HPC2N

PDC

# Shared memory computing in R

## Course: Parallel computing in R

# Overview

1. Execution types in R
2. Using optimized libraries
3. The parallel package (shared)
4. The foreach package (shared, distributed)

# Execution types in R

- Default R only uses serial computing
- Shared memory parallel computing is using…
  - Thread, Pthread (posix thread)
  - Sockets
  - Rarely OpenMP (Open Multi-Processing, multithreading)
- Distributed memory computing
  - Poor scaling of R above one node

# Using optimized libraries

# Faster linear algebra functions in R

## Basic Linear Algebra Subroutines (BLAS)

Provide standard building blocks for performing basic vector and matrix operations

# BLAS

- Usually installed by default
- Default **BLAS** is not optimized

## Installation

1. In *UBUNTU* install **OpenBLAS** via

```
$ sudo apt-get install libopenblas-base
```

2. in *OSX* install **OpenBLAS** via

```
$ brew install openblas
```

3. In windows, please check out *Microsoft R* https://mran.microsoft.com

# Installation of OpenBLAS

1. Before installation

```
> La_library()
[1] "/usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3.9.0"
> extSoftVersion()["BLAS"]
                                             BLAS
"/usr/lib/x86_64-linux-gnu/blas/libblas.so.3.9.0"
```

2. After installation (**POSIX threads available**)

```
> La_library()
[1] "/usr/lib/x86_64-linux-gnu/openblas-pthread/liblapack.so.3"
> extSoftVersion()["BLAS"]
                                                     BLAS
"/usr/lib/x86_64-linux-gnu/openblas-pthread/libblas.so.3"
```

# Linear algebra computing comparison

- simulate a matrix of 1 million observations by n predictors and generate an outcome y.
- Compute the least squares estimates of the linear regression coefficents when regressing the response y on the predictor matrix X

## Function

```
simLR <- function(n) {
   x <- matrix(rnorm(1e6 * n), 1e6, n)
   b <- rnorm(n)
   y <- drop(x %*% b) + rnorm(n)
   b <- solve(crossprod(x), crossprod(x, y))
   }
```

# Testing with different BLAS

```
start_time <- Sys.time()
simLR(100)
print(Sys.time() - start_time)
```

| Without optimized BLAS | With optimized BLAS |
|---|---|
| **Execution time of 16.81325 secs** | **Execution time of 7.17849 secs** |

# The parallel package

# Parallel package

1. Installed by default

    ```
    library(parallel)
    ```

2. Contains many functions
3. Can detect how many cores you have

    ```
    detectCores(logical = TRUE/FALSE)
    ```

**Hyperthreading**

Process where a **CPU** splits each of its physical cores into virtual cores, which are known as threads

*LOGICAL* = *TRUE* detects the hyperthreaded cores

# Types of parallel calculations

## Forking

- Faster than sockets.
- Because it copies the existing version of R, your entire workspace exists in each process.
- Trivially easy to implement.
- Only works on POSIX systems (Mac, Linux, Unix, BSD) and not Windows.
- Because processes are duplicates, it can cause issues specifically with random number generation

# Types of parallel calculations

## Socket

- Works on any system (including Windows).
- Each process on each node is unique so it can’t cross-contaminate
- Each process is unique so it will be slower
- Things such as package loading need to be done in each process separately. Variables defined on your main version of R don’t exist on each core unless explicitly placed there.
- More complicated to implement.

# Forking parallel example

## List of functions

```
mclapply(X, FUN, ...)
mcmapply(X, FUN, ...)
```

## Example

```
library(parallel)
no_cores <- detectCores() - 1
mclapply(1:100, FUN,
  mc.cores = no_cores)
```

Decrease *no_cores* by one for not interfering with master thread

With small tasks, the overhead of scheduling the task and returning the result can be greater than the time to execute the task itself

# Socket parallel example

## List of functions

```
parLapply(cluster, X, FUN)
parSapply(cluster, X, FUN)
```

## Example

```
library(parallel)
no_cores <- detectCores() - 1
cl <- makeCluster(no_cores)
parLapply(cl, 1:100, FUN)
stopCluster(cl)
```

Decrease *no_cores* by one for not interfering with master thread

# Sockets: Passing objects

- If libraries are used you need to load the package within the process

```
clusterEvalQ(cl, library([LIBRARY NAME]))
```

- Variables are not present within the parallelisation and must be passed.

```
# Move variable x to the cluster
clusterExport(cl, "x")
# Register the variable x on the cluster
clusterEvalQ(cl, x)
```

# The foreach package

**Works for serial, shared and distributed memory computing**

# Serial Foreach example

- We have a list of vectors (Default dataset of co2 concentrations in ppm 1959 to 1997)

```
# Create a list with values from each year
x <- split(co2, ceiling(seq_along(co2)/12))
foreach(i = x) %do%
  mean(i)
```

# Using Foreach package in parallel

1. Must be installed

   ```
   library(parallel)
   library(foreach)
   library(doParallel)
   ```

2. Similar use as **OpenMP** https://en.wikipedia.org/wiki/OpenMP
3. More information at https://cran.r-project.org/web/packages/foreach/foreach.pdf

# How about parallel calculations: doParallel package

1. Packages needed

```
library(parallel)
library(foreach)
library(doParallel)
```

2. Shows how many parallel processes are available

```
getDoParWorkers()
```

3. Register the number of processes

```
registerDoParallel()
```

4. Substitute **%do%** with **%dopar%**

# Parallel shared memory example

```
library(parallel)
library(foreach)
library(doParallel)
x <- split(co2, ceiling(seq_along(co2)/12))
registerDoParallel()
foreach(i = x) %do%
  mean(i)
```

1. Adding extra libraries
2. Register the number of processes (Default: all)
3. Switch *do* with *dopar*

# Defining the number of parallel processes

1. Set the number of processes

   ```
   nproc <- makeCluster(<number of processes>)
   ```

2. Register the number of processes

   ```
   registerDoParallel(nproc)
   ```

3. End all processes

   ```
   stopCluster(nproc)
   ```

# Userful parameters

Process the tasks results as they are generated

```
foreach(i = 1:100,
   .combine='[type]') %do%
  FUN(i)
```

| Type | Process |
|---|---|
| c | returns an array |
| cbind | returns a column matrix |
| rbind | returns a row matrix |
| + | summarize the results |
| * | Multiplicate the results |

# Working examples

1. Equivalent to **lapply(x, mean)**

```
x <- split(co2, ceiling(seq_along(co2)/12))
foreach(i = x) %do%
  mean(i)
```

2. Return array of means

```
x <- split(co2, ceiling(seq_along(co2)/12))
foreach(i = x, .combine='c') %do%
  mean(i)
```

3. replace *do* with *dopar* for parallelisation

# Distributed computing

## How to use foreach for distributed computing

- Can be used for distributed computing using *doMPI* library
- R seldomly scales above one node on a supercomputer

```
library(doMPI)
cl <- startMPIcluster()
# You define N processes when allocating your job
registerDoMPI(cl)
x <- split(co2, ceiling(seq_along(co2)/12))
foreach(i = x) %dopar%
  mean(i)
closeCluster(cl)
mpi.quit()
```