CBS COPENHAGEN BUSINESS SCHOOL
HANDELSHØJSKOLEN

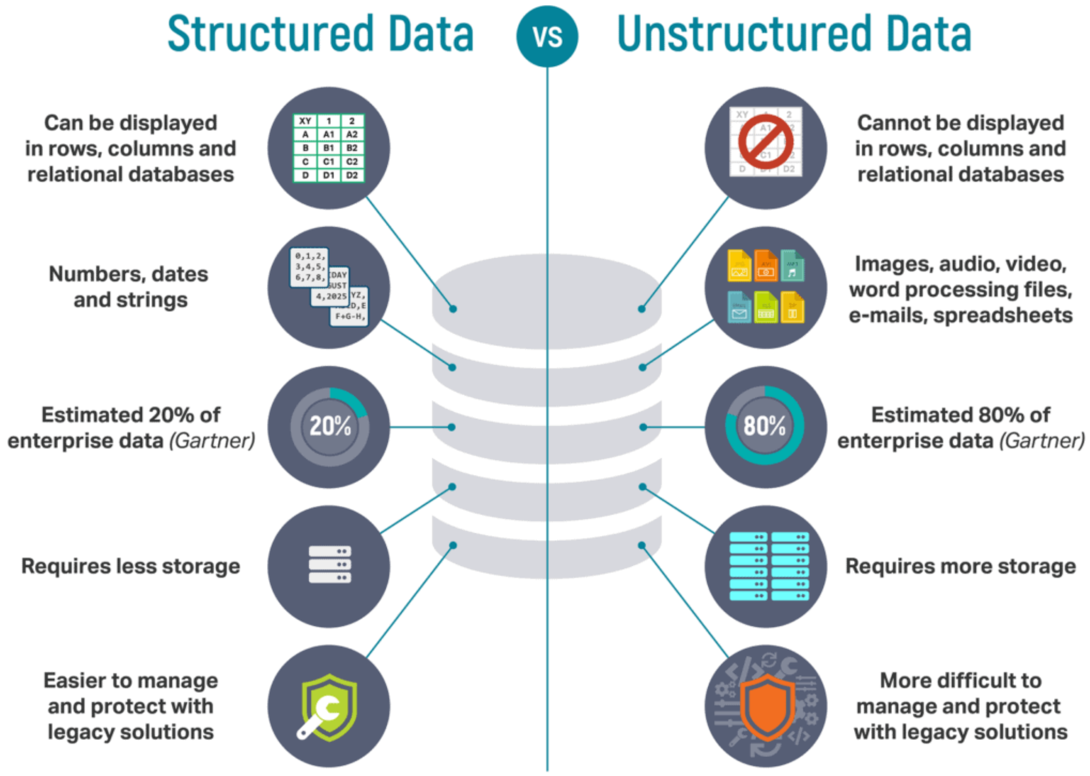
HPC & PARALLEL PROGRAMMING IN R

New cloud computing possibilities for researchers & students

Kristoffer Gulmark Poulsen & Lars Nondal
CBS

About You?

1. What **type and size** of data do you work with?
2. What programming languages do you work in (e.g. R, Python..)?
3. Are you familiar with parallel programming?
4. Are you familiar with high performance computing
5. In particular UCloud?



CBS

Program Today

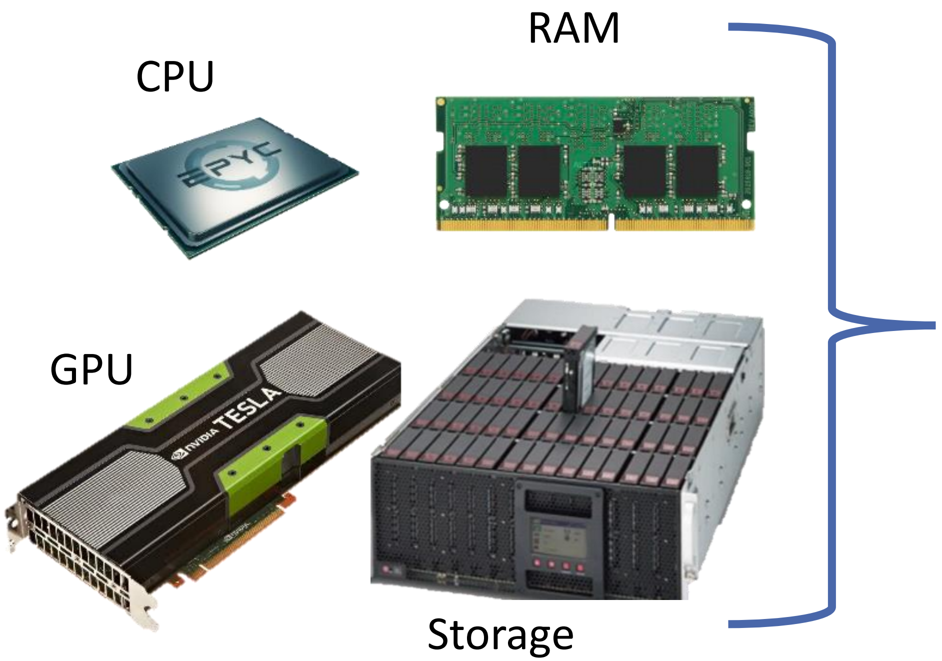
- Introduction to UCloud platform (if needed)
- Basic theory of parallel programming
- Parallel programming basics within R.
- Parallelization of a ML models within the Tidymodels framework.
- https://cbs-hpc.github.io/

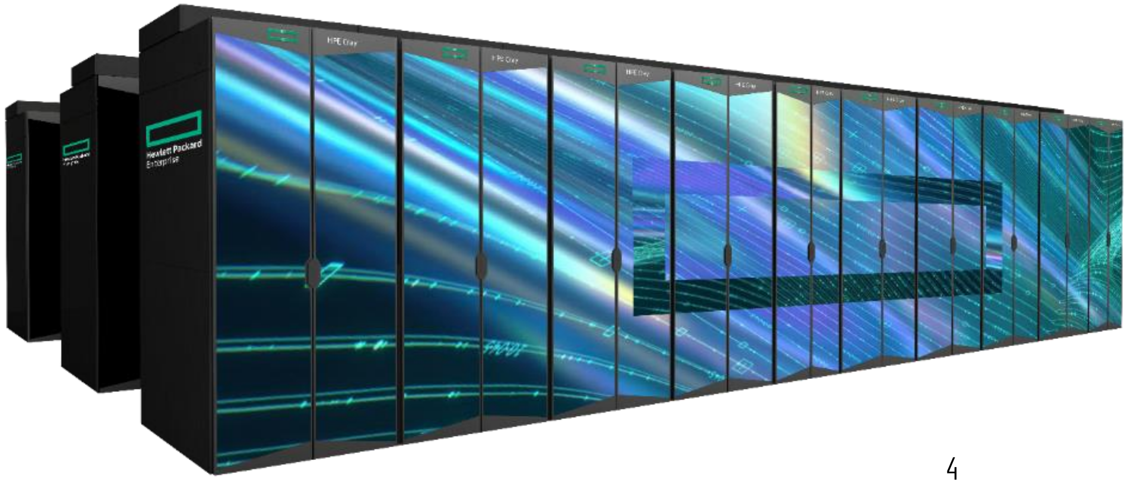
What is High Performance Computing (supercomputer)?

- Network of processors, hard drives & other hardware

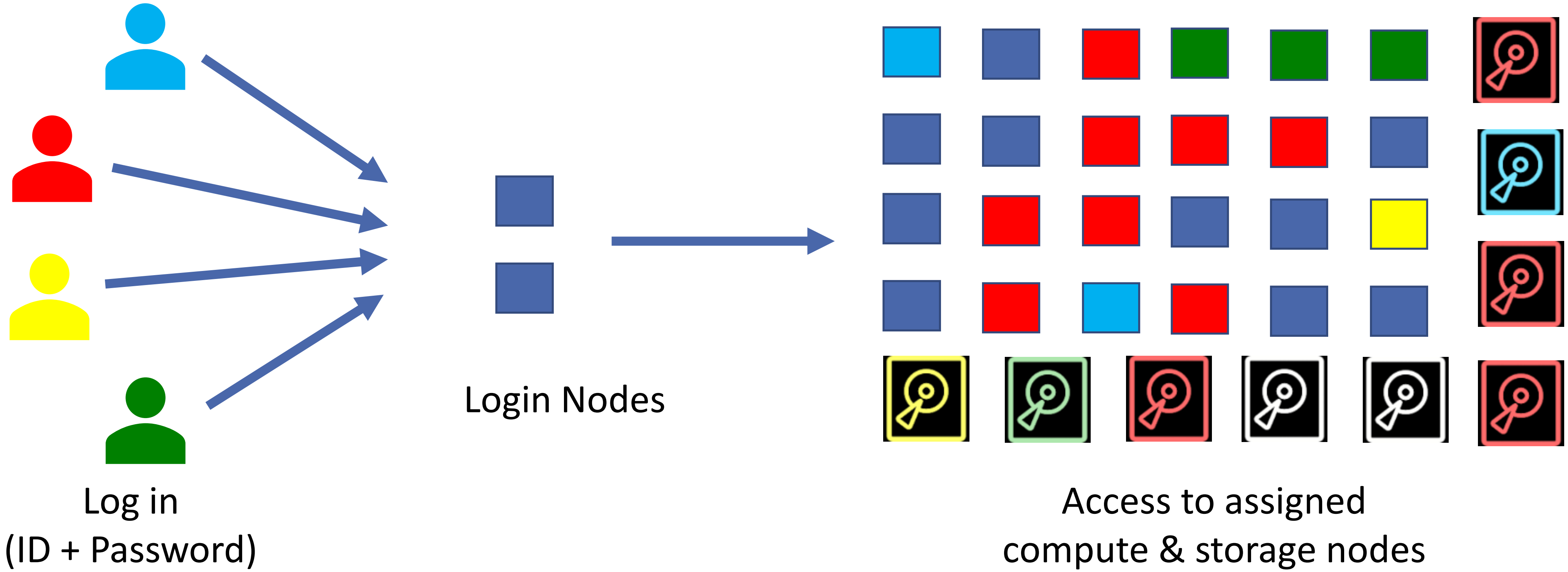
Hardware

- **Core**: Processing unit on a single machine.
- **Node**: A single machine.
- **Cluster**: Network of multiple nodes.





