INTRO TO PARALLELIZING R CODE

Jacalyn Huband

22 September 2022

University of Virginia | Research Computing

University of Virginia
Research Computing

AGENDA

- Discussion of Parallelizing Code
- Examples of Parallel Packages
- Running Parallel Jobs on Rivanna

DISCUSSION OF PARALLELIZING CODE

What does it mean to parallelize code?

- Most programs are written to run *serially* – each line is executed, one at a time.

- In parallel code, **independent tasks** are assigned to separate processors so that they can run simultaneously.

Serial vs. Parallel Concept

- Serial Example vs. Parallel Example

Saturday Chores

- I will mow the lawn
- I will do the laundry
- I will buy groceries
- I will vacuum the family room

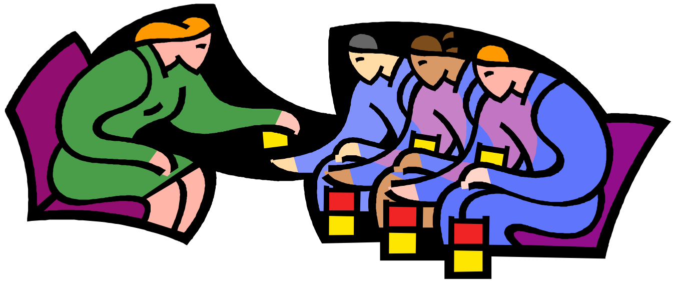
Saturday Chores

- Greg will mow the lawn
- Bobby will buy groceries
- Marcia will do the laundry
- Jan will vacuum the family room

Instead of people doing chores, a parallel program assigns tasks to individual processors (cores).

A Note about parallel R

- R uses a manager/worker model



  - Manager sends tasks to the workers
  - Workers perform the tasks and return the results to the manager
  - The manager combines the results

- The parallel packages in R make the communications between the manager and workers transparent to the programmer.

Why parallelize your code?

- Speed
  Computers have reached a limit for how fast they can perform a computation.

- Time
  When work is being done simultaneously, it can reduce the total amount of time for the code to complete.

Note: If not done properly, parallelizing your code can slow down its performance.

R as a parallel language

- R by itself does not have parallel capabilities.

- There are lots of packages available that can extend R to have parallel capabilities.
  - parallel
  - doParallel
  - multidplyr`

- Today, we will look at a limited number of functions from two packages:
  - *future*
  - *future.apply*
  - *furrr*

Activity

- If you do not have future & furrr installed, go ahead and do that now.

```
install.packages(“future”)
install.packages(“future.apply”)

install.packages(“furrr”)
```

- We also will be using a timing package tictoc. Install tictoc if you do not have it already:

```
install.packages(“tictoc”)
```

Activity

- To get a copy of today's scripts, log onto Rivanna and type:

```
cp -r /project/rivanna-training/multicore_R_2022-09-22 ~
```

Or

```
git clone https://github.com/jhuband/Multicore_R.git
```

DETERMINING THE NUMBER OF CORES

The *future* package

What is a core?

- Most computers come with multiple processors or cores.
- On your laptop, the cores allow you to do several tasks at the same time (e.g., typing a report while watching your slack messages and googling the time that your favorite coffee shop closes).
- To take advantage of having multiple cores, you will need to know how many core are available and how to send tasks to the cores

availableCores() vs detectCores()

- **Caution:** Lots of websites tell you to use `parallel::detectCores()` to determine the number of cores.
  - This will report the number of physical cores.
  - On Rivanna, you may not have access to all of the physical cores

- A safer technique is to use `future::availableCores()`

- An even better approach:

```
numCores <- max(1, future::availableCores() - 1)
```

MULTICORE COMPUTING

The *future* packages

future.lapply package

- A frequently-used functions from the future.apply package:
  - future_lapply
    - Allows you to apply the same function to a list of elements
    - Most basic syntax: **results <- future_lapply(x, FUN)**

      where x = a list of elements, and

      FUN = a function, either built-in or defined by you

      results = list of results from the application of FUN to x

multicore Example 1

future_lapply

01_future_lapply.R:

```
library(future.apply)
library(tictoc)

#Define the function to be passed to the cores
myFunc <- function(value){
   Sys.sleep(1)
   paste0("I received ", value, ". Hello from ",
Sys.getpid())
}

#Define the type of parallelization
plan("multisession", workers=3)

#  Launch the jobs and collect the results
tic()
results <- future_lapply(1:10, myFunc)
toc()
print(unlist(results))
```

Define the task that will be done in parallel.

Define the type of parallelization.

