CBS COPENHAGEN BUSINESS SCHOOL
HANDELSHØJSKOLEN

HPC & PARALLEL PROGRAMMING IN PYTHON

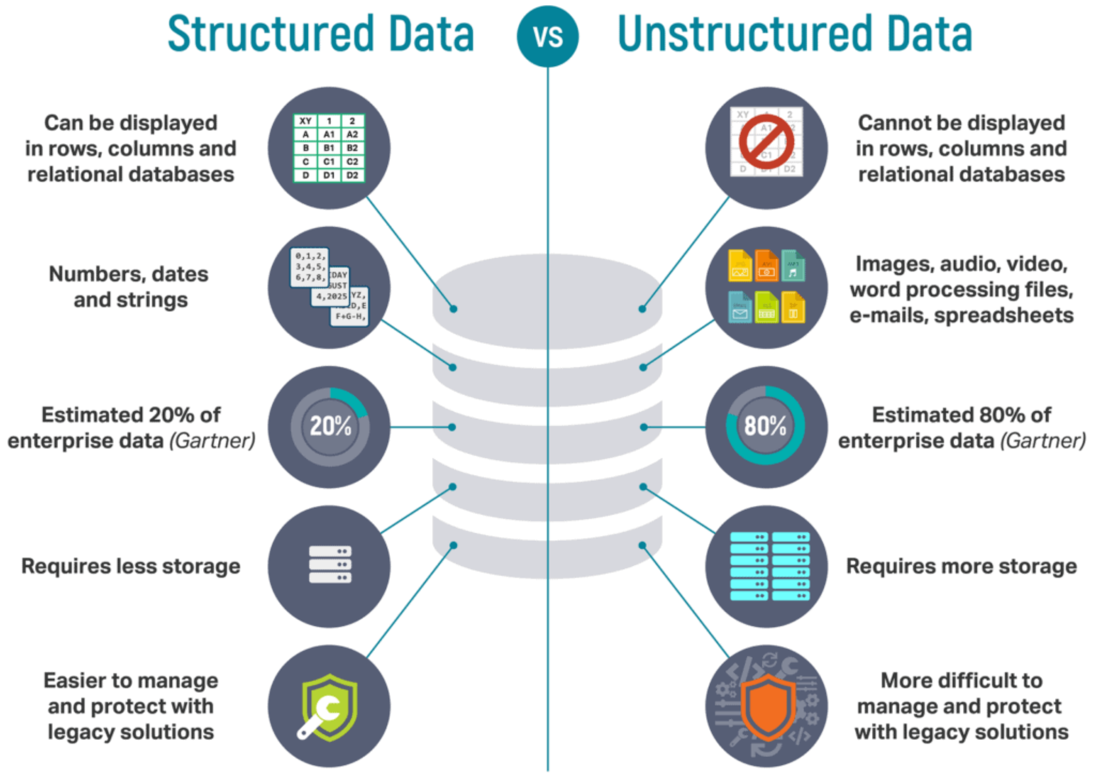
New cloud computing possibilities for researchers & students

Kristoffer Gulmark Poulsen & Lars Nondal
CBS

EQUIS ACCREDITED AACSB ACCREDITED ASSOCIATION OF MBAS ACCREDITED CEMS PIM

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

- Basic theory of parallel programming
- Parallel programming basics within Python
- Parallelization of a ML models scikit-learn framework.
- Distributed parallelization on a SLURM Cluster.

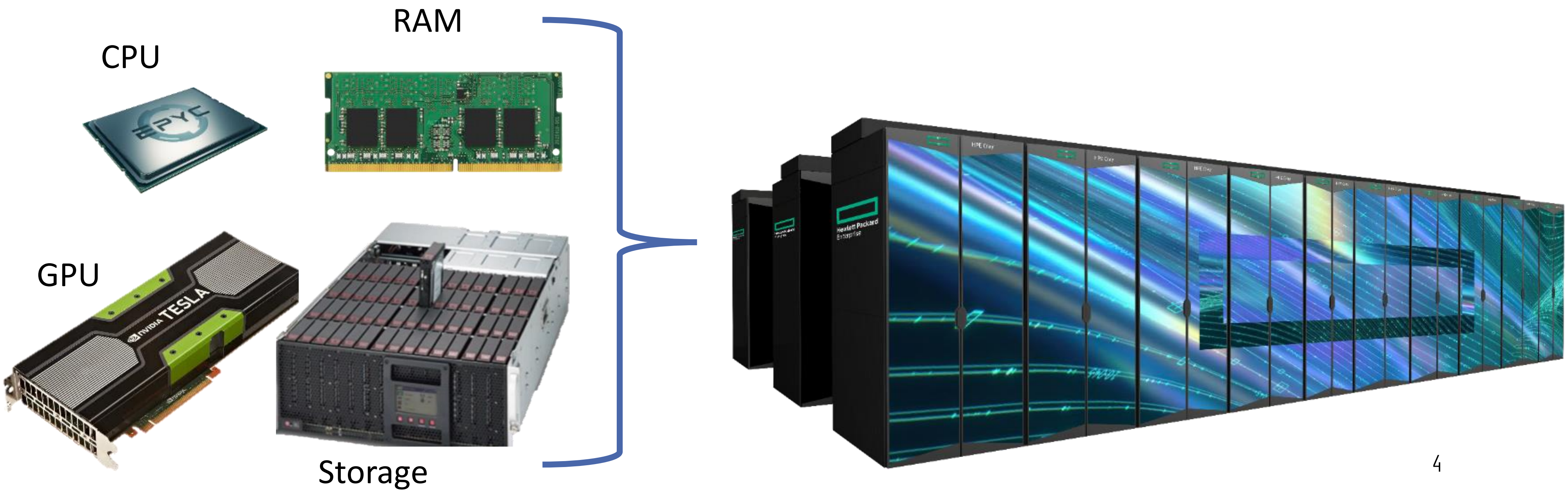
- https://cbs-hpc.github.io/

What is High Performance Computing (supercomputer)?

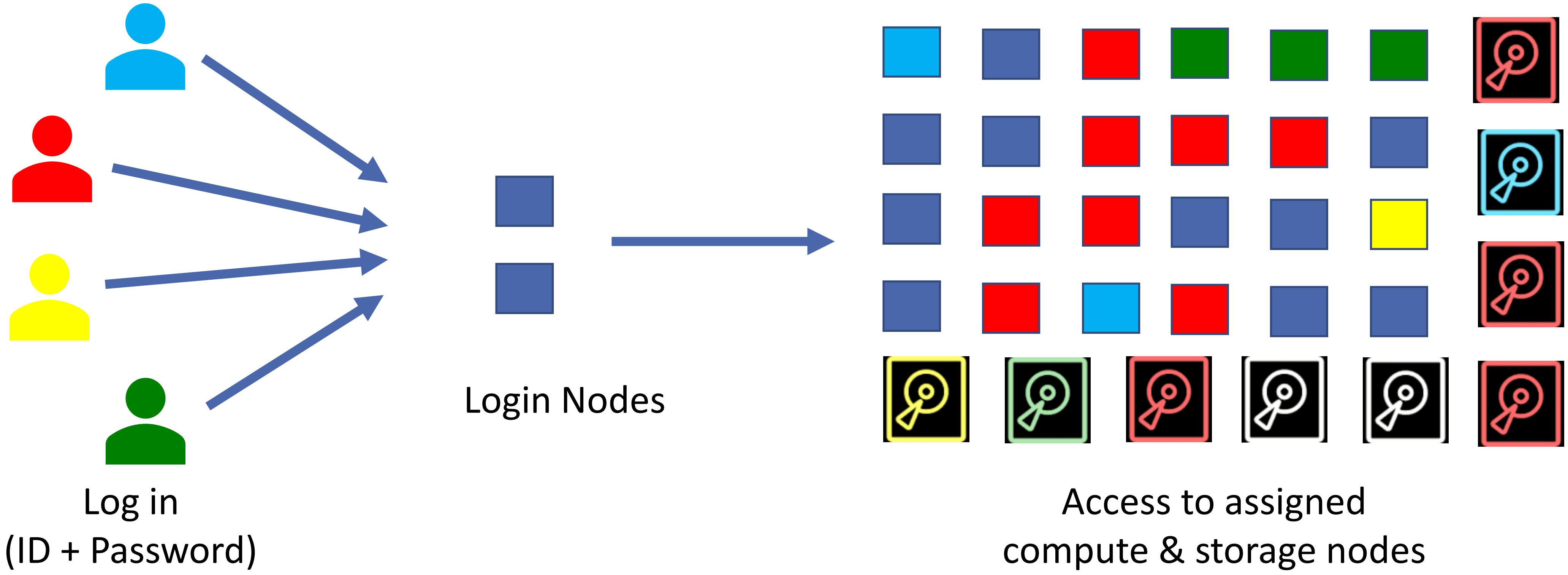
- Network of processors, hard drives & other hardware

