used to turn this into a job file. We provide examples of such templates in the `examples` directory of the package which will run on many systems with only minor modifications.

While the above 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[1] and the source code of the existing implementations.

3.9. Package configuration and status mailer

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 package **BatchJobs** is loaded. To understand its content, let us look at a simple example configuration:

[1] `https://github.com/tudo-r/BatchJobs/wiki/Cluster-Functions`

```
R> cluster.functions <- makeClusterFunctionsTorque("torque.tmpl")
R> mail.start <- "first"; mail.done <- "last"; mail.error <- "all"
R> mail.from <- "<me@cluster.system>"
R> mail.to <- "<me@my.domain>"
R> mail.control <- list(smtpServer = "mx.uni.edu")
```

The first line specifies the batch system you are using. Here, we use a TORQUE-based system and have therefore specified the path to the template portable batch system (PBS) file as already explained in the previous section. Working on another architecture simply means exchanging this single line in your configuration. This enables you to swap the back end on the fly if you need to switch to a different cluster system. Examples are scaling 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.

Package **BatchJobs** also contains a configurable status mailer that internally uses the package **sendmailR** (Mersmann 2014). Mail sending is triggered at the start of a 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. 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 directory $\prec$ 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.

Other configuration options include setting default resources, raising R warnings to errors, enabling a debug mode and switching to the staged SQL queries already mentioned in Section 3.7. All these options are in-depth explained in the wiki of the project web page[2].

4. The package BatchExperiments

The package **BatchExperiments** expands package **BatchJobs** with an abstraction layer for the very general task of applying a set of algorithms $\mathcal{A}$ to a set of problems $\mathcal{P}$ and recording some arbitrary results. Both, problems and algorithms, may be parametrized using statistical designs $\mathcal{D}$. Here, the elements $D \in \mathcal{D}$ contain both problem specific parameters D_P and algorithm specific parameters D_A so that D potentially consists of two parts: $D = (D_P, D_A)$. We call a problem $P \in \mathcal{P}$ applied to specific parameter settings D_P a problem instance $I = P(D_P)$. An experiment E is defined to be an application of an algorithm $A \in \mathcal{A}$ to a problem instance I and it returns a final result R: $E(P, A, D) = A(P(D_P), D_A) = A(I, D_A) = R$. As experiments might be stochastic, each one can be replicated any number of times. Such a replicated experiment constitutes a batch job.

[2] https://github.com/tudo-r/BatchJobs/wiki/Configuration

In our opinion, a wide range of 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, for a lot of statistical domains comprehensive 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; in a clean, convenient and reproducible fashion.

Package **BatchExperiments** combines exactly these two aspects. It allows you to compare many candidate methods to each other and work with large problem domains instead of a few, possibly unrepresentative instances. Another option is to 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 package **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, Blockeel, Pfahringer, and Holmes 2012 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, we believe that an organized, machine-readable, publicly accessible collection 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 (Anderson 1935), vary a few of their hyper-parameters and evaluate the classification performance.

Just as in package **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, which is created with `makeExperimentRegistry` in the same way as before:

```
R> library("BatchExperiments")
R> reg <- makeExperimentRegistry(id = "iris_example")
```

Once the registry is created, you can start to add problems, algorithms and designs. Experiments can afterwards 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 are of a more "dynamic" nature, e.g., random numbers drawn from