Launch the parallel work with "future_lapply"

multicore Example 1

future_lapply

01_future_lapply.R:

```
library(future.apply)
library(tictoc)

#Define the function to be passed to the cores
myFunc <- functi
   Sys.sleep(1)
   paste0("I rec
Sys.getpid())
}

#Define the type
plan("multisessi

#  Launch the jobs and collect the results
tic()
results <- future_lapply(1:10, myFunc)
toc()
print(unlist(results))
```

Define the task that will be done in parallel

Results:
4.253 sec elapsed
[1] "I received 1. Hello from 18341" "I received 2. Hello from 18341"
[3] "I received 3. Hello from 18341" "I received 4. Hello from 18342"
[5] "I received 5. Hello from 18342" "I received 6. Hello from 18342"
[7] "I received 7. Hello from 18342" "I received 8. Hello from 18340"
[9] "I received 9. Hello from 18340" "I received 10. Hello from 18340"

Launch the parallel work with "future_lapply"

An advantage of the future packages

- Because the future package allows you to specify the type of paralleliztion, you can easily change to the from serial to parallel, or vice versa.

```
plan("multisession", workers=3)
plan("sequential")
```

Activity

- Copy 01_future_lapply.R to 01_serial.R
- Change the plan to sequential and run the code.

- What happened to the timing results?
- Is this what you expected?

furr package

- A frequently-used function from the furrr package:
  - future_map
    - Allows you to apply the same function to a list of elements
    - Most basic syntax: **results <- future_map(x, FUN)**

      where x = a list of elements, and

      FUN = a function, either built-in or defined by you

      results = list of results from the application of FUN to x

multicore Example 2

future_map

02_future_map.R:

```
library(future); library(future.map)
library(tictoc)

#Define the function to be passed to the cores
myFunc <- function(value){
   Sys.sleep(1)
   paste0("I received ", value, ". Hello from ",
Sys.getpid())
}

#Define the type of parallelization
plan("multisession", workers=3)

#  Launch the jobs and collect the results
tic()
results <- future_map(1:10, myFunc)
toc()
print(unlist(results))
```

Define the task that will be done in parallel.

Define the type of parallelization.

Launch the parallel work with "future_lapply"

multicore Example 2

future_map

02_future_map.R:

```
library(future); library(future.map)
library(tictoc)

#Define the function to be passed to the cores
myFunc <- functi
   Sys.sleep(1)
   paste0("I rec
Sys.getpid())
}

#Define the type
plan("multisessi

#  Launch the jobs and collect the results
tic()
results <- future_map(1:10, myFunc)
toc()
print(unlist(results))
```

Define the task that

Results:
4.615 sec elapsed
[1] "I received 1. Hello from 24858" "I received 2. Hello from 24858"
[3] "I received 3. Hello from 24858" "I received 4. Hello from 24859"
[5] "I received 5. Hello from 24859" "I received 6. Hello from 24859"
[7] "I received 7. Hello from 24859" "I received 8. Hello from 24857"
[9] "I received 9. Hello from 24857" "I received 10. Hello from 24857"

Launch the parallel work with "future_lapply"

DoFuture package

- If you absolutely must have a loop
- The DoFuture package allows you to run a foreach loop in parallel
- You will need an additional step of registering the parallelization

multicore Example 3

DoFuture

03_doFuture.R:

```
library(foreach); library(future); library(doFuture)
library(tictoc)
# Define the function to be passed to the cores
myFunc <- function(value){
   Sys.sleep(1)
   paste0("I received ", value, ". Hello from ", Sys.getpid())
}
# Define the type of parallelization
registerDoFuture()
plan("multisession", workers=3)
# Launch the jobs and collect the results
tic()
results <- foreach(x = 1:10, .combine=c) %dopar%
   myFunc(x)
}
toc()
print(unlist(results))
```

Define the task that will be done in parallel.

Define the type of parallelization.

Launch the parallel work with “foreach” and “%dopar%”

multicore Example 3

DoFuture

03_doFuture.R:

```
library(foreach); library(future); library(doFuture)
library(tictoc)
# Define the function to be passed to the cores
myFunc <- function(value){
   Sys.sleep(1)
   paste0("I receive
}
# Define the type of
registerDoFuture()
plan("multisession",
# Launch the jobs an
tic()
results <- foreach(x = 1:10, .combine=c) %dopar%
   myFunc(x)
}
toc()
print(unlist(results))
```

Define the task that will be done in parallel.

Results:

4.328 sec elapsed

```
[1] "I received 1. Hello from 320"  "I received 2. Hello from 320"
[3] "I received 3. Hello from 320"  "I received 4. Hello from 319"
[5] "I received 5. Hello from 319"  "I received 6. Hello from 319"
[7] "I received 7. Hello from 319"  "I received 8. Hello from 321"
[9] "I received 9. Hello from 321"  "I received 10. Hello from 321"
```

Launch the parallel work with "foreach" and "%dopar%"

Activity

I have a program (compute_pi.R) that we can try to parallelize.

```
numDarts <- 1000
circleHits <- 0

for (n in 1:numDarts){
   x <- runif(1);  y <- runif(1)
   d <- sqrt(x*x + y*y)
   if (d <= 1.0){
      circleHits <- circleHits + 1
   }
}
#  Use formula to estimate pi
pi = 4.0 * circleHits/numDarts
cat("\nThe estimate for pi is",pi,".\n")
```

How would you parallelize this program?

Need more help?

Office Hours via Zoom

Tuesdays: 3 pm - 5 pm

Thursdays: 10 am - noon

To connect to the Zoom sessions, go to https://www.rc.virginia.edu/support/#office-hours and click on the "Join us via Zoom" button

Website:

https://rc.virginia.edu

Email:

https://www.rc.virginia.edu/form/support-request/

Questions?