Hardware

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

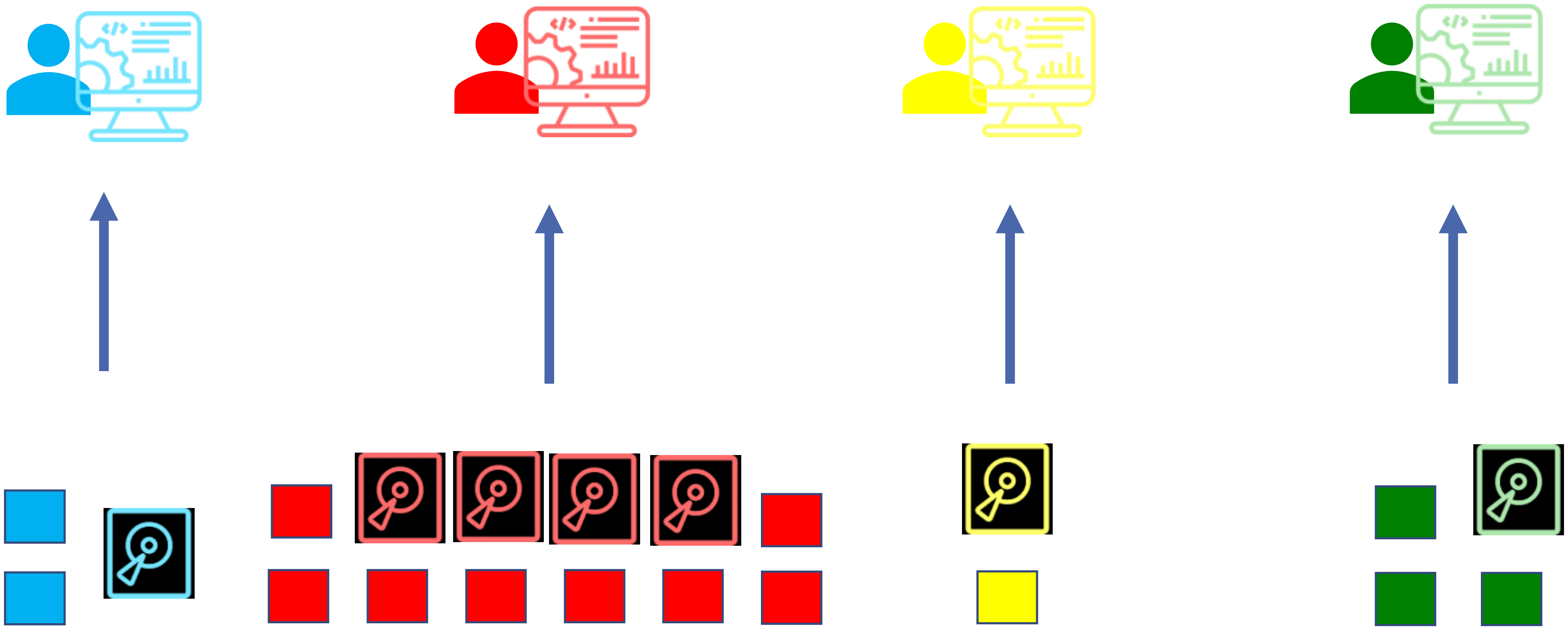


Accessing an HPC...



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

Type 1: 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.

Type 1 (SDU, AAU)

Interactive HPC

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

CBS

Support at CBS

Local CBS support

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

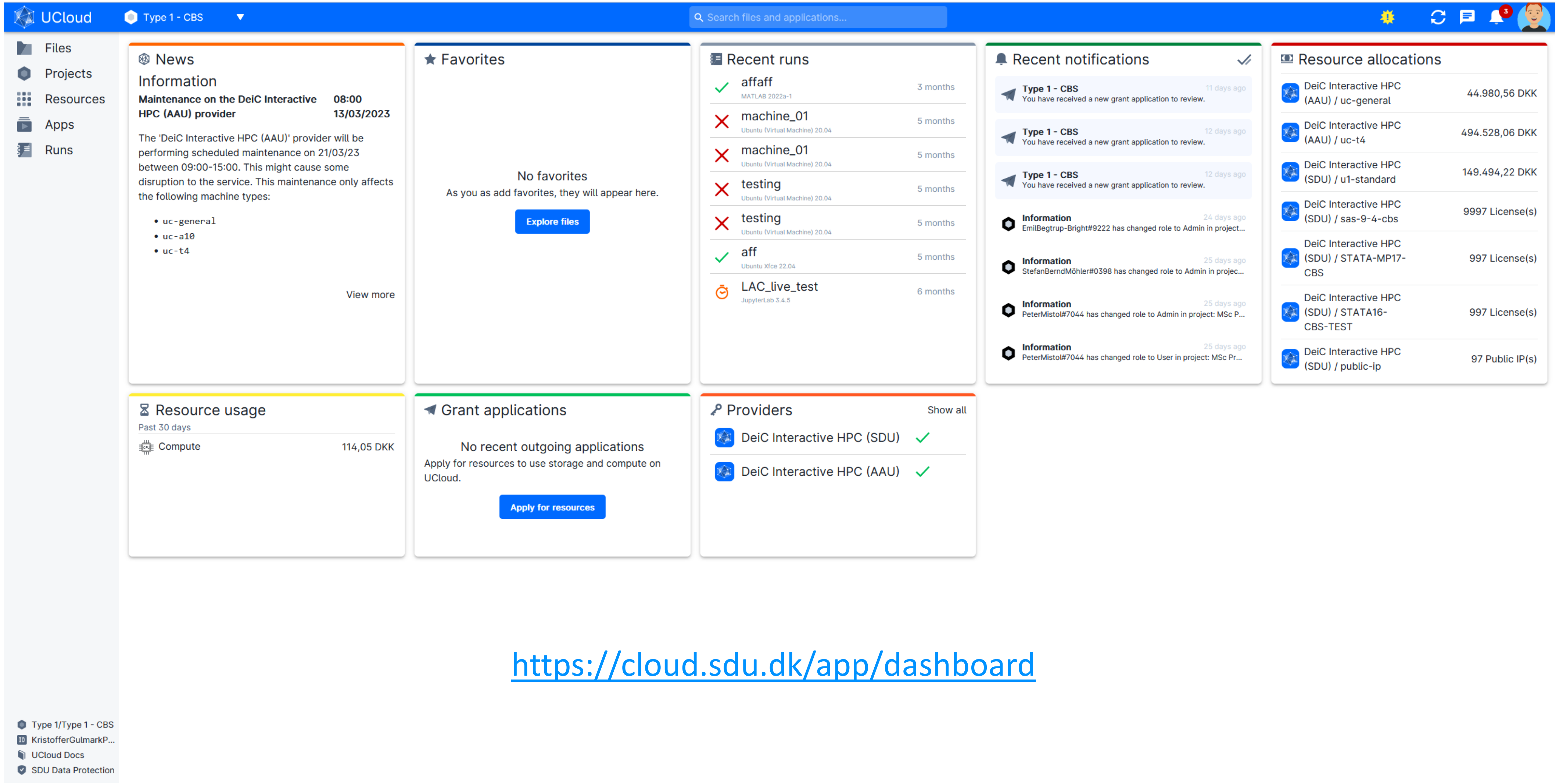
User support: Advising and granting resources, technical problems.