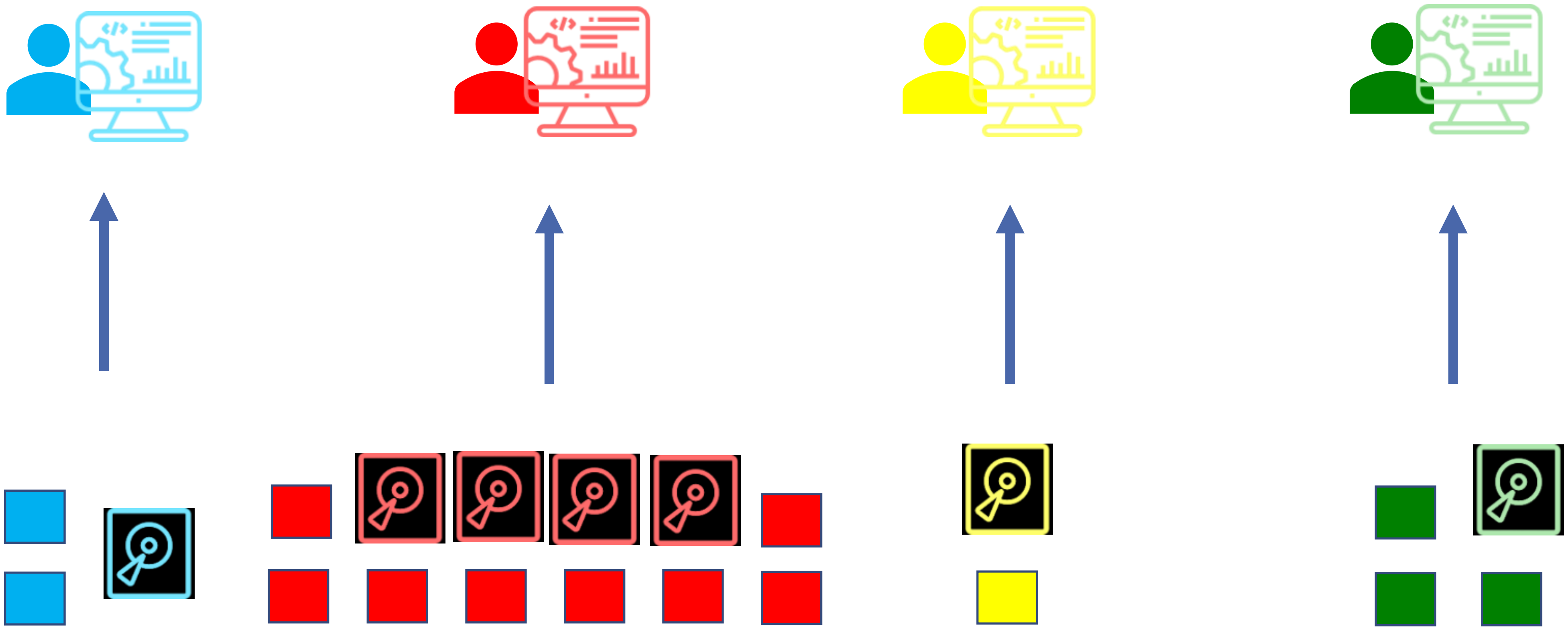
Accessing an HPC...



CBS

Accessing an HPC...

- Your assigned resources (HW + SW) can be used from your PC



When HPC might be for you

- Applying ML/AI
- Running simulation and resampling techniques
- Working with large datasets
- My laptop runs out of memory
- My workflow is running very slow

National HPC facilities

- Collaboration between Universities and DeiC (Danish e-Infrastructure Cooperation)

Type 1 (SDU, AAU)
[Interactive HPC]

Type 2 (AU,KU & DTU)
[Throughput HPC]

Type 3 (SDU)
[Large Memory HPC]

Type 5 (EuroHPC Consortium)
[LUMI Capability HPC]

https://www.deic.dk/en/supercomputing/national-hpc-facilities

CBS

Type 1: Interactive HPC

Type 1 (SDU, AAU)

[Interactive HPC]

Cloud-based (HPC) systems (e.g. similar to google colab, amazon aws)

User friendly with Graphical User Interface (GUI).

Lots of preinstalled software (R, Python, Stata & Matlab)

Collaborative projects – work & share files with others

GDPR-Compliant

Access with university credentials from https://cloud.sdu.dk

- xxx@student.cbs.dk
- xxx@cbs.dk
- 1000 DKK Free credit.

CBS

Type 1: SDU

- **CPU** resources
- GUI based
- Wide range of applications
- Slurm and Spark Cluster

Type 1 (SDU, AAU)

[Interactive HPC]

Name	vCPU	Memory (GB)	GPU	Price
DeiC Interactive HPC (SDU): u1-standard				
u1-standard-1	1 (Intel Xeon Gold 6130)	6	None	0,07 DKK/hour
u1-standard-2	2 (Intel Xeon Gold 6130)	12	None	0,16 DKK/hour
u1-standard-4	4 (Intel Xeon Gold 6130)	24	None	0,33 DKK/hour
u1-standard-8	8 (Intel Xeon Gold 6130)	48	None	0,67 DKK/hour
u1-standard-16	16 (Intel Xeon Gold 6130)	96	None	1,36 DKK/hour
u1-standard-32	32 (Intel Xeon Gold 6130)	192	None	2,74 DKK/hour
u1-standard-64	64 (Intel Xeon Gold 6130)	384	None	5,49 DKK/hour

Support at CBS

Local CBS support

- Lars Nondal & Kristoffer Gulmark Poulsen
- Contact: rdm@cbs.dk or directly to Kristoffer (kgp.lib@cbs.dk)

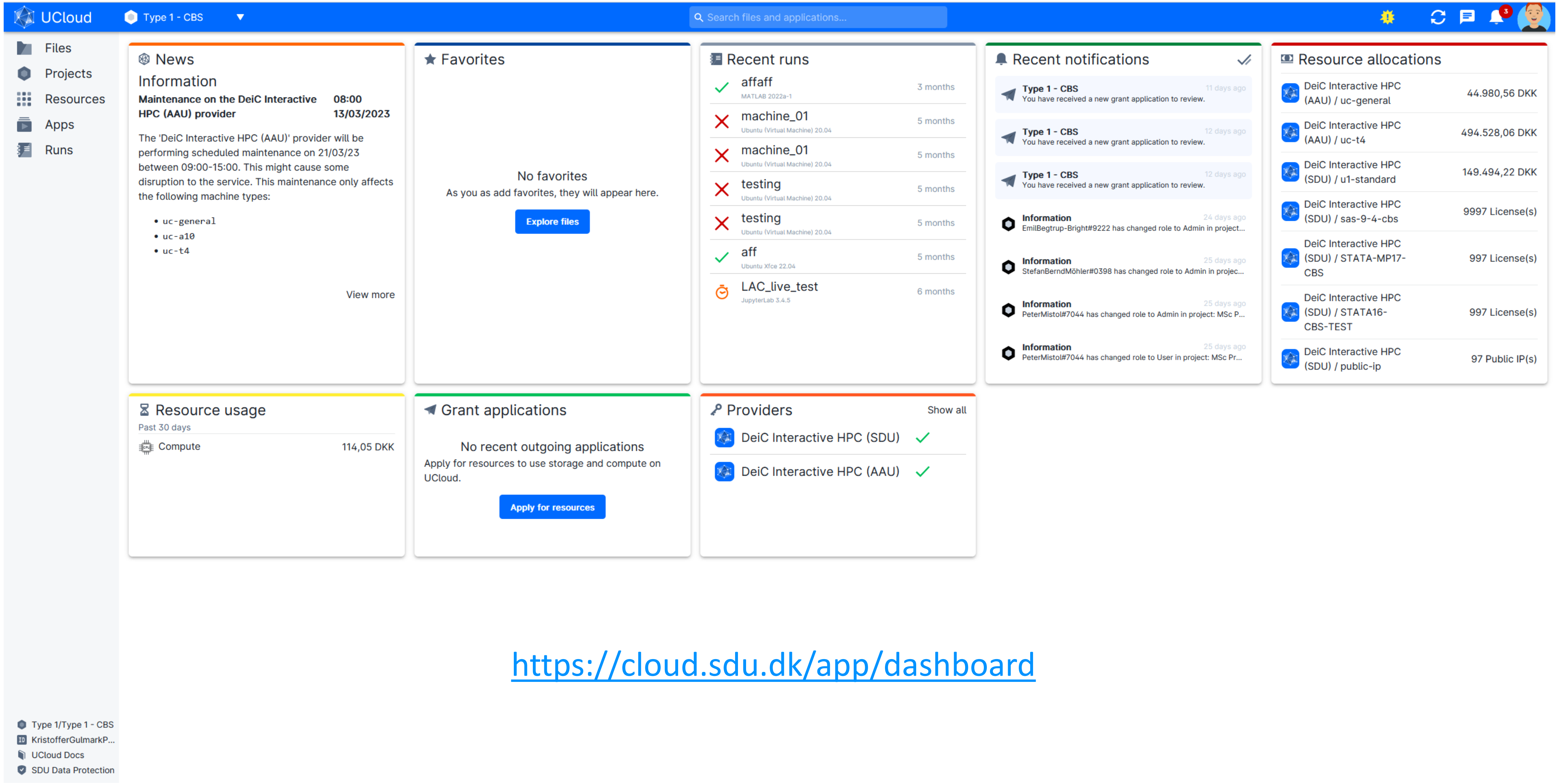
User support: Advising and granting resources, technical problems.

Consultation: Code development etc.

Teaching: “High Performance Computing”, “HPC & Parallel Programming in R and Python” and “Train your ML/AI Model on GPUs”.

Documentation and Tutorials: https://cbs-hpc.github.io/

UCloud Dashboard



https://cloud.sdu.dk/app/dashboard

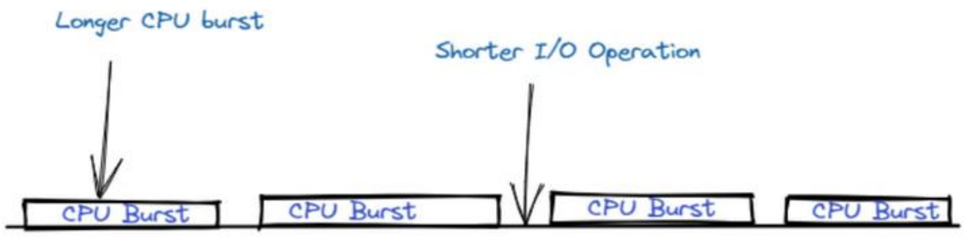
Why is it taking so long?

