BatchJobs and BatchExperiments

Bernd Bischl
(joint work with Michel Lang from Dortmund)
(note to myself: Never do meetup talks on my birthday!)

Motivation

- Long-running, independent jobs
- Very often these are benchmarking / comparison experiments
- These types of experiments can get complex very quickly
  - Some jobs can fail
  - Heterogeneous runtimes
  - Very often want to add / remove some experiments later, when partial results are there
  - We often learn about problems during / after calculations

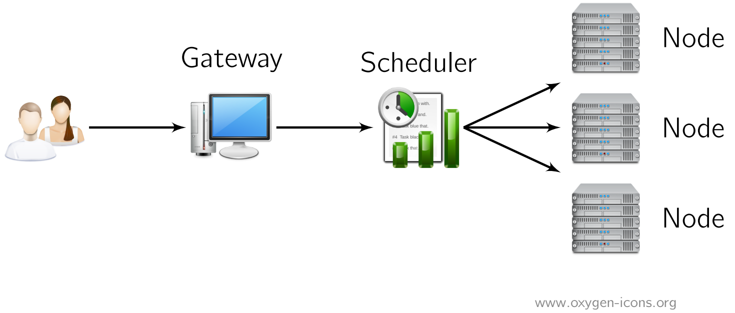
Naive batch computing

Computing on multicore machines (non-cluster)

- Prepare standalone script(s) that run your jobs, save results at end
- Parameters must be hard coded or retrieved through commandline
- Login on a machine per SSH
- Start job(s) with `R CMD BATCH myscript1.R`, combine this with nohup, screen or tmux
- Start remaining jobs when resources get available (argh...)
- Check manually for completion / errors (argh again...)
- Write script to collect results

No automation, no resource management or fair share,
neither extensible nor scalable.
— Don't do it this way —

High Performance Computing (HPC) clusters



- User log onto the gateway server (master or head node)
- Network of multiple computing nodes, managed by the scheduler
- Scheduler orchestrates the computation and organizes queues to fairly distribute compuation times among users
- Nodes either share a file system or support for file staging

Manual working on a batch system

- You have to specify
  - (a) Resource specifications (number CPUs, expected runtime and memory)
  - (b) Where to redirect the output streams
  - (c) Command to execute (e.g. `R CMD BATCH <myscript.R>`)
- Specs passed to CLI tools either directly as arguments or encoded in a shell script
- Check status of jobs via CLI tools (e.g. `qstatus`) or wait for mails
- Write script to collect results

Example Job Description File

Portable Batch System (PBS) syntax:

```
#PBS -N jobname
#PBS -j oe
#PBS -o job.output
#PBS -M lang@statistik.tu-dortmund.de
#PBS -l nodes=1,walltime=500,vmem=1024MB
#PBS -q short_eth

### commands to run
module add R
cd $HOME/my_experiments
R CMD BATCH myjob.R /dev/stdout
```

Submit from a shell:

```
> qsub myjob.pbs
```

Usual workflow on a batch system

- **Unroll** your R **loop(s)** so that your script computes a single iteration
- Write a **script that writes** R **scripts** for each iteration setting the iteration counter(s) at the beginning
- Write a **script that writes job description files** for each R script
- Write a **script that submits** your job description files
- **Crawl** through file system checking for existence of results or log files
- Write a **script that combines** your scattered result files
- Found a bug in your code? Write a **script that kills** all running jobs, fix the bug, submit everything again
- Some jobs have hit the wall time? Write a **script that finds** out which jobs you need to resubmit with weaker constraints
- Want to try your model on another data set or using other parameters? Eventually **start from scratch**, it might get ugly

Synchronous vs. asynchronous computation

Why not simply ask for 100 cores, then use the parallel package?