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

CBS

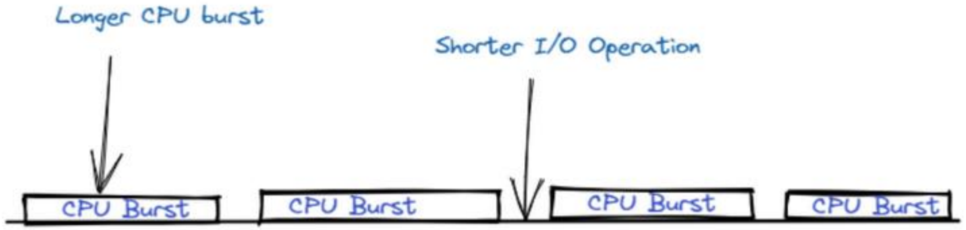
Why is it taking so long?

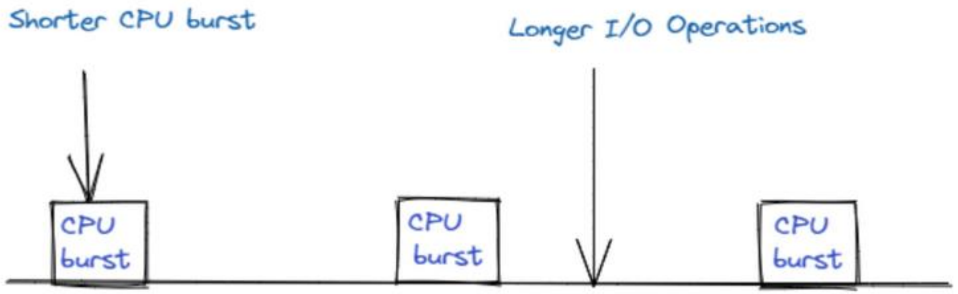
Computation can be slow for one of three reasons:

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.

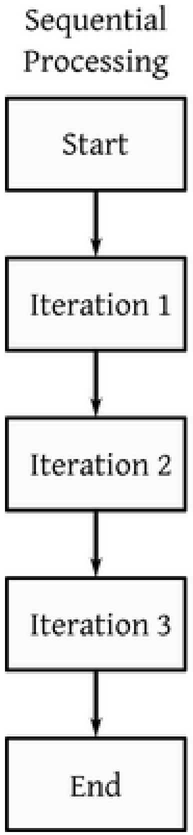




Parallel Programming

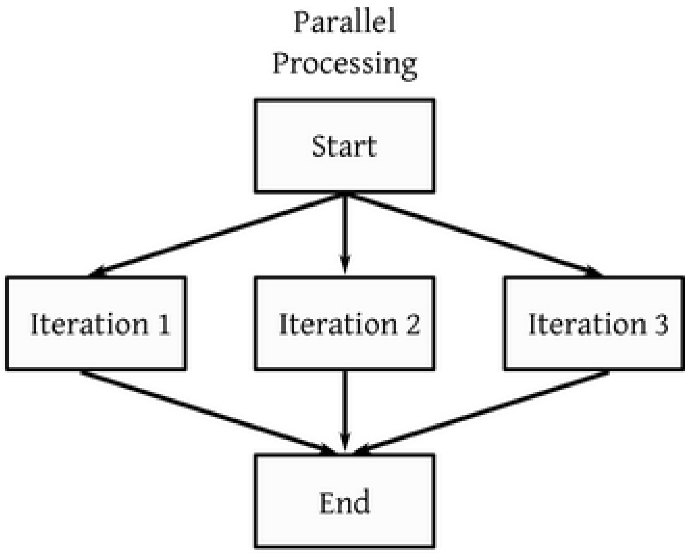
Sequential Computing

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



Parallel Computing

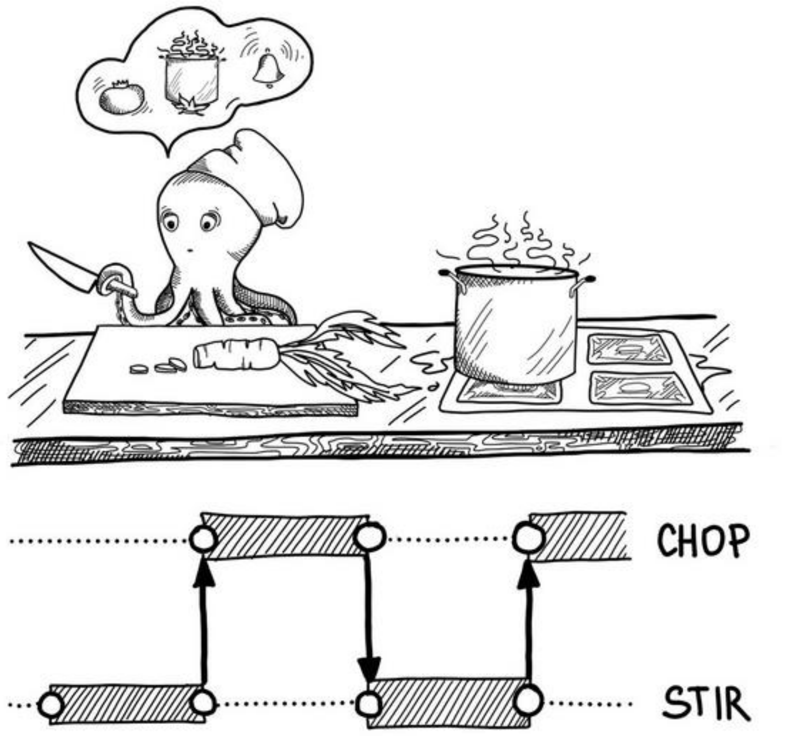
- Multi-core processor
- Multiple tasks which runs overlapping.
- Synchronous/Asynchronous



