

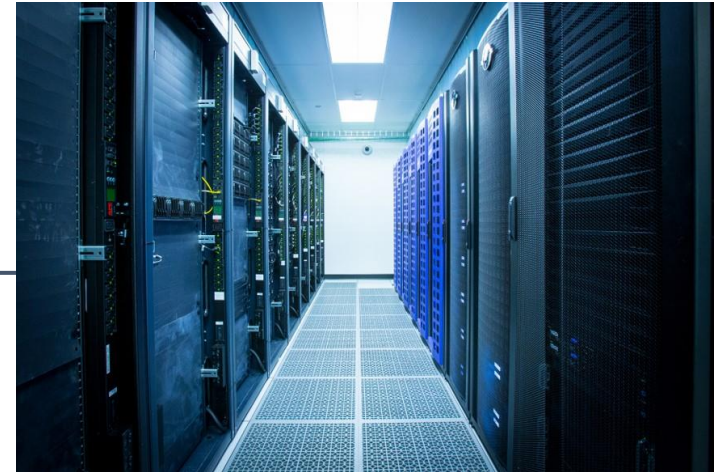
# MAKING R RUN FASTER ON RIVANNA

---

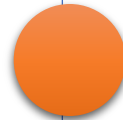
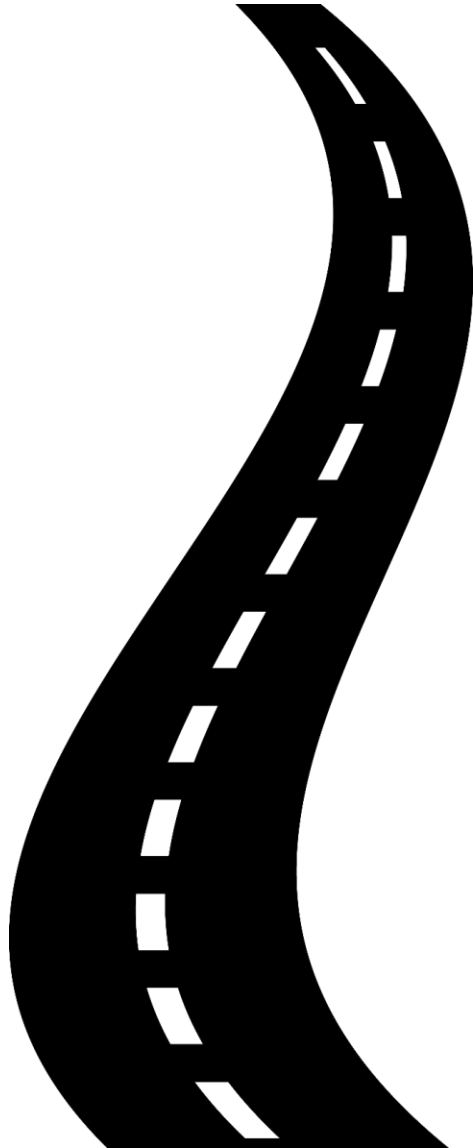
Jacalyn Huband

Senior Computational Scientist  
UVA Research Computing

16 Mar. 2023



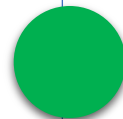
# Roadmap for this Workshop



Talk about optimization to speed up codes.



Look at general techniques for speeding up R

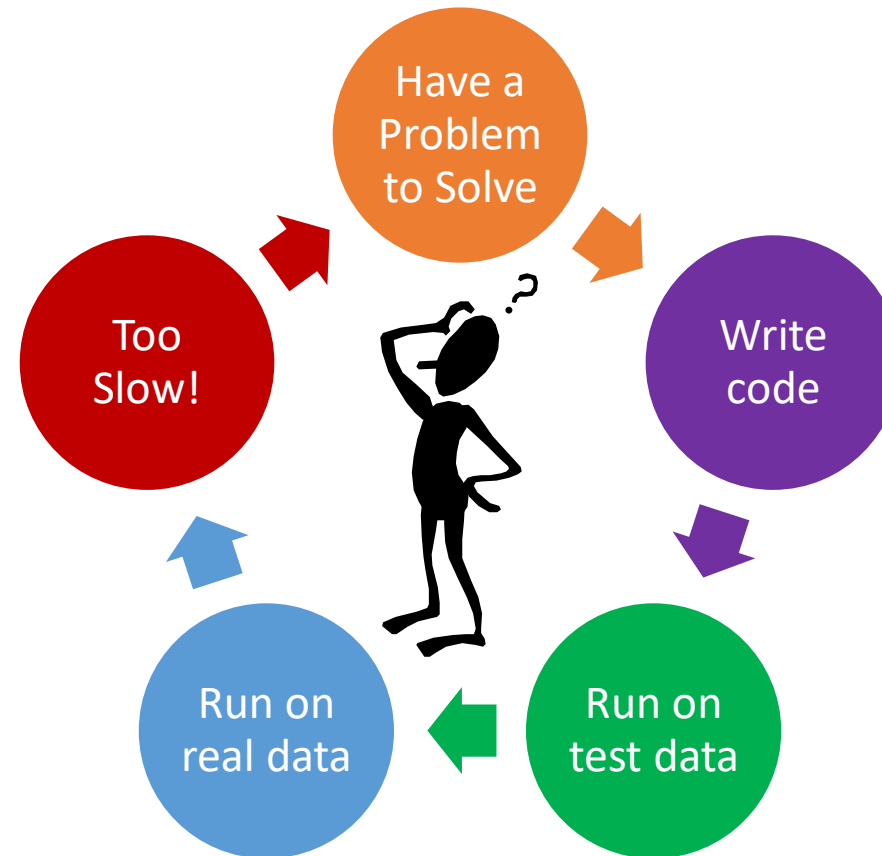


Walk through three case studies places where code can be slow

# INTRODUCTION TO OPTIMIZING R

---

# Your Software Project

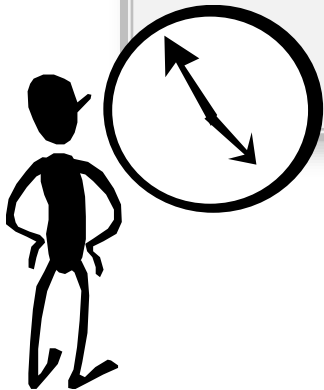


# It's About Time

- After getting your code to work, you may find that the code does not *scale* for larger data sets.

- Suppose it takes only 40 sec for a simple data set.
- But, when you run it on a data set that is twice the size of the first, the time takes 400 secs!

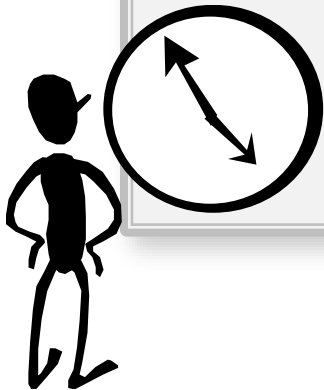
*400 secs = 6.7 mins*



# It's About Time

- Or, you may find that repeating the algorithm multiple times takes too long.

