Parallel programming in R

Bjørn-Helge Mevik

Research Infrastructure Group, USIT, UiO

NORBIS Course Week, april 2018

Introduction

Simple example

Practical use

The end...

Background

- `R` is (more-or-less) single-threaded
- There are several contributed packages for parallel computation in `R`, some of which have existed a long time, e.g. `Rmpi`, `nws`, `snow`, `sprint`, `foreach`, `multicore`
- `R` itself ships with a package `parallel`
- `R` can also be compiled against multi-threaded linear algebra libraries (BLAS, LAPACK) which can speed up some calculations

Today's focus is the `parallel` package.

To Parallelise or not to Parallelise?

- ▶ Parallelisation is all about increasing speed
- ▶ It can be tricky to get good speed increase by parallelisation
  - ▶ Overhead
  - ▶ Algorithmic constraints
- ▶ Start with optimising your code serially and linking with optimised libraries
- ▶ If you still need more speed: parallelise

How Wide to Parallelise?

▶ If you have one calculation to perform, and a number of CPUs that would otherwise be idle: Increase number of cpus until walltime starts to increase.

▶ In most other situations: Increase number of cpus until cputime (walltime x nCPUs) increases too much.

▶ If you are developing an R-package: Let the users choose.

Overview of `parallel`

- Uses *processes*, not *threads*.
- Based on packages `multicore` and `snow` (slightly modified)
- Includes a parallel random number generator (RNG); important for simulations (see `?nextRNGStream`)
- Particularly suitable for 'single program, multiple data' (SPMD) problems
- Main interface is parallel versions of `lapply` and similar
- Can use the CPUs/cores on a single machine (`multicore`/`snow`), or several machines, using MPI (`snow`)
- MPI support depends on the `Rmpi` package (installed on Abel)

Simple example: serial

▶ parallel provides substitutes for lapply, etc.

▶ Silly example for illustration: caluclate $(1:100)\text{\textasciicircum}2$

Serial version:

```
## The worker function to do the calculation:
workerFunc <- function(n) { return(n^2) }

## The values to apply the calculation to:
values <- 1:100

## Serial calculation:
res <- lapply(values, workerFunc)

print(unlist(res))
```

Simple example: `mclapply`

- ▶ Performs the calculations in parallel on the local machine
- ▶ (+) Very easy to use; no set-up
- ▶ (+) Can be used interactively
- ▶ (+) Less overhead
- ▶ (-) Can only use the cores of *one* machine
- ▶ (-) Uses fork, so it will not work on MS Windows
- ▶ (-) Can leave orhpaned `R` processes
- ▶ => Good for within-one machine, but perhaps not for repeated calculations

Simple example: `mclapply`

```
workerFunc <- function(n) { return(n^2) }
values <- 1:100

library(parallel)

## Number of workers (R processes) to use:
numWorkers <- 8

## Parallel calculation (mclapply):
res <- mclapply(values, workerFunc, mc.cores = numWorkers)

print(unlist(res))
```

Simple example: `parLapply`

- Performs the calculations in parallel, possibly on several nodes
- Can use several types of communications, including `PSOCK` and `MPI`
- `PSOCK`:
    - (+) Can be used interactively
    - (+) Portable; works 'everywhere'
    - (-) Not good for running on several machines
    - => Good for within-one-machine and tests
- `MPI`:
    - (-) Needs the `Rmpi` package (installed on Abel)
    - (-) Cannot be used interactively
    - (+) Good for running on several machines
    - (+) Works everywhere where `Rmpi` does
    - => Good for multi-machine

Simple example: `parLapply` (PSOCK)

```
workerFunc <- function(n) { return(n^2) }
values <- 1:100

library(parallel)

## Number of workers (R processes) to use:
numWorkers <- 8

## Set up the 'cluster'
cl <- makeCluster(numWorkers, type = "PSOCK")

## Parallel calculation (parLapply):
res <- parLapply(cl, values, workerFunc)

## Shut down cluster
stopCluster(cl)

print(unlist(res))
```

Simple example: `parLapply` (MPI)

`simple_mpi.R`:

```
workerFunc <- function(n) { return(n^2) }
values <- 1:100
library(parallel)
numWorkers <- 8
cl <- makeCluster(numWorkers, type = "MPI")
res <- parLapply(cl, values, workerFunc)
stopCluster(cl)
Rmpi::mpi.finalize() # or Rmpi::mpi.quit(), which quits R as well
print(unlist(res))
```

Running:

```
mpirun -n 1 R --slave -f simple_mpi.R
```

Preparation for calculations

- Write your calculations as a function that can be called with `lapply`
- Test interactively with `lapply` serially, and `mclapply` or `parLapply` (`PSOCK`) in parallel
- Deploy with `mclapply` on single node or `parLapply` on one or more nodes
- For `parLapply`, the worker processes must be prepared with any loaded packages with `clusterEvalQ` or `clusterCall`.
- For `parLapply`, large data sets can be exported to workers with `clusterExport`.