- You are now basically circumventing the scheduler
- How long are you willing to wait in queue? Infinitely?
- What if some jobs crash?
- What if you later want to add jobs?
- **One independent, embarrassingly parallel job should be submitted as such!**

Pros and Cons of Batch Systems

- + They are pretty fast!
- + Many statistical tasks are embarrassingly parallel
- - Job description files needed
- - We cannot control when jobs are started.
- - Jobs cannot really communicate, except by writing stuff on disk (or we have to allocate multiple cores and use something like MPI)
- - Requesting many nodes at once increases time spend in queue
- - Auxiliary scripts to create files and submit jobs necessary
- - Functions to collect results can get complicated and lengthy
- - If some jobs fail (e. g., singularities), debugging is awful

Packages

BatchJobs

- Basic infrastructure to communicate with a high performance cluster
- Tailored around Map-Reduce paradigm
- Can be incorporated into other packages
- Supported via `parallelMap` and `BiocParallel`

BatchExperiments

- Builds on top of BatchJobs
- Abstraction for “applying algorithms on problems”
- Assists the user in conducting comprehensive computer experiments

BatchJobs – Features

- Basic infrastructure to communicate with batch systems
- Complete control over the batch system from within R: submit, supervise, kill
- Implementation of higher order functions
  Map, **Reduce** and **Filter**
  to define jobs and collect results
- Persistent state of computation for experiments
- R code independent from the underlying batch system
- Reproducibility in distributed environments, even if the architecture changes
- Convenient result collection capabilities
- Debugging tools
- Configurable integrated status mailer

Supported Systems

Real batch systems:

- Torque/PBS based systems
- Sun Grid Engine / Oracle Grid Engine
- Load Sharing Facility (LSF)
- SLURM
- Still missing: Condor

Other modes:

- Interactive: Jobs executed in current interactive `R` session
- Multicore: local multicore execution with spawned processes
- SSH: distributed computing on loosely connected machines which are accessible via SSH (makeshift cluster)

Links and references

- `https://github.com/tudo-r/BatchJobs`
  - Installation infos
  - Intro slides by Henrik Bengtsson
  - R documentation
  - Wiki + FAQ
  - Mailing list + Issue tracker
  - Recent development version in git
- Our tech report:
  Bischl, Lang, Mersmann, Rahnenführer, Weihs:
  "Computing on high performance clusters with R: Packages BatchJobs and BatchExperiments"
  Available on project page
- Shortened version as JSS paper: Cite us!

First Time Configuration

- Specification of scheduling system and template file
- Templates can be downloaded at our website, easy to adept
- Optional mail notifications
- Defaults to interactive mode: no distributed computing, no mails

`BatchJobs.conf`

```
cluster.functions = makeClusterFunctionsTorque("~/lido.tmpl")
mail.to = "<lang@statistik.tu-dortmund.de>"
mail.start = "none"
mail.done = "first+last"
mail.error = "all"
```

Simple brew template for PBS

- One guy for one site has to this once
- Needed for flexibility and specifities of systems
- Talk to your admin for help
- Templates can be downloaded at our website
- Stuck? Put up a issue or email our list

```
#PBS -N <%= job.name %>
## merge standard error and output
#PBS -j oe
## direct streams to our logfile
#PBS -o <%= log.file %>
#PBS -l walltime=<%= resources$walltime %>,
#    nodes=<%= resources$nodes %>,vmem=<%= resources$memory %>M

## Run R:
## we merge R output with stdout from PBS,
## which gets then logged via -o option
R CMD BATCH --no-save --no-restore "<%= rscript %>" /dev/stdout
```

	BatchJobs' functions	Common functions	BatchExperiments' functions
(1) Create Registry	`makeRegistry`		`makeExperimentRegistry`
(2) Define Jobs	`batchMap` `batchReduce` `batchExpandGrid`	`batchMapResults` `batchReduceResults`	`addProblem` `addAlgorithm` `makeDesign` `addExperiments`
(3) Subset Jobs	`findJobs`	`findDone` `findErrors` ...	`findExperiments`
(4) Submit Jobs		`submitJobs`	
(5) Status & Debugging		`showStatus` `testJob` `showLog`	`summarizeExperiments`
(6) Collect Results		`loadResult[s]` `reduceResults` `filterResults` `reduceResults[AggrType]`	`reduceResultsExperiments`

	BatchJobs' functions	Common functions	BatchExperiments' functions
(1) Create Registry	`makeRegistry`		`makeExperimentRegistry`

