



Parallel Computing in R

UCF Office of Research Cyberinfrastructure – November 12, 2025

Satyar Foroughi
PhD Student Big Data Analytics – GRA UCF Research IT

Office of Research Cyberinfrastructure's Mission

Our Mission:

Enable researchers through the effective use of research technology

Improve Research Computing and Data (RCD) Capabilities

Improve Technology Support for Researchers

CONTENTS

- 1** What is Parallel Processing?
- 2** When to Parallelize
- 3** High Performance Computing
- 4** HPC in R
- 5** Purdue Anvil
- 6** Parallel Computing Demonstrations
- 7** Command Line Interface and SLURM Scheduling
- 8** Resources and Contact Information

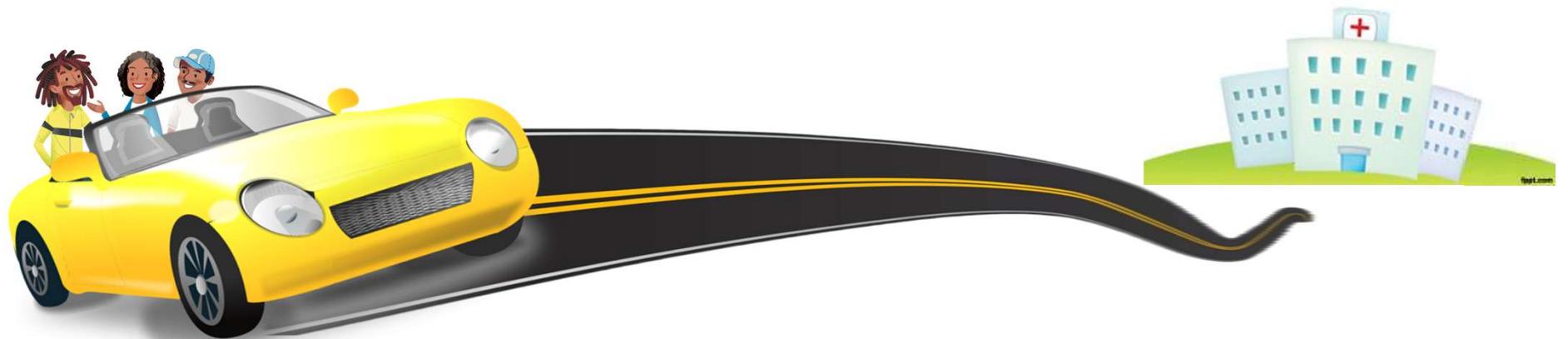
What is Parallel Processing?

Imagine there are **3 people**, we need to **urgently take them somewhere**.



What is Parallel Processing?

We can simply use a **car**.



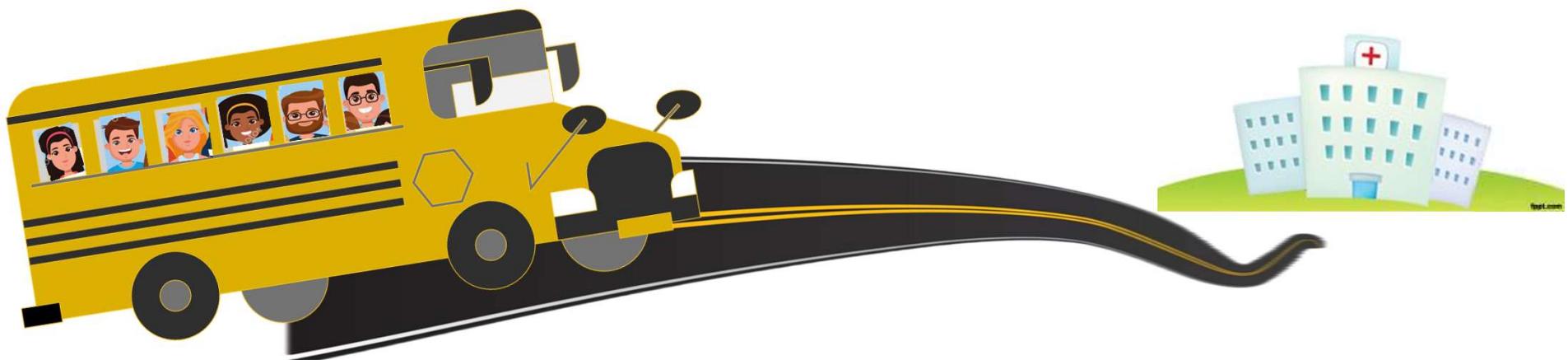
What is Parallel Processing?

Imagine there are **10 people**, we need to **urgently take them somewhere**.



What is Parallel Processing?

We can use a **bus** this time.



What is Parallel Processing?

Now imagine we have **200** people, what is the **fastest way**?



What is Parallel Processing?

We can use a **multiple buses**, all working **at the same time**.



When to parallelize

- Our **goal** when doing parallel processing **reduce time-to-solution** by decomposing the work into **chunks that run simultaneously** on multiple cores/GPUs/nodes.

- Small Job



(small dataset, basic models, etc.)

Can be run on a **basic laptop CPU** using a **single core**



- Medium Job



(moderate dataset, heavier models, etc.)

Can be done on a **powerful laptop** with **multi-core parallelization** on a single node.



- Heavy Job



(very large dataset, LLM fine-tuning, etc.)

We need **super-computers** with powerful CPUs, utilizing **many cores / multiple CPUs in parallel**.



High Performance Computing (HPC)

- Parallel computing is the *technique* (splitting work to run at the same time). **HPC** is the *ecosystem* that uses those techniques at **large scale**.
- Use of **clustered CPUs/GPUs**, fast interconnects, and parallel software to run massive computations simultaneously.
- At UCF, **ARCC** houses two HPC clusters **Stokes (CPU)** and **Newton (GPU)**.
- **NSF ACCESS** and its resource providers deliver this HPC ecosystem to researchers, offering national-scale **clusters and supercomputers**.



High-Performance Computing systems inside the Data Center at Purdue University. These HPC systems are maintained and operated by the Rosen Center for Advanced Computing (RCAC).

HPC in R

Benefits

- R is suited for **Classical statistics workflows**, which benefit greatly from CPU parallelization—often simpler to parallelize than deep learning.
- **Scales** from a laptop to clusters with the same workflow and **minimal code changes**.
- Predictable speedups for independent tasks; easy to schedule lots of small jobs.

Limitations

- **GPU** in R is **niche**; most acceleration is CPU/threading, not ideal for deep learning tasks.
- There aren't many **direct parallel implementations** of specific statistical methods built into base R or CRAN packages.
- **Some** functions/packages just **won't run in parallel**.

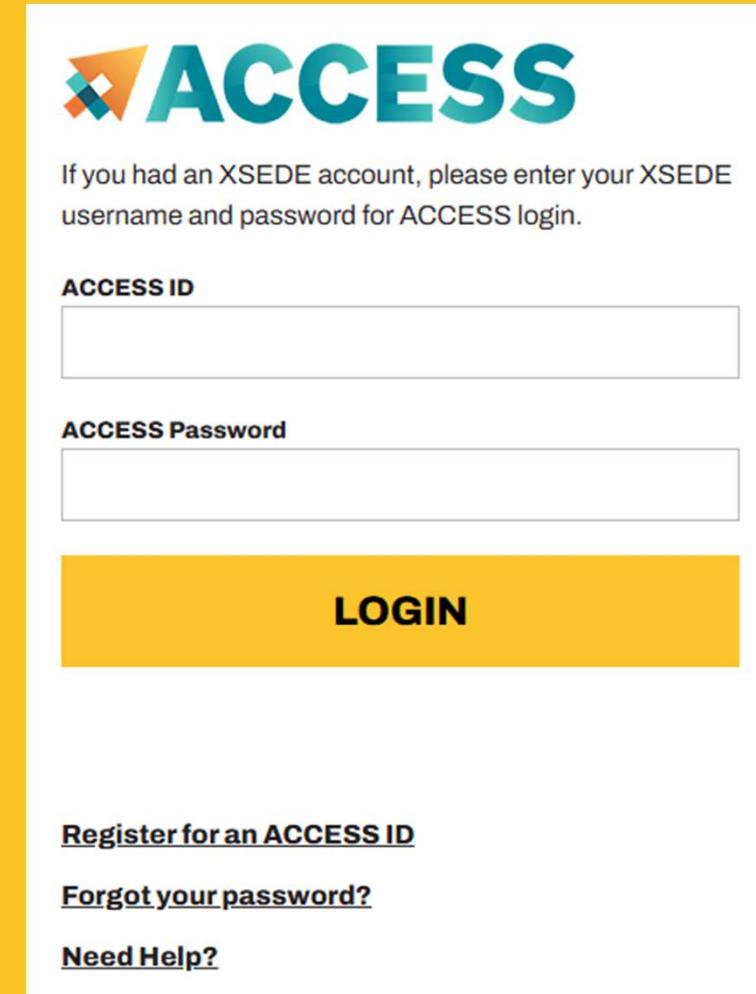
Accessing Purdue Anvil

Step 1

<https://ondemand.anvil.rcac.purdue.edu/>

If not logged in already, log into ACCESS

(Make sure you have Duo authentication)



If you had an XSEDE account, please enter your XSEDE username and password for ACCESS login.