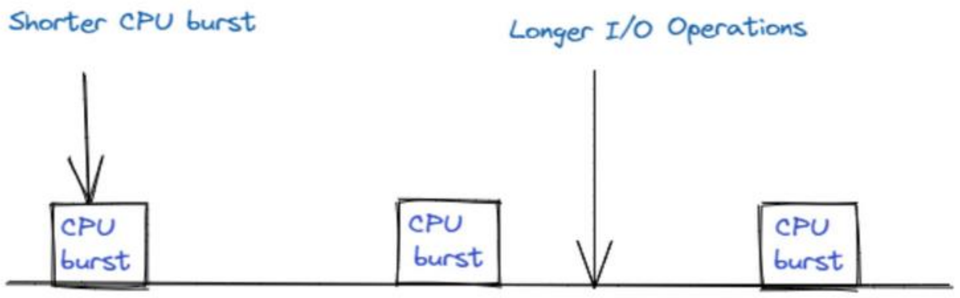
Computation can be slow for one of three reasons:

CPU bound when computational time is restricted by processor.

I/O bound when reading **from** and **to disk/database** is limiting factor.

Memory bound when limited by the memory required to hold the working data.



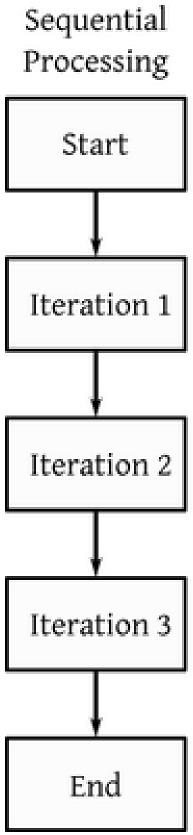


CBS

Parallel Programming

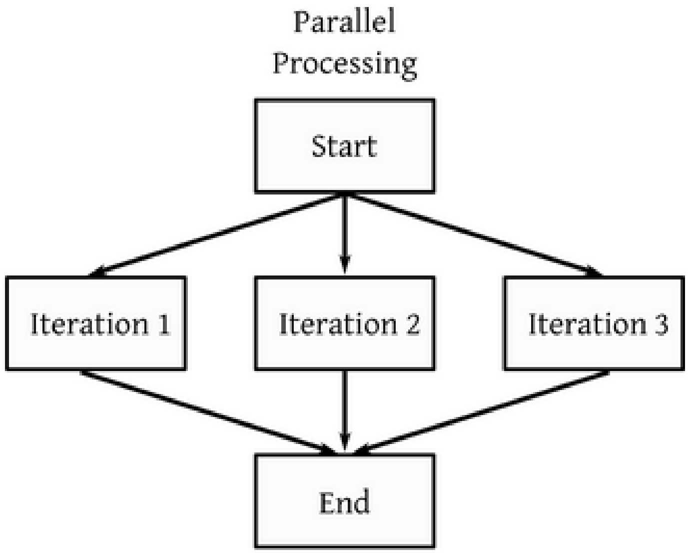
Sequential Computing

- Single core processor
- Multiple tasks which runs overlapping but **not** at same time
- Synchronous tasks



Parallel Computing

- Multi-core processor
- Multiple tasks which runs overlapping.
- Synchronous/Asynchronous

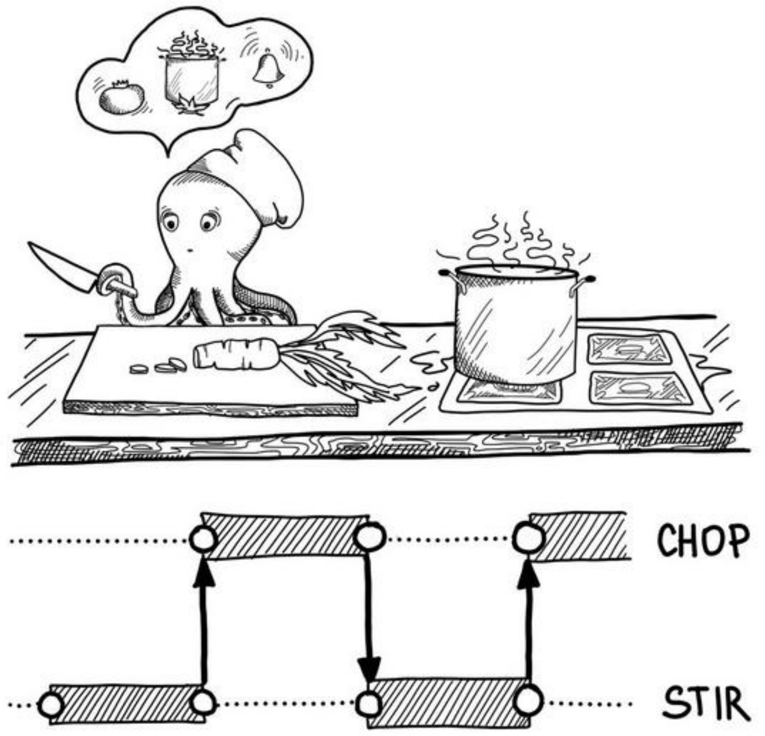


CBS

Parallel Programming

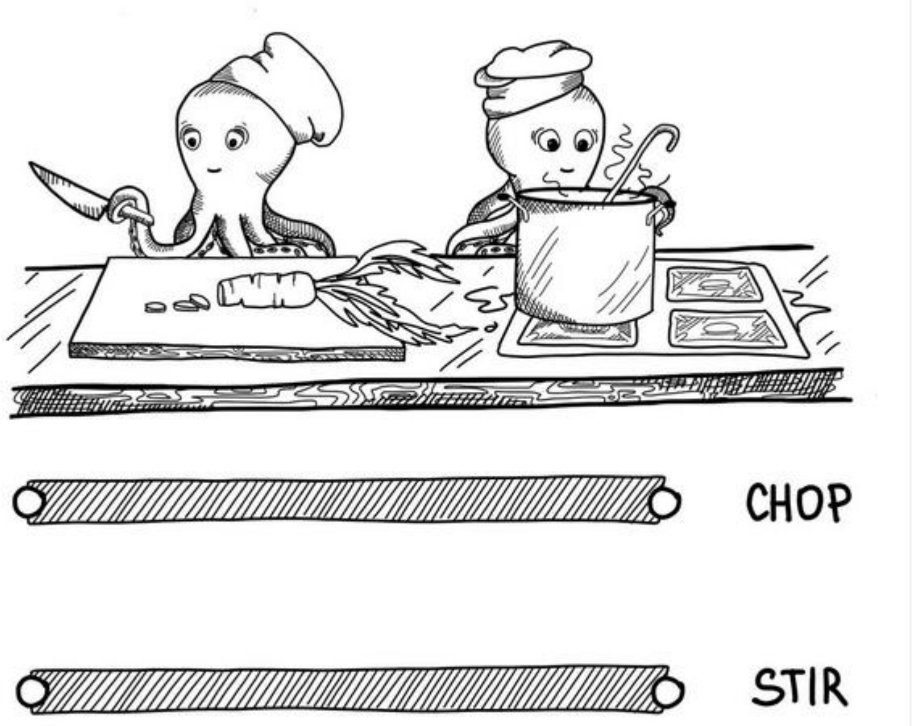
Sequential Computing

- Single core processor
- Multiple tasks which runs overlapping but **not** at same time.
- Synchronous tasks



Parallel Computing

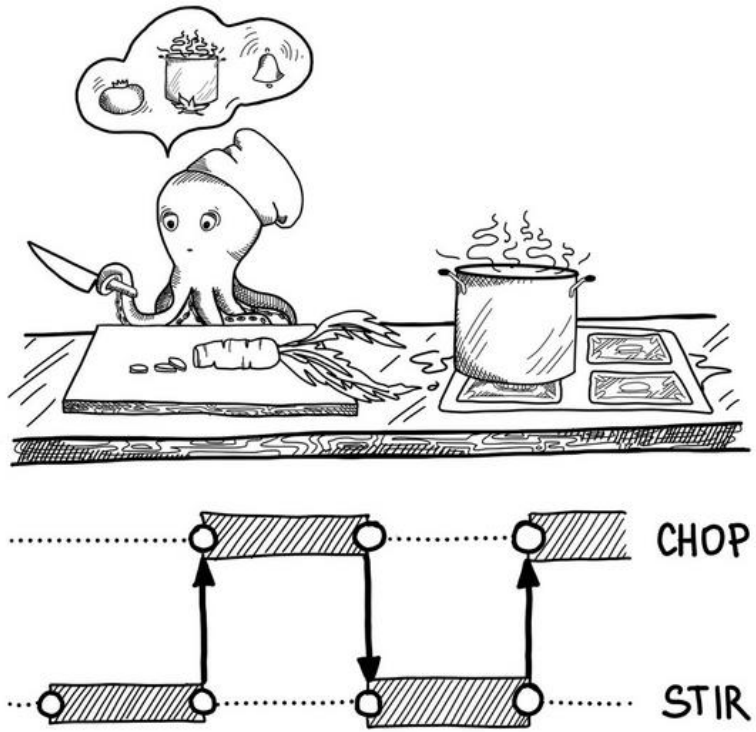
- Multi-core processor
- Multiple tasks which runs overlapping.
- Synchronous/Asynchronous



Parallel Programming

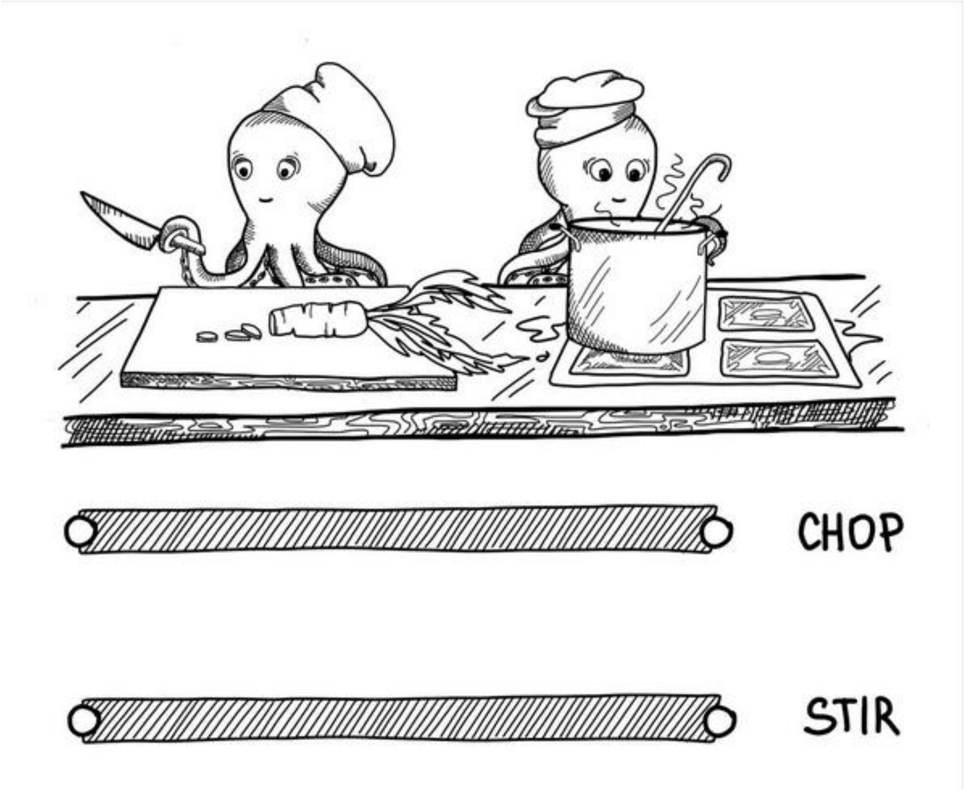
Concurrency

- Executing multiple tasks at the same time but not necessarily simultaneously.