Parallel Programming

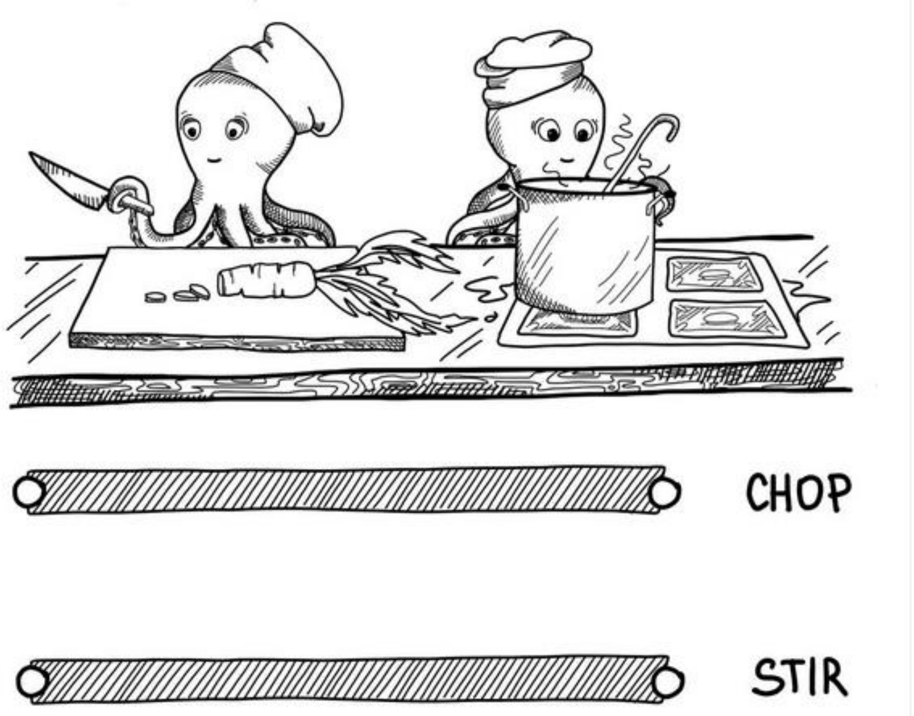
Sequential Computing

- 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

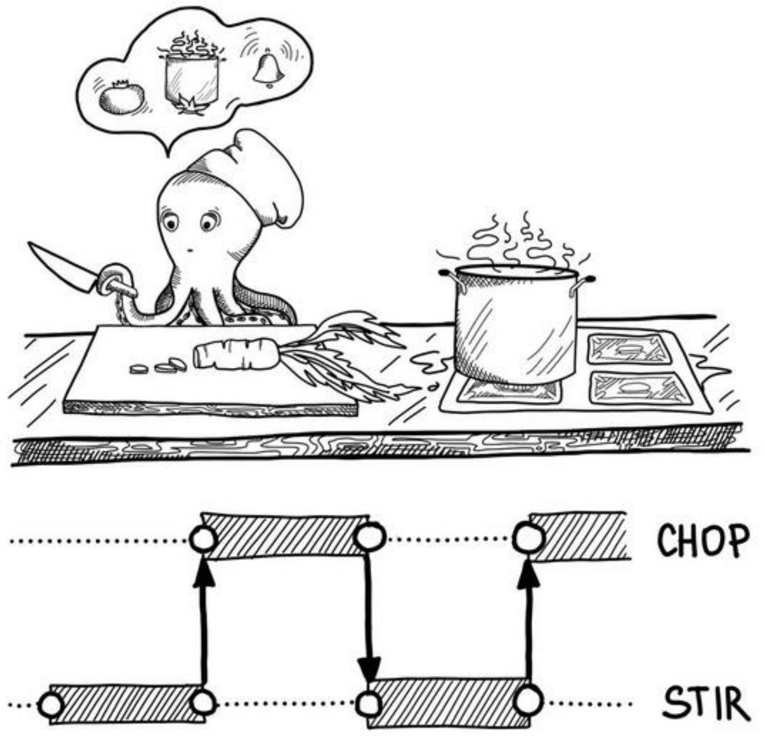


CBS

Parallel Programming

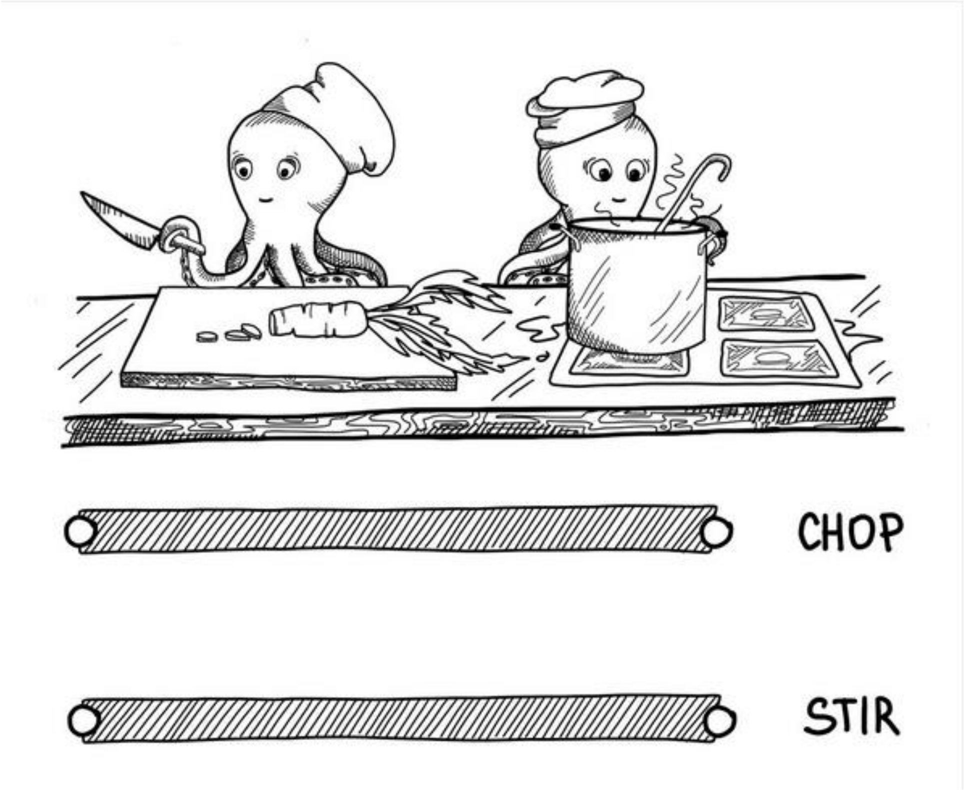
Concurrency

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

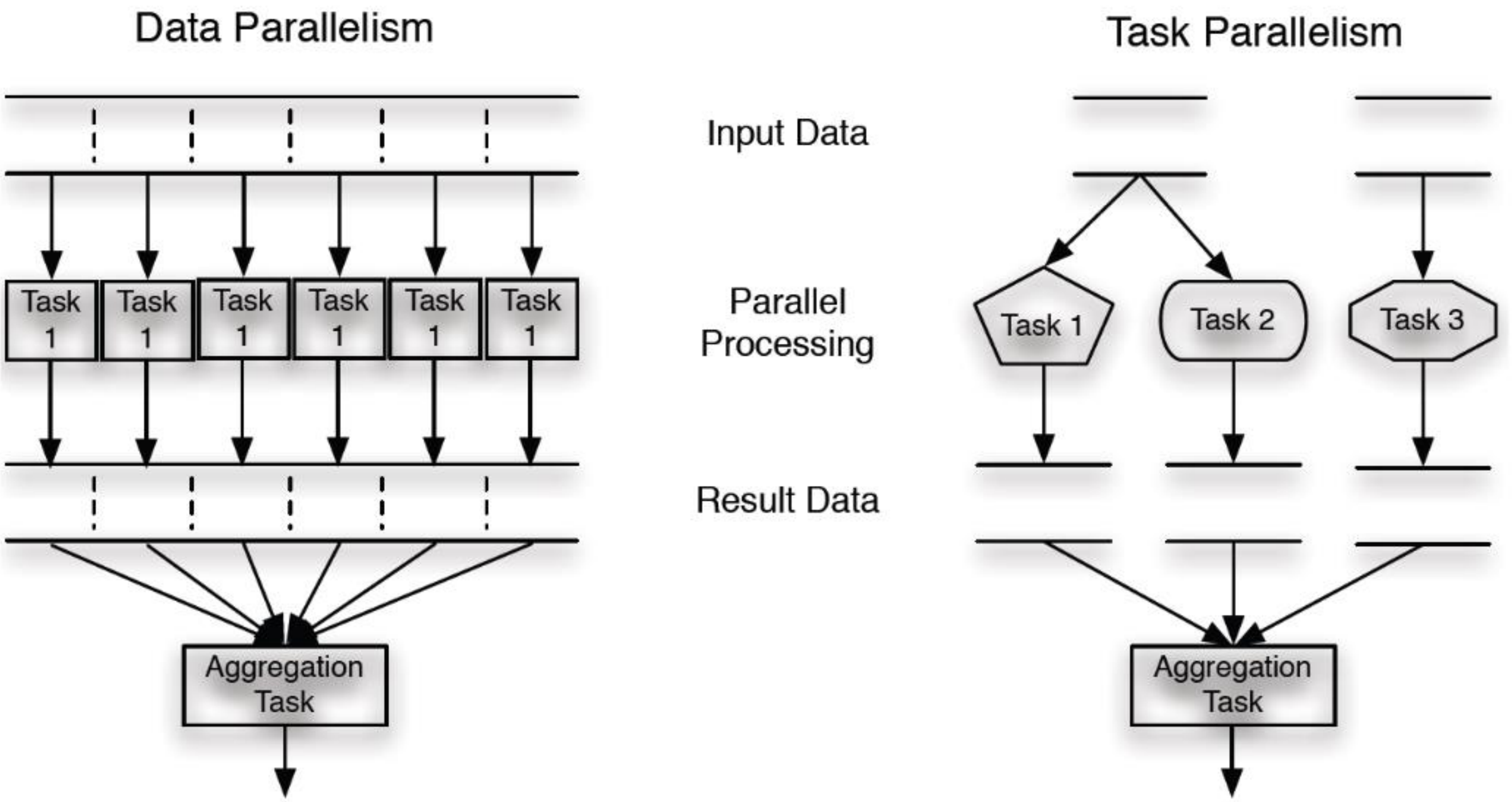


Parallelism

- 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



Parallel Programming



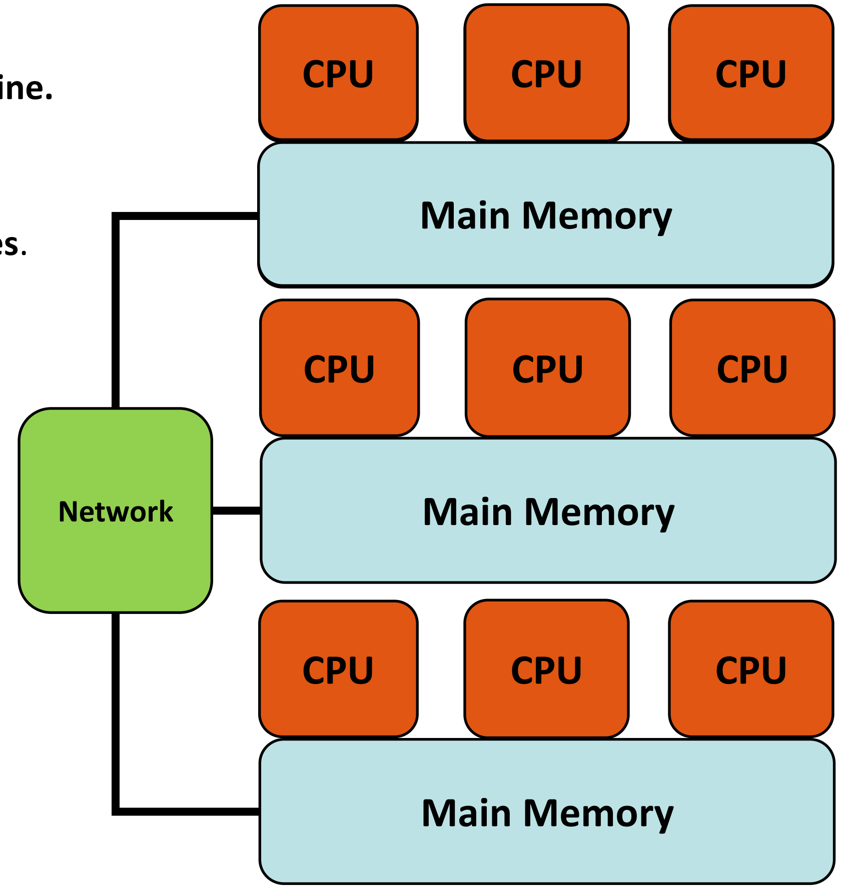
Models for Parallel Programming

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.

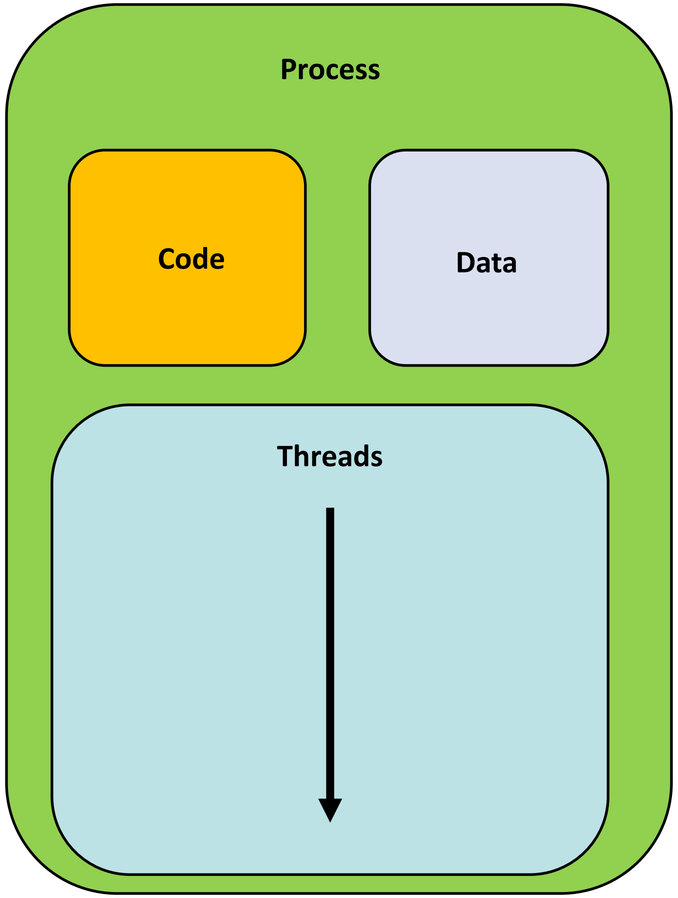