ACCESS ID

ACCESS Password

LOGIN

[Register for an ACCESS ID](#)

[Forgot your password?](#)

[Need Help?](#)

Step 2

The screenshot shows the 'Dashboard - Anvil' page with the URL 'ondemand.anvil.rcac.purdue.edu/pun/sys/dashboard'. The left sidebar displays 'System Status' for various clusters: wholenode, shared, highmem, debug, gpu, and ai. Each cluster entry includes the number of cores in use and available, and a utilization bar chart. The right side features a dropdown menu titled 'Interactive Apps' which lists categories like Bioinformatics Apps, Cryo-EM Apps, Desktops, GUIs, Molecular Simulation, Servers, and nf-core pipelines, each with a corresponding icon.

Step 3

Purdue Anvil Open OnDemand (OOD)

OOD

Open OnDemand is an open-source web portal from the Ohio Supercomputing Center that lets users **access HPC resources through a browser** to manage files, submit jobs, and run graphical applications without installing software.

Interactive Application Menu

Dropdown lists all the **applications supported on Anvil OOD**. Clicking one of the applications will take you to a form through which you can **request the resources** you want to use.

RStudio Server

This app will launch an RStudio server on a node.

Account (Allocation)

cis240473 (8745.1 SUs remaining)

Queue (partition)

shared

- GPU-only allocations MUST use the 'gpu' queue
- CPU-only allocations MAY NOT use the 'gpu' queue

R Version

4.1.0

Wall Time in Hours

0.5

Number of hours you are requesting for your job.

Cores

1

Number of cores (up to 128) for a shared job. Non-shared jobs will have exclusive nodes and be charged at 128 cores per node requested

I would like to receive an email when the session starts

Launch

Interactive RStudio Session

- **Account (Allocation):**

The project or allocation that pays for your compute time (shows remaining Service Units).

- **Queue (Partition):**

shared – Runs on a shared CPU node

wholenode – Gives you the entire node

highmem – CPU nodes with **extra RAM** per node.

gpu – Nodes with GPUs

- **Wall Time in Hours:**

The maximum runtime for your session before it's automatically stopped.

- **Cores:**

Number of CPU cores to request (shared or exclusive use depending on queue).



Hands-On Practice (OOD)

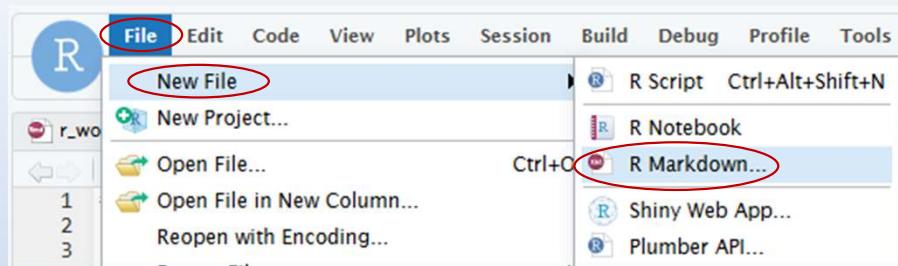
Connect to RStudio once session is ready

The screenshot shows a user interface for managing interactive sessions. At the top, there is a header with a folder icon and the text "My Interactive Sessions". Below this, there are several session cards. The first card is partially visible. The second card, which is highlighted with a green background, contains the following information:

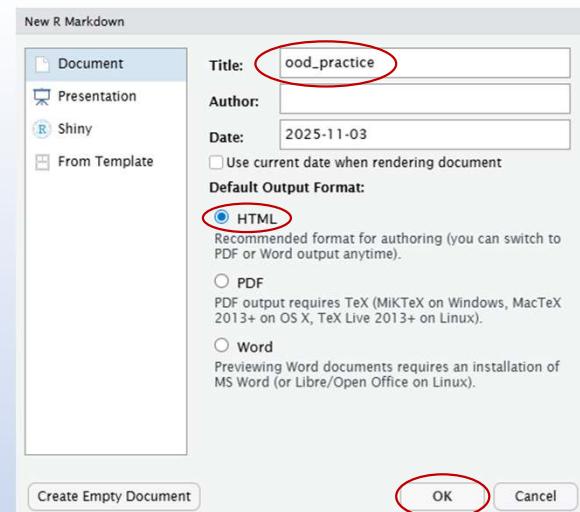
- RStudio Server (14354418)**
- Host:** >_a240.anvil.rcae.purdue.edu
- Created at:** 2025-11-06 09:07:01 EST
- Time Remaining:** 28 minutes
- Session ID:** 0efa7ee7-0820-4ac1-a63d-7b1e0dbaf2b2

Below the session details is a dark button with white text that reads "How to import Imod (environment) modules within RStudio". At the bottom of the card is a blue button with white text that reads "Connect to RStudio Server". This blue button is circled with a red oval.

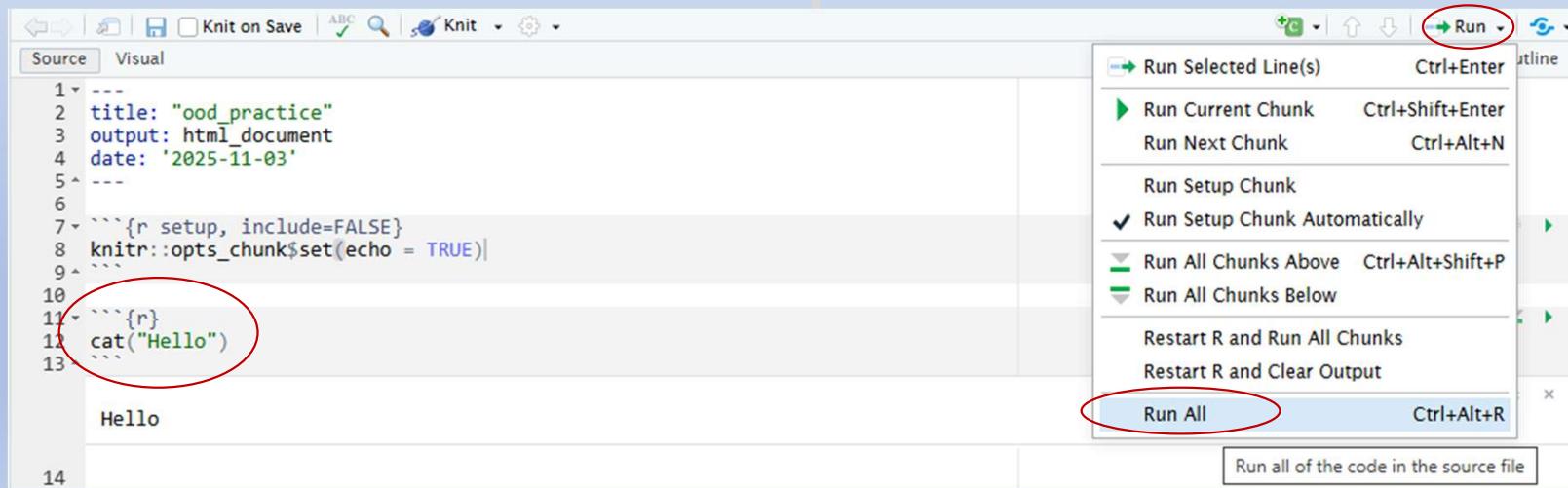
Step 1



Step 2



Step 3



Delete once done to avoid using resources

RStudio Server (14383514) 1 node | 8 cores | Running

Host: >[_a241.anvil.rcac.purdue.edu](#)

Created at: 2025-11-10 12:36:29 EST

Time Remaining: 50 minutes

Session ID: 80028504-d6e9-474d-8e0e-752b46f05ea8

How to import Imod (environment) modules within RStudio

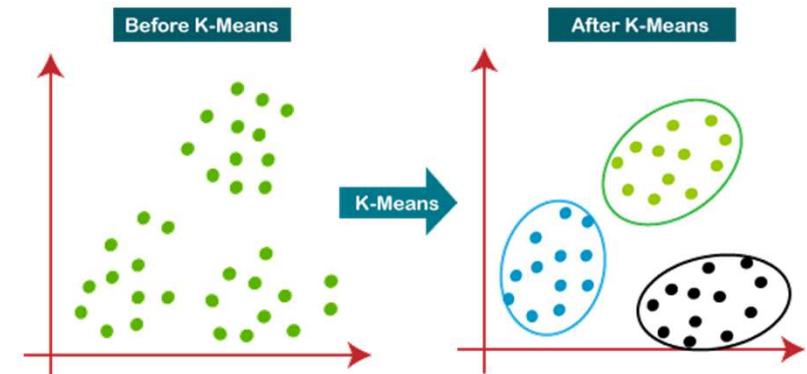
[@ Connect to RStudio Server](#)