Extended example

(Notes to self:)

- ▶ Submit jobs

- ▶ Go through scripts

- ▶ Look at results

Efficiency

- ▶ Don't use more processes than you have *physical* CPU cores!
- ▶ Avoid copying large things back and forth:
  - ▶ Export large datasets up front with `clusterExport` (for `parLapply`)
  - ▶ Iterate over indices or similar small things, not large data sets
  - ▶ Let the worker function return as little as possible
- ▶ The time spent in each invocation of the worker function should be long enough
- ▶ If the time spent in each invocation of the worker function vary very much, try the load balancing versions of the functions

Illustration: Don't use logical CPU cores

- ▶ Machine with 4 physical cores, 2 logical cpus per core
- ▶ Worker functions:
  ```
  sleeper <- function(n) {
    Sys.sleep(10)
    return(42)
  }
  worker <- function(n) {
    i <- 1
    while(i < 2.5e7) { tmp <- sqrt(42); i <- i + 1 }
    return(42)
  }
  ```
- ▶ Timings:
  ```
  > system.time(sleeper(1))
     user   system  elapsed
    0.000    0.000   10.006
  > system.time(worker(1))
     user   system  elapsed
    9.948    0.000    9.949
  ```

Illustration: Don't use logical CPU cores

- ▶ No real computation:

```
> system.time(mclapply(1:2, sleeper, mc.cores = 2))
  user  system elapsed
 0.008   0.000  10.013
> system.time(mclapply(1:4, sleeper, mc.cores = 4))
  user  system elapsed
 0.004   0.004  10.014
> system.time(mclapply(1:8, sleeper, mc.cores = 8))
  user  system elapsed
 0.000   0.008  10.015
> system.time(mclapply(1:16, sleeper, mc.cores = 16))
  user  system elapsed
 0.004   0.008  10.015
```

Illustration: Don't use logical CPU cores

- Actual computation:
  ```
  > system.time(mclapply(1:2, worker, mc.cores = 2))
     user   system  elapsed
   10.100    0.012   10.122
  > system.time(mclapply(1:4, worker, mc.cores = 4))
     user   system  elapsed
   42.048    0.060   10.774
  > system.time(mclapply(1:8, worker, mc.cores = 8))
     user   system  elapsed
  160.056    0.176   21.889
  ```

Illustration: Return as little as possible

- ▶ Worker functions:
  ```
  small <- function(n) {
    Sys.sleep(0.01)
    return(42)
  }
  big <- function(n) {
    Sys.sleep(0.01)
    return(1:100000)
  }
  ```
- ▶ Timings:
  ```
  > system.time(small(1))
     user  system elapsed
     0.00    0.00    0.01
  > system.time(big(1))
     user  system elapsed
     0.00    0.00    0.01
  ```

Illustration: Return as little as possible

▶ Returning a single number:

```
> system.time(mclapply(1:4000, small, mc.cores = 2))
   user  system elapsed
  0.000   0.004  20.279
> system.time(mclapply(1:4000, small, mc.cores = 4))
   user  system elapsed
  0.112   0.052  10.135
> system.time(mclapply(1:4000, small, mc.cores = 8))
   user  system elapsed
  0.152   0.124   5.072
> system.time(mclapply(1:4000, small, mc.cores = 16))
   user  system elapsed
  0.104   0.040   2.542
```

Illustration: Return as little as possible

- Returing $1x10^5$ numbers:

```
> system.time(mclapply(1:4000, big, mc.cores = 2))
   user  system elapsed
  0.244   0.972  22.898
> system.time(mclapply(1:4000, big, mc.cores = 4))
   user  system elapsed
  1.792   2.952  12.034
> system.time(mclapply(1:4000, big, mc.cores = 8))
   user  system elapsed
  1.048   2.664   6.922
> system.time(mclapply(1:4000, big, mc.cores = 16))
   user  system elapsed
  1.756   3.672   4.296
```

Other topics

There are several things we haven't touched in this lecture:

- ▶ Parallel random number generation
- ▶ Alternatives to *apply (e.g. `mcparallel` + `mccollect`)
- ▶ Lower level functions
- ▶ Using multi-threaded libraries
- ▶ Other packages and tecniques

Resources:

- ▶ The documentatin for `parallel`: `library(help = parallel)`, `vignette("parallel")`
- ▶ The book *Parallel R*, McCallum & Weston, O'Reilly
- ▶ The HPC Task view on CRAN: `http://cran.r-project.org/web/views/HighPerformanceComputing.html`