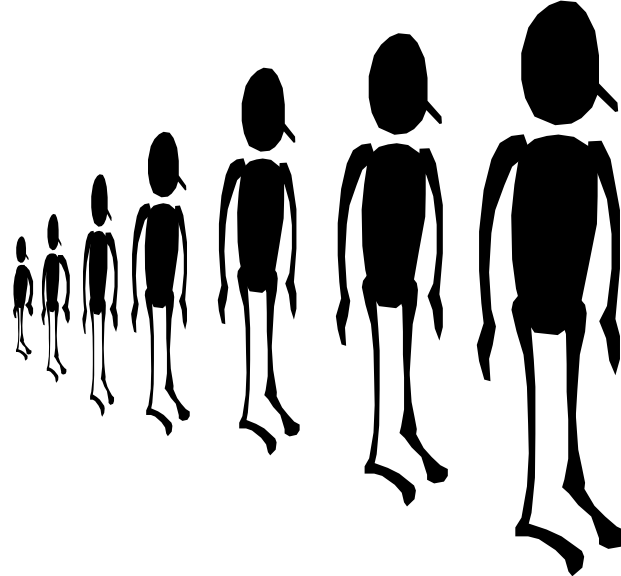
- Suppose it takes only 40 sec for one iteration (or for one data set).
- But, you need to run it for 5000 iterations (or on 5000 data sets).



$$40 \frac{\text{sec}}{\text{iteration}} * 5000 \text{ iter.} = 200,000 \text{ sec or } 55.6 \text{ hours}$$

# Should you parallelize it?

- Before requesting time on the cluster and parallelizing your code, you may want to **optimize** it.



# What is Code Optimization?

- The process of making code more efficient (either with time or memory)
- Important caveats:

- The correctness of the code must be preserved.
- The code should run faster “on average”.
- There is a tradeoff between your efforts and the computer’s efforts. (**Should you spend a week trying to make a program faster by 1 sec?**)



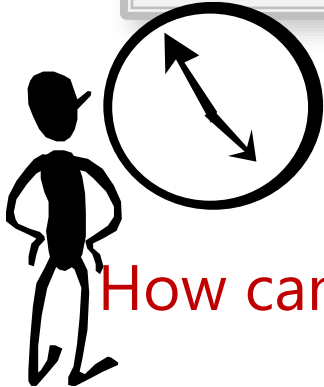


# Going Back to Time

- Returning to our timing example with multiple iterations, suppose you were able to reduce the time of one iteration to 36 seconds.

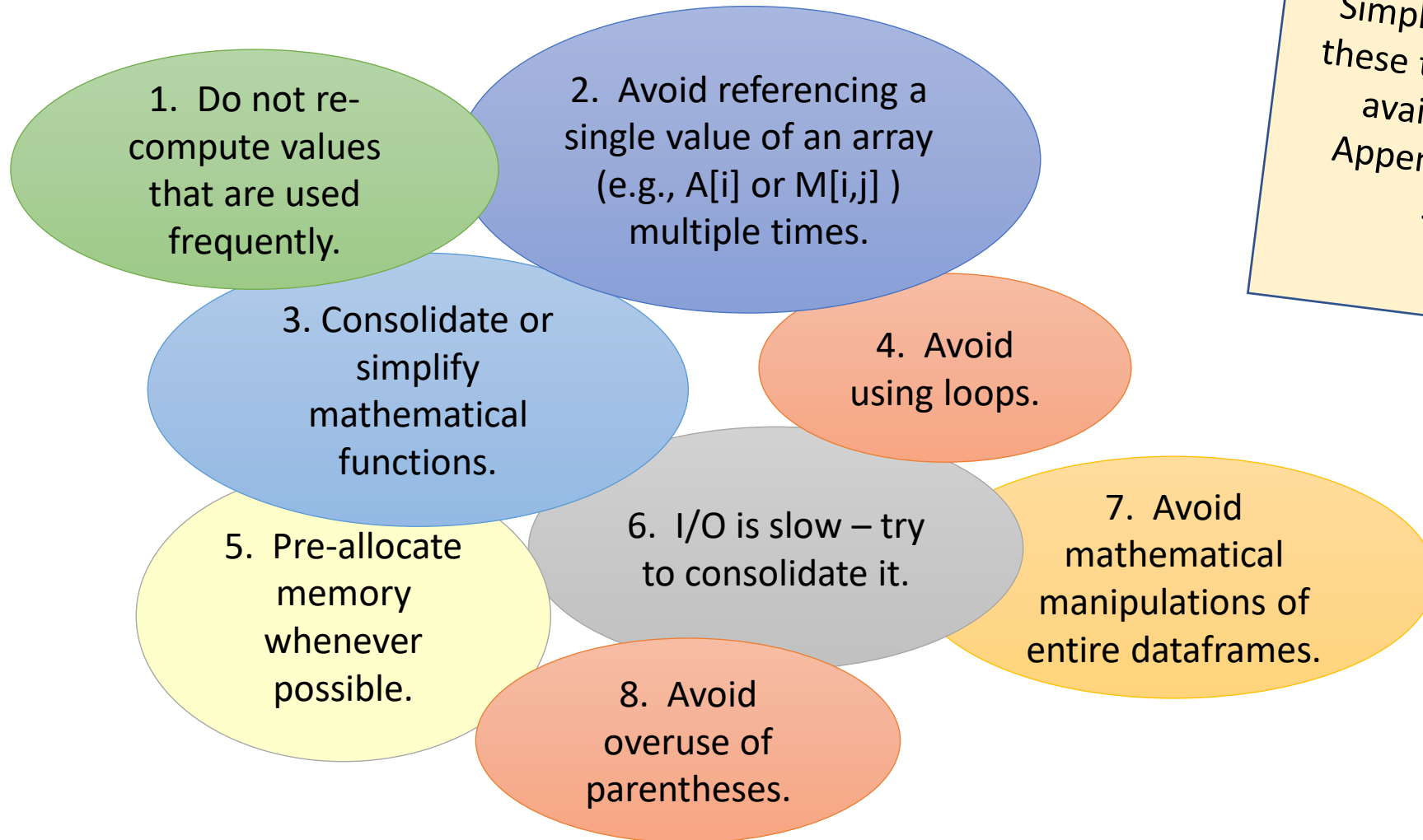
$$36 \frac{\text{sec}}{\text{iteration}} * 5000 \text{ iter.} = 180,000 \text{ sec or } 50.0 \text{ hours}$$

- You just saved 5.6 hours!



- How can we reduce the time for R code to run?

# General Advice



Simple examples of these techniques are available in the Appendix of these slides.

# R is different

- General advice that you hear with other language does not always work with R.
- We will look at 3 case studies
  1. Reading in a Large Data File
  2. Manipulating a column in a table
  3. Performing mathematical operations
- But first, to get these slides and the test cases with codes, you can use Open OnDemand.

# USING OPEN ONDEMAND

---

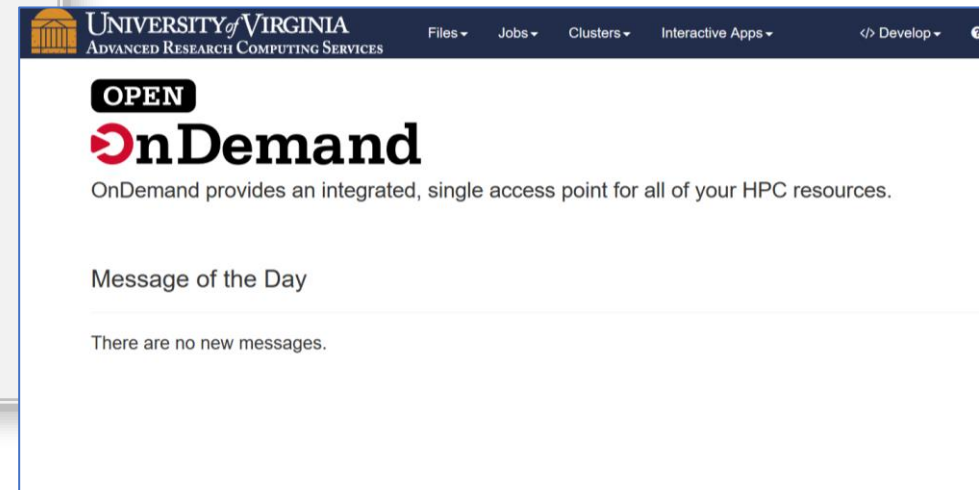
# Open On-Demand

- For today's activities, we are going to use our web-based portal for Rivanna: Open OnDemand. To access it, follow the instructions below:

In your web browser go to  
<https://rivanna-portal.hpc.virginia.edu>

You will need to "Netbadge" in

This will take you to a dashboard with some nice features.