Delete
Delete Session

Parallel K-means

K-Means Algorithm

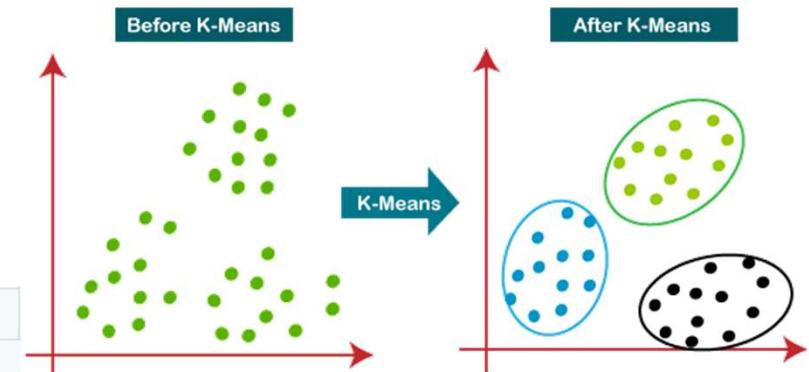
- K-means groups data into **K** clusters by similarity; you choose **K**, and the algorithm finds **centroids** (average points) for each group.
- It starts with **K** initial centroids and assigns every point to its **nearest** centroid.
- Then it **recomputes** each centroid as the mean of its assigned points and **repeats** assign → update until things stop changing much.



Task

- We have a synthetic dataset of ~ 500K patients
- We want to use the K-means Algorithm (in parallel) to group them based on their demographics
- Demographics are:

Race	String	true	Description of the patient's primary race.
Gender	String	true	Gender. M is male, F is female.
State	String	true	Patient's address state.
Healthcare_Expenses	Numeric	true	The total lifetime cost of healthcare to the patient (i.e. what the patient paid).
Healthcare_Coverage	Numeric	true	The total lifetime cost of healthcare services that were covered by Payers (i.e. what the insurance company paid).
Income	Numeric	true	Annual income for the Patient



```
library(parallel)
library(tictoc)
library(dplyr)
library(ggplot2)
library(fastDummies)
```

Libraries

- **parallel**
 - Built-in R package for multicore processing.
 - Provides functions to distribute work across CPU cores.
 - Ideal for running independent tasks simultaneously.
- **Tictoc:**
 - For timing
- **Dplyr, ggplot2:**
 - For visualizations.
- **fastDummies:**
 - To turn categorical variables into numeric dummy variables.

Parallel K-means Function

Goal: run many K-means initializations **in parallel** and keep the best solution.
Inputs: `data` (numeric matrix), `centers` (k), `n_cores` (workers), `n_starts` (restarts)
Output: a standard kmeans object (best run), identical to base R's `kmeans()`

- K-means is sensitive to initialization \Rightarrow we try `n_starts` independent runs.
- If `n_cores` is 1, we run basic `kmeans()`
- **`makeCluster()`:**
 - Launches `n_cores` independent R worker processes
 - Each worker starts with a clean GlobalEnv
- **`clusterExport()`:**
 - Copies objects from function environment to each worker's GlobalEnv
- **`parLapply()`:**
 - Distributes `n_starts` tasks to workers

```
# Function to run K-means with specified number of cores
parallel_kmeans <- function(data, centers = 10, n_cores = 1, n_starts = 25) {

  if (n_cores == 1) {
    # Sequential execution
    tryCatch({
      result <- kmeans(data, centers = centers, nstart = n_starts, iter.max = 100, algorithm = "Lloyd")
    }, error = function(e) {
      cat("Error in kmeans:", e$message, "\n")
      cat("Trying with fewer centers...\n")
      result <- kmeans(data, centers = min(5, centers), nstart = n_starts, iter.max = 100, algorithm = "Lloyd")
    })
  } else {
    # Parallel execution
    cl <- makeCluster(n_cores)

    # Export data to cluster
    clusterExport(cl, varlist = c("data", "centers"), envir = environment())

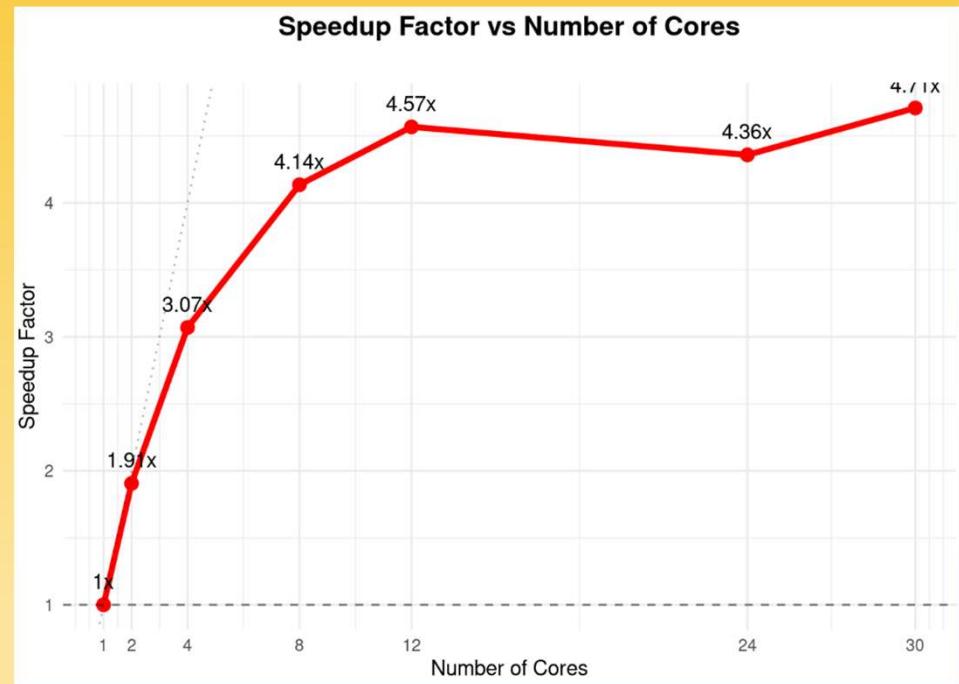
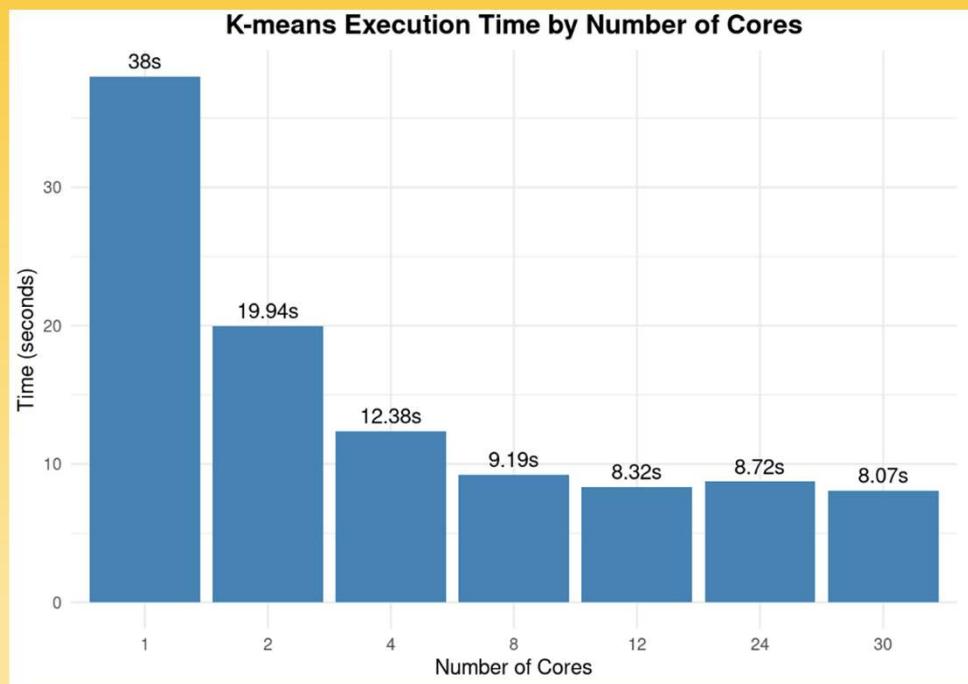
    # Run multiple K-means in parallel with error handling
    results <- parLapply(cl, 1:n_starts, function(i) {
      tryCatch({
        kmeans(data, centers = centers, nstart = 1, iter.max = 100, algorithm = "Lloyd")
      }, error = function(e) {
        # If error, try with fewer centers
        kmeans(data, centers = min(5, centers), nstart = 1, iter.max = 100, algorithm = "Lloyd")
      })
    })

    stopCluster(cl)

    # Select best result (lowest total within-cluster sum of squares)
    best_idx <- which.min(sapply(results, function(x) x$tot.withinss))
    result <- results[[best_idx]]
  }

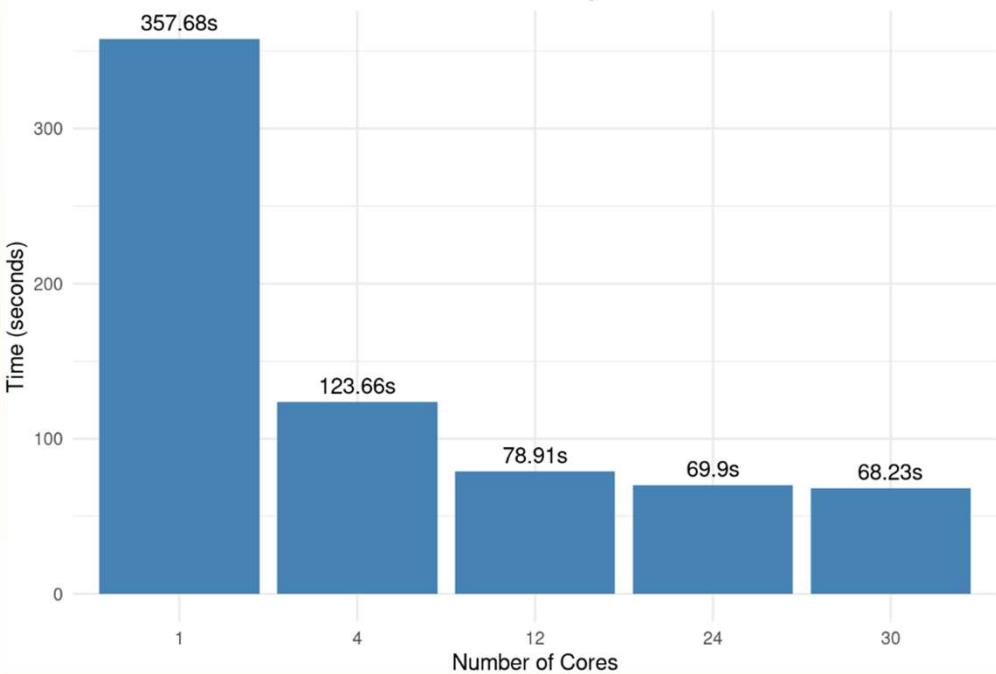
  return(result)
}
```

