Machine Learning with R at LRZ

Introduction to batchtools

Janek Thomas, Johannes Albert-von der Gönna

2018-11-27

MOTIVATION

EFFICIENCY: PARALLELIZATION

- Parallelization minimizes the effective computation time by distributing the CPU time to many cores
- Speedups linear to the number of independently run processes (in theory!)
- Debugging parallel code is especially hard
- Coding discipline even more important to minimize errors and frustration

WHAT CAN BE EASILY PARALLELIZED?

- Independent replications
- Resampling, cross-validation
- Model averaging
- Parameter variations in simulations, sensitivity analysis ...
- "Single program, multiple data"
- In other words: everything expressible as loop of independent iterations (if you can write it with `(l|m|)apply`, you are fine)

Popular counterexample: MCMC (single chain). Can be parallelized, but is **alot** harder

Many statistical problems are "embarrassingly parallel"

IDEAL PERFORMANCE IMPROVEMENT

- p processors should be p times faster than one processor
- Note: This is rarely possible in practice
- If you want a bit more theory, look up Amdahl's law and Gustafson's law

Some time scales

Single processor	30 Processors
1 minute	2 seconds
1 hour	2 minutes
1 day	1 hour
1 month	1 day
1 year	2 weeks

IDEAL PROGRAMMING REQUIREMENTS

- Minimal effort for simple problems
- Be able to use existing high level (i.e. R) code
- Ability to test code in sequential setting
- Debugging parallel problems possible
- Seeding / Reproducibility (with different CPU settings)
- Scale up to larger systems with minimal effort

As often: Cannot have your cake and eat it, too...

MASTER / SLAVE MODEL

- Have embarrassingly parallel problem: job1 job2 job3 (reduce) done
- Divide jobs among slave processes and collect results

machine	operation
master	init
slave1	job1
slave2	job2
slave3	job3
master	collect and reduce

- Ideal: p times faster with p slaves

A MORE REALISTIC PICTURE

- Jobs vary in complexity
- Machines vary in speed/load
- Communication takes time
- Dividing up jobs and collecting results takes time

SITUATION IN R

- R is single-threaded, so parallelization is not really built-in or free
- There exists a jungle of packages for parallel computation in R, some of which have existed a long time: `multicore`, `Rmpi`, `nws`, `snow`, `sprint`, `parallel`, `foreach`, `snowfall`, `batchtools`, `parallelMap`, `BiocParallel`
- As of 2.14.0, *R* ships with a package `parallel`
- *R* can also be compiled against multi-threaded linear algebra libraries (BLAS, LAPACK) which can speed up calculations
- The package `parallelMap` is developed to combine different communication backend-ends and provide convenient usage

PARALLELMAP

- Convenience wrapper around `parallel` and `batchtools`
- Modes: local, interactive, socket, mpi, batchtools.
- Even if you use `parallel` you need to touch code if you change backend
- None of the alternatives support BatchJobs
- You basically need to learn ONE function: `parallelMap` for ALL modes
- `parallelLapply` and `parallelSapply` also exist
- Perfect for interactive usage and in packages
- Supports tagging with levels and customization options
- Much convenience stuff

github.com/berndbischl/parallelMap

PARALLELMAP

```
library(parallelMap)
parallelStartSocket(cpus = 2L)
f = function(x) x^2
y = parallelMap(f, 1:10)
parallelStop()
```

PARALLELMAP

- `parallelLibrary` and `parallelSource` to load packages and sources
- `parallelExport` to export R objects
- Warnings / messages: `parallelMap` has a logging mode
- Random number generators are properly initialized

