Lecture 13:

Introduction to Parallel Computing and CUDA

Markus Hohle

University California, Berkeley

Machine Learning Algorithms

MSSE 277B, 3 Units

Fall 2024

"What was initially thought to be a simple process is in fact an incredibly complicated, intricate, and complex system that I've codified and organized into a few easy-to-follow rules that are more difficult to implement than you'd think."

Outline

- **Parallel Processing**

- **Using Map**

- **Using Process**

- **CUDA & PyTorch**

© MARK ANDERSON, WWW.ANDERTOONS.COM

"What was initially thought to be a simple process is in fact an incredibly complicated, intricate, and complex system that I've codified and organized into a few easy-to-follow rules that are more difficult to implement than you'd think."
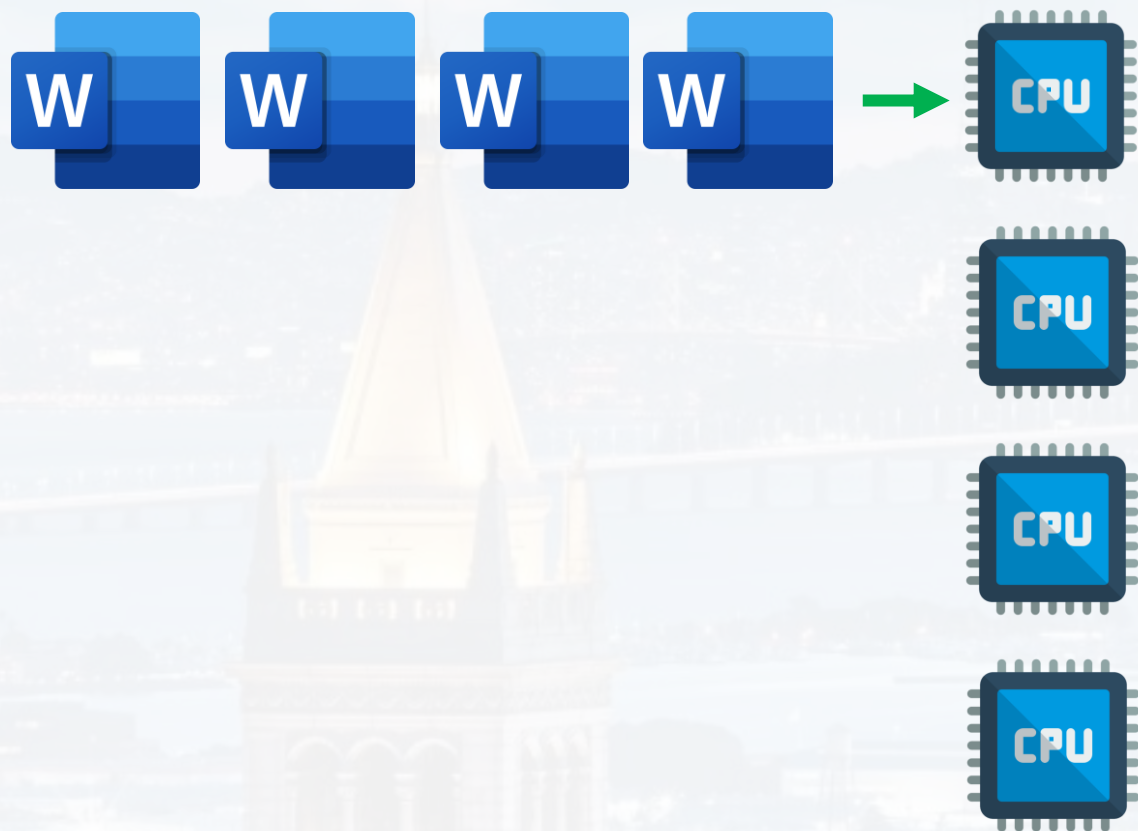
Outline