Performance

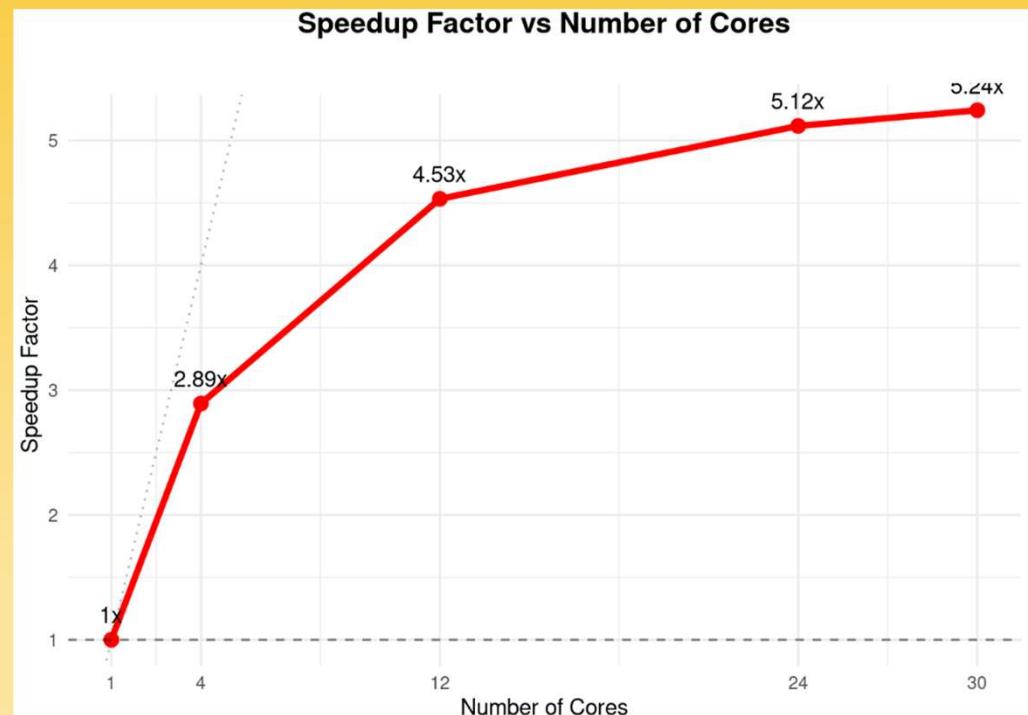


Performance (on 10x larger dataset)

K-means Execution Time by Number of Cores



Speedup Factor vs Number of Cores





Hands-On Practice (Parallel K-means)

First, we get access to the files needed for the rest of the workshop

Step 1

The screenshot shows the Anvil web interface. At the top, there is a navigation bar with buttons for 'Anvil', 'Files', 'Jobs', 'Clusters' (which is highlighted with a red circle), 'Interactive Apps', and 'My Interactive Sessions'. Below the navigation bar, there is a section titled 'Announcements' with a 'View' button. A second red circle highlights the 'Anvil Shell Access' button, which is located just below the announcements section.

Step 2

Type command `git clone https://github.com/satyarforoughi/workshop`

```
(base) x-sforoughi@login05.anvil:[~] $ git clone https://github.com/satyarforoughi/workshop
```

Checking

You should now see the folder "workshop" in Home Directory

The screenshot shows the Anvil interface with the 'Files' button highlighted with a red circle in the top navigation bar. Below the navigation bar, there is a section titled 'Announcements' with a 'Home Directory' button highlighted with a red circle. The main area is a file browser showing the contents of the user's home directory. A folder named 'workshop' is visible and highlighted with a red circle.

File/Folder	Description
logs	Logs directory
ondemand	On-demand files
privatemodules	Private modules
R	R directory
testing_git	Testing Git repository
Untitled Folder	Untitled folder
workshop	Workshop folder (highlighted)
.bash_history	Bash history file
.bash_logout	Bash logout file

RStudio Server

This app will launch an RStudio server on a node.

Account (Allocation)

cis240473 (8740.7 SUs remaining)

Queue (partition)

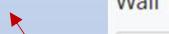
shared

- GPU-only allocations MUST use the 'gpu' queue
- CPU-only allocations MAY NOT use the 'gpu' queue

R Version

4.1.0

Request multiple cores



Wall Time in Hours

0.5

Number of hours you are requesting for your job.

Cores

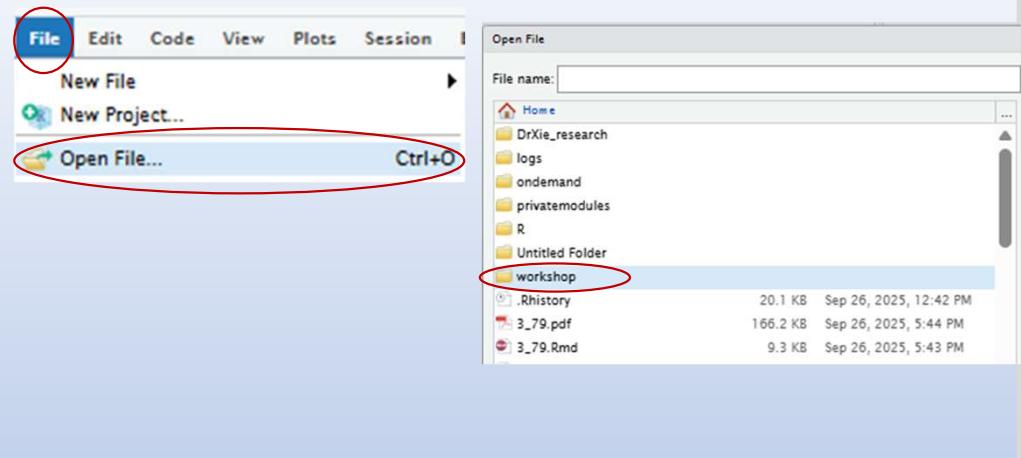
5

Number of cores (up to 128) for a shared job. Non-shared jobs will have exclusive nodes and be charged at 128 cores per node requested