Explicit Parallelism - is written into the source code by the Programmer.



CBS

Terminology

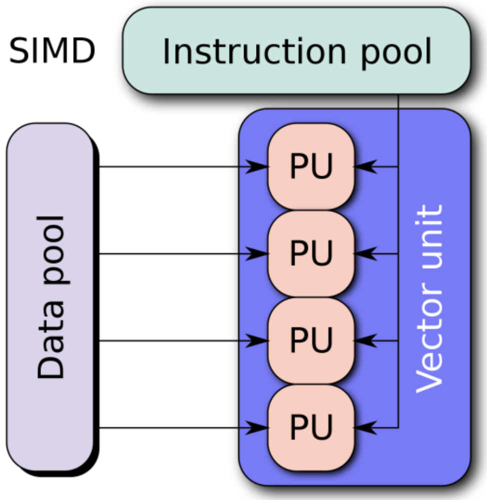
- **Embarrassing/ Perfectly Parallel -** the tasks can be run independently, and they don't need to communicate.
- **Process**: Execution of a program . A given executable (e.g., Python or R) may start up multiple processes.
- **Thread**: Path of execution within a single process.
- **Interpreted -** 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)**



SIMD & Multi-Threading

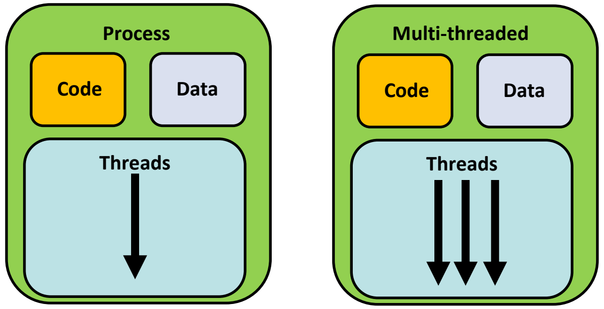
Single Instruction, Multiple Data (SIMD)