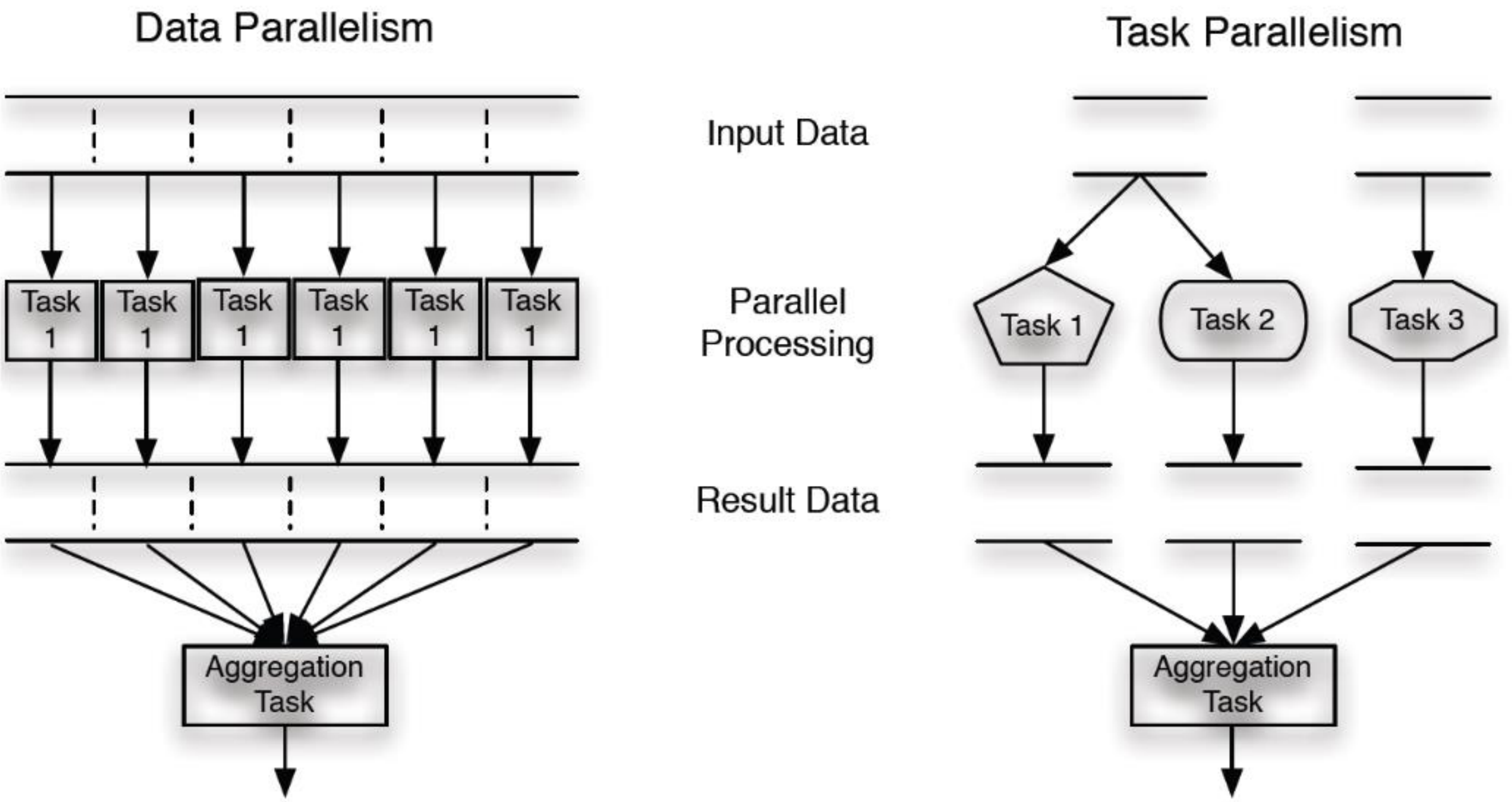
Parallelism

- One task is split into subtasks and run in parallel at the exact same time.
- Run multiple tasks in in parallel on multiple CPUs at the exact same time



CBS

Parallel Programming



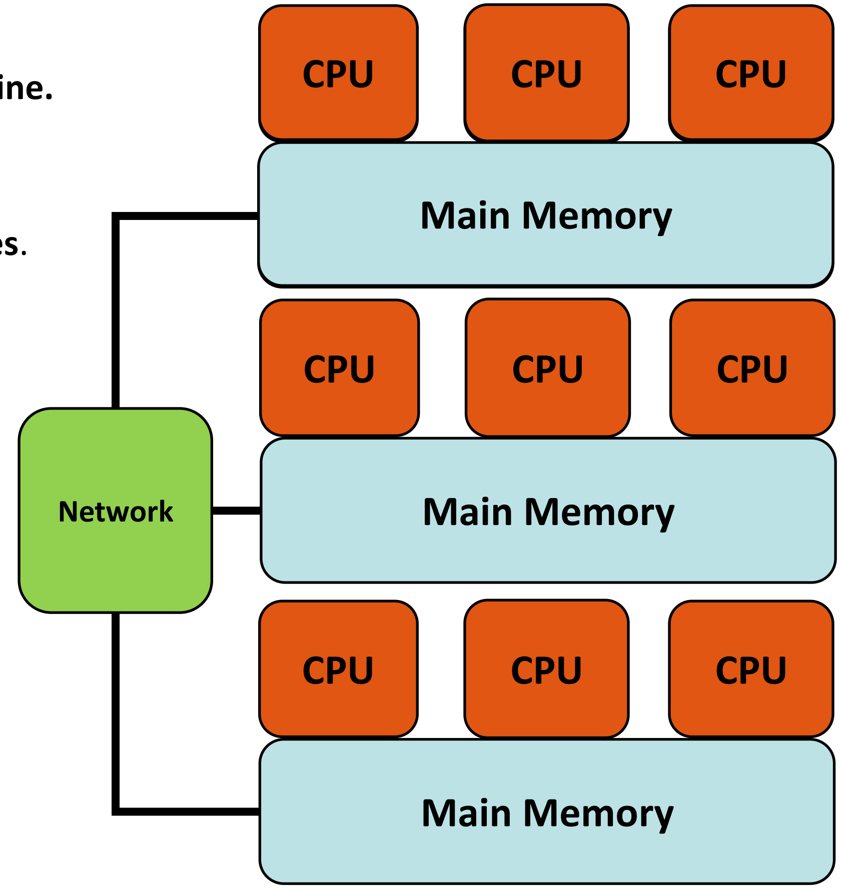
Models for Parallel Programming

Shared Memory Parallelism (SMP) work is divided between **multiple cores** running on a **single machine.**

Distributed Memory Parallelism (Distributed Computing) work is divided between **multiple machines**.

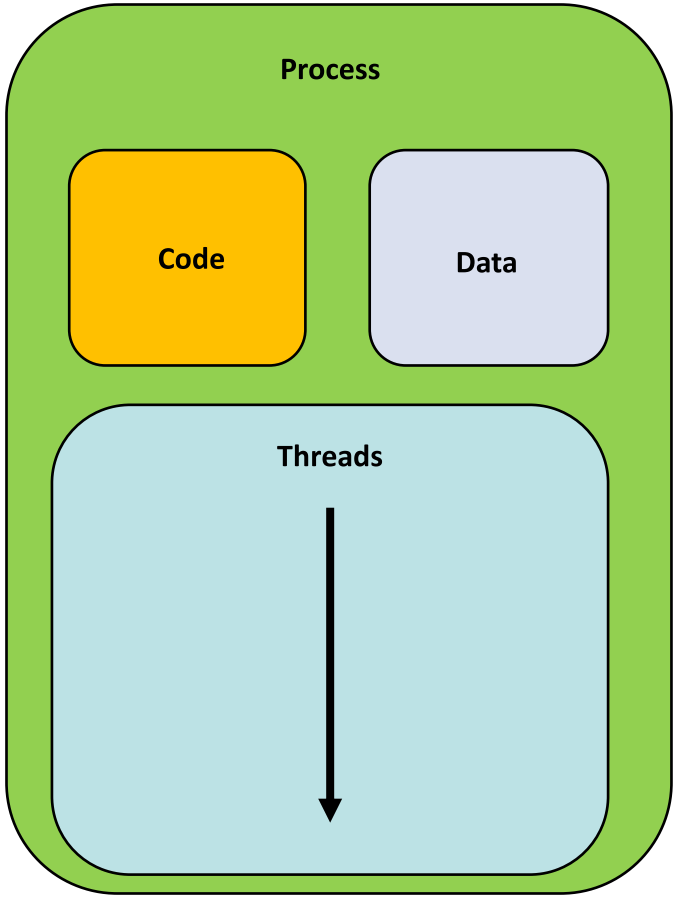
Implicit/Hidden Parallelism - is implemented automatically by the Compiler, Interpreter or Library.

Explicit Parallelism - is written into the source code by the Programmer.