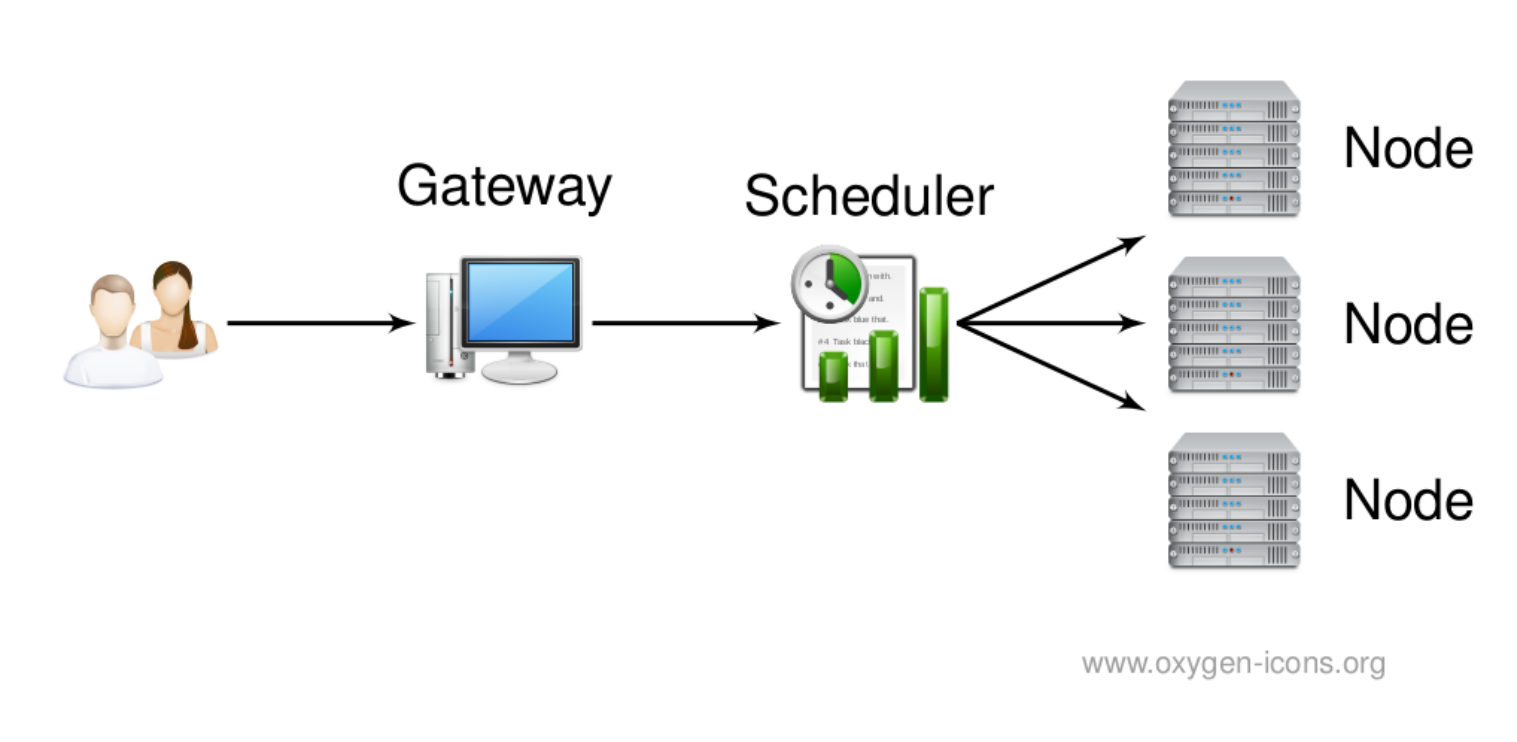
NAIVE BATCH COMPUTING

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.

HIGH PERFORMANCE COMPUTING (HPC) CLUSTERS



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

MANUAL WORKING ON A BATCH SYSTEM

You have to specify

- Resource specifications (number CPUs, number of tasks, expected runtime and memory)
- Which cluster/partition use
- Command to execute (e.g. `R CMD BATCH <myscript.R>`)

You have to manually

- Pass specks to CLI tools, either directly as arguments or encoded in a shell script
- Check status of jobs via CLI tools (e.g. `squeue`)
- Write script to collect results

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

USUAL WORKFLOW ON A BATCH SYSTEM

- 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

CONCLUSIONS AND FURTHER REMARKS

- Clusters are pretty fast!
- Many statistical tasks are embarrassingly parallel

But:

- 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

SLURM DEMO

BATCHTOOLS

BATCHTOOLS

- 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`
- Additional abstraction for "applying algorithms on problems"
- Assists the user in conducting comprehensive computer experiments
- Successor package (and combination) of **BatchJobs** and **BatchExperiments**.