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



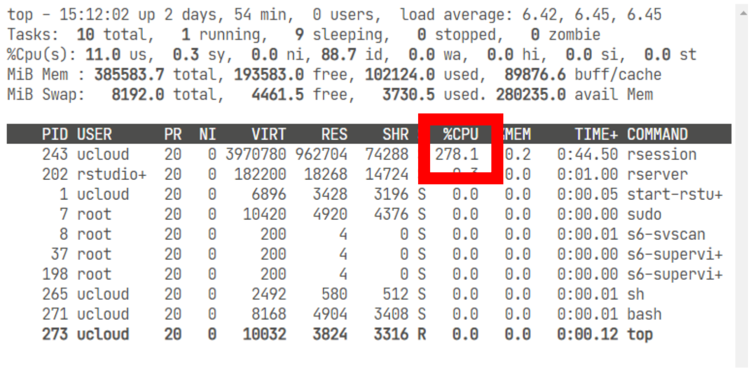
Multi-Threading

- **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
```

SIMD & Multi-Threading in Python and R

SIMT is achieved in several ways:

Through external libraries

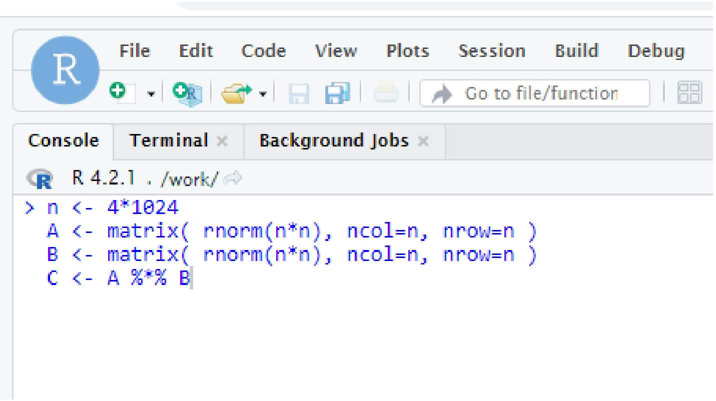
- Written in other languages (e.g. C, C++, Fortran) that run multi-threaded.
- Linear algebra routines (BLAS & LAPACK) implemented in libraries such as MKL, OpenBLAS or BLIS.
- NumPy, SciPy and Pandas
- built-in R functions

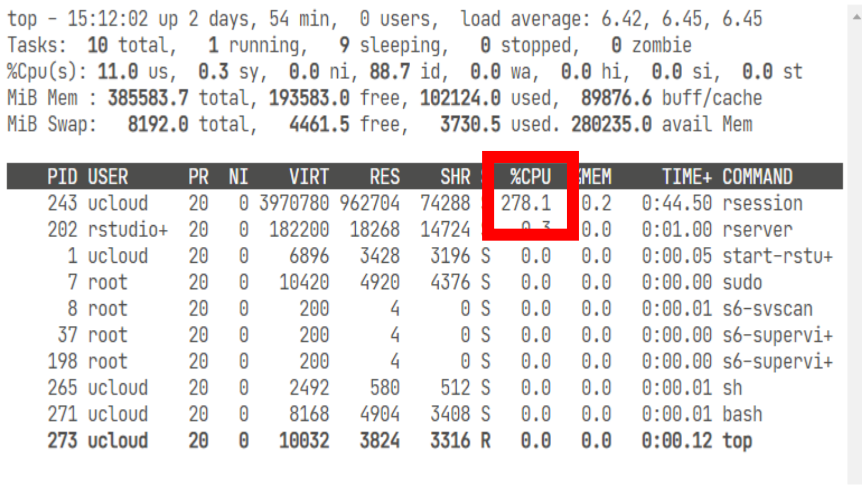
“Static Compilers”

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

Dynamic/JIT Compilers:

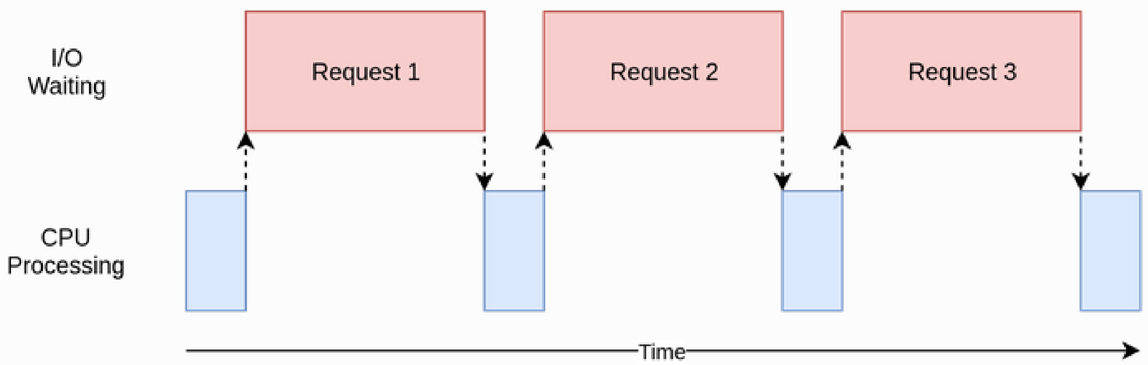
- Numba
- JITR





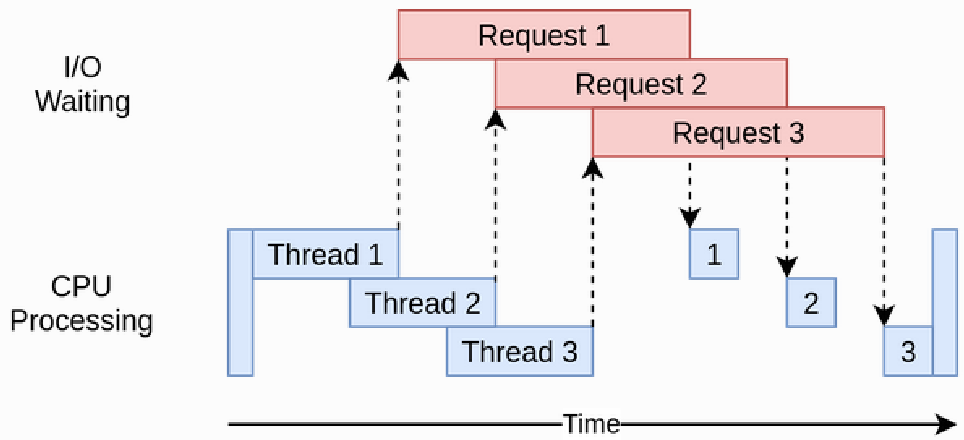
Multi-Threading I/O

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

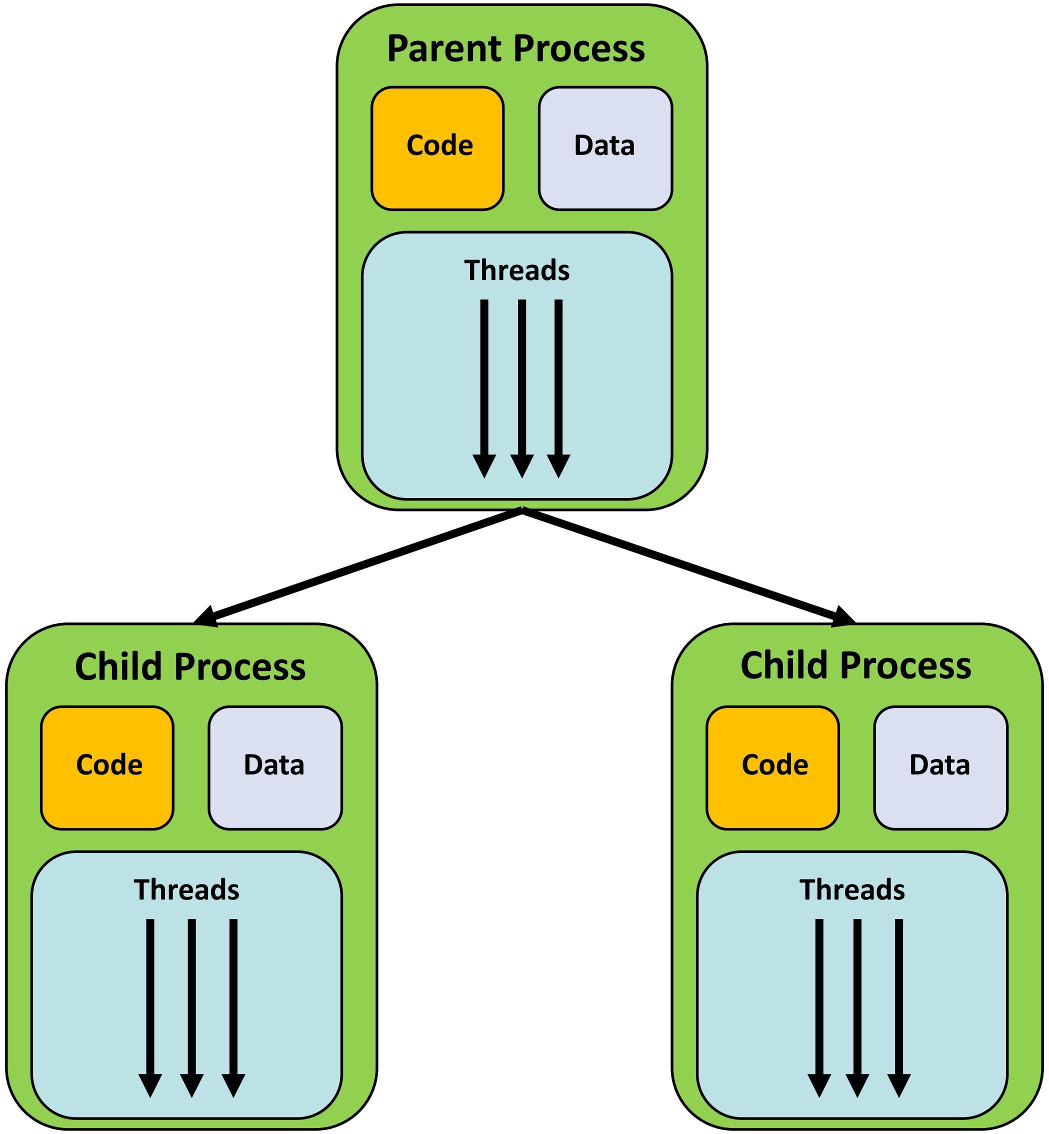
Multi-Processing

Fork

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

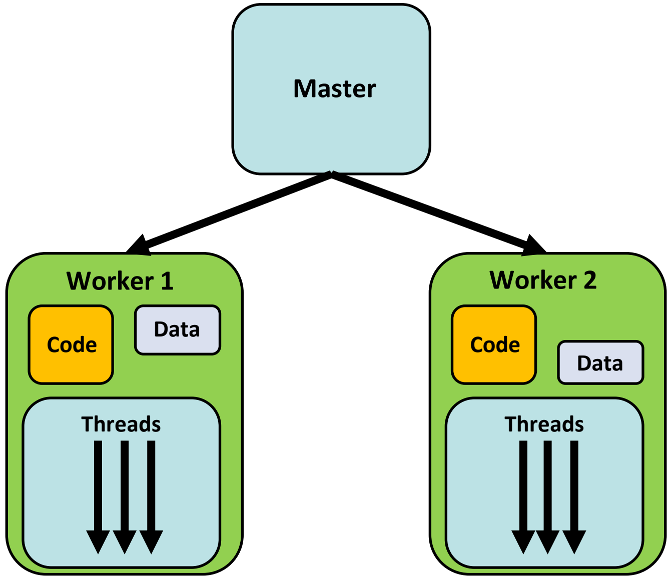
Spawn/Socket (PSOCK)

- 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



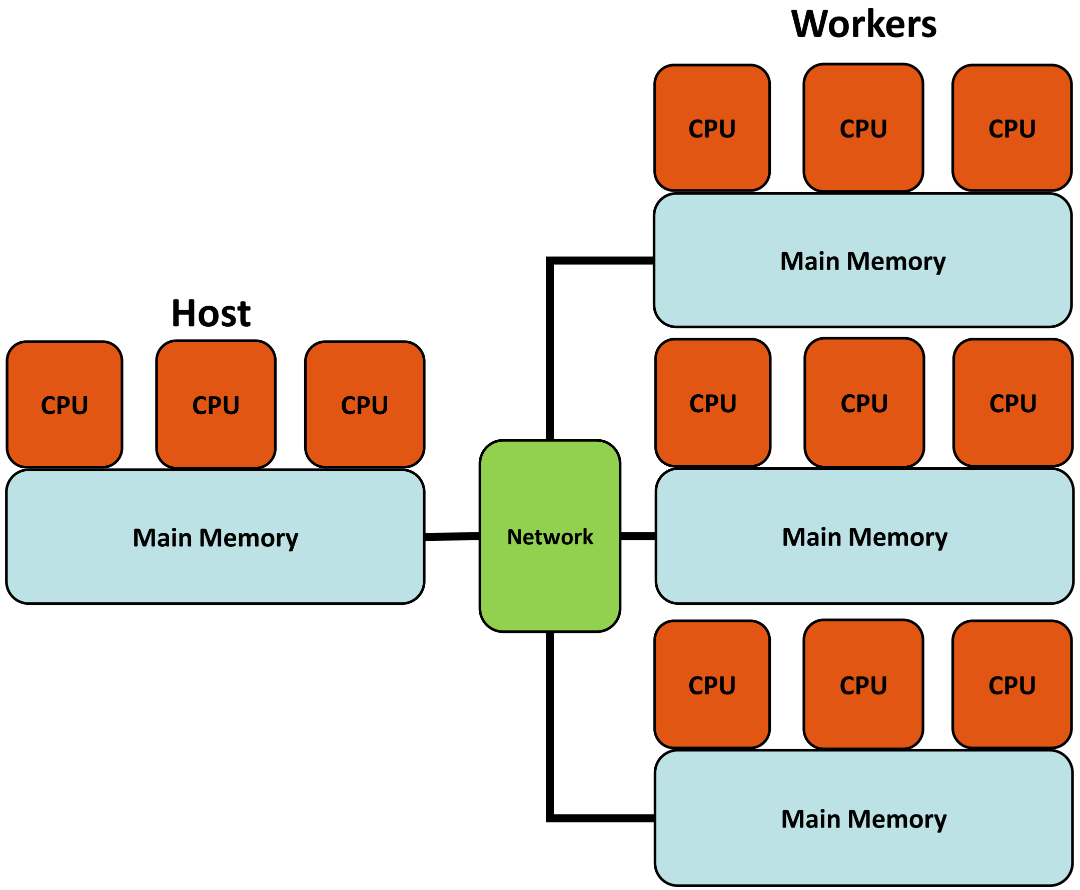
Big chunks are generally better than little chunks

