Applied Machine Learning

Parallelization:
Batchtools package

Learning goals

- Understand parallelization concepts
- Introduction to batchtools package

PARALLELIZATION

- **Goal**: Minimize computation time by distributing tasks across CPUs/GPUs.
  $\Rightarrow$ Speedup is ideally linear but often limited by overhead and dependencies.
- Debugging parallel code is especially hard.
- Coding discipline is even more important to minimize errors and frustration.
- **What can be easily parallelized?**
  - Independent replications
  - Resampling, cross-validation
  - Model averaging
  - Parameter variations in simulations ...
  - "Single program, multiple data"
  - Everything expressible as a loop of independent iterations (if you can write it with `(l|m|)apply`, you are fine)
- **Many statistical problems are "embarrassingly parallel"**

NAIVE BATCH COMPUTING (NON-CLUSTER)

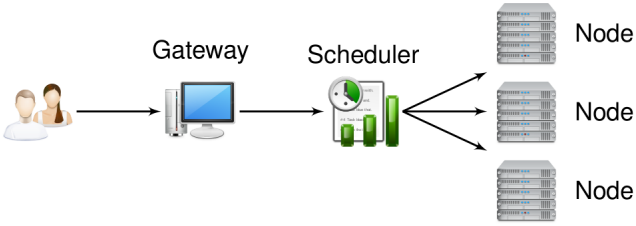
Workflow on multicore machines:

- Write standalone script(s) to run jobs and save outputs.
- Hard-code parameters or pass via Command-Line Interface (CLI).
- Log in via SSH; run with `R CMD BATCH myscript.R`.
- Use `nohup`, `screen`, or `tmux` to persist after logout.
- Manually start jobs when CPUs free up.
- Check completion and errors by hand.
- Write scripts to merge results.

Drawbacks:

- No resource management, automation, or fair scheduling.
- Poor scalability; hard to debug parallel issues.
- No guarantees for reproducibility (e.g., seeding).

HIGH-PERFORMANCE COMPUTING (HPC) CLUSTERS



- Users access a gateway server (head node).
- Cluster = multiple nodes managed by a scheduler (e.g., SLURM).
- Scheduler assigns jobs to nodes via a queuing system.
- Nodes share a common file system.

MANUAL WORKFLOW ON A HPC CLUSTER

Resource Specification:

- Define CPU count, memory, runtime, partition.
- Set command (e.g., `R CMD BATCH script.R`).

Manual Tasks:

- Submit jobs via CLI or shell scripts.
- Monitor with tools like `squeue`.
- Write aggregation scripts for results.

Typical Workflow:

- Unroll `R` loops into single-iteration scripts.
- Auto-generate job and script files per task.
- Submit jobs; crawl logs and outputs.

Handling Issues:

- Kill + resubmit on failure.
- Adjust resources on wall-time hits.
- Full rerun for changes in data or params.

BATCHTOOLS OVERVIEW

- R package for structured access to batch systems.
- Built around Map-Reduce: apply algorithms to many problems.
- Full control from R: submit, monitor, kill jobs.
- Persistent state: resume and audit large experiments.
- Convenient debugging and result collection.
- Supports reproducibility across hardware and job schedulers.
- Supports multiple execution backends:
  - **Interactive:** Run jobs directly in the current `R` session
  - **Multicore:** Parallel execution on local CPU cores
  - **SSH:** Offload jobs to remote machines via SSH
  - **HPC schedulers:** SLURM, Torque/PBS, Load Sharing Facility (LSF), etc.

Project Page: `https://github.com/mllg/batchtools`

Paper: `https://doi.org/10.21105/joss.00135`

CREATING AND CONFIGURING A REGISTRY

Purpose:

- A registry object is used to access and exchange information: file paths, job parameters, and computational events, ...
- Stores all data in a single, portable directory for easy tracking and reproducibility.