BATCHTOOLS FEATURES

- Basic infrastructure to communicate with batch systems from within **R**
- Complete control over the batch system from within **R**: submit, supervise, kill
- 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

SUPPORTED SYSTEMS

- Torque/PBS based systems
- Sun Grid Engine / Oracle Grid Engine
- Load Sharing Facility (LSF)
- SLURM
- DockerSwarm

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/mllg/batchtools

- Installation infos
- R documentation
- Vignettes
- Issue tracker
- Recent development version in git

Paper:

```
batchtools: Tools for R to work on batch systems.
The Journal of Open Source Software 2.10 (2017).
Lang, Michel, Bernd Bischl, and Dirk Surmann.
```

CREATE A REGISTRY

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

```
library(batchtools)
reg = makeRegistry(
  file.dir = "~/project",           # accessible on all nodes
  seed = 1                          # initial seed for first job
)
```

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

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 $1 + 9$, $2 + 8$, $\ldots$, $9 + 1$

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

- Stores function on file system
- Creates jobs as rows in a **data.table**
- Parameters also serialized into the **data.table** for fast access
- All jobs get unique positive integers as IDs
- `reg =` can be omitted in most cases. See `?getDefaultRegistry`.

SUBSET JOBS

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

```
findJobs(j==1)
findNotSubmitted()
findDone()
```

- Set operations on `job.id` **data.tables**: `merge`
- **data.table** of `job.id`'s can be passed to basically all functions interacting with the batch system

SUBMIT JOBS

- 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(resources = res)
```

- Submits all jobs per default
- Subsets of jobs can be providing as **data.table** or `vector`

```
submitJobs(ids = 1:5, ressources = res)
```

SUPERVISE AND DEBUG

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

```
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(findErrors()[1])`
- You can also `grepLogs(pattern)`
- Found a bug? `killJobs(findRunning())`
- Run a job in the current **R** session: `testJob(id)`

COLLECT RESULTS

Reduce:

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

- Convenience wrappers around `reduceResults`: `reduceResults[DataTable|List]`

Simple Loading:

```
loadResult(id = 1)
```

MORE CONFIGURATION

Configuration file `~/.batchtools.conf.R`:

```
cluster.functions = makeClusterFunctionsSlurm("~/slurm_lmulrz.tmpl",
  clusters = "serial")
default.resources = list(walltime = 3600, memory = 1024,
  ntasks = 1)
debug = FALSE
max.concurrent.jobs = 999L
```

EXPERIMENTS IN BATCHTOOLS

Intended as abstraction for typical statistical tasks:

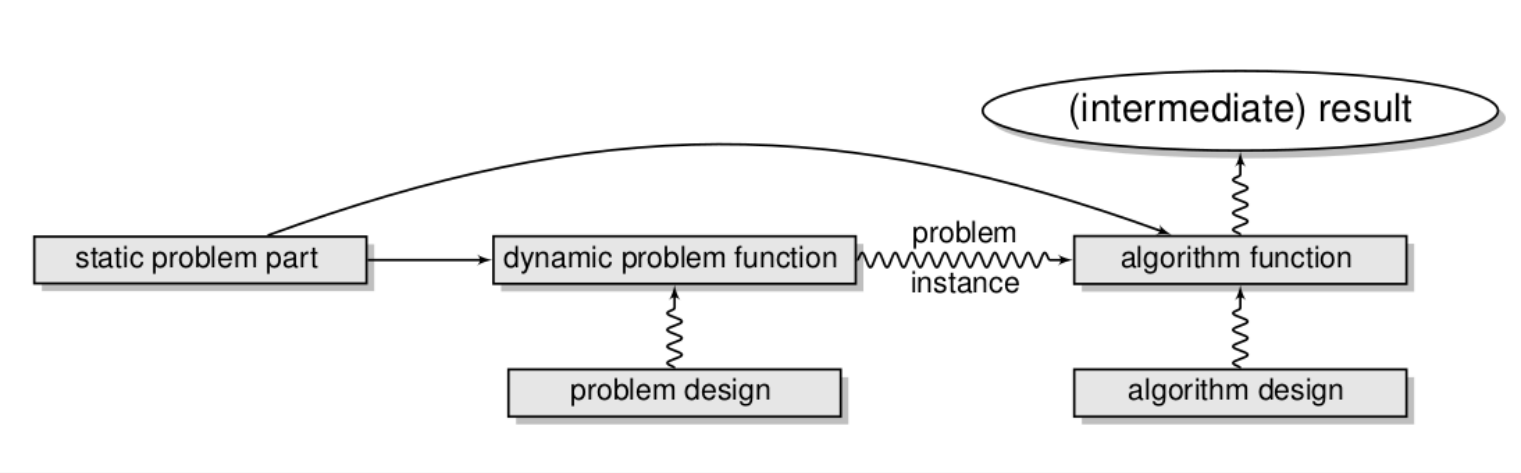
Applying algorithms on problems

- 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
- ...

ABSTRACTION OF COMPUTER EXPERIMENTS



- Problem definition split into static and dynamic part
  - Immutable **R** objects: matrix, data frames, ...
  - 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

EXPERIMENT DEFINITION STEPS

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

A SIMPLE EXAMPLE

```
reg = makeExperimentRegistry("test_reg")
addProblem(name = "p1", data = 1, fun = function(data, job) runif(data))
addAlgorithm(name = "a1",
  fun = function(job, data, instance) 2 * instance)
addAlgorithm(name = "a2",
  fun = function(job, data, instance) data + instance)
addExperiments(repls = 2)
submitJobs()
res = reduceResultsDataTable()
getJobPars()[res]

#   job.id problem algorithm        V1
#1:      1      p1        a1 0.8617642
#2:      2      p1        a1 0.9606042
#3:      3      p1        a2 1.9529027
#4:      4      p1        a2 1.5205857
```

SUMMARY

- Reproducibility: Every computation is seeded, seeds are stored in a **data.table**
- 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
- Greatly simplifies the work with batch systems
- Interactively control batch systems from within R (no shell required)
- Do reproducible research
- Exchange code and results with others

BATCHTOOLS DEMO

BEST PRACTICES

USE A SEPARATE CONFIGURATION FILE

- config.R

```
FILEDIR = "~/project/bla"
SEED = 12345
REPLS = 10

...
```

- benchmark.R

```
library(batchtools)
source("config.R")

reg = makeExperimentRegistry(file.dir = FILEDIR, seed = SEED)

...

addExperiments(repls = REPLS)
```

TEST STUFF LOCALLY

- `config.R`

```
LOCAL = TRUE
FILEDIR = "~/project/bla"
SEED = 12345

ITERATIONS = ifelse(LOCAL, 1, 10^6)
REPLS = ifelse(LOCAL, 1, 10)
...
```

Have a flag, e.g. `LOCAL`, in your config file that sets all expensive parameters (number of iterations, size of a neural network etc.) to very small values. This allows to **quickly** check if your code runs on a technical level.

DO NOT PUT SUBMITJOBS() IN YOUR SCRIPT

```
source("benchmark.R")
reg$cluster.functions # check the backend
reg$default.resources # check resources
summarizeExperimens() # check experiments
submitJobs()
```

Before submitting you jobs it is always a good idea to do one final check of the resources and of jobs.

It is always bad when you have to kill a large number of jobs because something was set wrong.

EXPONENTIAL BACKOFF

- Oftentimes it is not clear how much runtime and memories jobs really require.
- Especially in machine learning run times can be vastly different depending on many factors.
- The idea of an exponential backoff is to start with low resources on all jobs such that the majority of the jobs should be able to finish.
- For expired jobs gradually increase runtime and/or memory until they can finish.

For example start all jobs with a runtime of 10 minutes. 80% of the jobs finish, restart the remaining 20% with 1 hour runtime. Again 80% finish while 20% (out of the remaining 20%) still expire. Increase runtime to 6 hours...

ADDITIONALLY

- Use `tmux` or a different tool to keep terminal sessions alive.
- Think about what objects you want to save.
  - Everything you don't save is gone!
  - **But**: Registries can become very large.
- Use relative paths (ideally from the root of a git repository). This allows transportability between machines.
- If code (e.g., for algorithms) gets complex, put it in separate files and add them to the source of the registry
  `makeRegistry(..., source = c("file1.R", "file2.R")`