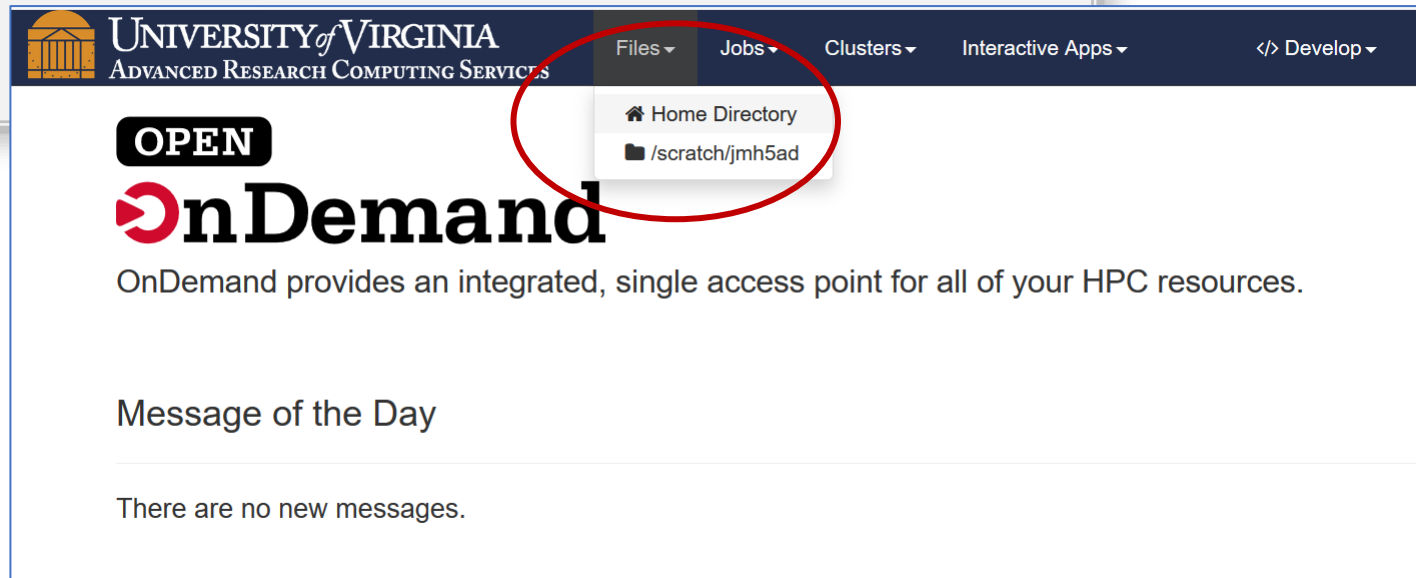


# The Files Page

- First, you will use the Files tool to copy some files to your directory.

Click on Files > Home Directory

This will take you to a page where you can see the files in your home directory.