Initialization of a new registry:

```
library(batchtools)
reg = makeRegistry(
  file.dir = "registry",  # Directory accessible on all nodes
  seed = 1                # Initial seed for reproducibility
)
```

Configure the system:

```
# Set interactive mode and start jobs in external R sessions
reg$cluster.functions = makeClusterFunctionsInteractive(external = TRUE)
```

- Each supported system has its own `makeClusterFunctions*` function.

Load an existing registry to continue work:

```
loadRegistry("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 Status and Parameters

- Use `find*` functions to query job IDs by computational status:
  - `findError` to get job IDs for failed jobs
  - `findDone` to get job IDs for successful jobs
  - `findNotSubmitted` to get job IDs in order resume jobs
  - ...
- Query job IDs by parameters with `findJobs(pars)` (here: i and j), e.g.,:

```
job = findJobs(i == 2 & j == 8)
job
## Key: <job.id>
##    job.id
##     <int>
## 1:      2
```

- Pass the `data.table` containing the `job.ids` to functions interacting with the batch system, e.g., `submitJobs(ids = job)`

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, resources = res)
```

- Collect/reduce results:

```
# get results of each job in a list
reduceResultsList(ids = findDone())

# get result of single job
loadResult(id = 1)
```

SUPERVISE AND DEBUG

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

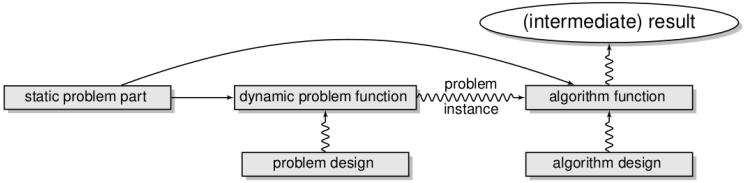
```
## Status for 9 jobs at 2019-10-10 17:49:48:
##   Submitted    : 9 (100.0%)
##   -- Queued    : 0 (  0.0%)
##   -- Started   : 9 (100.0%)
##   ---- Running : 0 (  0.0%)
##   ---- Done    : 9 (100.0%)
##   ---- Error   : 0 (  0.0%)
##   ---- Expired : 0 (  0.0%)
```

- 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)`

EXPERIMENTS IN BATCHTOOLS

- **Purpose**: Abstraction for typical statistical tasks.
- **Applying Algorithms to Problems**:
  - Ideal for simulations, benchmark experiments, sensitivity analyses, ...
  - Simplifies workflow with a focus on job definition.
- **Scenarios**:
  - Compare machine learning algorithms on multiple datasets.
  - Compare one/many estimation procedure(s) on simulated data.
  - Compare optimizers on various objective functions.

ABSTRACTION OF COMPUTER EXPERIMENTS



- **Problem Definition**:
  - **Static part**: Immutable R objects (e.g., matrices, data frames).
  - **Dynamic part**: Arbitrary R functions (e.g., transformations of static objects, data extraction from external sources, data generation functions).
- **Parametrization**: Specify experimental designs for problems and algorithms.
- **Seeding and Reproducibility**:
  - Each step is automatically seeded.
  - Random seeds are stored in a database for reproducibility.

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, seed = 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]

## Key: <job.id>
##    job.id problem prob.pars algorithm algo.pars   result
##     <int>  <char>    <list>    <char>    <list>   <list>
## 1:      1      p1 <list[0]>        a1 <list[0]>   0.5310
## 2:      2      p1 <list[0]>        a1 <list[0]>   0.3698
## 3:      3      p1 <list[0]>        a2 <list[0]>   1.2655
## 4:      4      p1 <list[0]>        a2 <list[0]>   1.1849
```

SUMMARY

- **Reproducibility**:
  - Every computation is seeded.
  - Seeds are stored in a `data.table`.
- **Extensibility**:
  - Easily add more problems or algorithms.
  - Try different parameters or increase replications at any stage.
- **Portability**: Data, algorithms, results, and job information in a single directory.
- **Exchangeability**: Share your file directory to allow others to extend your study with their data sets and algorithms.
- Simplifies working with batch systems.
- Control batch systems interactively from within R (no shell required).
- Facilitates reproducible research.
- Enables easy exchange of code and results with others.