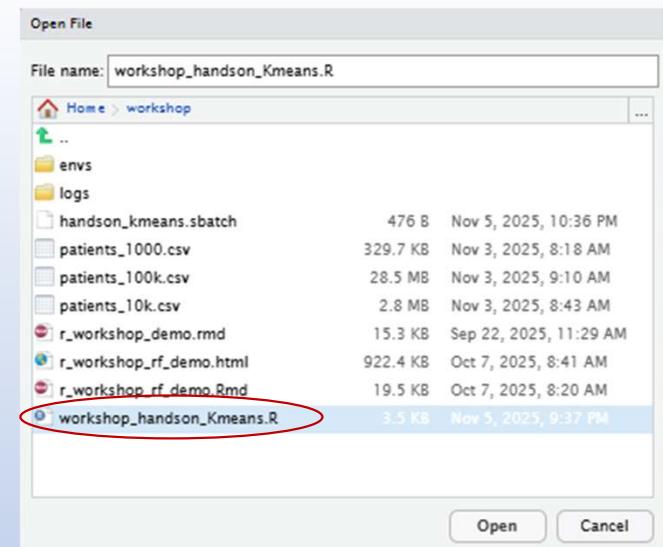
I would like to receive an email when the session starts

Launch

Step 1

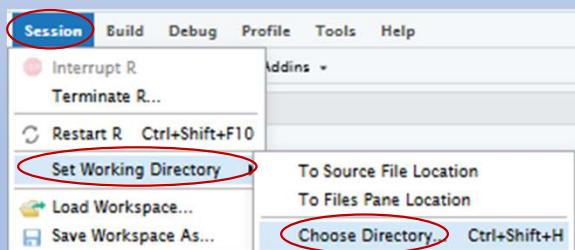


Step 2

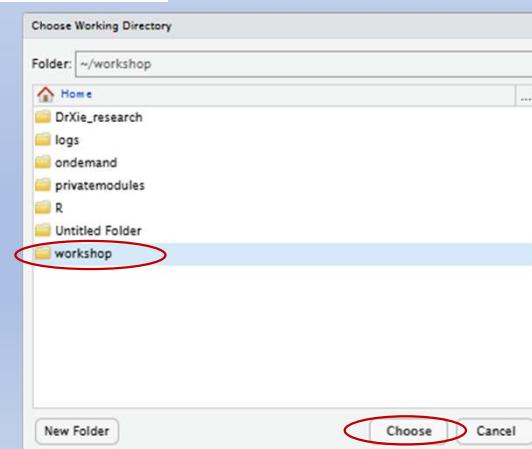


Step 3

Set your session directory to the workshop folder



Step 4



Step 5

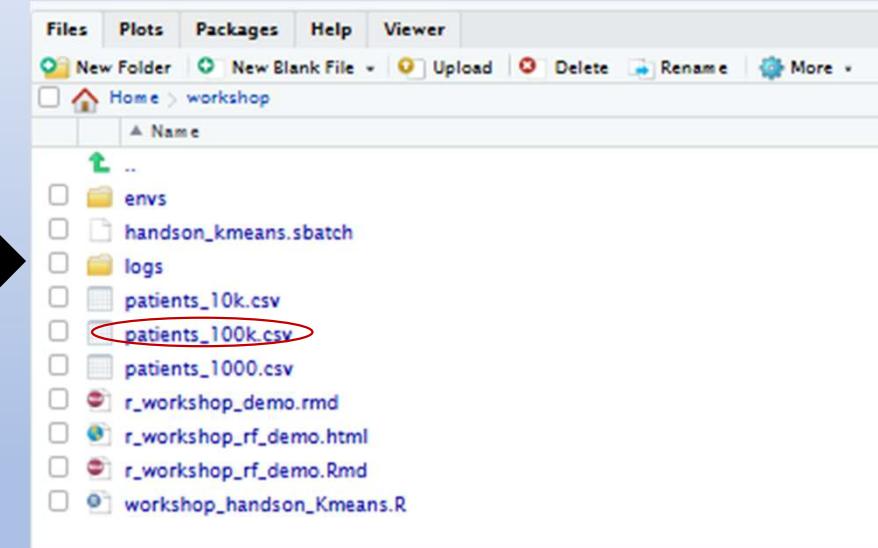
The screenshot shows the RStudio interface. On the left is the script editor with the following R code:

```
61 # Export data to cluster
62 clusterExport(cl, varlist = c("data", "centers"), envir = environment())
63 
64 # Run multiple K-means in parallel with error handling
65 results <- parLapply(cl, 1:n_starts, function(i) {
66   tryCatch({
67     kmeans(data, centers = centers, nstart = 1, iter.max = 100, algorithm = "Lloyd")
68   }, error = function(e) {
69     if (error == "Error in kmeans") {
70       kmeans(data, centers = min(5, centers), nstart = 1, iter.max = 100, algorithm = "Lloyd")
71     }
72   })
73 })
74 stopCluster(cl)
75 
76 # Select best result (lowest total within-cluster sum of squares)
77 best_idx <- which.min(sapply(results, function(x) x$tot.withinss))
78 result <- results[[best_idx]]
79 
80 return(result)
81 }
82 
83 tic("K-means with 1 core")
84 km_1core <- parallel_kmeans(cluster_data_scaled, centers = 5, n_cores = 1)
85 time_1core <- toc()
86 cat("Number of iterations: ", km_1core$iter, "\n")
87 time_1core_val <- time_1core$toc - time_1core$tic
88 
89 tic("K-means with 4 cores")
90 km_4cores <- parallel_kmeans(cluster_data_scaled, centers = 5, n_cores = 4)
91 time_4cores <- toc()
92 time_4cores_val <- time_4cores$toc - time_4cores$tic
93 speedup_4 <- time_1core_val / time_4cores_val
94 cat("\nSpeedup compared to 1 core!:", round(speedup_4, 2), "x\n")
95 cat("Number of iterations: ", km_4cores$iter, "\n")
96 
97 patients_10k.csv
98 patients_100k.csv
99 patients_1000.csv
100 r_workshop_demo.Rmd
101 r_workshop_rf_demo.html
102 r_workshop_rf_demo.Rmd
103 workshop_hanson_Kmeans.R
```

On the right is the file browser showing a folder structure under 'workshop':

- ..
- envs
- handson_kmeans.sbatch
- logs
- patients_10k.csv
- patients_100k.csv (highlighted with a red circle)
- patients_1000.csv
- r_workshop_demo.Rmd
- r_workshop_rf_demo.html
- r_workshop_rf_demo.Rmd
- workshop_hanson_Kmeans.R

Make sure you see *patients_100k.csv* in files



Ctrl + Shift + Enter (Windows/Linux)
or **Cmd + Shift + Enter** (Mac)



To run full script

Using our parallel function and timing we compare using
a single core vs 4 cores in parallel.

```
tic("K-means with 1 core")
km_1core <- parallel_kmeans(cluster_data_scaled, centers = 5, n_cores = 1)
time_1core <- toc()
cat("Number of iterations:", km_1core$iter, "\n")
time_1core_val <- time_1core$toc - time_1core$tic

tic("K-means with 4 cores")
km_4cores <- parallel_kmeans(cluster_data_scaled, centers = 5, n_cores = 4)
time_4cores <- toc()
time_4cores_val <- time_4cores$toc - time_4cores$tic

speedup_4 <- time_1core_val / time_4cores_val
cat("\nSpeedup compared to 1 core:", round(speedup_4, 2), "x\n")
cat("Number of iterations:", km_4cores$iter, "\n")
```

Output

```
Console Terminal Background Jobs <
R 4.1.0 . ~/workshop/ >
> km_1core <- parallel_kmeans(cluster_data_scaled, centers = 5, n_cores = 1)
> time_1core <- toc()
K-means with 1 core: 11.198 sec elapsed
> cat("Number of iterations:", km_1core$iter, "\n")
Number of iterations: 17

> time_1core_val <- time_1core$toc - time_1core$tic
> tic("K-means with 4 cores")
> km_4cores <- parallel_kmeans(cluster_data_scaled, centers = 5, n_cores = 4)
> time_4cores <- toc()
K-means with 4 cores: 4.448 sec elapsed
> time_4cores_val <- time_4cores$toc - time_4cores$tic
> speedup_4 <- time_1core_val / time_4cores_val
> cat("\nSpeedup compared to 1 core:", round(speedup_4, 2), "x\n")
Speedup compared to 1 core: 2.52 x
> cat("Number of iterations:", km_4cores$iter, "\n")
Number of iterations: 11
> |
```