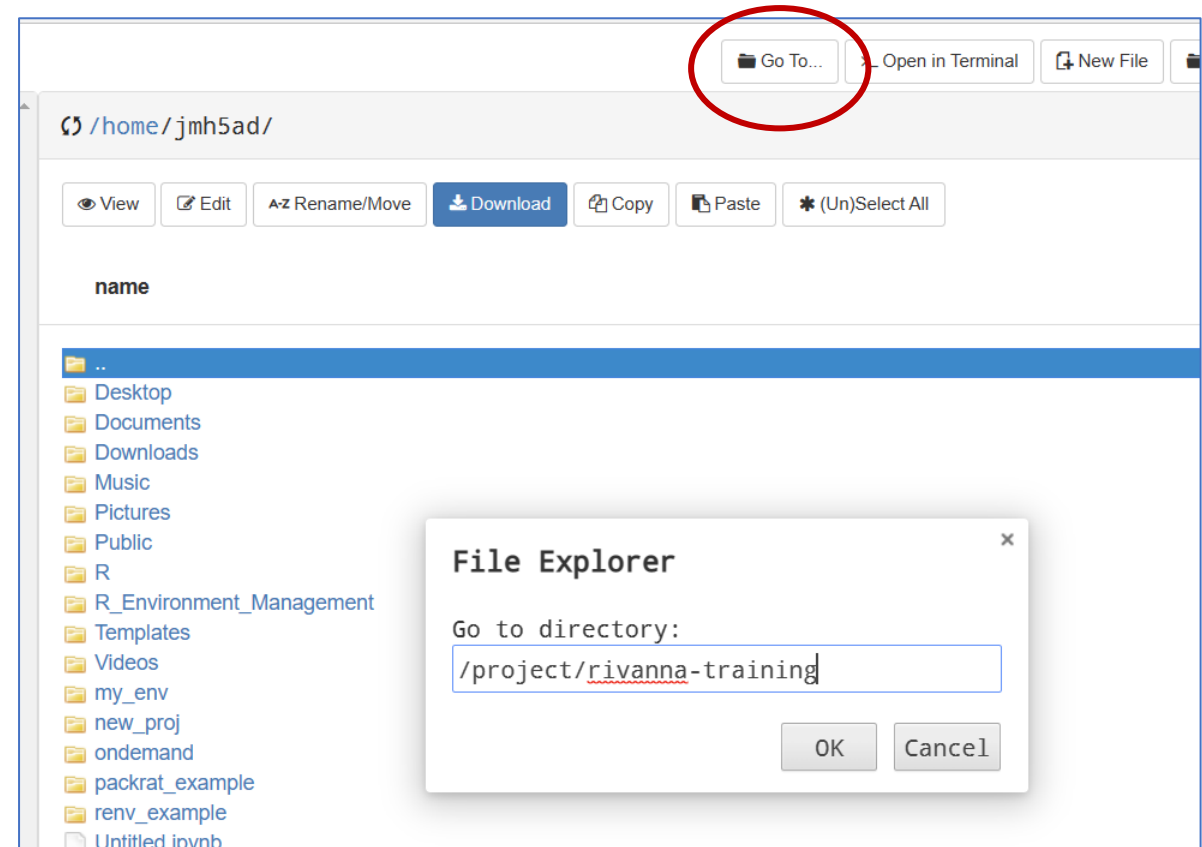


# The Go To Feature

- Click on “Go To . . .” and enter

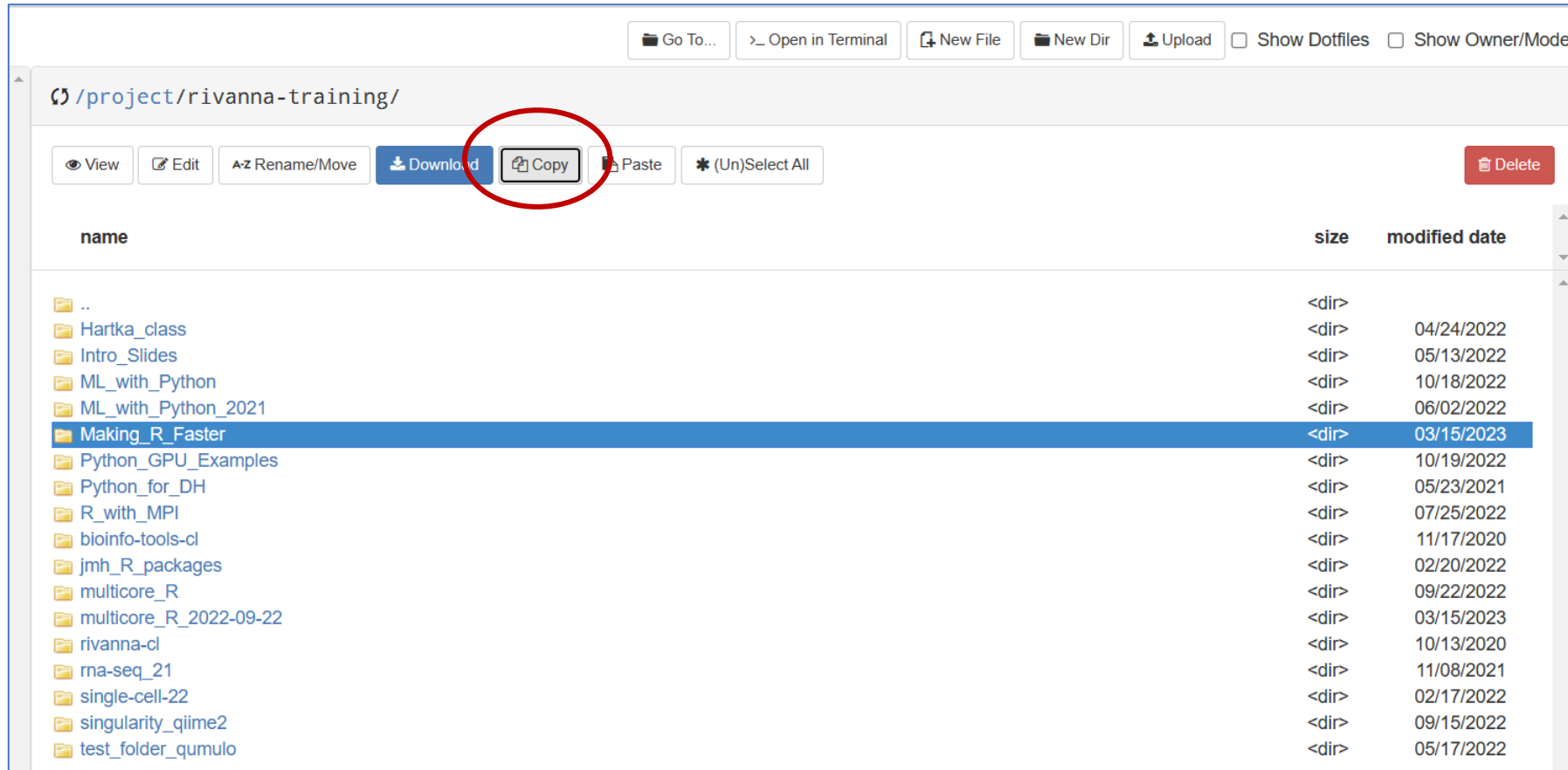
`/project/rivanna-training`

in the dialog box.



# The Copy Feature

Highlight the folder called “Making\_R\_Faster” and click on “Copy”.



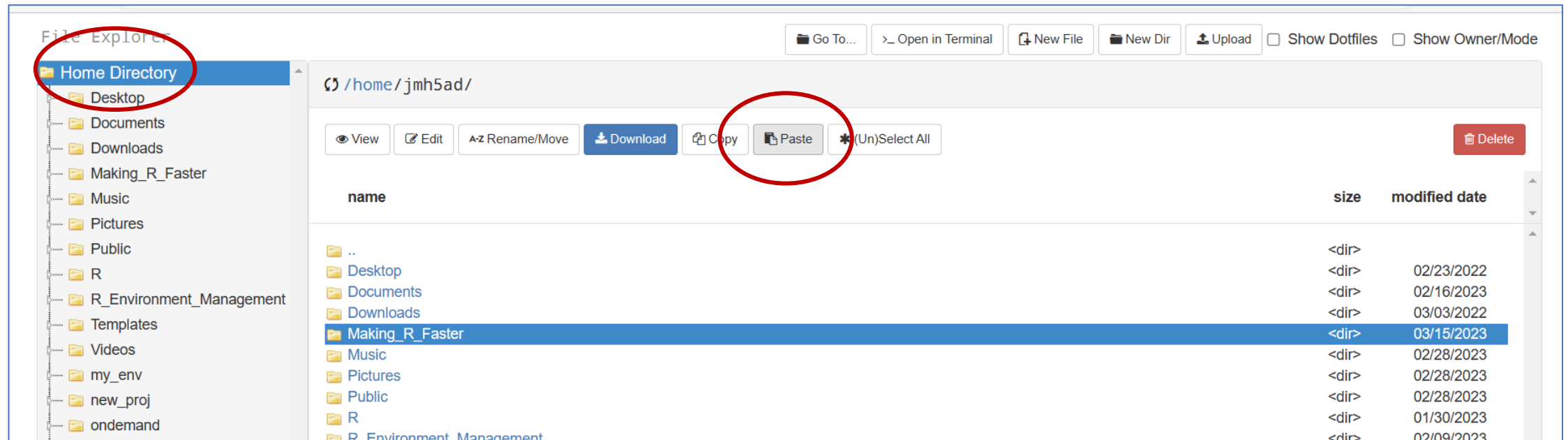
The screenshot shows a web-based file manager interface for the directory `/project/rivanna-training/`. The toolbar includes buttons for View, Edit, Rename/Move, Download, Copy, Paste, (Un)Select All, and Delete. The 'Copy' button is circled in red. Below the toolbar is a table listing the contents of the directory.

name	size	modified date
..	<dir>	
Hartka_class	<dir>	04/24/2022
Intro_Slides	<dir>	05/13/2022
ML_with_Python	<dir>	10/18/2022
ML_with_Python_2021	<dir>	06/02/2022
<b>Making_R_Faster</b>	<dir>	<b>03/15/2023</b>
Python_GPU_Examples	<dir>	10/19/2022
Python_for_DH	<dir>	05/23/2021
R_with_MPI	<dir>	07/25/2022
bioinfo-tools-cl	<dir>	11/17/2020
jmh_R_packages	<dir>	02/20/2022
multicore_R	<dir>	09/22/2022
multicore_R_2022-09-22	<dir>	03/15/2023
rivanna-cl	<dir>	10/13/2020
rna-seq_21	<dir>	11/08/2021
single-cell-22	<dir>	02/17/2022
singularity_qiime2	<dir>	09/15/2022
test_folder_qumulo	<dir>	05/17/2022