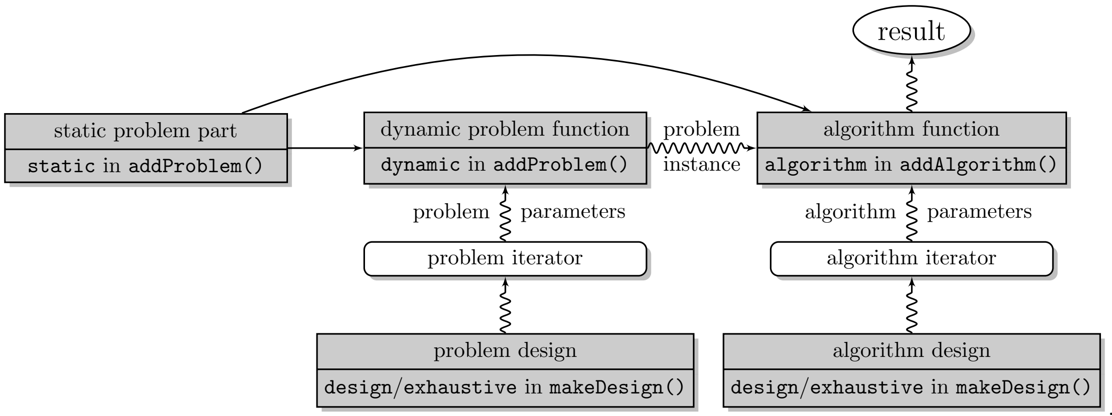

Figure 2: Relationship of **BatchExperiment** functions. Grey rectangles require user input. White boxes represent internal functions. A straight arrow stands for direct passing of the object or function, a squiggly line denotes passing of the evaluated result.

a probability distribution. And yet others depend on a static object but are also stochastic (e.g., subsampling) and/or depend on parameters so that the resulting problem instance can also be marked as being dynamic in that regard. With this in mind, we opted for a unified interface, which deals with all three possibilities.

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

The problem-defining function `addProblem` requires the registry `reg` and a unique problem identifier `id` as its first two arguments. The latter is solely 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, which may have arbitrary parameters, which can later be filled in from a connectible statistical design. If the dynamic function has `static` as a 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 now illustrate a typical workflow with our already mentioned toy classification example for the `iris` data set. 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 a simple subsampling strategy and therefore define a function `subsample`, which takes the additional argument `ratio` to define the ratio of training to test observations.

```
R> subsample <- function(static, ratio) {
+    n <- nrow(static)
+    train <- sample(n, floor(n * ratio))
+    test <- setdiff(seq(n), train)
+    list(test = test, train = train)
+  }
R> data("iris", package = "datasets")
```

```
R> 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 file system and the problem gets recorded in the registry. The dynamic function call 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 in each distinct replication. 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., hyper-parameters or strategy parameters), which can later be connected to a statistical design may be analogously defined as for the dynamic function in `addProblem`. When a job gets executed on the node the `static` object and the evaluated result of the `dynamic` function (the problem instance) are passed to your algorithm. Both arguments are optional, and if they do not show up in the algorithm's signature, they will neither be loaded nor calculated. The return value of an algorithm can be any R object and will automatically be stored the file system for later retrieval.

For our example we define functions for a classification tree (package **rpart**, Therneau and Atkinson 1997) 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.

```
R> 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)
+  }
R> addAlgorithm(reg, id = "tree", fun = tree.wrapper)
R> 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)
+  }
R> addAlgorithm(reg, id = "forest", fun = forest.wrapper)
```

Here, we compute a confusion matrix for the predictions 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 circumvent naming specific design parameters for now. This is an advantage if we later want to extend the set of algorithm parameters in the experiment. The algorithms get recorded in the registry and the corresponding functions are stored on the file system.

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. This way you are free to use any statistical design – and corresponding R package as listed on the CRAN Task View on experimental designs (Grömping 2014) – you deem fit for your experiments. The argument `exhaustive` can be used to create exhaustive grid designs, i.e., crossproducts of vectors. `makeDesign` expects `exhaustive` to be a named list of atomic vectors or factors. 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 classification tree, as well as the number of trees in the random forest.

```
R> pars <- list(ratio = c(0.67, 0.9))
R> iris.design <- makeDesign("iris", exhaustive = pars)
R> pars <- list(minsplit = c(5, 10, 20), cp = c(0.01, 0.1))
R> tree.design <- makeDesign("tree", exhaustive = pars)
R> pars <- list(ntree = c(100, 500, 1000))
R> forest.design <- makeDesign("forest", exhaustive = pars)
```

It is worth mentioning that the exhaustive grid will never be expanded in memory. 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.

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 all 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 define experiments in the registry `reg`. You may pass any number of designs, either as single object or wrapped inside a list. String IDs for problems/algorithms mean that these objects are applied unparametrized in the experiments. 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` data set 50 times and apply both classifiers. To do this, we combine both algorithm designs into a list and then pass this list to `addExperiments`:

```
R> addExperiments(reg, repls = 50, prob.designs = iris.design,
+    algo.designs = list(tree.design, forest.design))

  Adding 18 experiments / 900 jobs to DB.
```

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 package **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 data frame, which gives an overview over the defined experiments:

```
R> summarizeExperiments(reg)
```

```
  prob   algo .count
1 iris   tree    600
2 iris forest    300
```

Once the experiments are added to the registry, you can use `testJob` to test and `submitJobs` to submit your jobs, as described in Section 3.2.

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 the jobs vastly differ in runtime. Another reoccurring task is the collection of results for only a subset of experiments. 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 then 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`.

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

As `findExperiments` returns job IDs, you can combine `findExperiments` with any member of the status querying `find<State>`-family of functions (see Section 3.3) using set operations: The next snippet would kill all currently running jobs which use the classification tree algorithm:

```
R> ids1 <- findExperiments(reg, algo.pattern = "tree")
R> ids2 <- findRunning(reg)
R> killJobs(reg, intersect(id1, id2))
```

4.5. Collecting results

To collect the results of your experiments you are free to use the collection functions introduced in Section 3.4. 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 automatically prepends problem and algorithm IDs and parameters.

The next snippet reduces the confusion matrix returned by the algorithms in our toy example to a misclassification rate:

```
R> reduce <- function(job, res) {
+    n <- sum(res)
+    list(mcr = (n - sum(diag(res))) / n)
+ }
R> res <- reduceResultsExperiments(reg, fun = reduce)
R> print(res[c(1:2, 899:900), ])
```

```
 id prob ratio   algo   cp minsplit repl        mcr     ntree
 1     1 iris  0.67   tree 0.01        5    1 0.08000000    NA
 2     2 iris  0.67   tree 0.01        5    2 0.06000000    NA
 899 899 iris  0.90 forest   NA       NA   49 0.06666667  1000
 900 900 iris  0.90 forest   NA       NA   50 0.06666667  1000
```

After the results have been collected, the data can be summarized, explored, visualized, modeled or analyzed with any method of your choice. An encompassing overview of these topics is of course out of the scope of this paper. But 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 package **plyr** (Wickham 2011b) to partition the data frame into groups and aggregate the misclassification rates w. r. t. the problem, the algorithms and their parameters.

```
R> library("plyr")
R> vars <- setdiff(names(res), c("id", "repl", "mcr"))
R> print(head(ddply(res, vars, summarise, mean.mcr = mean(mcr))))
```