Example 2

Changing parameters could also influence runtime.

```
tic("K-means with 1 core")
km_1core <- parallel_kmeans(cluster_data_scaled, centers = 3, n_cores = 1)
time_1core <- toc()
cat("Number of iterations:", km_1core$iter, "\n")
time_1core_val <- time_1core$toc - time_1core$tic

tic("K-means with 4 cores")
km_4cores <- parallel_kmeans(cluster_data_scaled, centers = 3, n_cores = 4)
time_4cores <- toc()
time_4cores_val <- time_4cores$toc - time_4cores$tic

speedup_4 <- time_1core_val / time_4cores_val
cat("\nSpeedup compared to 1 core:", round(speedup_4, 2), "x\n")
cat("Number of iterations:", km_4cores$iter, "\n")
```

Changing number of centers

```
> tic("K-means with 1 core")
> km_1core <- parallel_kmeans(cluster_data_scaled, centers = 3, n_cores = 1)
> time_1core <- toc()
K-means with 1 core: 5.93 sec elapsed
> cat("Number of iterations:", km_1core$iter, "\n")
Number of iterations: 30
> time_1core_val <- time_1core$toc - time_1core$tic
>
>
> tic("K-means with 4 cores")
> km_4cores <- parallel_kmeans(cluster_data_scaled, centers = 3, n_cores = 4)
> time_4cores <- toc()
K-means with 4 cores: 3.035 sec elapsed
> time_4cores_val <- time_4cores$toc - time_4cores$tic
>
> speedup_4 <- time_1core_val / time_4cores_val
> cat("\nSpeedup compared to 1 core:", round(speedup_4, 2), "x\n")
Speedup compared to 1 core: 1.95 x
> cat("Number of iterations:", km_4cores$iter, "\n")
Number of iterations: 22
```

Example 3

What happens if we try to use 8 cores?

```
### 5 vs 8 ###
tic("K-means with 5 cores")
km_5core <- parallel_kmeans(cluster_data_scaled, centers = 5, n_cores = 5)
time_5core <- toc()
cat("Number of iterations:", km_5core$iter, "\n")
time_5core_val <- time_5core$toc - time_5core$tic

tic("K-means with 8 cores")
km_8cores <- parallel_kmeans(cluster_data_scaled, centers = 5, n_cores = 8)
time_8cores <- toc()
time_8cores_val <- time_8cores$toc - time_8cores$tic

speedup_8 <- time_5core_val / time_8cores_val
cat("\nSpeedup compared to 5 cores:", round(speedup_8, 2), "x\n")
cat("Number of iterations:", km_8cores$iter, "\n")
```

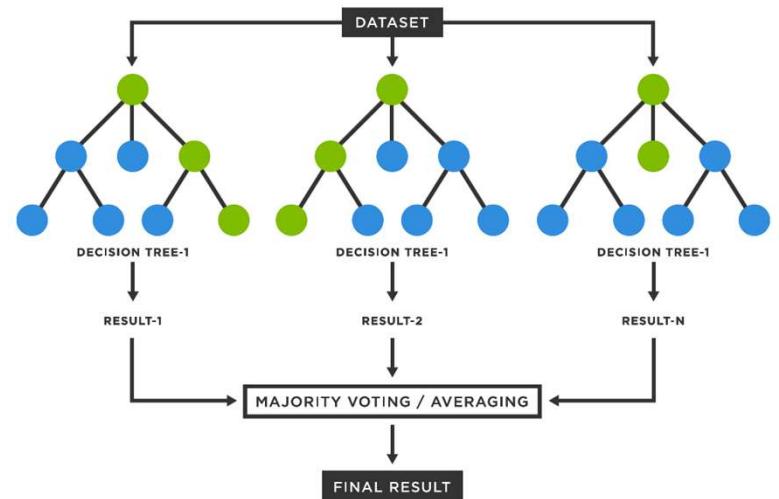
5 vs 8 cores

```
> tic("K-means with 5 cores")
> km_5core <- parallel_kmeans(cluster_data_scaled, centers = 5, n_cores = 5)
> time_5core <- toc()
K-means with 5 cores: 4.248 sec elapsed
> cat("Number of iterations:", km_5core$iter, "\n")
Number of iterations: 21
> time_5core_val <- time_5core$toc - time_5core$tic
>
>
> tic("K-means with 8 cores")
> km_8cores <- parallel_kmeans(cluster_data_scaled, centers = 5, n_cores = 8)
> time_8cores <- toc()
K-means with 8 cores: 4.565 sec elapsed
> time_8cores_val <- time_8cores$toc - time_8cores$tic
>
> speedup_8 <- time_5core_val / time_8cores_val
> cat("\nSpeedup compared to 5 cores:", round(speedup_8, 2), "x\n")
Speedup compared to 5 cores: 0.93 x
> cat("Number of iterations:", km_8cores$iter, "\n")
Number of iterations: 37
```

Parallel Random Forest

Random Forest Algorithm

- A Random Forest is like asking **many decision trees** the same question and then taking a **majority vote** (for categories) or an **average** (for numbers).
- Each tree learns from a **slightly different slice** of the data and looks at **random subsets of features**, so they don't all make the same mistakes.



Task

- We have a synthetic dataset of ~ 500K patients
- We want to use a Random Forest model (in parallel) to predict their total hospital encounters from 2020 – 2025 based on their attributes (until 2020).

- Attributes are:

Race	String	true	Description of the patient's primary race.
Gender	String	true	Gender. M is male, F is female.
State	String	true	Patient's address state.
Healthcare_Expenses	Numeric	true	The total lifetime cost of healthcare to the patient (i.e. what the patient paid).
Healthcare_Coverage	Numeric	true	The total lifetime cost of healthcare services that were covered by Payers (i.e. what the insurance company paid).
Income	Numeric	true	Annual income for the Patient
Total Conditions	Numeric	true	Total diagnosed conditions before 2020.

```
library(parallel)
library(randomForest)
library(ranger)
library(tictoc)
library(dplyr)
library(ggplot2)
```

Libraries

- **randomForest**
 - Classic R implementation of the random forest algorithm.
- **ranger:**
 - Pre-built library for running RF in parallel.
- **Parallel, tictoc, Dplyr, ggplot2:**
 - Same as before.

“Manual” Parallel RF Function

Goal: build a Random Forest by **splitting trees across cores** and then **combining** the sub-forests.

Inputs: X (feature matrix), y (target), n_cores (workers), ntree (total trees), classification (TRUE/FALSE)

Output: a standard **randomForest** model (combined from per-core sub-forests).

clusterEvalQ():

- Executes given expression on each worker; here, library(randomForest).

clusterExport():

- Copies X, y, trees_per_core, classification to each worker's GlobalEnv so the worker function can see them.

do.call(randomForest::combine()):

- Merges sub-forests into one final Random Forest model.

makeCluster(), parLapply():

- Same as before.

```
# Function to run Random Forest with specified number of cores using randomForest package
parallel_rf_manual <- function(X, y, n_cores = 1, ntree = 500, classification = FALSE) {

  if (n_cores == 1) {
    # Sequential execution
    if (classification) {
      model <- randomForest(x = X, y = y, ntree = ntree, importance = TRUE)
    } else {
      model <- randomForest(x = X, y = y, ntree = ntree, importance = TRUE)
    }
  } else {
    # Parallel execution - split trees across cores
    trees_per_core <- ceiling(ntree / n_cores)

    # Create cluster
    cl <- makeCluster(n_cores)

    # Export necessary objects and Load Library on each worker
    clusterEvalQ(cl, library(randomForest))
    clusterExport(cl, varlist = c("X", "y", "trees_per_core", "classification"),
                 envir = environment())
  }

  # Train Random Forest models in parallel
  rf_list <- parLapply(cl, 1:n_cores, function(i) {
    set.seed(i * 1000) # Different seed for each worker
    if (classification) {
      randomForest(x = X, y = y, ntree = trees_per_core, importance = FALSE)
    } else {
      randomForest(x = X, y = y, ntree = trees_per_core, importance = FALSE)
    }
  })

  stopCluster(cl)

  # Combine forests using randomForest's combine function
  # Explicitly use randomForest::combine to avoid conflicts with dplyr
  model <- do.call(randomForest::combine, rf_list)
}

return(model)
}
```