Terminology

- **Embarrassing/ Perfectly Parallel -** the tasks can be run independently, and they don't need to communicate.
- **Process**: Execution of a program . A given executable (e.g., Python or R) may start up multiple processes.
- **Thread**: Path of execution within a single process.
- **Interpreted -** High-level code converted to machine code and executed **line by line**. **(Python & R)**
- **Compiled -** All code is converted to machine code and then program is executed. **(C & Fortran)**

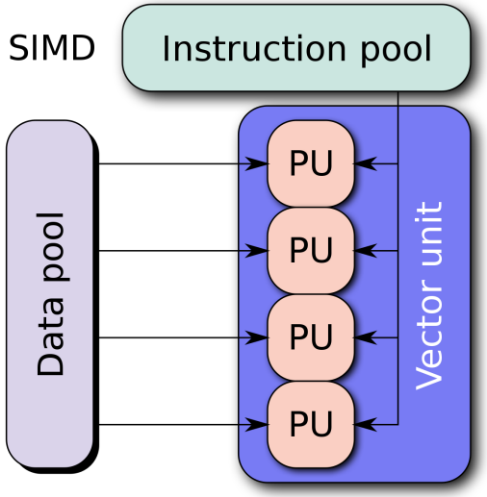


CBS

SIMD & Multi-Threading

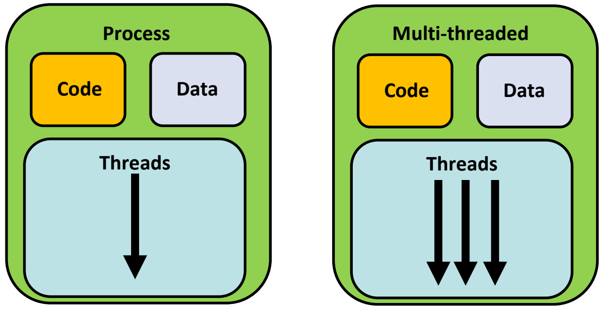
Single Instruction, Multiple Data (SIMD)

- single thread/processor where each processing unit (PU) performs the same instruction on different data.
- **Vectorization**.



Multi-Threading

- **Threads** are multiple paths of execution within a single process.
- Appears as a single process.



Single instruction, multiple threads (SIMT)

<u>**Python** and **R** are examples of single-threaded programming languages.</u>

```
top - 15:12:02 up 2 days, 54 min,  0 users,  load average: 6.42, 6.45, 6.45
Tasks:  10 total,   1 running,   9 sleeping,   0 stopped,   0 zombie
%Cpu(s): 11.0 us,  0.3 sy,  0.0 ni, 88.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem : 385583.7 total, 193583.0 free, 102124.0 used,  89876.6 buff/cache
MiB Swap:   8192.0 total,   4461.5 free,   3730.5 used. 280235.0 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR    %CPU  MEM     TIME+ COMMAND
    243 ucloud    20   0 3970780 962704  74288    278.1  0.2   0:44.50 rsession
    202 rstudio+  20   0  182200  18268  14724           0.0   0:01.00 rserver
      1 ucloud    20   0    6896   3428   3196 S   0.0  0.0   0:00.05 start-rstu+
      7 root      20   0   10420   4920   4376 S   0.0  0.0   0:00.00 sudo
      8 root      20   0     200      4      0 S   0.0  0.0   0:00.01 s6-svscan
     37 root      20   0     200      4      0 S   0.0  0.0   0:00.00 s6-supervi+
    198 root      20   0     200      4      0 S   0.0  0.0   0:00.00 s6-supervi+
    265 ucloud    20   0    2492    580    512 S   0.0  0.0   0:00.01 sh
    271 ucloud    20   0    8168   4904   3408 S   0.0  0.0   0:00.01 bash
    273 ucloud    20   0   10032   3824   3316 R   0.0  0.0   0:00.12 top
```

CBS

SIMD & Multi-Threading in Python and R

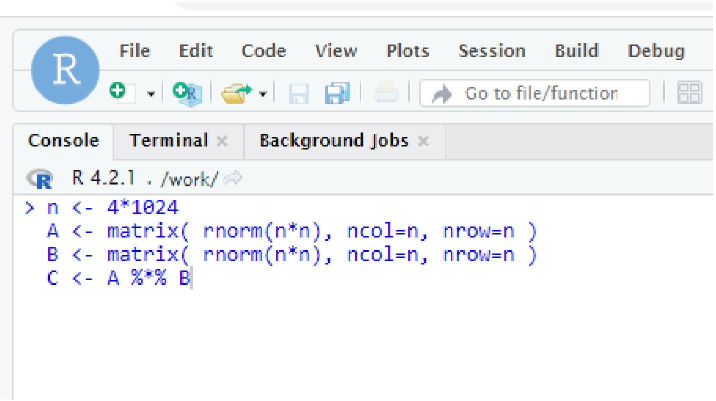
SIMT is achieved in several ways:

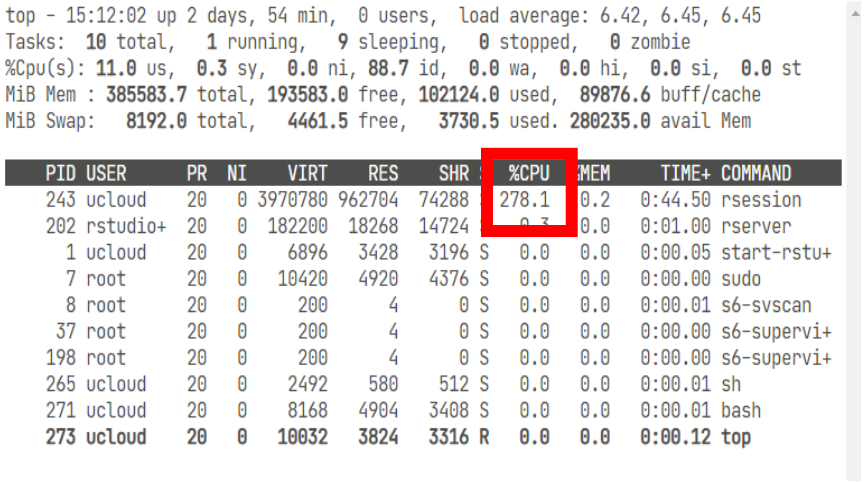
Through external libraries

- Written in other languages (e.g. C, C++, Fortran) that run multi-threaded.
- Linear algebra routines (BLAS & LAPACK) implemented in libraries such as MKL, OpenBLAS or BLIS.
- NumPy, SciPy and Pandas
- built-in R functions

"Static Compilers"

- OpenMP/GCC (GNU Compiler Collection)
- Rcpp
- Cython





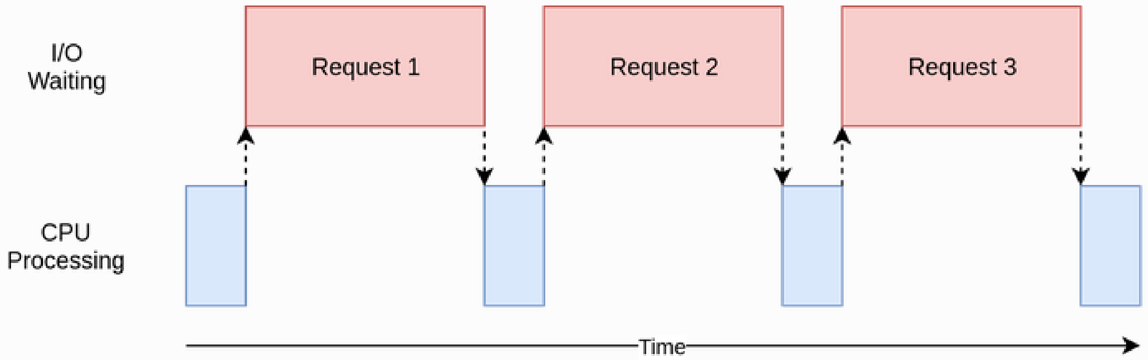
Dynamic/JIT Compilers:

- Numba
- JITR

CBS

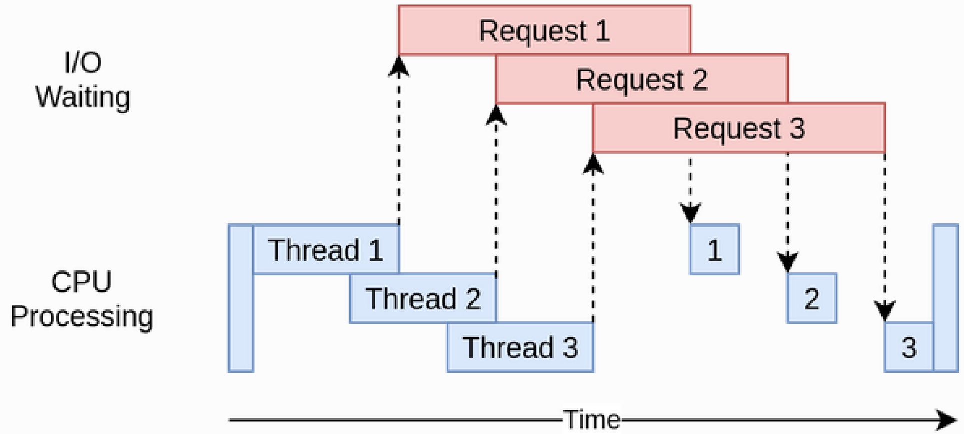
Multi-Threading I/O

This is how an I/O-bound application might look:



From https://realpython.com/, distributed via a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported licence

The speedup gained from multithreading I/O bound problems can be understood from the following image.



From https://realpython.com/, distributed via a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported licence