```
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)

COPENHAGEN BUSINESS SCHOOL
HANDELSHØJSKOLEN

PARALLEL PROGRAMMING IN PYTHON

Kristoffer Gulmark Poulsen & Lars Nondal
CBS

Python Libraries - Overview

Built-in Libraries

- *Threading*
- *Multiprocessing*
- *concurrent.futures*

Compilers

- *Numba*

Parallelization *Libraries*

- *Joblib*
- *Loky*
- *Ipyparallel*
- *Ray*
- *Dask*

AI/ML Frameworks

- *Scikit-Learn*
- *Pytorch (torch.multiprocessing ,torch.distributed)*
- *Tensorflow*

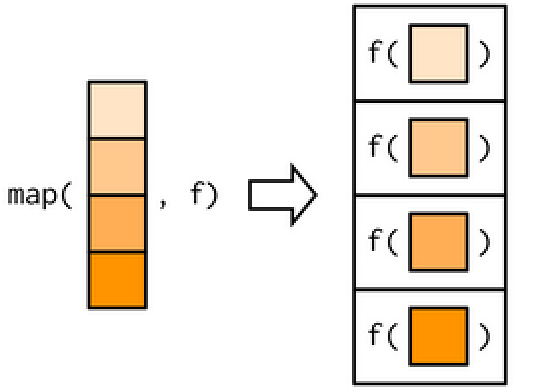
Iterations

There are two styles of iterations

```
for i in range(3):
np.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()**

CBS

Python Library - *Numba*

- Numba a dynamic just-in-time (JIT) compiler.
- Write a pure Python function which can be JIT compiled to native machine instructions.
- Similar in performance to C, C++ and Fortran, by simply adding the decorator @jit in your function.
- @jit compilation adds overhead to the runtime of the function (first time it is run).
- CPU and GPU support.

