STATS 769
Parallel Computing

Paul Murrell

The University of Auckland

October 7, 2019

Overview

- This section of the course (two lectures) explores parallel computing.
- We will look at how to make code run faster by splitting the execution into pieces and running several pieces at the same time.

Stop and think

Speed of computation is not the only consideration.

- Your code must get the right answer.
- Your code must be understandable (by you and others).
- Your code must be shareable/runnable (by others).
- Human time is more expensive than computer time.

- Parallel computation comes with overheads (set up and communication between parallel computations).
- As was the case when working with large data, ALWAYS start small and then scale up.

Types of parallelism

- We will only consider "embarrassingly parallel" jobs.
- Embarrassingly parallel means the job splits into separate pieces that can be run completely independently. A master submits pieces of the job to workers and collects the results.
- More sophisticated parallelism involves processes or threads sending messages to each other to coordinate their actions. This allows you to create more complicated programming models, but it gets much harder to program.

Parallel issues

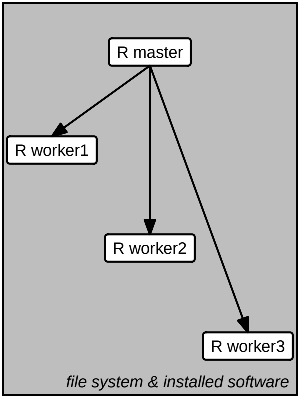
- How do we run multiple R sessions at once?
- How do we get R expressions and data to multiple R sessions?
- How do we get results back from multiple R sessions?

Multiple cores

- Modern computers have more than one CPU core.
- Use `detectCores()` to determine available number of cores.
- Use `mclapply(X, FUN, mc.cores=)` from **parallel** to run code on multipe cores at once
  (and get the result as a list).
- `X` is vector or list.
- `FUN` is function to be called on each element or component of `X`.
- `mc.cores` is number of cores to use.

- `X` is split into as many parts as there are cores and each core works on one part.
- The results from all cores are gathered back into a list (the same length as `X`).

Multiple cores



Monitoring CPU performance

- Still use `system.time()` in R
- Still use `time -p` in the shell
- BUT now take more notice of "elapsed" or "real" time.

- Use `top` to see percentage of CPU usage.
- Use `top -d` to change delay between updates.
- Press `1` (one) to get per-cpu information.
- Press `q` to quit `top`

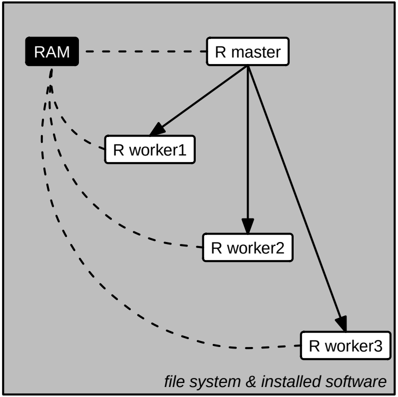
Shared memory

- The `mc*()` functions in **parallel** rely on “forking” an R session.
- A forked R session is a copy of the current session (rather than an entirely new R session), so things like global variables, packages, etc are shared (read-only) between sessions.
- **This is NOT available on Windows.**

- An alternative approach involves R sessions as separate processes, each with their own RAM (the processes communicate via “sockets”).

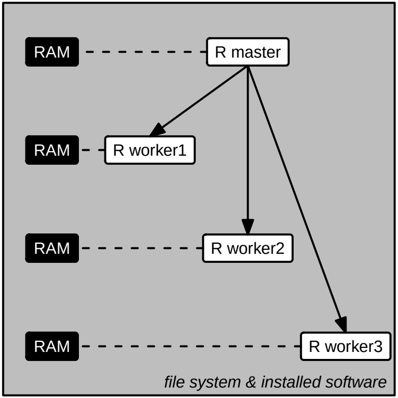
Multiple cores

Forked R sessions



Multiple cores

Independent R sessions



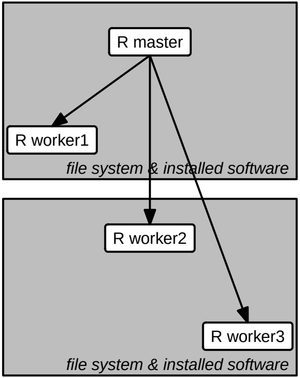
Multiple machines

It is also possible to run multiple independent R sessions.

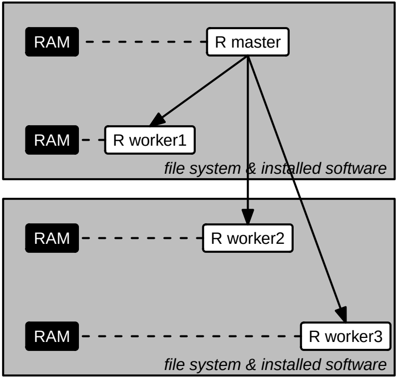
- Use `makeCluster()` to create cluster.
- Use `clusterExport()` to share R objects and functions between worker R sessions.
- Use `clusterEvalQ()` to run an R expression on worker R sessions.
- May also need to copy files to remote machines.
- Use `parLpply()` to run code on the cluster (and get the result as a list).
- Use `stopCluster()` to shut down the cluster.