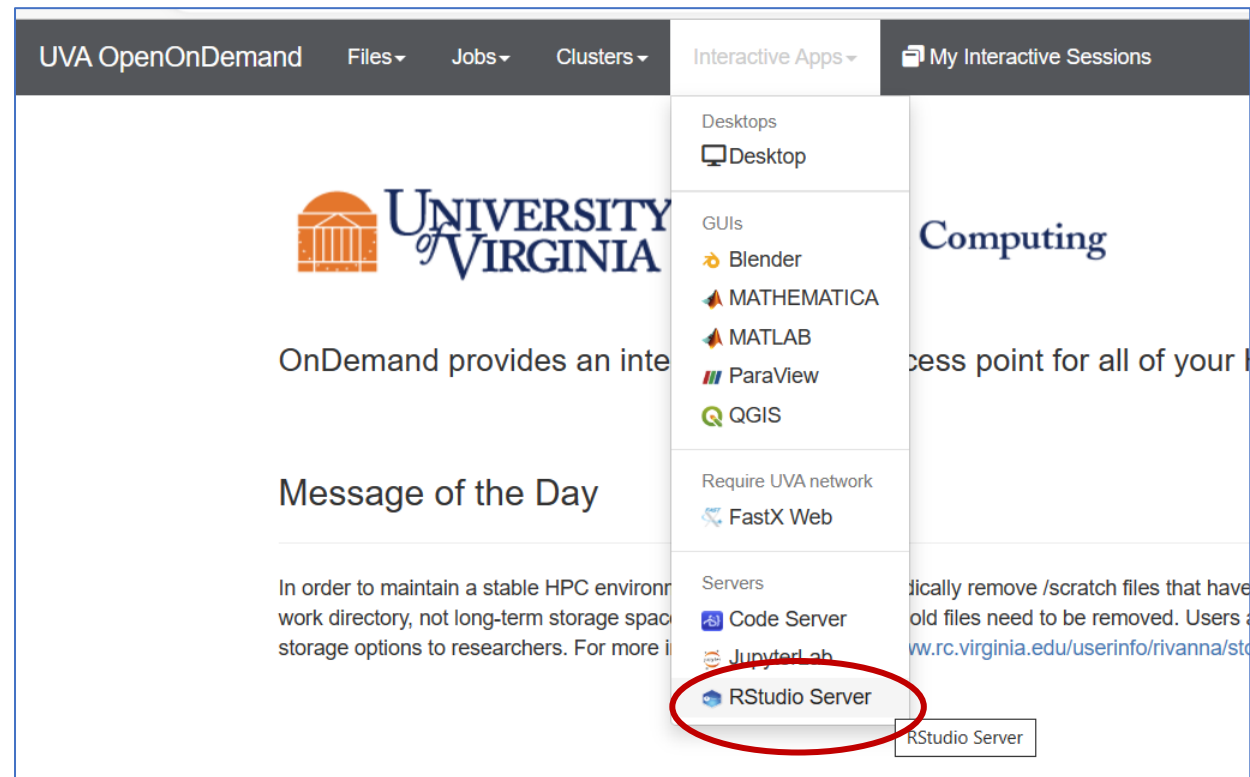
# The Paste Feature

- Click on “Home Directory” in the left panel. Then click on “Paste”.



# Return to the Dashboard

- Click the tab on your browser that is labeled “Dashboard”.
- Click on “Interactive Apps” and “RStudio Server”



# RStudio Server

**RStudio Server**

This app will launch [RStudio Server](#) an IDE for R on the [Rivanna cluster](#).  
**NOTE:** The plot package does not work properly in "RStudio 1.3.1073 - R 4.1.1". This is a known incompatibility with the R version + RStudio version. If your work requires the use of the plot package, please select "RStudio 1.3.1073 - R 4.0.2".

**R version**

RStudio 1.3.1073 - R 4.1.1

**Rivanna Partition**

Standard

- **Standard** - (1-40 cores) Rivanna node in the standard partition.
- **Bii,Bii-gpu** - (1-40 cores) Partition for Biocomplexity Institute and Initiative.
- **GPU** - (1-28 cores) Rivanna node that has NVIDIA GPU.
- **Dev** - (1-8 cores) For short sessions (= 1 hour) with no SU charge; walltime is strictly limited to an hour.
- **Instructional** - (1-20 cores) Rivanna node in the instructional partition.
- [Learn More - Rivanna Queuing Policies](#)

**Number of hours**

2

**Number of cores**

1

**Memory Request in GB ( maximum 384G )**

6

**Allocation (SUs)**

[rivanna-training](#)

**Optional: GPU type for GPU partition**

default

**Optional: Number of GPUs ( 1 ~ 4)**

**Optional: Slurm Option**

**Optional: Group (for access to software or storage)**

**Launch**

- Fill out the requirements for running RStudio on Rivanna and click on Launch:

R version: Rstudio 1.3.1073-R 4.1.1

Rivanna Partition: Standard

Number of Nodes: 1

Number of Hours: 2

Number of Cores: 1

Memory Request in GB: 6

Allocation: rivanna-testing

# And, Wait

- RStudio Server runs on a compute node; so, we need to wait for the resources. A button labeled ‘Connect to RStudio Server’ will pop up when it is ready. Click on the button.

The image displays two screenshots of the RStudio Server interface, illustrating the process of waiting for resources. The left screenshot shows the server in a 'Starting' state, while the right screenshot shows it in a 'Running' state.

**Left Screenshot (Starting State):**

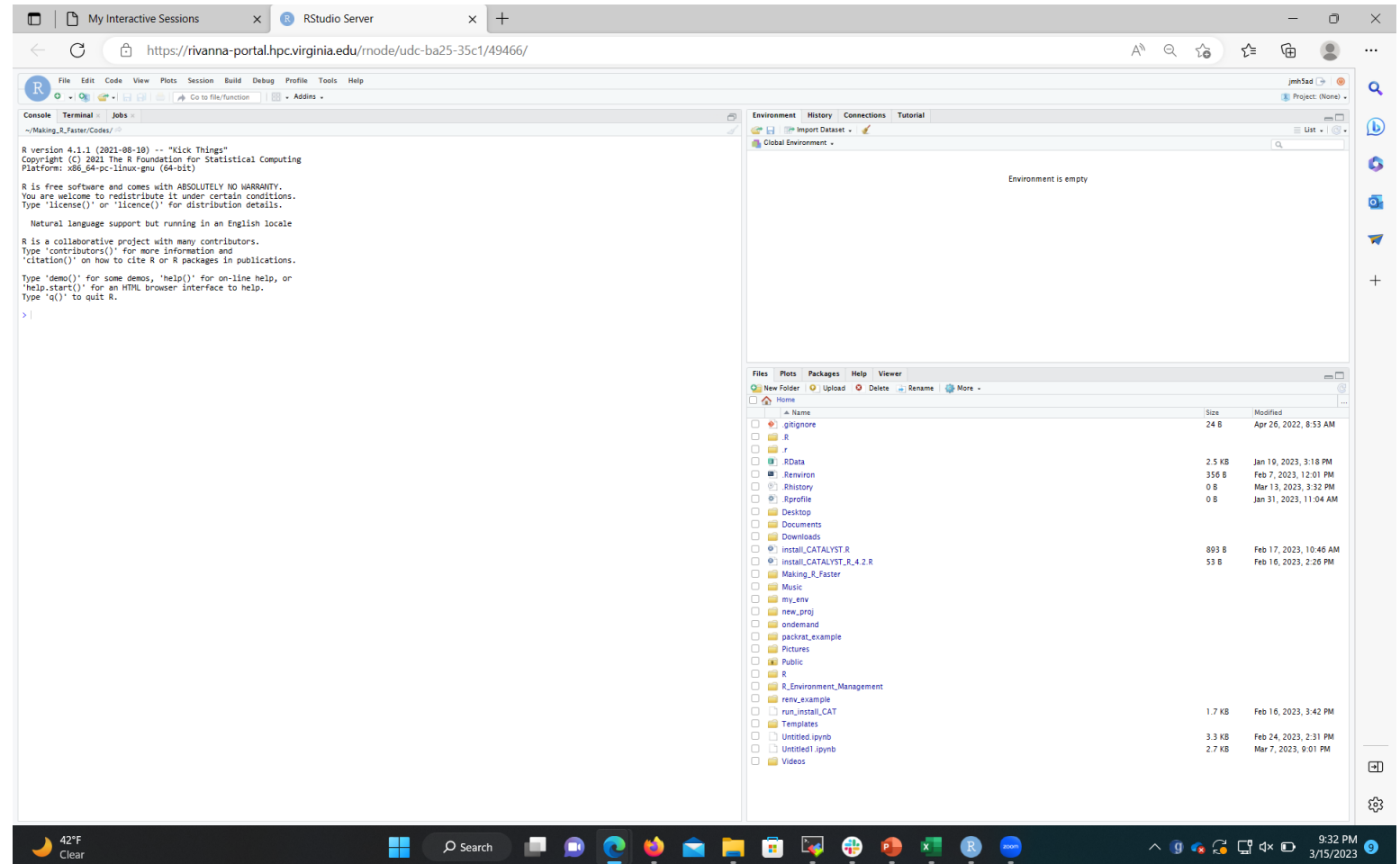
- RStudio Server (2515027)** (1 node | 1 core | Starting)
- Created at:** 2018-10-03 00:02:55 EDT
- Time Remaining:** about 1 hour
- Session ID:** c41a8098-83a9-4e52-84c3-71ce25a62c85
- Message:** Your session is currently starting... Please be patient as this process can take a few minutes.
- Buttons:** Delete