(2) Define Jobs	`batchMap`
	`batchReduce`
	`batchExpandGrid`
		`batchMapResults`
		`batchReduceResults`
			`addProblem`
			`addAlgorithm`
			`makeDesign`
			`addExperiments`

(3) Subset Jobs	`findJobs`	`findDone`
		`findErrors`
		`...`
			`findExperiments`

(4) Submit Jobs		`submitJobs`	

(5) Status & Debugging		`showStatus`
		`testJob`
		`showLog`
			`summarizeExperiments`

(6) Collect Results		`loadResult[s]`
		`reduceResults`
		`filterResults`
		`reduceResults[AggrType]`
			`reduceResultsExperiments`

(1) Create a registry

Registry

- Object used to access and exchange informations: file paths, job parameters, computational events, ...
- Manages SQLite and file system backend
- All information is stored in a single, portable directory
- Initialization of a new registry:

```
library(BatchJobs)
reg = makeRegistry(
  id = "tryout",                  # name of your study
  file.dir = "~/project/bj-files", # accessible on all nodes
  work.dir = "~/project",          # accessible on all nodes
  seed = 1                         # initial seed for first job
)
```

- `loadRegistry(dir)` to resume working with an existing registry

	BatchJobs' functions	Common functions	BatchExperiments' functions
(1) Create Registry	`makeRegistry`		`makeExperimentRegistry`

(2) Define Jobs	`batchMap`
	`batchReduce`
	`batchExpandGrid`
		`batchMapResults`
		`batchReduceResults`
			`addProblem`
			`addAlgorithm`
			`makeDesign`
			`addExperiments`

(3) Subset Jobs	`findJobs`	`findDone`
		`findErrors`
		...
			`findExperiments`

(4) Submit Jobs		`submitJobs`	

(5) Status & Debugging		`showStatus`
		`testJob`
		`showLog`
			`summarizeExperiments`

(6) Collect Results		`loadResult[s]`
		`reduceResults`
		`filterResults`
		`reduceResults[AggrType]`
			`reduceResultsExperiments`

(2) Define Jobs

batchMap

- Like `lapply` or `mapply`
- $(x_1, x_2) \times (y_1, y_2) \to (f(x_1, y_1), f(x_2, y_2))$
- 10 Jobs to calculate $1 + 9$, $2 + 8$, $\ldots$, $9 + 1$

```r
map = function(i, j) i+j
ids = batchMap(reg, f, i=1:9, j=9:1)
```

- Stores function on file system
- Creates jobs as rows of SQLite database
- Parameters also serialized into the database for fast access
- All jobs get unique positive integers as IDs

	BatchJobs' functions	Common functions	BatchExperiments' functions
(1) Create Registry	makeRegistry		makeExperimentRegistry
(2) Define Jobs	batchMap batchReduce batchExpandGrid	batchMapResults batchReduceResults	addProblem addAlgorithm makeDesign addExperiments
(3) Subset Jobs	findJobs	findDone findErrors ...	findExperiments
(4) Submit Jobs		submitJobs	
(5) Status & Debugging		showStatus testJob showLog	summarizeExperiments
(6) Collect Results		loadResult[s] reduceResults filterResults reduceResults[AggrType]	reduceResultsExperiments

(3) Subset Jobs

- Query job IDs by computational status: `find*` functions `findSubmitted`, `findRunning`, `findDone`, ...
- Query job IDs by parameters: `findJobs(reg, pars)`

```
findJobs(reg, pars = (j==1))
findNotSubmitted(reg)
findDone(reg)
```

- Set operations on ID vectors: `intersect`, `setdiff`, `union`
- Vector of IDs can be passed to basically all functions interacting with the batch system