```
   prob ratio   algo   cp minsplit ntree mean.mcr
 1 iris  0.67 forest   NA       NA   100   0.0468
 2 iris  0.67 forest   NA       NA   500   0.0456
 3 iris  0.67 forest   NA       NA  1000   0.0456
 4 iris  0.67   tree 0.01        5    NA   0.0524
 5 iris  0.67   tree 0.01       10    NA   0.0520
 6 iris  0.67   tree 0.01       20    NA   0.0596
```

5. Reproducibility and seeding

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 analyses 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 stumbling block 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, Eugster, and Hothorn (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 packages **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 multicore 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. Packages **BatchJobs** and **BatchExperiments** provide a seeding mechanism for each job to ensure reproducibility. The registry of package **BatchJobs** allows the definition of an initial seed. A job's seed is defined by incrementing this initial seed when the job is added to the database. For package **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 as in package **BatchJobs**. Each job has one unique seed, automatically defined by incrementing the initial seed. For dynamically generated problems we assign a second seed to the stochastic problem part. We differentiate between "unsynchronized" and "synchronized" problem generation. Per default, by not setting a problem seed in `addProblem`, problems are generated in an unsynchronized fashion where the problem seed is randomly defined and stored. For synchronized problems, i.e., when you explicitly provide a problem seed, this problem seed is incremented only depending on the experiment replication so that all your algorithms will retrieve the same problem instances for each distinct replication.

6. Conclusion and outlook

We have presented two packages for performing statistical calculations on high performance computing clusters. Package **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 package **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. Package **BatchJobs** is already applicable in many common environments, and, as pointed out in Section 3.8, the cluster functions interface is general enough for further extensions. The most important ones are probably Amazon EC2 (Amazon Inc. 2015) and standalone, multicore Windows machines. EC2 support would probably require a custom Amazon Machine Image (AMI) and some sort of globally shared file system between the nodes. We will of course support anybody who would like to integrate other systems and happily accept contributions of this kind.

Another option would also be to support architectures where the computational nodes only have local file systems. This requires a file staging mechanism which defines a set of files to transfer before and after each job's execution.

Furthermore, it might also be beneficial to have a system that allows for scheduling workflows of dependent jobs. 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 package **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 combined for such problems. Especially the notion of model-based optimizers originates from the desire to efficiently produce approximate solutions for expensive problems. We think it might be worthwhile to create an interface on top of package **BatchJobs** that allows such optimizers to create cluster jobs on-the-fly, collect their results and iterate asynchronously.

Acknowledgments

Bernd Bischl and Michel Lang contributed equally to this work.

This work was partly supported by the Graduate School of Energy Efficient Production and Logistics (http://www.nrw-forschungsschule.de/) and by the German Research Foundation (DFG) within the Collaborative Research Centers SFB 823 "Statistical Modelling of Nonlinear Dynamic Processes", Project C2 (http://www.statistik.tu-dortmund.de/sfb823-project_c2.html), and SFB 876 "Providing Information by Resource-Constrained Analysis", Project A3 (http://sfb876.tu-dortmund.de/SPP/sfb876-a3.html).

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.

References

Adaptive Computing Inc (2014). *Terascale Open-Source Resource and QUEue Manager (TORQUE)*. URL http://www.adaptivecomputing.com/products/open-source/torque/.

Amazon Inc (2015). *Amazon Elastic Compute Cloud (EC2)*. Seattle. URL http://aws.amazon.com/en/ec2/.

Anderson E (1935). "The Irises of the Gaspé Peninsula." *Bulletin of the American Iris Society*, **59**, 2–5.

Apache Software Foundation (2014). *Apache Hadoop*. URL http://hadoop.apache.org/.

Bischl B, Lang M, Bengtsson H (2015a). ***BatchJobs**: Batch Computing with R*. R package version 1.6, URL http://CRAN.R-project.org/package=BatchJobs.

Bischl B, Lang M, Mersmann O (2015b). ***BatchExperiments**: Statistical Experiments on Batch Computing Clusters*. R package version 1.4.1, URL http://CRAN.R-project.org/package=BatchExperiments.

Bui P, Yu L, Thrasher A, Carmichael R, Lanc I, Donnelly P, Thain D (2011). "Scripting Distributed Scientific Workflows Using Weaver." *Concurrency and Computation: Practice and Experience*, **24**(15), 1685–1707.

Burger M, Juenemann K, Koenig T (2015). ***RUnit**: R Unit Test Framework*. R package version 0.4.28, URL http://CRAN.R-project.org/package=RUnit.

Cleveland W, Guha S, Hafen R, Li J, Rounds J, Xi B, Xia J (2011). "Divide and Recombine for the Analysis of Complex Big Data." *Technical Report 2*, Department of Statistics, Purdue University.