- Parallel Processing

- Using Map

- Using Process

- CUDA & PyTorch

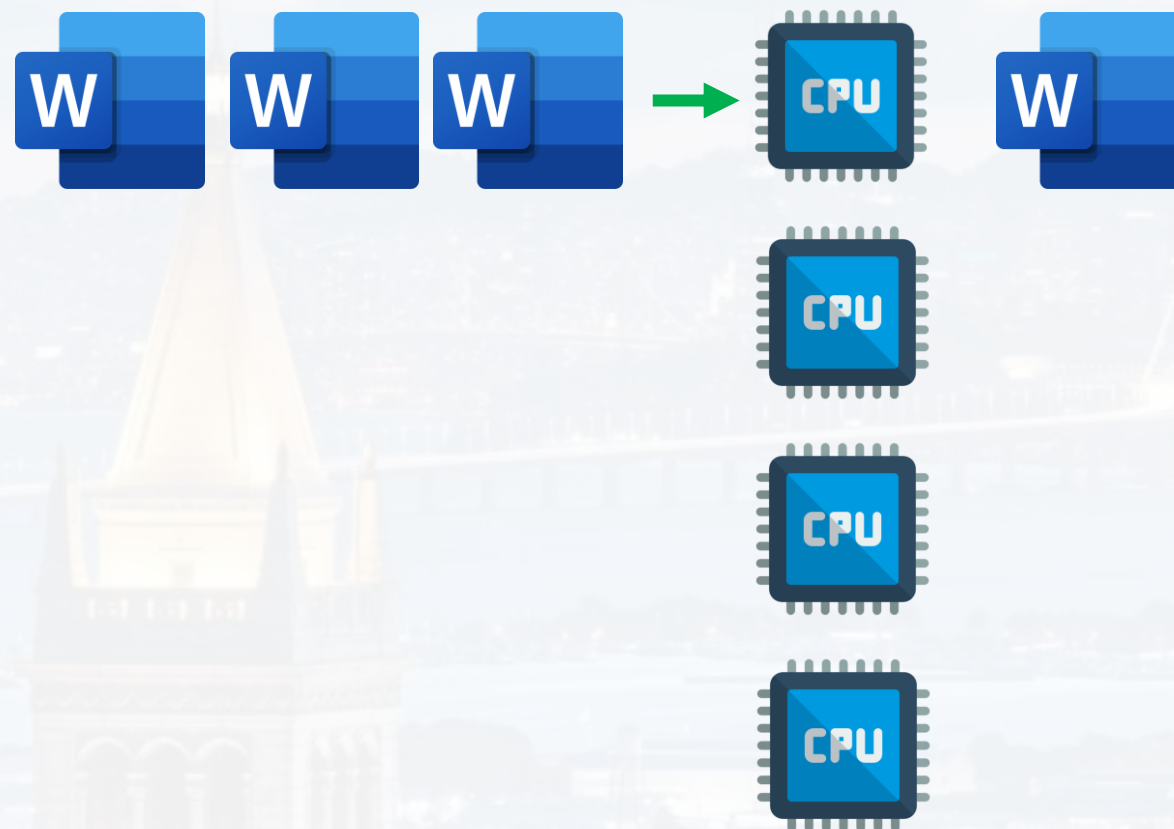even for moderate models and data: $\rightarrow$ computational limits

standard:

Parallel & CUDA:

even for moderate models and data: → computational limits

standard:

Parallel & CUDA:

even for moderate models and data: → computational limits
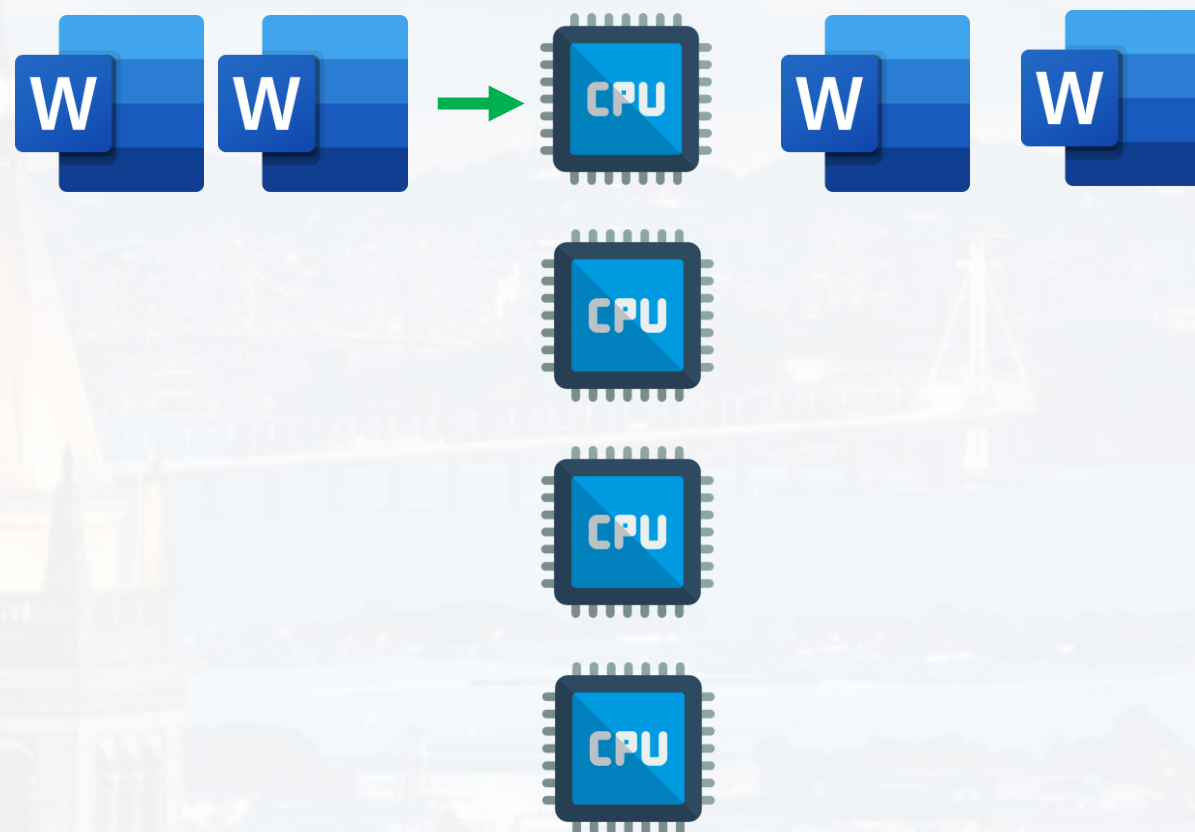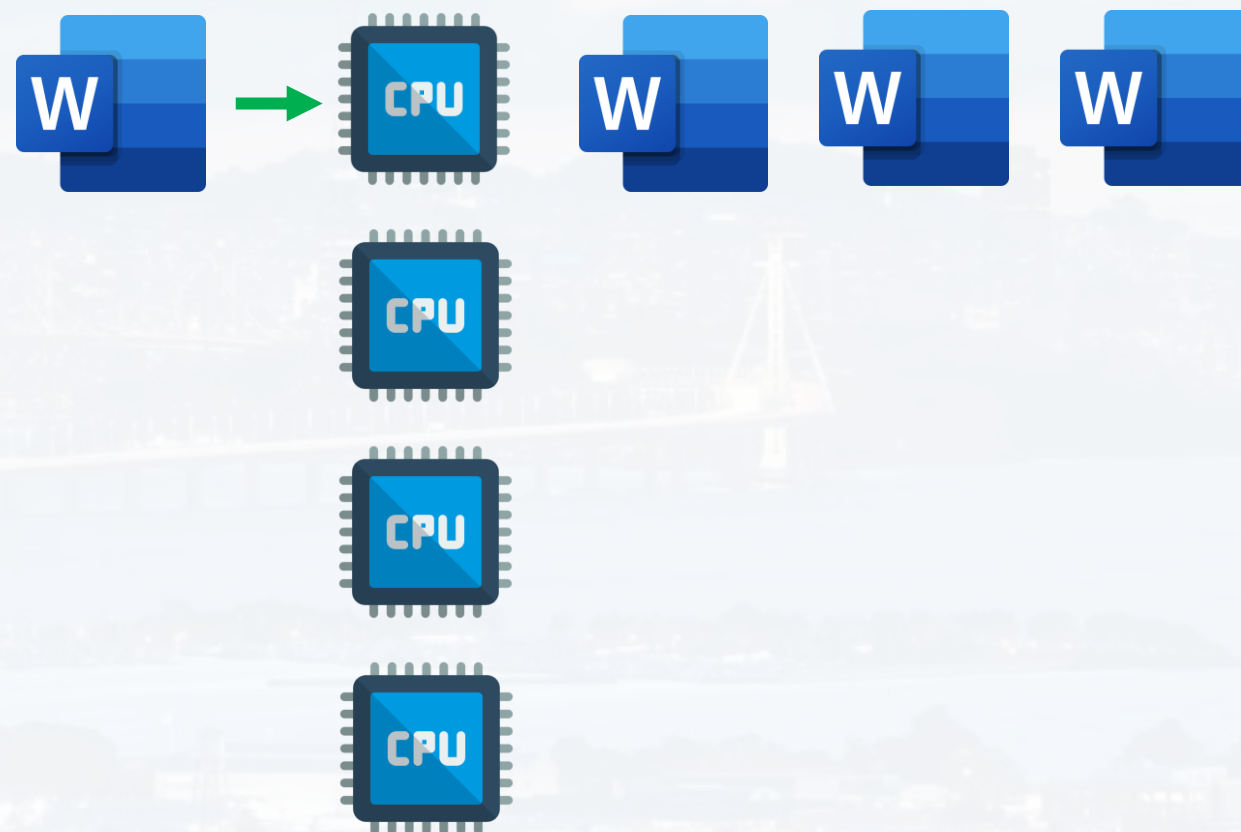
standard:

Parallel & CUDA:

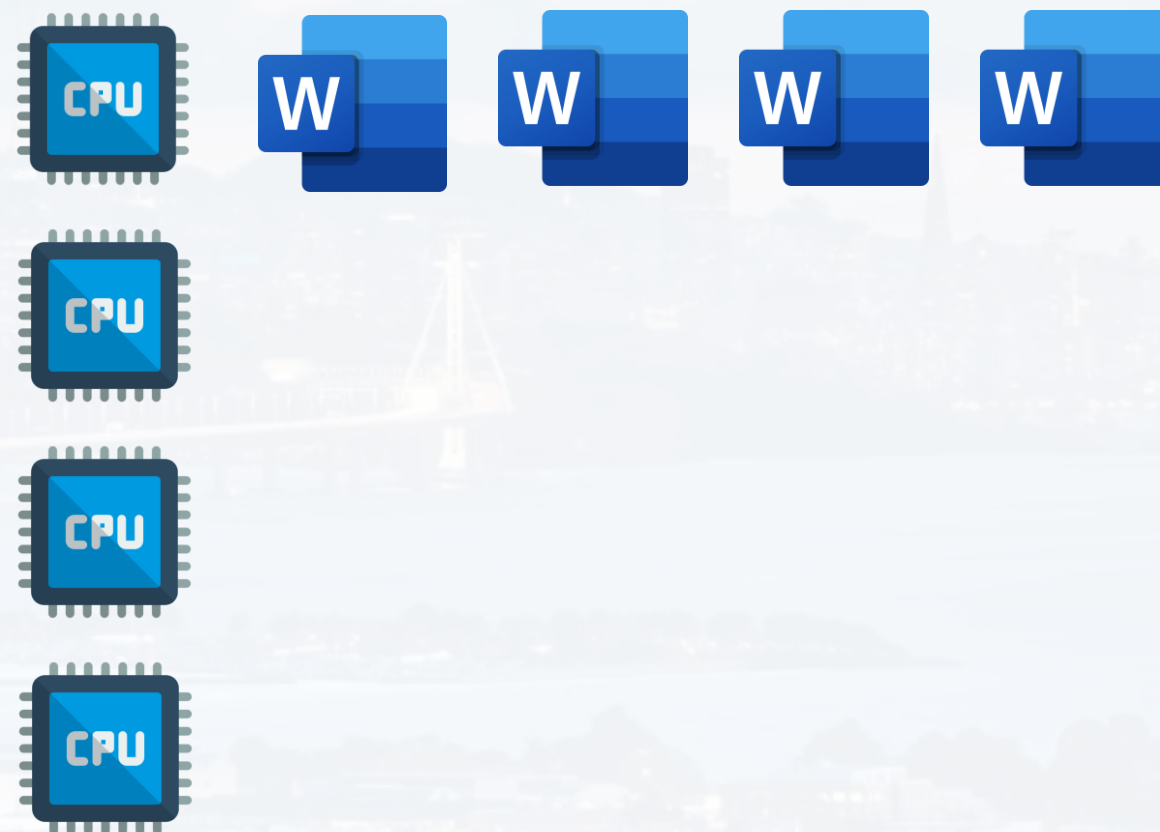even for moderate models and data: → computational limits