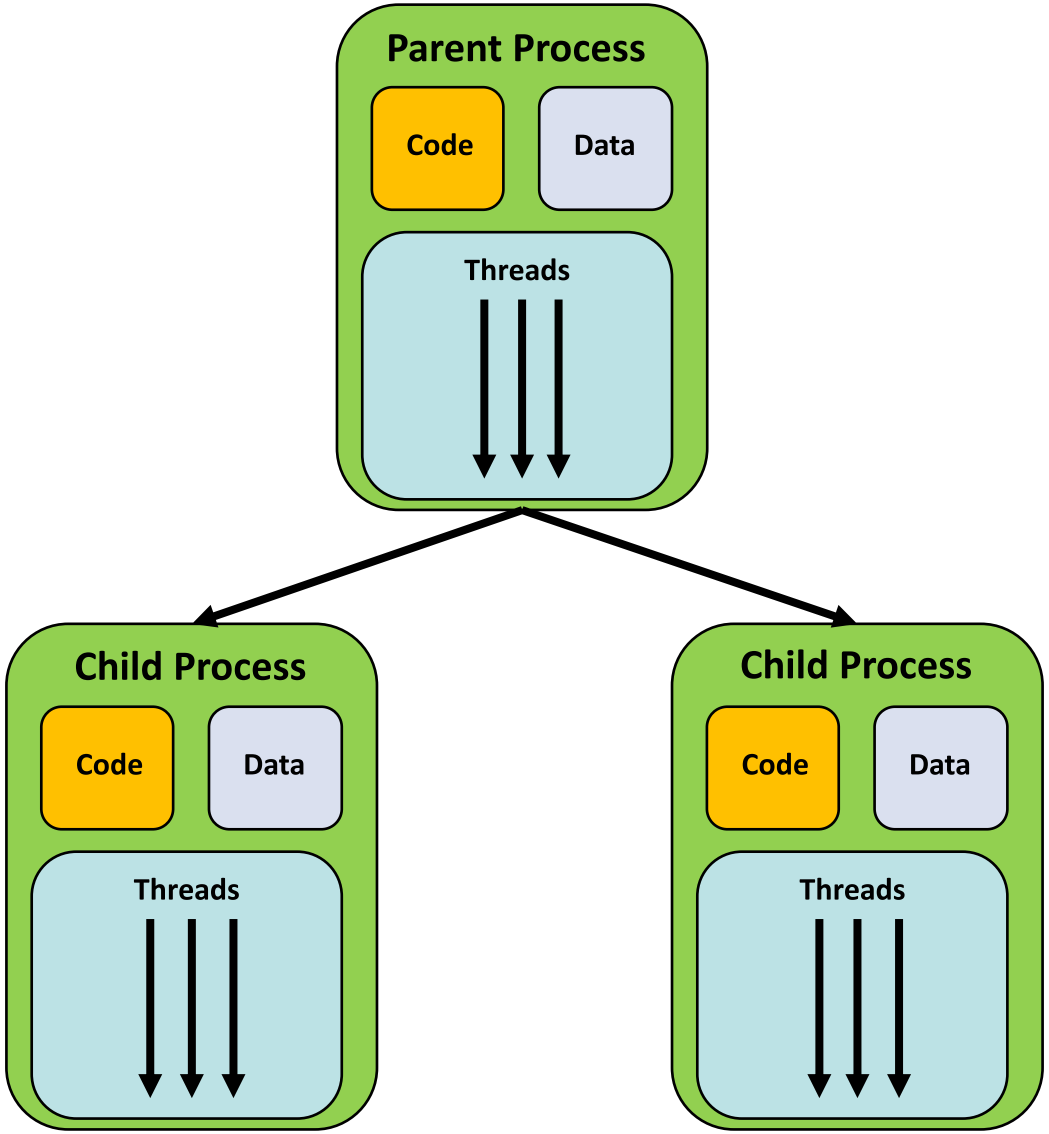
Multi-Processing

Fork

- Only available on UNIX machines (Linux, Mac, and the likes).
- The **child process** is an identical “cloned” of the **parent process**.
- Single machine

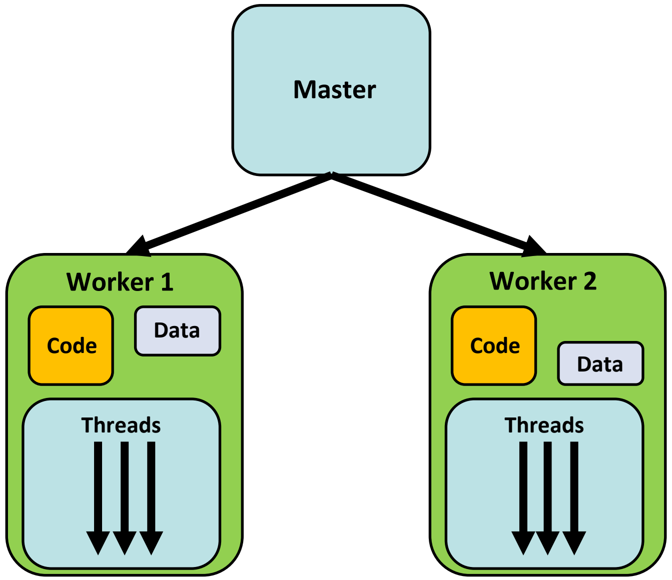
Spawn/Socket (PSOCK)

- Available on Unix and Windows.
- The **parent process** starts a fresh/empty process.
- Code & data needs to copied onto the new **child process**
- Can be scaled to multiple machines/cluster.



Multi-Processing – Splitting Data

Passing only data “chucks” to each worker



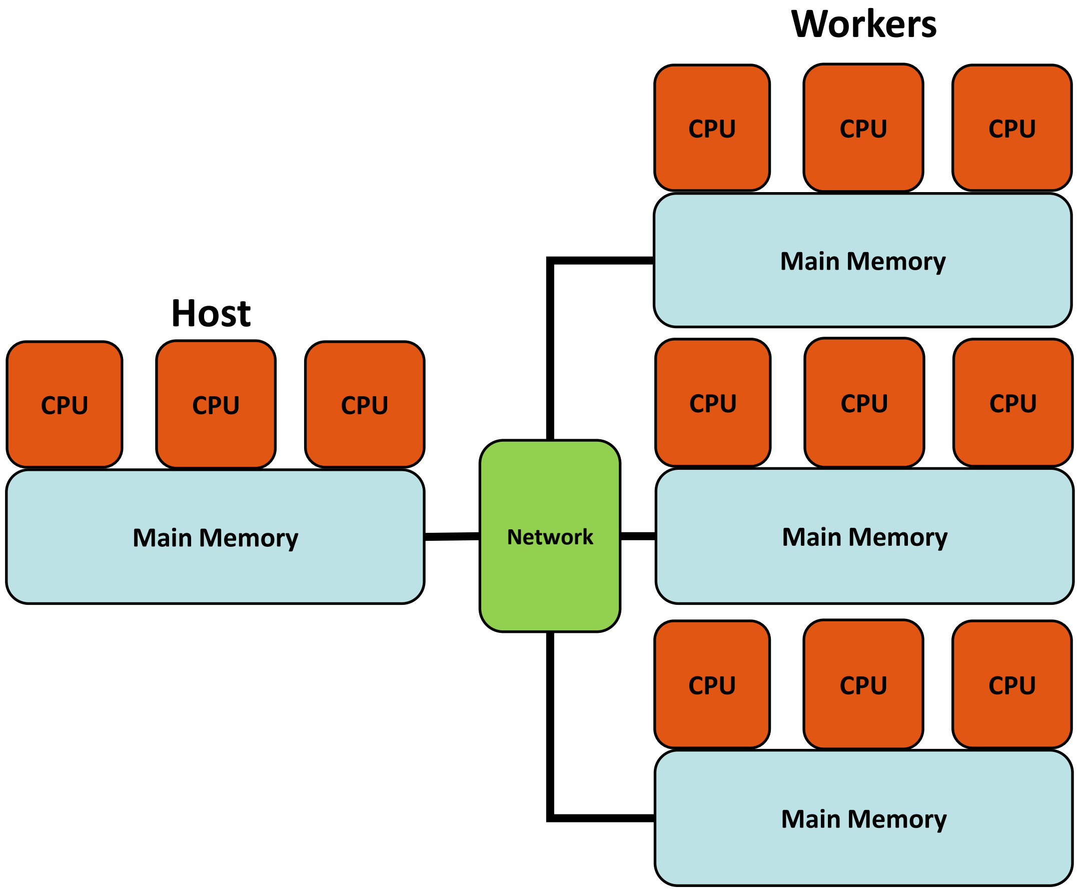
Big chunks are generally better than little chunks

```
for (i in 1:10) {
  for (j in 1:1000000) {

    # Execution of code

    }
  }
```

Distributed Computing on HPC

Distributed Memory Parallelism (Distributed Computing)

- **Multiple machines** with its own **private memory**.
- **Message Passing Interface (MPI)**
- **Host** schedules the work across the **workers**



HPC Job Schedulers:

- Portable Batch System (PBS)
- **Simple Linux Utility for Resource Management (SLURM)**
- IBM Spectrum LSF
- Sun Grid Engine (SGE)

CBS

CBS COPENHAGEN BUSINESS SCHOOL
HANDELSHØJSKOLEN

PARALLEL PROGRAMMING IN R

Kristoffer Gulmark Poulsen & Lars Nondal
CBS

EQUIS ACCREDITED AACSB ACCREDITED ASSOCIATION OF MBAS ACCREDITED CEMS PIM

R Packages - Overview

Compilers (Not covered)

- ***Rcpp***
- ***JIT***

parallel package

- ***multicore***
- ***Snow***

foreach loop adaptation of ***parallel***

- ***doParallel, doSnow**, **doMC & doMPI…***

Tidymodels framework

- Examples of parallel computing

Scalable Frameworks(Not covered)

- ***future***
- ***SparkR***

https://cran.r-project.org/web/views/HighPerformanceComputing.html

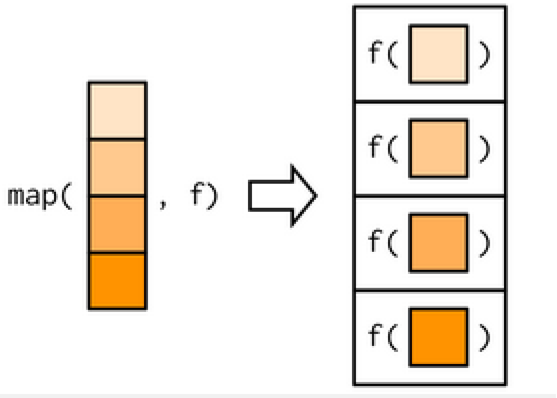
Iterations

There are two styles of iterations

```r
for (i in 1:3) print(sqrt(i))
```

for and *while* loops

- It is often the most intuitive way to begin.
- **Imperative programming .**



functional programming

- Readability & code redundancy
- **Functionals** are a functions that takes a function as an input and returns a vector as output.
- E.g. **apply()** or **map()**

R Packages- *Parallel*

- ***multicore***: Multi-processing on **single machine** through **forking** (Not Covered Today).

- ***Snow (Simple Network of Workstations)***: Can **fork/spawning** on multiple machines/clusters.

- ***paralellel*** serve as ***"parallel backend"*** to many/most packages, so worth understanding.

- It is all based on ***apply*** form of R iteration:

Base R	snow
lapply	parLapply
sapply	parSapply
vapply	-
apply(rowwise)	parRapply, parApply(,1)
apply(columnwise)	parCapply, parApply(,2)

CBS

R Packages- *Parallel /snow*

Snow (Simple Network of Workstations): Can **fork/spawning** on multiple machines/clusters.