**Right Screenshot (Running State):**

- RStudio Server (2515027)** (1 node | 1 core | Running)
- Host:** udc-ba26-11
- Created at:** 2018-10-03 00:02:55 EDT
- Time Remaining:** about 1 hour
- Session ID:** c41a8098-83a9-4e52-84c3-71ce25a62c85
- Buttons:** Delete, **® Connect to RStudio Server**

# RStudio

- RStudio will appear.
- Now we are ready to work on some code.



# CASE STUDY #1

---

Reading a Large Data File

# The Data

- The data are a collection of information about jobs running on Rivanna
  - The file is pipe-delimited
  - There are 514409 rows with 5 variables (features)

```
"NCPUS"|"CPUTime"|"Start"|"Elapsed"|"ReqTRES"  
1|"00:00:00"|"2023-01-11T21:07:20"|"00:00:00"|"billing=1,cpu=1,gres/gpu=1,mem=30G,node=1"  
6|"33-00:01:00"|"2022-12-26T16:25:46"|"5-12:00:10"|"billing=6,cpu=6,mem=100G,node=1"  
4|"27-15:52:12"|"2022-12-26T16:39:47"|"6-21:58:03"|"billing=4,cpu=4,mem=36G,node=1"  
80|"16-16:04:00"|"2022-12-31T19:36:12"|"05:00:03"|"billing=80,cpu=80,mem=720000M,node=2"  
80|"12-22:13:20"|"2022-12-31T20:24:47"|"03:52:40"|"billing=80,cpu=80,mem=720000M,node=2"  
80|"16-21:05:20"|"2022-12-31T20:47:16"|"05:03:49"|"billing=80,cpu=80,mem=720000M,node=2"
```

# The Task

- Determine the “fastest” way to read in the data

```
"NCPUS"|"CPUTime"|"Start"|"Elapsed"|"ReqTRES"  
1|"00:00:00"|"2023-01-11T21:07:20"|"00:00:00"|"billing=1,cpu=1,gres/gpu=1,mem=30G,node=1"  
6|"33-00:01:00"|"2022-12-26T16:25:46"|"5-12:00:10"|"billing=6,cpu=6,mem=100G,node=1"  
4|"27-15:52:12"|"2022-12-26T16:39:47"|"6-21:58:03"|"billing=4,cpu=4,mem=36G,node=1"  
80|"16-16:04:00"|"2022-12-31T19:36:12"|"05:00:03"|"billing=80,cpu=80,mem=720000M,node=2"  
80|"12-22:13:20"|"2022-12-31T20:24:47"|"03:52:40"|"billing=80,cpu=80,mem=720000M,node=2"  
80|"16-21:05:20"|"2022-12-31T20:47:16"|"05:03:49"|"billing=80,cpu=80,mem=720000M,node=2"
```



# Options for Reading the Data File

- *read.csv* from base packages
- *read\_delim* from readr/tidyverse

# Using read.csv for a pipe-delimited file

```
filename <- "collectedStats.csv"
start_time <- proc.time()

raw_data <- read.csv(filename, sep="|")

elapsedTime <- proc.time() - start_time

cat("\n*****\n")
print(elapsedTime)
cat("\n*****\n")
```

01\_read\_csv.R

Run #1:  
0.899 sec  
Run #2:  
0.917 sec  
Run #3:  
0.906 sec

# Using read\_delim for a pipe-delimited file

```
library(readr)

filename <- "collectedStats.csv"
start_time <- proc.time()

raw_data <- read_delim(filename, delim="|", show_col_types=FALSE)

elapsedTime <- proc.time() - start_time

cat("\n*****\n")
print(elapsedTime)
cat("\n*****\n")
```

02\_read\_delim.R

Run #1:  
0.607 sec  
Run #2:  
0.569 sec  
Run #3:  
0.565 sec

# Times can vary

- How can we determine an overall best approach?

Run #1:  
0.899 sec  
Run #2:  
0.917 sec  
Run #3:  
0.906 sec

Run #1:  
0.607 sec  
Run #2:  
0.569 sec  
Run #3:  
0.565 sec

- Replicate the tests and take an average

# REPEATING TIMING TEST

---

How to compare results

# Times can vary

- How can we determine an overall best approach?

03\_read\_csv\_repeated.R

Average Time across 10 runs:  
0.8616 sec

04\_read\_delim\_repeated.R

Average Time across 10 runs:  
0.5400 sec

- Replicate the tests and take an average of the times.
- The more efficient process is the one that is the fastest on average.

# CASE STUDY #2

---

## Manipulating a Column in a Table

# The Data

- We will use the same data as in the Case #1 study.
- Notice that the column CPUTime is in the format Days-HH:MM:SS

```
"NCPUS"|"CPUTime"|"Start"|"Elapsed"|"ReqTRES"  
1|"00:00:00"|"2023-01-11T21:07:20"|"00:00:00"|"billing=1,cpu=1,gres/gpu=1,mem=30G,node=1"  
6|"33-00:01:00"|"2022-12-26T16:25:46"|"5-12:00:10"|"billing=6,cpu=6,mem=100G,node=1"  
4|"27-15:52:12"|"2022-12-26T16:39:47"|"6-21:58:03"|"billing=4,cpu=4,mem=36G,node=1"  
80|"16-16:04:00"|"2022-12-31T19:36:12"|"05:00:03"|"billing=80,cpu=80,mem=720000M,node=2"  
80|"12-22:13:20"|"2022-12-31T20:24:47"|"03:52:40"|"billing=80,cpu=80,mem=720000M,node=2"  
80|"16-21:05:20"|"2022-12-31T20:47:16"|"05:03:49"|"billing=80,cpu=80,mem=720000M,node=2"
```