```
import numba

# Define a function to be JIT compiled
@numba.jit
def my_function(x):
    y = x ** 2 + 2 * x + 1
    return y

# Call the function
result = my_function(5)
print(result)
```

```
import math
import numba
import GPUtil

# No Compiling
def f(x,y):
    return math.pow(x,3.0) + 4*math.sin(y)

# JIT Compiling (CPUs)
@numba.vectorize([numba.float64(numba.float64, numba.float64)], target='cpu')
def f_numba_cpu(x,y):
    return math.pow(x,3.0) + 4*math.sin(y)

# JIT Compiling (GPUs)
if GPUtil.getAvailable():
    @numba.vectorize([numba.float64(numba.float64, numba.float64)],
target='cuda')
    def f_numba_gpu(x,y):
        return math.pow(x,3.0) + 4*math.sin(y)
```

CBS

Python Library - *Threading*

- Multi-theading
- Concurrent not parallel - **subject to the GIL**
- Can increase speed for I/O-bound applications.
- Single-machine

Functions:

- .Thread()
- .start()
- .join()

```python
import threading as th

def print_cube(num):
    # function to print cube of given num
    print("Cube: {}" .format(num * num * num))

def print_square(num):
    # function to print square of given num
    print("Square: {}" .format(num * num))

if __name__ =="__main__":
    # creating thread
    t1 = th.Thread(target=print_square, args=(10,))
    t2 = th.Thread(target=print_cube, args=(10,))

    # starting thread 1
    t1.start()
    # starting thread 2
    t2.start()

    # wait until thread 1 is completely executed
    t1.join()
    # wait until thread 2 is completely executed
    t2.join()

    # both threads completely executed
    print("Done!")
```

Python Library - *Multiprocessing*

Methods:

- ‘spawn’
- ‘fork’
- Single-machine

Functions:

P = mp.Process(target=x, args=y)

P.start()

P.join()

```
import multiprocessing as mp

def print_cube(num):
    # function to print cube of given num
    print("Cube: {}" .format(num * num * num))

def print_square(num):
    # function to print square of given num
    print("Square: {}" .format(num * num))

if __name__ =="__main__":
    mp.set_start_method('spawn')
    # mp.set_start_method('fork')

    # creating process
    p1 = mp.Process(target=print_square, args=(10,))
    p2 = mp.Process(target=print_cube, args=(10,))

    # starting process 1
    p1.start()
    # starting process 2
    p2.start()

    # wait until process 1 is completely executed
    p1.join()
    # wait until process 2 is completely executed
    p2.join()

    # both process completely executed
    print("Done!")
```

Python Library - *Multiprocessing*

Creating a worker pool:

- myPool = Pool(nworkers)

Functions:

- myPool.apply()
- myPool.apply_async()
- myPool.map()
- myPool.map_async()
- myPool.imap()
- myPool.imap_unordered()
- myPool.starmap()
- myPool.starmap_async()

```
import multiprocessing as mp

def print_cube(num):
    # function to print cube of given num
    print("Cube: {}" .format(num * num * num))

X = [100,500, 1000, 3044, 233]

# protect the entry point
if __name__ == '__main__':
    mp.set_start_method('spawn')
    # mp.set_start_method('fork')

    # create a process pool with 4 workers
    mypool = mp.Pool(processes=4)
    value = mypool.map(print_cube,X)
```

```
if __name__ == '__main__':
    mp.set_start_method('spawn')
    # mp.set_start_method('fork')

    with mp.Pool(processes=i) as mypool:
            value = mypool.map(cube, X)
```

CBS

Python Library - *concurrent.futures*

Multiprocessing Pool vs ProcessPoolExecutor

https://superfastpython.com/multiprocessing-pool-vs-processpoolexecutor/

ThreadPoolExecutor vs. Thread

https://superfastpython.com/threadpoolexecutor-vs-threads/#Similarities_Between_ThreadPoolExecutor_and_Thread

Concurrent not parallel- **subject to the GIL**

```
# create a thread pool
executor = ThreadPoolExecutor(max_workers=10)

# create a process pool
executor = ProcessPoolExecutor(max_workers=10)

# submit a task to the pool and get a future immediately
future = executor.submit(task, item)

# get the result once the task is done
result = future.result()

# Shutdown pool
executor.shutdown()
```

```
with ThreadPoolExecutor(max_workers=10) as executor:
# call a function on each item in a list and process results
for result in executor.map(task, items):
# process result...
# ...
# shutdown is called automatically
```

Python Library - Scikit-Learn

Depending on the type of estimator parallelism:

OpenMP:
Is used to parallelize code written in Cython or C, relying on multi-threading exclusively. By default, the implementations using OpenMP will use as many threads as possible, i.e. as many threads as logical cores.

```
OMP_NUM_THREADS=4 python my_script.py
```

MKL, OpenBLAS or BLIS:
Scikit-learn relies heavily on NumPy and SciPy, which internally call multi-threaded linear algebra routines (BLAS & LAPACK) implemented in libraries such as MKL, OpenBLAS or BLIS.

```
# You can control the exact number of threads used by BLAS
for each library using environment variables, namely:

MKL_NUM_THREADS  # sets the number of thread MKL uses,
OPENBLAS_NUM_THREADS # sets the number of threads OpenBLAS uses
BLIS_NUM_THREADS # sets the number of threads BLIS uses
```

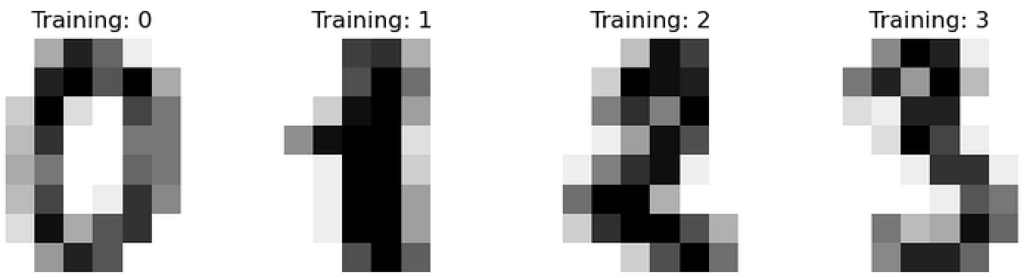
joblib backends:

```
from joblib import parallel_backend

# Default
with parallel_backend('loky'):
with parallel_backend('mulitprocessing'):
with parallel_backend('dask'):
with parallel_backend('ray'):
with parallel_backend('ipyparallel'):
with parallel_backend('threading'):
with parallel_backend('spark'):

    # Your scikit-learn code here
```

CBS

Scikit-Learn – joblib backends

```python
import numpy as np
from joblib import parallel_backend
from sklearn.datasets import load_digits
from sklearn.model_selection import RandomizedSearchCV
from sklearn.svm import SVC

param_space = {
'C': np.logspace(-6, 6, 30),
'gamma': np.logspace(-8, 8, 30),
'tol': np.logspace(-4, -1, 30),
'class_weight': [None, 'balanced'],
}

model = SVC(kernel='rbf')
search = RandomizedSearchCV(model, param_space, cv=10, n_iter=5,verbose=1)
digits = load_digits()
```




```python
with parallel_backend('multiprocessing',n_jobs=2):
 search.fit(digits.data,digits.target)

# Fitting 10 folds for each of 5 candidates, totaling 50 fits
# 8.597755701979622
```

```python
with parallel_backend('multiprocessing',n_jobs=16):
 search.fit(digits.data,digits.target)

# Fitting 10 folds for each of 5 candidates, totaling 50 fits
# 2.2656688350252807
```

```python
with parallel_backend('loky',n_jobs=16):
 search.fit(digits.data,digits.target)

# Fitting 10 folds for each of 5 candidates, totaling 50 fits
# 2.7689956098329276
```

```python
with parallel_backend('threading',n_jobs=16):
 search.fit(digits.data,digits.target)

# Fitting 10 folds for each of 5 candidates, totaling 50 fits
# 1.5711621041409671
```

Scikit-Learn – Ray

```python
import numpy as np
from joblib import parallel_backend
from sklearn.datasets import load_digits
from sklearn.model_selection import RandomizedSearchCV
from sklearn.svm import SVC

param_space = {
'C': np.logspace(-6, 6, 30),
'gamma': np.logspace(-8, 8, 30),
'tol': np.logspace(-4, -1, 30),
'class_weight': [None, 'balanced'],
}

model = SVC(kernel='rbf')
search = RandomizedSearchCV(model, param_space, cv=10, n_iter=5,verbose=1)
digits = load_digits()
```

```python
import ray
from ray.util.joblib import register_ray

# create local ray cluster
ray.init(num_cpus=16)
# connect to cluster
register_ray()
```

```python
with parallel_backend('ray’):
  search.fit(digits.data,digits.target)

# Fitting 10 folds for each of 5 candidates, totaling 50 fits
# 3.9540881011635065
```

```python
ray.shutdown()
```

QUESTIONS?