	BatchJobs' functions	Common functions	BatchExperiments' functions
(1) Create Registry	`makeRegistry`		`makeExperimentRegistry`
(2) Define Jobs	`batchMap` `batchReduce` `batchExpandGrid`	`batchMapResults` `batchReduceResults`	`addProblem` `addAlgorithm` `makeDesign` `addExperiments`
(3) Subset Jobs	`findJobs`	`findDone` `findErrors` `...`	`findExperiments`
(4) Submit Jobs		`submitJobs`	
(5) Status & Debugging		`showStatus` `testJob` `showLog`	`summarizeExperiments`
(6) Collect Results		`loadResult[s]` `reduceResults` `filterResults` `reduceResults[AggrType]`	`reduceResultsExperiments`

(4) Submit Jobs

submitJobs

- Creates R script files and job description files on the fly
- Resources can be provided as named list

```
# 1 hour maximal execution time, about 2 GB of RAM
res = list(walltime=60*60, memory=2000)

# ... and submit
submitJobs(reg, resources=res)
```

- Jobs can be grouped into chunks by providing a list of ID vectors. Each chunk gets executed sequentially as one single batch job

```
chunk(ids, chunk.size = 5)
```

	BatchJobs' functions	Common functions	BatchExperiments' functions
(1) Create Registry	`makeRegistry`		`makeExperimentRegistry`
(2) Define Jobs	`batchMap` `batchReduce` `batchExpandGrid`	`batchMapResults` `batchReduceResults`	`addProblem` `addAlgorithm` `makeDesign` `addExperiments`
(3) Subset Jobs	`findJobs`	`findDone` `findErrors` ...	`findExperiments`
(4) Submit Jobs		`submitJobs`	
(5) Status & Debugging		`showStatus` `testJob` `showLog`	`summarizeExperiments`
(6) Collect Results		`loadResult[s]` `reduceResults` `filterResults` `reduceResults[AggrType]`	`reduceResultsExperiments`

(5) Supervise and debug

- Quick overview of what is going on: `showStatus(reg)`

```
Status for jobs:  10
Submitted:        10 (100.0%)
Started:          10 (100.0%)
Errors:            0 (  0.0%)
Running:           2 ( 20.0%)
Expired:           0 (  0.0%)
Done:              8 ( 80.0%)
Time: min=1.50s avg=5.20s max=8.80s
```

- Display log files with a customizable pager (less, vi, ...): `showLog(reg, findErrors(reg)[1])`
- You can also `grepLogs(reg, pattern)`
- Found a bug? `killJobs(reg, findRunning(reg))`
- Run a job in the current R session: `testJob(reg, id)`

	BatchJobs' functions	Common functions	BatchExperiments' functions
(1) Create Registry	`makeRegistry`		`makeExperimentRegistry`

(2) Define Jobs	`batchMap`
	`batchReduce`
	`batchExpandGrid`
		`batchMapResults`
		`batchReduceResults`
			`addProblem`
			`addAlgorithm`
			`makeDesign`
			`addExperiments`

(3) Subset Jobs	`findJobs`	`findDone`
		`findErrors`
		...
			`findExperiments`

(4) Submit Jobs		`submitJobs`	

(5) Status & Debugging		`showStatus`
		`testJob`
		`showLog`
			`summarizeExperiments`

(6) Collect Results		`loadResult[s]`
		`reduceResults`
		`filterResults`
		`reduceResults[AggrType]`
			`reduceResultsExperiments`

(6) Collect results

Simple loading

```r
loadResult(reg, id = 1)
loadResults(reg)
loadResults(reg, ids)
```

Reduce

```r
# combine in numeric vector
reduceResults(reg, ids = findDone(reg), init = numeric(0),
  fun = function(aggr, job, res) c(aggr, res))
```

- Convenience wrappers around `reduceResults`: `reduceResults[Vector|Matrix|DataFrame|List]`
- `batchReduceResults` to reduce results on the nodes

Seeding

- Every job has a unique seed
- Registry stores initial seed x_0, all job seeds are defined as $x_0 + \mathrm{ID}$
- In some rare cases you need to manually change seeds, therefore all seeds are stored in DB
- Jobs become reproducible, even on another site
- But note all other ugly problems due to changes in hardware, software, compilers, versions, ...