- The independent R sessions can also be running on a remote machine.

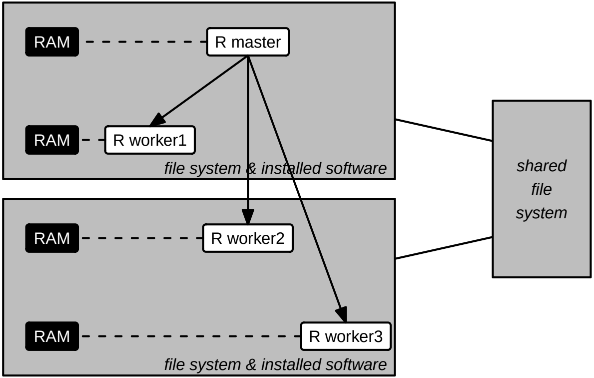
Multiple machines



Multiple machines



Multiple machines



Multiple machines

- `makeCluster()` starts an R session per node.
- Nodes can run on local cores or on remote machines.
- May require specification of how to run R on remote machine.
- Remote machines need to know the name of the local machine.
- Running R on a remote machine requires authentication (username and password).

Multiple machines

- We can set up automatic authentication (via `ssh`).
- Run `ssh-keygen` on your home machine to generate keys (with empty passphrase).
- `scp` your public key from your home machine, `/.ssh/id_rsa.pub`, to the remote machine.
- `ssh` into the remote machine.
- Append your public key from your home machine to `/.ssh/authorized_keys` on the remote machine.
- If you have done it correctly, you should be able to `exit` the remote machine and then `ssh` back into the remote machine without being prompted for a password.
- The `ssh-copy-id` can do the copying across of keys in one step.

Multiple machines

- You can now create a cluster from an R session on the home machine that includes R sessions on the remote machine.

- NOTE that with /.ssh/authorized_keys set up, you no longer have access to SONAS_HOME
  (because of how the VM is set up).
- You can reacquire SONAS_HOME with ...

  ```
  mv .ssh/authorized_keys .ssh/authorized_keys_hidden
  ```

  ... then `exit` and `ssh` back in.
- Reverse that step to restore automatic authentication.

Parallel complexities

- Reproducible random number generation
- Job scheduling and load balancing

Random numbers

Some extra care should be taken with random number generation when performing computations in parallel.

- Random number generators are **pseudo**-random number generators.
- The stream is deterministic with properties of randomness.
- A **seed** is used to start the stream.
- This is good for reproducibility, but bad if it happens accidentally.

Some solutions ...

- Generate all random values in the master (not in the workers).
- Use `RNGkind("L'Ecuyer-CMRG")` to choose a RNG that can produce multiple streams that do not overlap.
- Use `set.seed()` (plus `mclapply()`) or `clusterSetRNGStream()` (plus `clusterApply()`) to set a different known seed for each worker.

Load Balancing

If some jobs take longer than others ...

- Schedule jobs as previous jobs complete (rather than all at the start).
- Run longer jobs first.

- On a managed cluster a sophisticated job scheduler will take care of all of this for you.

Abstraction

- The **foreach** package.
- `foreach (i = values) %dopar% ...`
- `library(do*)` plus `registerDo*()`

- Advantage is convenience (do not have to worry about details, plus have standard interface for multiple back-ends).
- Disadvantage is convenience (loss of flexibility, plus do not notice the details).

Implicit parallelism

Some R functions automatically parallelise computations for you.

- The `boot()` function in the **boot** package has a `parallel` argument.
- The **caret** package automatically works in parallel if a **foreach** back-end package is registered.
- The **data.table** package automatically runs code in parallel (you will not see much, if any, speed up from running **data.table** code via `mclapply()`).

- Advantage is convenience (do not have to worry about details).
- Disadvantage is convenience (loss of flexibility, plus do not notice the details).

Parallel Shell

- Use `&` to run a program in the background (so that you can continue to do other work in the shell).
- Use `nice` to de-prioritise your code.

- GNU `parallel`

  `parallel <cmd> ::: <input>`
- Replacements in `<cmd>`: `{}`, `{/}`, `{.}`, `{//}`, `{/.}`

Resources

- The "parallel" vignette from the **parallel** package
  `vignette("parallel")`
  `https://stat.ethz.ch/R-manual/R-devel/library/parallel/doc/parallel.pdf`
- The **foreach** package
  `http://cran.stat.auckland.ac.nz/web/packages/foreach/vignettes/foreach.pdf`
- Running **caret** calls in parallel
  `http://topepo.github.io/caret/parallel.html`
- GNU parallel tutorial
  `https://www.gnu.org/software/parallel/parallel_tutorial.html`