Best Practices for R in HPC

Dec. 2022 | Henric Zazzi & Pedro Ojeda-May | Course: Parallel computing in R

Start with a working single core program

- This code should produce correct results without any optimizations

> "Premature optimization is the root of all evil" --- Donald Knuth

```
N <- 10000
h <- pi/N
mysum <-0

for (i in 1:N) {
  x <- h*(i-0.5)
  for (j in 1:N) {
    y <- h*(j-0.5)
    mysum <- mysum + sin(x+y)
  }
}

mysum*h*h
```

- This initial program will be your baseline for further optimizations

- Write clean code and write useful comments

```
# Code for computing a 2D sinus integral
N <- 10000  # Number of grid points
h <- pi/N   # Size of grid
mySum <-0   # Camel convention for variables' names

for (i in 1:N) {                # Discretization in the x direction
  x <- h*(i-0.5)                # x coordinate of the grid cell
  for (j in 1:N) {              # Discretization in the y direction
    y <- h*(j-0.5)              # y coordinate of the grid cell
    mySum <- mySum + sin(x+y)   # Computing the integral
  }
}

mySum*h*h     # Printing out the result
```

- Use some Version Control software
  - Git
    - HPC2N/UPPMAX 2021 (https://tinyurl.com/2p8cusyu)
    - HPC2N/UPPMAX 2022 (https://tinyurl.com/jk6a6cez)
    - CodeRefinery (https://tinyurl.com/yckrkx2b)
  - Mercurial
- and some Repository
  - Local
  - GitHub (https://github.com)
  - GitLab (https://gitlab.com)

- Organize your code by enclosing repetitive/important code into a function

```
integral <- function(N){
  # Function for computing a 2D sinus integral
  h <- pi/N   # Size of grid
  mySum <-0   # Camel convention for variables' names

  for (i in 1:N) {     # Discretization in the x direction
     x <- h*(i-0.5)    # x coordinate of the grid cell
     for (j in 1:N) {  # Discretization in the y direction
       y <- h*(j-0.5)  # y coordinate of the grid cell
       mySum <- mySum + sin(x+y)  # Computing the integral
     }
  }
  return(mySum*h*h)
}

integral(N)     # Calling the function
```

Now go for the parallel version

Once this initial program is written, one can start looking for optimizations

Here, it is handy to time parts of the code with built-in functions:

```
system.time( integral(N) )
```

or use existing packages to get more statistics:

```
library(microbenchmark)
bench <- microbenchmark( integral(N) , times=4 )
```

Organize your scripts in a Project

it may contain for instance:

```
Project
|      Readme.md
|      License.txt
|
+------scripts/
|      |    function1.R
|      |    function2.R
|
+------documentation/
       |    doc.md
```

Porting heavy parts of the code to other languages

If even by using the parallel schemes mentioned in this course the performance of your code is low, maybe you could consider the porting of expensive parts in a lower level language such as C/C++, Fortran or Julia. R has several tools to make interfaces with several languages.

Working on the login nodes

Run only lightweight tasks on the login nodes otherwise use the queueing system.

Be cautious about implicit parallelism

Be aware that some libraries create **threads** by default. You may read about the libraries that you are using and fix the appropriate number of threads so that the job does not overload the allocated CPU resources. The following command line tool is your friend:

```
job-usage Job_ID
```

Search if there is a faster version of expensive functions

For instance, for principal components: **prcomp**, **princomp**

For memory expensive jobs

The memory available per core on Kebnekaise is between 4.5-7 GB, if your code is memory demanding, you may request more CPUs than needed with the **-n** option in the batch script

For many single core jobs

Job Arrays feature of *SLURM* is your friend

Keep your .bashrc file simple