# The Task

- We would like to convert it to convert the CPUTime to be the number of hours

```
"NCPUS"|"CPUTime"|"Start"|"Elapsed"|"ReqTRES"  
1|"00:00:00"|"2023-01-11T21:07:20"|"00:00:00"|"billing=1,cpu=1,gres/gpu=1,mem=30G,node=1"  
6|"33-00:01:00"|"2022-12-26T16:25:46"|"5-12:00:10"|"billing=6,cpu=6,mem=100G,node=1"  
4|"27-15:52:12"|"2022-12-26T16:39:47"|"6-21:58:03"|"billing=4,cpu=4,mem=36G,node=1"  
80|"16-16:04:00"|"2022-12-31T19:36:12"|"05:00:03"|"billing=80,cpu=80,mem=720000M,node=2"  
80|"12-22:13:20"|"2022-12-31T20:24:47"|"03:52:40"|"billing=80,cpu=80,mem=720000M,node=2"  
80|"16-21:05:20"|"2022-12-31T20:47:16"|"05:03:49"|"billing=80,cpu=80,mem=720000M,node=2"
```

# Options for Converting the CPUTime to Hours

- Using a for loop
- Using tidyverse manipulations
- Using a multicore technique
  - mclapply
  - furrr

# Basic Code for the Conversion

```
##----- Compute CPU Hours-----##  
## Given a single value, say 5-01:30:10 or 12:15:05  
## Split the string into the numeric parts and convert each part to hours.  
##  
convert_time_to_hrs <- function(cputime){  
  parts <- as.numeric(unlist(strsplit(cputime, "[[:punct:]]")))  
  cpuhours <- 0  
  if (length(parts)==4){  
    cpuhours <- parts[1]*24  
    parts <- parts[-1]  
  }  
  cpuhours <- cpuhours + parts[1] + parts[2]/60 + parts[3]/3600  
  return(cpuhours)  
}
```

To be consistent with all the approaches, I'm creating a function that will convert a single CPUTime to Hours.

# Using a for loop

```
raw_data <- read_delim(filename, delim="|", show_col_types=FALSE)
N <- nrow(raw_data)
CPUHours <- rep(0.0, N)

start_time <- proc.time()
for (i in 1:N) {
  CPUHours[i] <- convert_time_to_hrs(raw_data$CPUTime[i])
}

raw_data$CPUHours <- CPUHours
elapsed_time <- proc.time() - start_time

print(elapsed_time)
print("*****")
print(head(raw_data))
```

05\_for\_loop.R

Run #1:  
5.577 sec  
Run #2:  
5.626 sec  
Run #3:  
5.569 sec

# Using tidyverse manipulations

```
raw_data <- read_delim(filename, delim="|", show_col_types=FALSE)
N <- nrow(raw_data)

start_time <- proc.time()
raw_data <- raw_data %>%
  rowwise() %>%
  mutate(CPUHours=convert_time_to_hrs(CPUTime))
elapsed_time <- proc.time() - start_time
print(elapsed_time)
print("*****")
print(head(raw_data))
```

06\_tidyverse.R

Run #1:  
7.504 sec  
Run #2:  
7.568 sec  
Run #3:  
7.518 sec

# Using multicore lapply

```
raw_data <- read_delim(filename, delim="|", show_col_types=FALSE)
N <- nrow(raw_data)

options(mc.cores=4)

start_time <- proc.time()
CPUHours <- unlist(mclapply(raw_data$CPUTime, convert_time_to_hrs))
raw_data$CPUHrs <- CPUHours
elapsed_time <- proc.time() - start_time
print(elapsed_time)
print("*****")
print(head(raw_data))
```

07\_mclapply.R

Run #1:  
2.081 sec  
Run #2:  
1.978 sec  
Run #3:  
2.310 sec

# Using furr::map

```
raw_data <- read_delim(filename, delim="|", show_col_types=FALSE)
N <- nrow(raw_data)

plan("multisession", workers=4)

start_time <- proc.time()
raw_data$CPUHours <- unlist(future_map(raw_data$CPUTime, convert_time_to_hrs))
elapsed_time <- proc.time() - start_time

print(elapsed_time)
print("*****")
print(head(raw_data))
```

08\_furr.R

Run #1:

2.712 sec

Run #2:

3.097 sec

Run #3:

2.850 sec

# Average Times for Converting the CPUTime to Hours

- Using a for loop

Average Time:  
5.590

- Using tidyverse manipulations

Average Time:  
7.530

- Using a multicore technique

- mclapply

Average Time:  
2.123

- furrr

Average Time:  
2.886



# CASE STUDY #3

---

## Performing Mathematical Operations

# The Data

- For this case, we will use two sets of data: a table of x, y, coordinate values, and an array of electric charges for each location in the table
- The data are saved in .RData files and can be loaded with the load function.

```
head(coords)
  [,1] [,2] [,3]
[1,] 0.566 0.611 0.506
[2,] 0.817 0.183 0.585
[3,] 0.025 0.316 0.061
[4,] 0.977 0.978 0.874
[5,] 0.269 0.092 0.881
[6,] 0.664 0.981 0.962
```

```
head(q)
[1] 0.180 0.422 0.084 0.053 0.563 0.139
```

# The Task

- Compute a value,  $E$ , defined as

$$E = \sum_{j < i} \frac{e^{r_{ij} * q_i} * e^{r_{ij} * q_j}}{r_{ij}} - \frac{1}{a} \quad \text{for } r_{ij} \leq cut$$

where  $r_{ij}$  is the distance between  $(x_i, y_i, z_i)$  and  $(x_j, y_j, z_j)$

# Inefficient Code for the Computations

```
total_e = 0.0
cut_count = 0

load("coords.RData")
load("q_values.RData")
natom <- 5000
cut <- 0.01
a <- 3.2

time1 <- proc.time()
```

```
for (i in 2:natom) {
  for (j in 2:natom) {

    if (j < i) {

      vec2 = (coords[i-1, 1]-coords[j-1, 1])^2.0 +
              (coords[i-1, 2]-coords[j-1, 2])^2.0 +
              (coords[i-1, 3]-coords[j-1, 3])^2.0
      rij = sqrt(vec2);
      if ( rij <= cut ) {
        # Increment the counter of pairs below cutoff
        cut_count = cut_count + 1;
        # Compute the term to be added to E and add
        current_e = (exp(rij*q[i-1])*exp(rij*q[j-1]))/rij;
        total_e = total_e + current_e - 1.0/a;
      } # end rij <= cut
    } # end j < i
  } # end for j
} # end for i
```

09\_slow\_math\_code.R

```
time2 = proc.time(); #time calculation
elapsedTime2 = time2 - time1
cat("Value of system clock after E calc = \n")
```

# Techniques for Reducing the Mathematical Computations

- Do not recompute values that have already been computed, like  $i-1$ .
- Use vectorization whenever possible.
- Reduce the number of inner loops, if possible.
- Simplify the math whenever possible:  
$$x/2 + y/2 + z/2 = (x+y+z)/2$$
$$\exp(ax) * \exp(ay) = \exp(a*(x+y))$$
- Check that you have the right boundaries on your *for* loops – changing the boundaries could reduce the need for computing values like  $i-1$