Dean J, Ghemawat S (2008). "MapReduce: Simplified Data Processing on Large Clusters." *Communications of the ACM*, **51**, 107–113.

Eddelbuettel D (2015). *CRAN Task View: High-Performance and Parallel Computing with R*. Version 2015-01-15, URL http://CRAN.R-project.org/view=HighPerformanceComputing.

Gentzsch W (2001). "Sun Grid Engine: Towards Creating a Compute Power Grid." In *First IEEE/ACM International Symposium on Cluster Computing and the Grid*, pp. 35–36.

Grömping U (2014). *CRAN Task View: Design of Experiments (DoE) & Analysis of Experimental Data*. Version 2014-12-07, URL http://CRAN.R-project.org/view=ExperimentalDesign.

Hill J, Hambley M, Forster T, Mewissen M, Sloan T, Scharinger F, Trew A, Ghazal P (2008). "**SPRINT**: A New Parallel Framework for R." *BMC Bioinformatics*, **9**(558).

Hipp DR (2015). *SQLite*. URL http://www.sqlite.org/.

Hoffmann TJ (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. URL http://www.jstatsoft.org/v39/c01/.

Horner J (2011). ***brew**: Templating Framework for Report Generation*. R package version 1.0-6, URL http://CRAN.R-project.org/package=brew.

Hothorn T, Leisch F (2011). "Case Studies in Reproducibility." *Briefings in Bioinformatics*, **12**(3), 288–300.

Kane M, Emerson JW, Weston S (2013). "Scalable Strategies for Computing with Massive Data." *Journal of Statistical Software*, **55**(14), 1–19. URL http://www.jstatsoft.org/v55/i14/.

Knaus J (2013). ***snowfall**: Easier Cluster Computing (Based on **snow**)*. R package version 1.84-6, URL http://CRAN.R-project.org/package=snowfall.

Leisch F, Eugster M, Hothorn T (2011). "Executable Papers for the R Community: The R^2 Platform for Reproducible Research." *Procedia Computer Science*, **4**, 618–626.

Liaw A, Wiener M (2002). "Classification and Regression by **randomForest**." *R News*, **2**(3), 18–22. URL http://CRAN.R-project.org/doc/Rnews/.

Mersmann O (2014). ***sendmailR**: Send Email Using R*. R package version 1.2-1, URL http://CRAN.R-project.org/package=sendmailR.

Mersmann O, Bischl B (2012). ***soobench**: Single Objective Optimization Benchmark Functions*. R package version 1.0-73, URL http://CRAN.R-project.org/package=soobench.

Platform Computing (2012). *Load Sharing Facility (LSF)*. URL http://www.platform.com/.

R Core Team (2014). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL http://www.R-project.org/.

Revolution Analytics (2014). ***foreach**: Foreach Looping Construct for R*. R package version 1.4.2, URL http://CRAN.R-project.org/package=foreach.

Revolution Analytics, Pfizer (2010). ***nws**: R Functions for NetWorkSpaces and Sleigh.* R package version 1.7.0.1, URL http://CRAN.R-project.org/package=nws.

Schmidberger M, Morgan M, Eddelbuettel D, Yu H, Tierney L, Mansmann U (2009). "State of the Art in Parallel Computing with R." *Journal of Statistical Software*, **31**(1), 1–27. URL http://www.jstatsoft.org/v31/i01/.

Therneau TM, Atkinson EJ (1997). "An Introduction to Recursive Partitioning Using the **rpart** Routine." *Technical Report 61*, Section of Biostatistics, Mayo Clinic, Rochester. URL http://www.mayo.edu/hsr/techrpt/61.pdf.

Tierney L, Rossini AJ, Li N, Sevcikova H (2013). ***snow**: Simple Network of Workstations.* R package version 0.3-13, URL http://CRAN.R-project.org/package=snow.

Urbanek S (2014). ***multicore**: Parallel Processing of R Code on Machines with Multiple Cores or CPUs.* R package version 0.2, URL http://CRAN.R-project.org/package=multicore.

Vanschoren J, Blockeel H, Pfahringer B, Holmes G (2012). "Experiment Databases. A New Way to Share, Organize and Learn from Experiments." *Machine Learning*, **87**(2), 127–158.

Wickham H (2011a). "**testthat**: Get Started with Testing." *The R Journal*, **3**(1), 5–10.

Wickham H (2011b). "The Split-Apply-Combine Strategy for Data Analysis." *Journal of Statistical Software*, **40**(1), 1–29. URL http://www.jstatsoft.org/v40/i01/.

Wickham H (2014). ***testthat**: Testthat Code. Tools to Make Testing Fun :).* R package version 0.9.1, URL http://CRAN.R-project.org/package=testthat.

Wickham H, James DA, Falcon S (2014). ***RSQLite**: SQLite Interface for R.* R package version 1.0.0, URL http://CRAN.R-project.org/package=RSQLite.

Yoo A, Jette M, Grondona M (2003). "SLURM: Simple Linux Utility for Resource Management." In D Feitelson, L Rudolph, U Schwiegelshohn (eds.), *Job Scheduling Strategies for Parallel Processing*, volume 2862 of *Lecture Notes in Computer Science*, pp. 44–60. Springer-Verlag.

Yu H (2002). "**Rmpi**: Parallel Statistical Computing in R." *R News*, **2**(2), 10–14.

Yu H (2014). ***Rmpi**: Interface (Wrapper) to MPI (Message-Passing Interface).* R package version 0.6-5, URL http://CRAN.R-project.org/package=Rmpi.

Affiliation:

Bernd Bischl, Michel Lang, Olaf Mersmann, Jörg Rahnenführer, Claus Weihs
Fakultät Statistik
TU Dortmund

44227 Dortmund, Germany
E-mail: bischl@statistik.tu-dortmund.de, lang@statistik.tu-dortmund.de,
olafm@statistik.tu-dortmund.de, rahnenfuehrer@statistik.tu-dortmund.de,
weihs@statistik.tu-dortmund.de
URL: https://github.com/tudo-r/BatchJobs,
https://github.com/tudo-r/BatchExperiments

Journal of Statistical Software http://www.jstatsoft.org/
published by the American Statistical Association http://www.amstat.org/
Volume 64, Issue 11 *Submitted:* 2012-05-30
March 2015 *Accepted:* 2014-08-04