Configuration again

.BatchJobs.conf

```
cluster.functions = makeClusterFunctionsTorque("~/lido.tmpl")
mail.to = "<lang@statistik.tu-dortmund.de>"
mail.start = "none"
mail.done = "first+last"
mail.error = "all"
default.resources = list(walltime = 3600, memory = 1024)
debug = FALSE
raise.warnings = FALSE
max.concurrent.jobs = 400
```

BatchExperiments

- Builds on top of BatchJobs
- Intended as abstraction for typical statistical tasks:

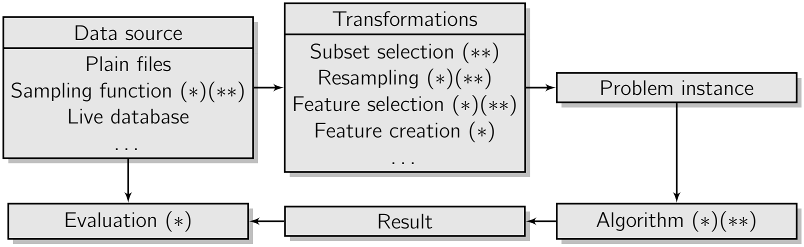
  Applying algorithms on problems
- More aimed at the end user
- Convenient for simulation studies, comparison and benchmark experiments, sensitivity analysis, . . .
- Workflow differs only in job definition

Scenarios

- Compare machine learning algorithms on many data sets
- Compare one/many estimation procedure(s) on simulated data
- Compare optimizers on objective functions
- ...

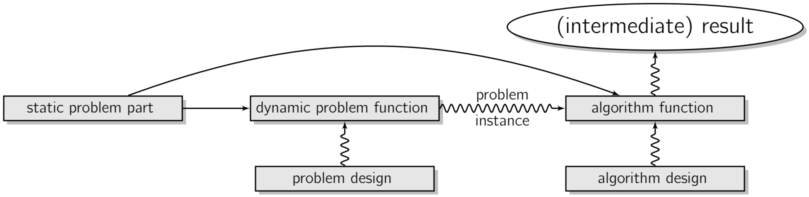
	BatchJobs' functions	Common functions	BatchExperiments' functions
(1) Create Registry	`makeRegistry`		`makeExperimentRegistry`
(2) Define Jobs	`batchMap` `batchReduce` `batchExpandGrid`	`batchMapResults` `batchReduceResults`	`addProblem` `addAlgorithm` `makeDesign` `addExperiments`
(3) Subset Jobs	`findJobs`	`findDone` `findErrors` `...`	`findExperiments`
(4) Submit Jobs		`submitJobs`	
(5) Status & Debugging		`showStatus` `testJob` `showLog`	`summarizeExperiments`
(6) Collect Results		`loadResult[s]` `reduceResults` `filterResults` `reduceResults[AggrType]`	`reduceResultsExperiments`

Abstraction of Computer Experiments



- (∗): Parameter requirements
- (∗∗): Stochastic, seeding needed
- Evaluation step often needs raw data
- Efficient and flexible abstraction?

Abstraction of Computer Experiments



- Problem definition split into static and dynamic part
  - Static: immutable R objects: matrix, data frames, ...
  - Dynamic: Arbitrary R function: transformations of static part, extraction of data from external sources, ...
- Parametrization through specifying experimental designs for both problems and algorithms
- Each step automatically seeded, random seeds stored in a database

Job definition steps in BatchExperiments

1. Add problems to registry: `addProblem`
   - Efficient storage: Separation of static and dynamic problem parts
   - Can be connected with an experimental design
2. Add algorithms to registry: `addAlgorithm`
   - Problem instance gets passed to algorithm
   - Can be connected with an experimental design
   - Return value will be saved on the file system
3. Connect experimental designs: `makeDesign`
4. Add experiments to registry: `addExperiments`
   - Experiment: problem instance + algorithm + algorithm parameters
   - Job: Experiment + replication number

What you get

- Reproducibility: Every computation is seeded, special seeding mechanism for synchronized problem generation
- Portability: Data, algorithms, results and job information reside in a single directory
- Extensibility: Add more problems or algorithms, try different parameters or increase the replication numbers at any computational state
- Exchangeability: Share your file directory to allow others to extend your study with their data sets and algorithms