Functions:

- ***cl<- makeCluster(n,type ="PSOCK")*** - (Default)
- ***cl<- makeCluster(n,type ="FORK")***
- ***stopCluster(cl)*** – stops clusters
- ***clusterExport(cl,data)*** - Copies data to processes
- ***clusterApply(cl,data,func)*** – Runs analysis in parallel
- ***clusterApplyLB()*** – dynamic load balancing
- ***clusterEvalQ(cl, expr)*** – Evaluating an expression
- ***clusterSplit(cl,data)*** – data splitting

ClusterApply

```
cl <- makeCluster(4)
system.time(clusterExport(cl, "jan2010"))

user  system elapsed
  0.128   0.027   0.574


system.time(cares <- clusterApply(cl, rep(5,4), do.n.kmeans))

  user  system elapsed
  0.357   0.039  11.064
```

CBS

R Packages- *foreach* loop adaptation of *parallel*

Parallelization using the "**for loop"** iteration through the ***foreach*** package.

```
for (i in 1:3) print(sqrt(i))
```

```
library(foreach)
foreach (i=1:3) %do% sqrt(i)
```

```
library(doParallel)
registerDoParallel(3)  # use multicore-style forking
foreach (i=1:3) %dopar% sqrt(i)
```

```
cl <- makePSOCKcluster(3)
registerDoParallel(cl)  # use the just-made PSOCK cluster
foreach (i=1:3) %dopar% sqrt(i)
```

Many different backends:

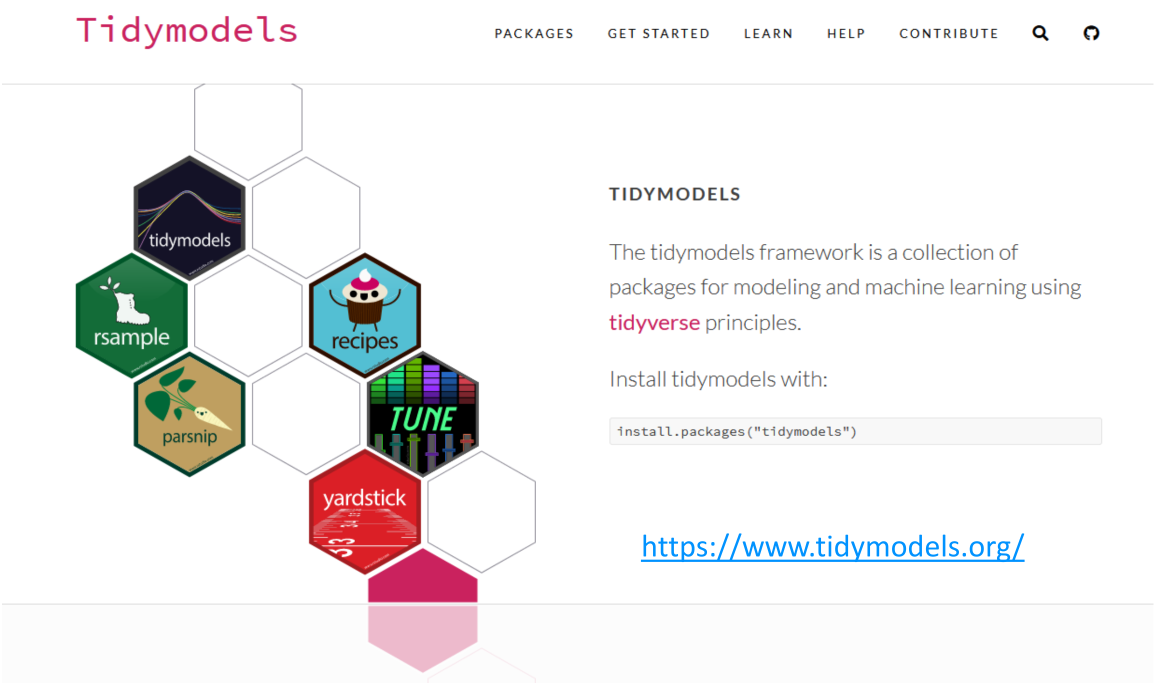
- ***doParallel*** - https://cran.r-project.org/web/packages/doParallel/index.html
- ***doSnow*** - https://cran.r-project.org/web/packages/doSNOW/index.html
- ***doMC*** - https://cran.r-project.org/web/packages/doMC/index.html
- ***doMPI*** - https://cran.r-project.org/web/packages/doMPI/index.html

CBS

Tidymodels

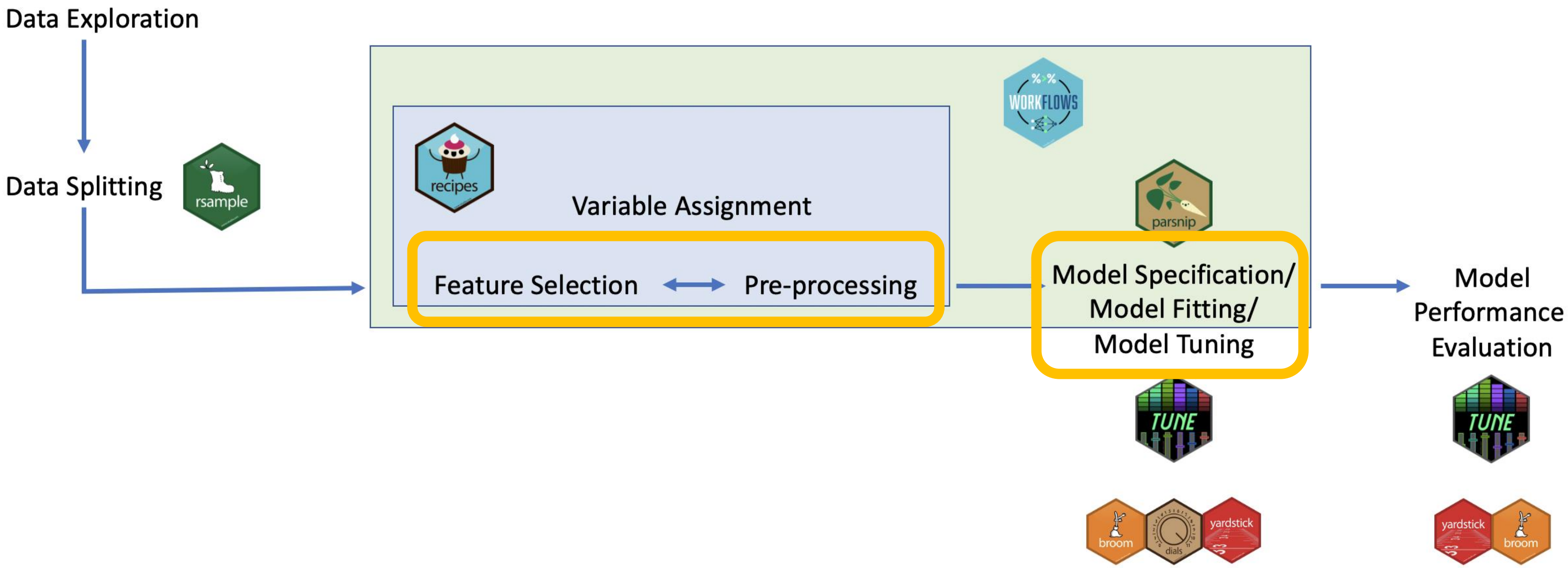
Tidyverse/Tidymodels

- The **tidyverse** is a language for solving data science challenges with R code.
- Both **tidymodels** is built on the **tidyverse** principles**:**
  - Should be intuitive
  - Consistence syntax: function naming, arguments.



https://www.tidymodels.org/

Tidymodels - Workflow

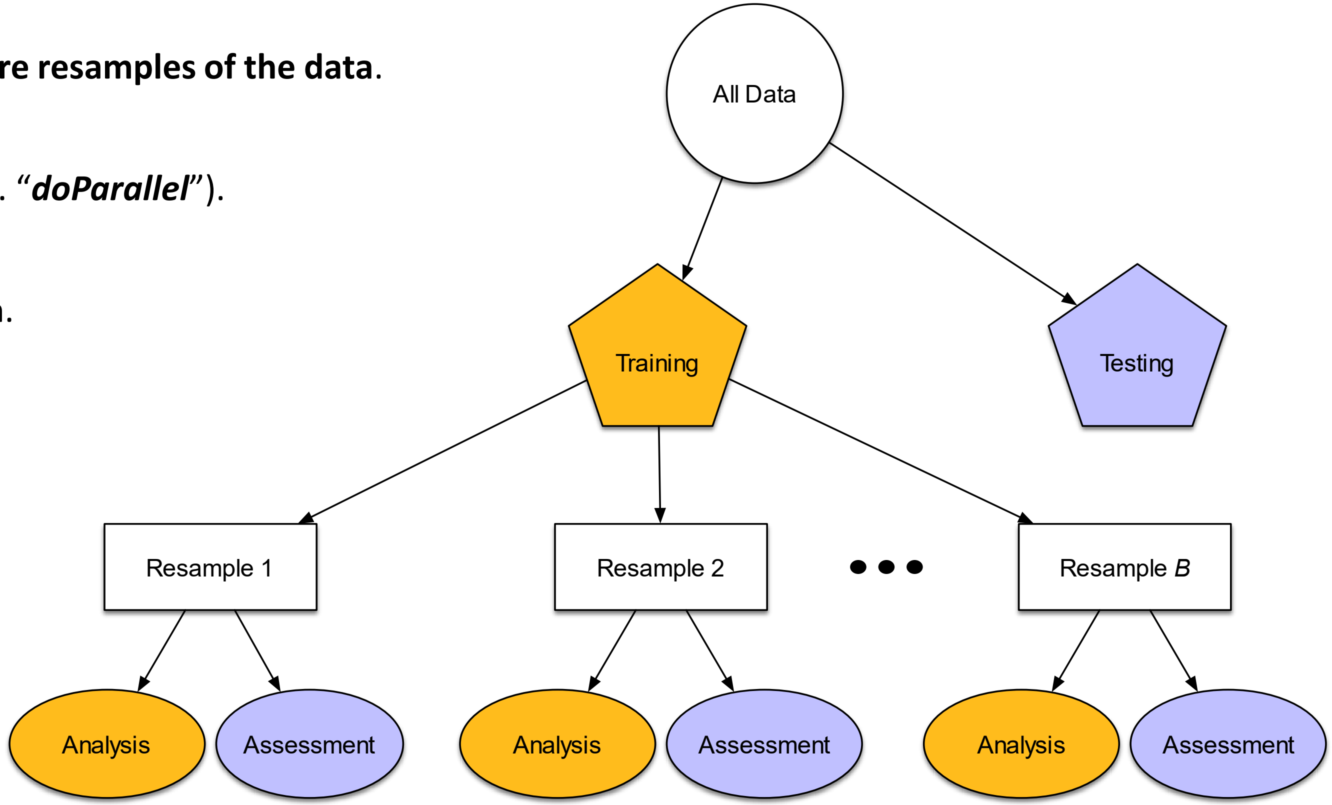


https://jhudatascience.org/tidyversecourse/model.html

Tidymodels - Model Fitting and Tuning

Model **performance** and **optimization** is based on **resampling methods** which are just **embarrassing parallel!!**

- **fit_resamples()** computes a set of performance metrics across **one** or **more resamples**.
- **tune_grid()** of performance for **tuning parameters** across **one** or **more resamples of the data**.
- ***foreach*** package is used in combination with a backend package (e.g. "***doParallel***").
- Many ML/AI packages within Tidymodels have built-in parallelisation.