# Before and After Changes

10\_faster\_math\_code.R

```
for (i in 2:natom) {  
  for (j in 2:natom) {  
  
    if (j < i) {  
  
      vec2 = (coords[i-1, 1]-coords[j-1, 1])^2.0 +  
             (coords[i-1, 2]-coords[j-1, 2])^2.0 +  
             (coords[i-1, 3]-coords[j-1,3])^2.0  
      rij = sqrt(vec2);  
      if ( rij <= cut ) {  
        # Increment the counter of pairs below cutoff  
        cut_count = cut_count + 1;  
        # Compute the term to be added to E and add  
        current_e = (exp(rij*q[i-1])*exp(rij*q[j-1]))/rij;  
        total_e = total_e + current_e - 1.0/a;  
      } # end rij <= cut  
    } # end j < i  
  } # end for j  
} # end for i
```

```
N <- natom = 1  
for (i in 2:N){  
  j <- 1:(i-1)  
  
  vec2 = (coords[i, 1]-coords[j, 1])^2.0 +  
         (coords[i, 2]-coords[j, 2])^2.0 +  
         (coords[i, 3]-coords[j,3])^2  
  ri <- sqrt(vec2)  
  ndx <- which(ri <= cut)  
  num_making_cut <- length(ndx)  
  
  if (num_making_cut > 0){  
    cut_count <- cut_count + 1  
    jcut <- j[ndx]  
    qj <- q[jcut] ;    rij <- ri[jcut]  
    current_e = sum( exp( rij*(q[i]+qj) ) )/rij  
    total_e <- total_e + current_e - a_inv  
  } # end if  
} #end for
```

# CONCLUSIONS

---

# Optimization Strategy

- During optimization, your goal is to minimize the amount of work the computer is required to do. The strategy we recommend is a two-step approach:
  1. Write code that is efficient from the start (e.g., use vectors instead of loops)
  2. After your code is debugged and working, try more aggressive optimization techniques (e.g., manipulating the mathematical formulas to reduce calls to built-in math functions).



# Wallclock Time is Important!

- The only metric that ultimately matters is the Wallclock time.
  - Wallclock is how long you have to wait for your program to run.
  - Wallclock is (sometimes) how much you get charged for.
  - Wallclock is how long your code is blocking other users from using the machine.

# Disadvantages?

## Optimizing code is time consuming

- Do not waste weeks optimizing code that will run once for 1 hour.
- Some optimizations can make the code harder to read and debug.
  - Which is more readable:  
$$y = a_3 * x^3 + a_2 * x^2 + a_1 * x + a_0$$

or

$$y = a_0 + x * (a_1 + x * (a_2 + x * a_3))$$
- Be aware that different architectures can respond in different ways.
  - Just because code is optimized on your laptop does not necessarily mean that it is optimized on your colleague's computer.
- Some optimizations can adversely affect parallel scaling.

# When to optimize?

- Code optimization is an iterative process requiring time, energy and thought. It is recommended for:
  - Codes that will be widely distributed and used often by the research community.
  - Projects that have time limitations, so that you can maximize the available time on the compute resources.

# When optimization isn't enough

- When you have done everything possible to optimize your code, and it still isn't fast enough, you can
  - Find a better algorithm (if one exists).
  - Use parallelizing your code.

# Need more help?

## Office Hours

Tuesdays: 3 pm - 5 pm

Thursdays: 10 am - noon

Zoom links for the office hours are available at  
<https://www.rc.virginia.edu/support/#office-hours>

Lots of information is available on our website:

[rc.virginia.edu](https://rc.virginia.edu)

Or, you submit a request for help through the web for at:

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

# Questions?



# APPENDIX

---

Simple examples for optimizing R code

# General Advice #1

- Do not re-compute values that are used frequently.

Compute once and store in a variable. This is especially critical in loops. Re-computing a single value thousands or millions of times will increase the time it takes for your code to run.

- Before:

```
variance = 0

for (x in values) {
    distToMean = x - mean(values)
    variance = variance +
        (distToMean^2)/length(values)
}
```

- After:

```
variance = 0
meanOfValues = mean(values)
N = length(values)

for (x in values){
    distToMean = x - meanOfValues
    variance = variance + (distToMean^2)/N
}
```



# General Advice #2

- Avoid referencing a single value of an array (e.g., `A[i]` or `M[i,j]` ) multiple times.

If you know that you will be using the value frequently in a block of code, save it to a variable.

- Before:

```
y = 0

for (x in values) {
    y = y + ( sin(M[i, j] * x) * exp(M[i, j]) )/M[i, j]
}
```

- After:

```
y = 0
mij = M[i, j]

for (x in values) {
    y = y + ( sin(mij * x) * exp(mij) )/mij
}
```

# General Advice #3

- Consolidate/minimize mathematical functions.
  - This may require some analytical manipulations. Keep in mind that math intrinsics (e.g., trig functions, exponential/logarithmic function) tend to be slow.

$$\exp\left(\frac{a*x}{L}\right) * \exp\left(\frac{b*x}{2*L}\right) = \exp\left(\frac{(2*a + b)*x}{2*L}\right)$$

4 multiplications  
2 divisions  
2 exponentials

3 multiplications  
1 divisions  
1 addition  
1 exponential

# General Advice #4

- Avoid using loops.
  - Take advantage of vectorizations.

- Before:

```
variance = 0
meanOfValues = mean(values)
N = length(values)

for (x in values){
  diffMean = x - meanOfValues
  variance = variance + (diffMean^2)/N
}
```

After:

```
diffMean = values - mean(values)
variance = sum((diffMean)^2 /length(values))
```

# General Advice #5

- Pre-allocate memory whenever possible.
  - If you know the size and type of the list or array, reserve memory for the entire list/array before doing calculations.

• Before:

```
theValues = c( )  
  
for (xValue in seq(1, N)){  
  theValues = c(theValues, sqrt(xValue))  
}
```

After:

```
theValues = rep(0.0, N)  
i = 1  
  
for (xValue in seq(1, N)){  
  theValues[i] = sqrt(xValue)  
  i = i + 1  
}
```

# General Advice #6

- I/O is slow – try to consolidate it.
  - When processing data and writing results to a file, don't write each result separately. Save and write everything with one command.

- Before:

```
filename = "outputTest1.txt"
outFile = file(filename, "w")

for (xValue in x){
  line = paste("The square root of",
              xValue, "is", sqrt(xValue))
  write(line, outFile)
}
close(outFile)
```

After:

```
lines = rep(" ", numValues)
for (xValue in x){
  lines[i] = paste("The square root of",
                  xValue , "is", sqrt(xValue))
}

filename = "outputTest2.txt"
outFile = file(filename, "w")
write(lines, outFile)
close(outFile)
```

# General Advice #7

- Avoid mathematical manipulations of entire dataframes.
  - If the data is all numeric, convert the dataframe to a matrix. Otherwise, pull out only the needed columns.

• Before:

```
ndx = which(theData$temps > 85.3)
lastYear = max(theData$years)
numHotDays =
    sum(theData$years[ndx]==lastYear)
```

After:

```
theData = as.matrix(theData)

ndx = which(theData[, "temps"] > 85.3)
lastYear = max(theData[ , "years"])
numHotDays =
    sum(theData[ndx , "years"]==lastYear)
```

# General Advice #8

- Avoid overuse of parentheses.
  - The contents inside parentheses are evaluated and stored in a special list structure. There is overhead for allocating the list, storing, and retrieving the results.

• Before:

$$y = ( 1 + (x) ) / ( (1) + (x)^2 )$$

After:

$$y = ( 1 + x ) / ( 1 + x^2 )$$

# Need more help?

## Office Hours

Tuesdays: 3 pm - 5 pm

Thursdays: 10 am - noon

Website:

[rc.virginia.edu](https://rc.virginia.edu)

Or, submit a request for help through the web for at:

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



# Questions?