Using ranger

- The ranger library already has an implementation of random forest for parallel computing.
- We just put our data into the correct format
- Choose whether we want regression or classification

```
# Function to run Random Forest using ranger (has built-in parallel support)
parallel_rf_ranger <- function(X, y, n_cores = 1, num_trees = 500, classification = FALSE) {

  # Combine X and y for ranger
  data_combined <- cbind(X, target = y)

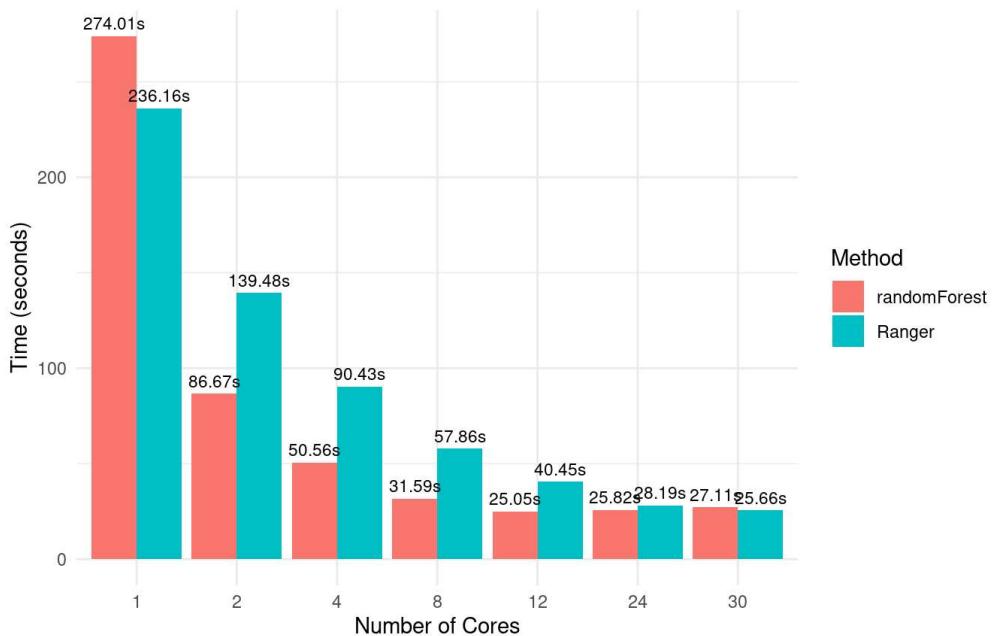
  if (classification) {
    model <- ranger(target ~ .,
                    data = data_combined,
                    num.trees = num_trees,
                    importance = "impurity",
                    num.threads = n_cores,
                    classification = TRUE)
  } else {
    model <- ranger(target ~ .,
                    data = data_combined,
                    num.trees = num_trees,
                    importance = "impurity",
                    num.threads = n_cores)
  }

  return(model)
}
```

Performance

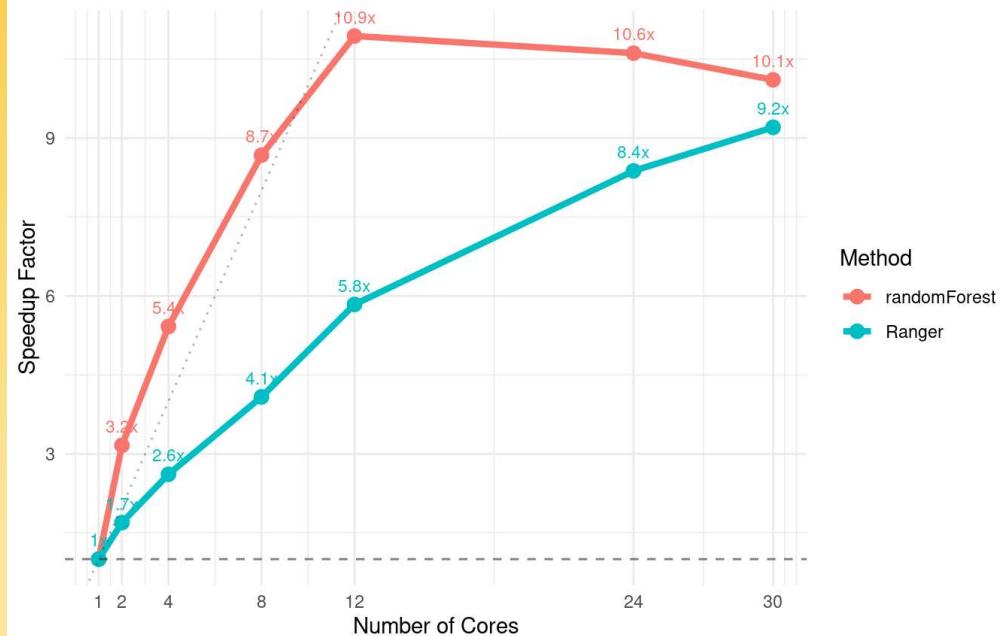
Random Forest Execution Time by Number of Cores

Comparing randomForest and ranger packages



Speedup Factor vs Number of Cores

Dotted line represents ideal linear speedup





Hands-On Practice (Parallel RF)

RStudio Server

This app will launch an RStudio server on a node.

Account (Allocation)

cis240473 (8740.7 SUs remaining)

Queue (partition)

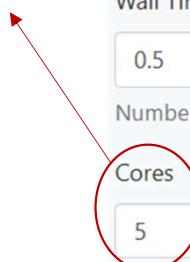
shared

- GPU-only allocations MUST use the 'gpu' queue
- CPU-only allocations MAY NOT use the 'gpu' queue

R Version

4.1.0

Request multiple cores



Wall Time in Hours

0.5

Number of hours you are requesting for your job.

Cores

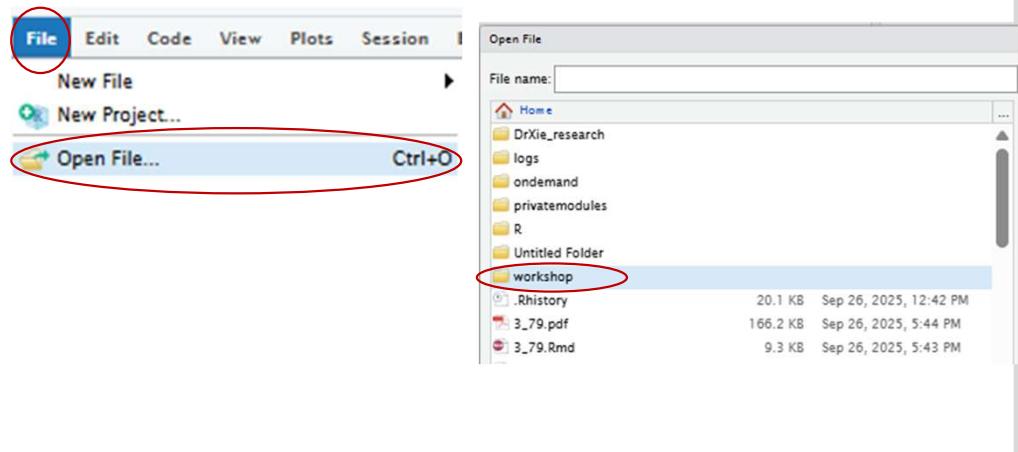
5

Number of cores (up to 128) for a shared job. Non-shared jobs will have exclusive nodes and be charged at 128 cores per node requested

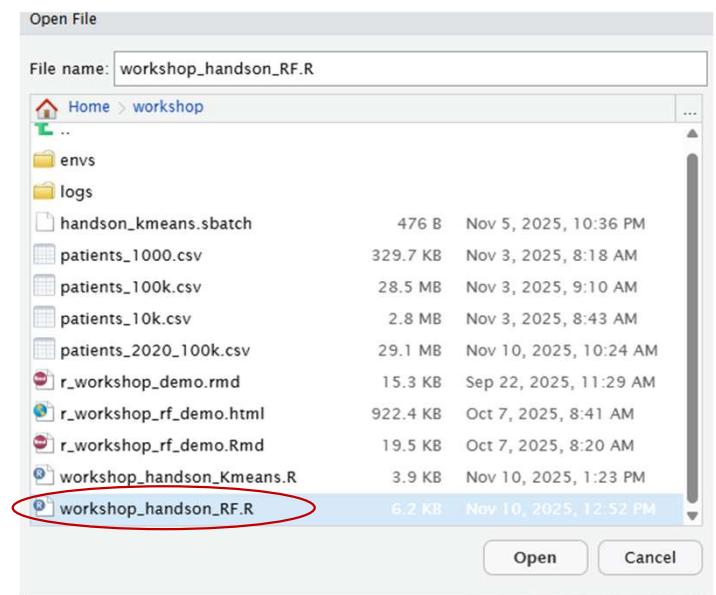
I would like to receive an email when the session starts