CBS

Tidymodels - Model Fitting and Tuning

```
nnet_tune <-
  nnet_workflow  %>%
  tune_grid(hotel_validation,
        grid = nGrid,
        control = control_grid(save_pred = TRUE,parallel_over = "everything"),
        metrics = metric_set(roc_auc))
```

"resamples"

- then tuning will be performed in parallel over **resamples alone.**
- Within each resample, the **preprocessor (i.e. recipe or formula)** is reused across all models.

"everything"

- An outer parallel loop will iterate over **resamples**.
- An inner parallel loop will iterate over all **unique combinations of preprocessor** and **model tuning parameters** for that specific resample.
- This will result in the preprocessor being re-processed multiple times
- Pre-processing depended.

Tidymodels - Case

- Hotel bookings data from [Antonio, Almeida, and Nunes (2019)]
- **Aim:** to predict which hotels are preferred by families with children.
- Data frame: 50.000 entries and 23 variables

Data Splitting

```
set.seed(123)

# Split into Training and Testing set
splits  <- initial_split(hotels, strata = children)
hotel_train <- training(splits)
hotel_test  <- testing(splits)

# Split Validation set from Training set (Alternative to Cross Validation)
set.seed(234)
hotel_validation <- validation_split(hotel_train,
                        strata = children,
                        prop = 0.80)
```

Methodology used: Classification

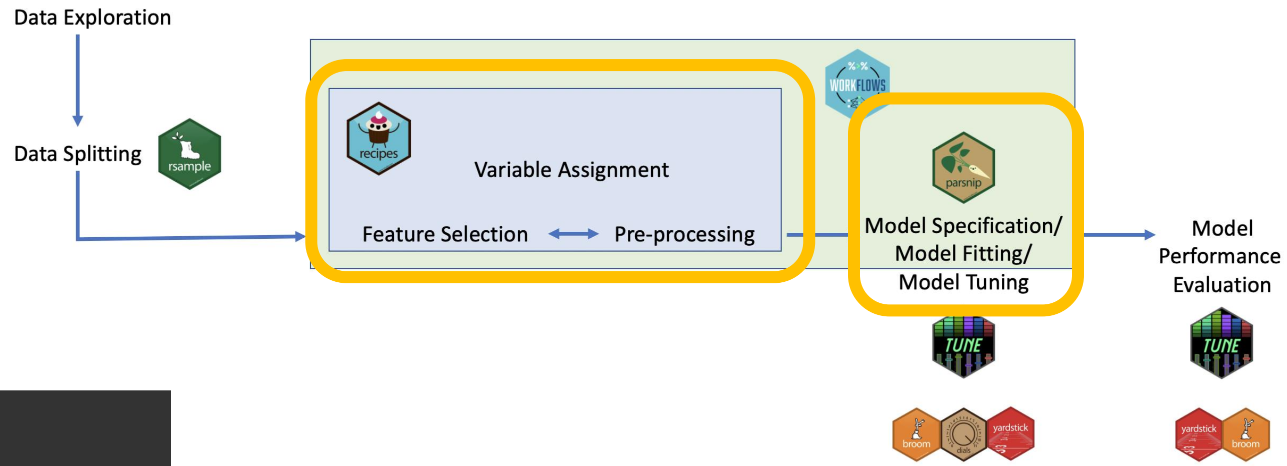
- Random Forrest - ***ranger::ranger()***
- Neural Network - ***nnet::nnet()***

	x
1	hotel
2	lead_time
3	stays_in_weekend_nights
4	stays_in_week_nights
5	adults
6	children
7	meal
8	country
9	market_segment
10	distribution_channel
11	is_repeated_guest
12	previous_cancellations
13	previous_bookings_not_canceled
14	reserved_room_type
15	assigned_room_type
16	booking_changes
17	deposit_type
18	days_in_waiting_list
19	customer_type
20	average_daily_rate
21	required_car_parking_spaces
22	total_of_special_requests
23	arrival_date

https://www.tidymodels.org/start/case-study/

Tidymodels - Model Tuning

Random Forrest - ***ranger::ranger()***



https://jhudatascience.org/tidyversecourse/model.html

Pre-processing with recipes

```
rf_recipe <-
  recipe(children ~ ., data = hotel_train) %>%
  step_date(arrival_date) %>%  # creates predictors for the year, month, and day of the week.
  step_holiday(arrival_date) %>% # generates a set of indicator variables for specific holidays.
  step_rm(arrival_date) #removes variables;
```

Model Specifications

```
Cores = parallel::detectCores()
set.seed(345)
# Define Model
rf_model <-
  rand_forest(mtry = tune(), min_n = tune(), trees = 1000) %>%
  set_engine("ranger", num.threads = Cores) %>%
  set_mode("classification")

# Define Workflow
rf_workflow <-
  workflow() %>%
  add_model(rf_model) %>%
  add_recipe(rf_recipe)
```

Grid Tuning – In Parallel

```
# Grid-Tune with 8(4) Cores
tic()
rf_tune <-
  rf_workflow %>%
  tune_grid(hotel_validation,
         grid = nGrid,
         control = control_grid(save_pred = TRUE),
         metrics = metric_set(roc_auc))
toc()

47.3 sec elapsed
```

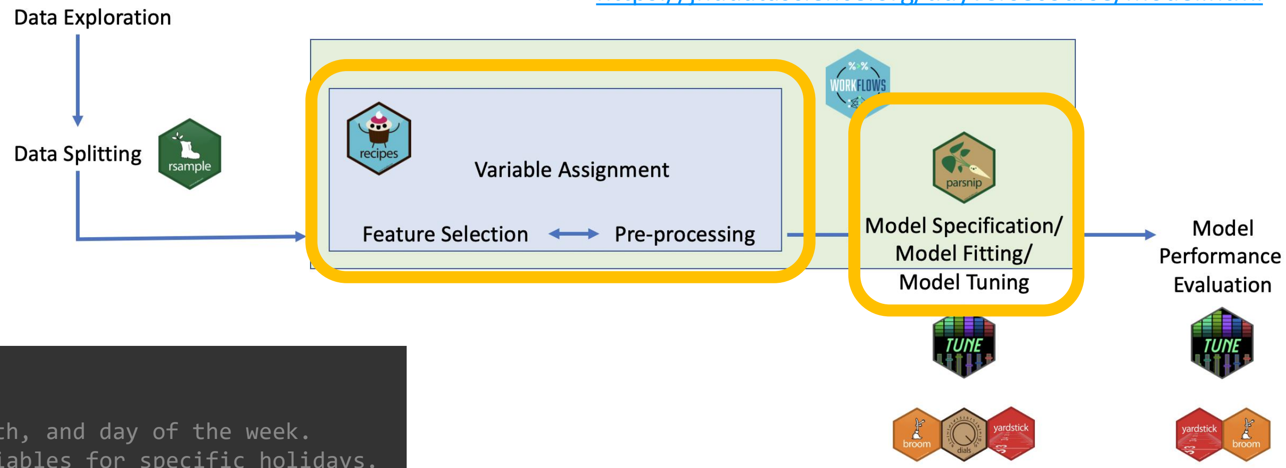
Grid Tuning – Not in Parallel

```
# Grid-Tune with 1 Core
tic()
rf_tune <-
  rf_workflow %>%
  tune_grid(hotel_validation,
         grid = nGrid,
         control = control_grid(save_pred = TRUE),
         metrics = metric_set(roc_auc))
toc()

208.8 sec elapsed
```

CBS

Tidymodels - Model Tuning

https://jhudatascience.org/tidyversecourse/model.html



Neural Network - ***nnet::nnet()***

Pre-processing with recipes

```
nnet_recipe <-
  recipe(children ~ ., data = hotel_train) %>%
  step_date(arrival_date) %>%                     # creates predictors for the year, month, and day of the week.
  step_holiday(arrival_date, holidays = holidays) %>%   # generates a set of indicator variables for specific holidays.
  step_rm(arrival_date) %>%                       # removes orginal variables;
  step_dummy(all_nominal_predictors()) %>%        # Converts characters or factors dummy variables.
  step_zv(all_predictors()) %>%                   # removes orginal variables;
  step_normalize(all_predictors())
```

Model Specifications

```
# Define Model
nnet_model <-
  mlp(hidden_units = tune(), penalty = tune(),
epochs = tune()) %>%
  set_engine("nnet", trace = 0,MaxNWts = 10000) %>%
  set_mode("classification")


# Define Workflow
nnet_workflow <-
  workflow() %>%
  add_model(nnet_model) %>%
  add_recipe(nnet_recipe)
```

Grid Tuning – In Parallel

```
# Grid-Tune with Multiple Cores
library(doParallel)
library(foreach)
cl <- makePSOCKcluster(parallel::detectCores())
registerDoParallel(cl)


tic()
nnet_tune <-
  nnet_workflow  %>%
  tune_grid(hotel_validation,
        grid = 5,
        control = control_grid(save_pred = TRUE,parallel_over = "everything"),
        metrics = metric_set(roc_auc))
toc()
stopCluster(cl)

293.41 sec elapsed
```

Grid Tuning – Not in Parallel

```
# Grid-Tune with Multiple Cores
library(doParallel)
library(foreach)
cl <- makePSOCKcluster(parallel::detectCores())
registerDoParallel(cl)


tic()
nnet_tune <-
  nnet_workflow  %>%
  tune_grid(hotel_validation,
        grid = 5,
        control = control_grid(save_pred = TRUE,parallel_over = "resamples"),
        metrics = metric_set(roc_auc))
toc()
stopCluster(cl)

600.77 sec elapsed
```

```
# Grid-Tune with No Parallelisation
tic()
nnet_tune <-
  nnet_workflow  %>%
  tune_grid(hotel_validation,
        grid = 5,
        control = control_grid(save_pred = TRUE,parallel_over = "everything"),
        metrics = metric_set(roc_auc))
toc()

676.7 sec elapsed
```

CBS COPENHAGEN BUSINESS SCHOOL
HANDELSHØJSKOLEN

QUESTIONS?

Kristoffer Gulmark Poulsen & Lars Nondal
CBS

EQUIS ACCREDITED AACSB ACCREDITED ASSOCIATION OF MBAS ACCREDITED CEMS PIM