Parallel Processing

William Lowe

Hertie School

19th November 2020

Problem

"Muuuh, my code is so slow…"

Four types of slow

Code can be slow because it's constrained by

→ network access e.g. web scraping, where *you* may be the one enforcing the slow
→ memory e.g. big data needed in memory
→ disk access e.g. searching through huge files
→ processing power e.g. difficult optimization or lots of bootstrap samples

Solutions to memory-constrained slow

More memory :-)

Solutions to memory-constrained slow

More memory :-)

- → Memory efficient R packages: `biglm`, `speedglm`, `biglars`
- → Use a database: `sqlite3` or `monet`, accessed via `RSQLite` and `MonetDBLite` respectively, and best approached through `dbplyr` unless you know SQL
- → 'Memory map' the files: `mmap` but also others
- → Use a 'distributed file system': `sparklyr` (built into RStudio), `SparkR` (not Dropbox, NFS partition, etc.)

Sampling:

- → Sometimes a stratified sample is as good as a census…

External options:

- → Use the command line tools Janssens, J. (forthcoming)
- → or Python instead of R: same problems, same solutions, different package names

Solutions to processor-constrained slow

Step 0:

- → Profile: casual: system.time and serious: microbenchmark package

Then, in rough order of preference:

- → Write better code (!)
- → No really. Write better code. See 'Advanced R' (ch. 23-25) for excellent advice
- → Run your code on somebody else's machine too
- → Run your code on more of your own machine
- → Run your code on more of somebody else's machine

Maybe…

- → Write *different* code, e.g. C++ using Rcpp

Faster R?

Two good tips for fast R:

- → Don't write Python in R. Use vectorised functions where they exist
- → Stop R copying things in the background

```r
collapse <- function(xs) {
  out <- ""
  for (x in xs)
    out <- paste0(out, x)
  out
}
loop10  <- collapse(strings10),
loop100 <- collapse(strings100),
vec10   <- paste(strings10, collapse = ""),
vec100  <- paste(strings100, collapse = "")
```

expression	min(μs)	median(μs)	itr/sec
loop10	50.7	53.2	18490.
loop100	954.1	976.1	1007.
vec10	10.3	11.1	88582.
vec100	45.8	46.8	20638.

What can you expect?

- → Better code: *Potentially* orders of magnitude speedup
- → Parallel: <2 x speedup
- → Parallel: <16 x speedup (on my laptop)
- → Parallel: <28 x speedup (An expensive AWS instances)

Architecture

hardware architecture

- → Several CPUs a.k.a. ‘sockets’
- → each with several cores

A ‘cluster’ just ties several machines together

Architecture

hardware architecture

- → Several CPUs a.k.a. 'sockets'
- → each with several cores

A 'cluster' just ties several machines together

software architecture

- → A 'process' is a bunch of data and code in memory (different processes can't see each others' memory)
- → A 'thread' is a path of execution in a process (all threads see the same memory)

POSIX compliant operating systems (basically everything except Windows) can parallelize across 'processes' *or* across cores.

- → Where possible, parallelise across cores

Two types of parallel

Inside the same CPU(s):

→ easier
→ shared memory
→ fast communication
→ fewer cores to work with

Between distinct CPU(s):

→ harder
→ memory contents need duplicating
→ slower comunication
→ more cores to work with

How much faster can we get?

Parallelisation big picture

- → *Split* Break the computation into parts
- → *Apply* Send to each computing unit (core / processor) and do the work
- → *Combine* Gather the results together and hand back

Transaction costs! Diminishing returns!

How much faster can we get?

Parallelisation big picture

- → *Split* Break the computation into parts
- → *Apply* Send to each computing unit (core / processor) and do the work
- → *Combine* Gather the results together and hand back

Transaction costs! Diminishing returns!

Bottom line:

- → K-way parallelisation doesn't usually make things K times faster

Are we *already* running in parallel?

```
> library(quanteda)

Package version: 2.1.2
Parallel computing: 2 of 16 threads used.
See https://quanteda.io for tutorials and examples.

Attache Paket: ‘quanteda’

The following object is masked from ‘package:utils’:

    View
```

Less obviously, matrix operations e.g. `%*%` are often *threaded*.

Base R resources

Since R 2.14.0 (in 2011) has contained the `parallel` package

- → You may read about `snow` and `multicore`
- → `parallel` supersedes these but some packages still use them

They reflect the two ways we can run parallel on a single machine

Base r resources

Since R 2.14.0 (in 2011) has contained the `parallel` package

- → You may read about `snow` and `multicore`
- → `parallel` supersedes these but some packages still use them

They reflect the two ways we can run parallel on a single machine

Alternative parallel frameworks

- → `parallel` offers `mclapply`, `mcmapply`, `clusterMap`, `parApply`, `parLapply`, `parSapply`, etc.
- → `foreach` offers `foreach`, `times`.
- → `future.apply` brand new and pretty cool
- → Not quite *ready* for prime time: `multidplyr`

Use cases and problems

Basically: *Lots* of separate actions that *do not need to know about each other*.

Use cases and problems

Basically: *Lots* of separate actions that *do not need to know about each other*.

Where we need to be careful what kind of parallel we're using

- → file pointers 'connections'
- → database connections
- → parsed html pages (e.g. from `rvest`)
- → `Rcpp` and `rJava` objects

And don't do this stuff inside `dplyr` pipes…

Cases where reproducibility is important

- → We usually have to set the random number seed on the cluster itself
- → Don't forget!

Example: (re)sampling

Sampling and re-sampling, e.g. the bootstrap, are natural tasks to parallelise

Reminder about the bootstrap:

- → We would like the sampling distribution of some statistic
- → We can do lots of *hard math* and get an asymptotic answer
- → We can do lots of *computing* and get an (often better but still) asymptotic answer

Intuition:

- → The best idea we have about the structure of the population is the sample
- → Treat the sample as the population
- → Resample with replacement to make a new sample
- → Computer the statistic on that bootstrap sample

Summarise all the bootstrapped values of the statistic and that as the sampling distribution

Example: (re)sampling

Example statistics (univariate, but there's almost no constraint here)

- → Value of coefficient 1
- → Whether coefficient 1 is larger than coefficient 2
- → Predicted outcome when input is…
- → Position of party X in 1990

Let's try this.