Launch

Step 1

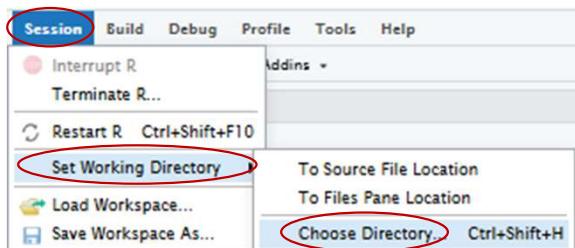


Step 2

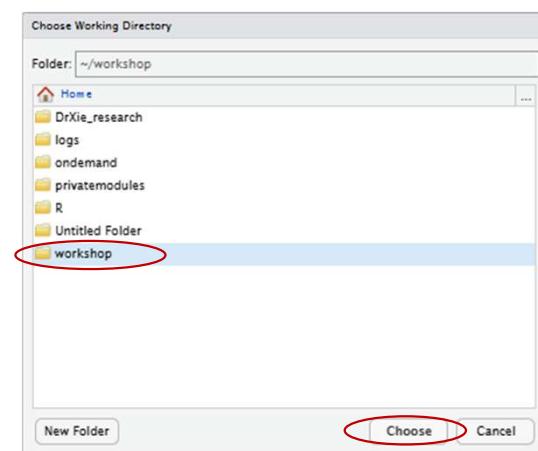


Step 3

Set your session directory to the workshop folder



Step 4



Ctrl + Shift + Enter (Windows/Linux)
or **Cmd + Shift + Enter** (Mac)



To run full script

Using our parallel function and timing we compare using a **single core vs 4 cores** in parallel for both **manual** and **ranger library**.

```
tic("Random Forest Classification with 1 core (randomForest)")
rf_1core <- parallel_rf_manual(X_train_large, y_train_class_large,
                                n_cores = 1, ntree = 200, classification = TRUE)
time_1core <- toc()
time_1core_val <- time_1core$toc - time_1core$tic
# Evaluate model
pred_1core <- predict(rf_1core, X_test)
accuracy_1core <- mean(pred_1core == y_test_class)
cat("\nAccuracy:", round(accuracy_1core * 100, 2), "%\n")

tic("Random Forest Classification with 4 cores (randomForest)")
rf_4cores <- parallel_rf_manual(X_train_large, y_train_class_large,
                                  n_cores = 4, ntree = 200, classification = TRUE)
time_4cores <- toc()
time_4cores_val <- time_4cores$toc - time_4cores$tic

speedup_4 <- time_1core_val / time_4cores_val
cat("\nSpeedup compared to 1 core:", round(speedup_4, 2), "x\n")
pred_4cores <- predict(rf_4cores, X_test)
accuracy_4cores <- mean(pred_4cores == y_test_class)
cat("Accuracy:", round(accuracy_4cores * 100, 2), "%\n")

cat("\n==== RANGER PACKAGE COMPARISON ====\n")
# Test with different core counts using ranger
ranger_times <- c()
ranger_cores <- c()

for (n_cores in c(1, 4)) {
  tic(paste("Ranger with", n_cores, "cores"))
  rf_ranger <- parallel_rf_ranger(X_train_large, y_train_class_large,
                                    n_cores = n_cores, num_trees = 200,
                                    classification = TRUE)
  time_ranger <- toc()
  ranger_times <- c(ranger_times, time_ranger$toc - time_ranger$tic)
  ranger_cores <- c(ranger_cores, n_cores)

  # Calculate accuracy
  pred_ranger <- predict(rf_ranger, data = cbind(X_test, target = y_test_class))$predictions
  accuracy_ranger <- mean(pred_ranger == y_test_class)
  cat("Accuracy:", round(accuracy_ranger * 100, 2), "%\n")
}
```

Output

```
> time_1core <- toc()
Random Forest Classification with 1 core (randomForest): 20.865 sec elapsed
> cat("\nAccuracy:", round(accuracy_1core * 100, 2), "%\n")
Accuracy: 52.17 %

> tic("Random Forest Classification with 4 cores (randomForest)")
> time_4cores <- toc()
Random Forest Classification with 4 cores (randomForest): 4.734 sec elapsed
Speedup compared to 1 core: 4.41 x

> pred_4cores <- predict(rf_4cores, X_test)

> accuracy_4cores <- mean(pred_4cores == y_test_class)
> cat("Accuracy:", round(accuracy_4cores * 100, 2), "%\n")
Accuracy: 51.95 %

Ranger with 1 cores: 14.728 sec elapsed
Accuracy: 51.86 %

Ranger with 4 cores: 4.518 sec elapsed
Accuracy: 52.45 %
```

Command-Line Interface

Command-Line Interface (CLI)

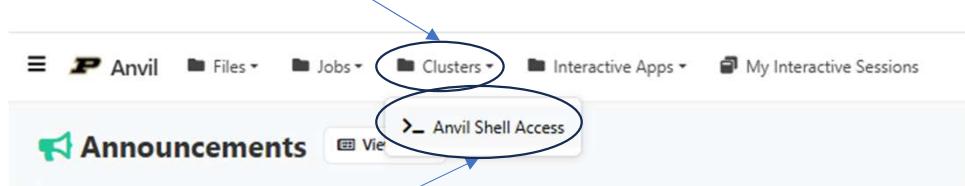
- The **CLI** is a text-based terminal where you run commands, and scripts for **full control** and reproducibility.
- Rather than waiting for an “interactive” session, such as OOD, we can utilize a **Slurm scheduler**.
- **Slurm** is the workload manager (scheduler) that allocates resources and runs your jobs on compute nodes. You **request resources** (CPUs, memory, GPUs, time) and Slurm **queues and dispatches** your job when those resources are available.





Hands-On Practice (CLI)

Step 1



Step 2

Accessing Terminal

```
Host: anvil.rcac.purdue.edu
=====
===== Welcome to the Anvil Cluster =====
=====
== Anvil consists of:
=====
== Nodes:
== Anvil-A ppn=128 256 GB memory (wholenode, wide, shared, debug)
== Anvil-B ppn=128 1024 GB memory (highmem)
== Anvil-G ppn=128 512 GB memory (gpu, gpu-debug)
== + 4 NVIDIA A100 GPUs
=====
== Scratch:
== Quota: 100 TB / 1 million files
== Path: $SCRATCH
== Type command: "myquota"
=====
== Partitions:
== Type command: "showpartitions" or "sinfo -s"
=====
== Software:
== Type command: "module avail" or "module spider"
=====
== User guide:
== www.rcac.purdue.edu/knowledge/anvil
=====
== ACCESS Help Desk:
== https://support.access-ci.org
=====
== News:
== www.rcac.purdue.edu/news/anvil
=====

=====
Last login: Mon Oct  6 12:12:52 2025 from syn-035-145-041-141.res.spectrum.com
Tip of the day (use "touch $HOME/.no.tips" to stop):
=====
Have you every tried to find the man-page for a function that has the same name as the command? Try "man 2 <name>" or "man 3 <name>" for example "man 2 time".
(base) x-sforoughi@login00.anvil:[~] $
```

Creating Folders and Files

- We create a **folder** in our home directory where our script **will install** the necessary **packages**

```
(base) x-sforoughi@login01.anvil:[~] $ mkdir -p ~/R/workshop_libs
```

- Change directory to workshop folder

```
(base) x-sforoughi@login01.anvil:[~] $ cd workshop
(base) x-sforoughi@login01.anvil:[workshop] $ █
```

Writing Slurm File

Similar to OOD

Folders to write outputs and errors

Load R

Set library folder

Run R script

```
#!/bin/bash
#SBATCH --account=cis240473
#SBATCH --partition=shared
#SBATCH --cpus-per-task=5
#SBATCH --time=00:30:00
#SBATCH --job-name=kmeans_cli
#SBATCH --output=logs/%x_%j.out
#SBATCH --error=logs/%x_%j.err

# Create logs dir if doesn't already exist
mkdir -p logs

echo "Job started at $(date)"
echo "This is being ran from $(pwd)"

module load r/4.4.1
export R_LIBS_USER=/home/${whoami}/R/workshop_libs

Rscript workshop_handson_Kmeans.R

echo "Job completed at $(date)"
```

Creating Folders and Files

- Run the Slurm file

```
(base) x-sforoughi@login01.anvil:[workshop] $ sbatch handson_kmeans.sbatch
```

- Check status of your job iteratively

```
(base) x-sforoughi@login01.anvil:[workshop] $ watch squeue -u $USER
```

Cntrl + C to quit



- Once done, look at **output** and **error** files in logs

```
(base) x-sforoughi@login01.anvil:[workshop] $ cat logs/kmeans_cli_14352226.out
```

```
(base) x-sforoughi@login01.anvil:[workshop] $ cat logs/kmeans_cli_14352226.err
```



Thank You!

RESOURCES AND CONTACT INFO

Office of Research Cyberinfrastructure:

ResearchIT@ucf.edu

Contacts:

satyar@ucf.edu

nandan.tandon@ucf.edu