standard:

even for moderate models and data: → computational limits
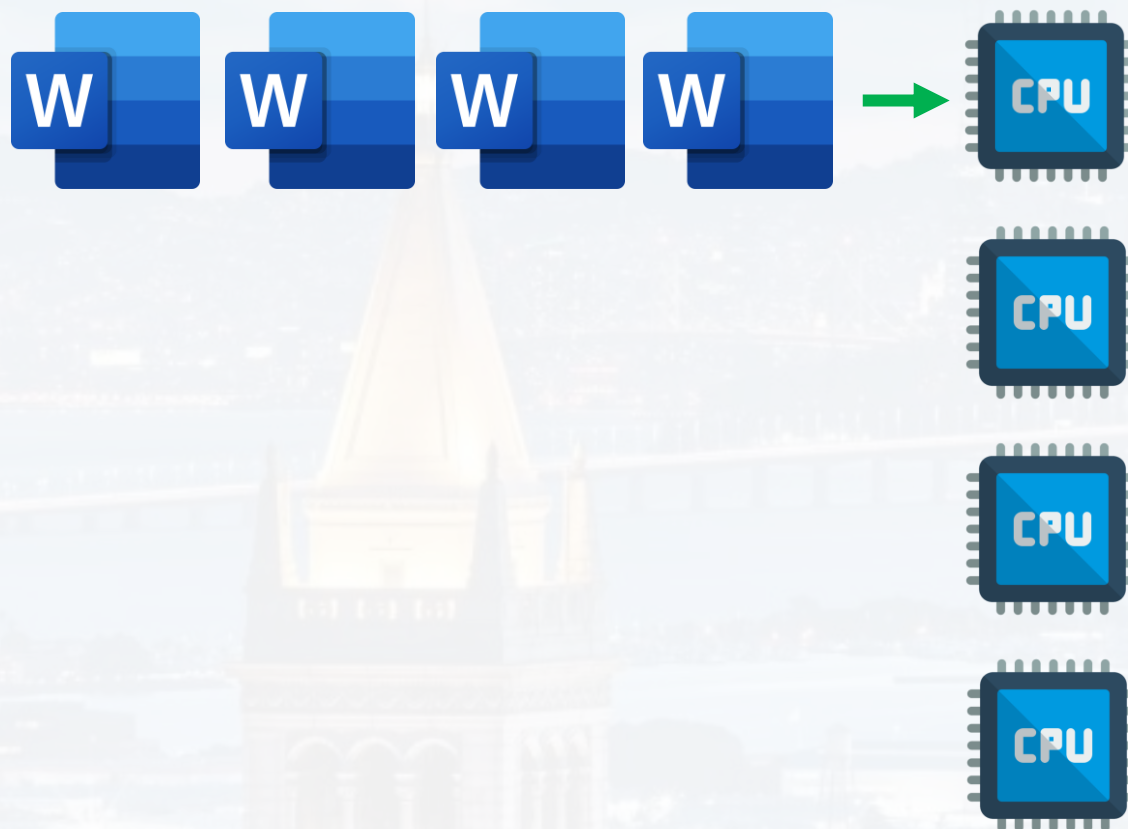
standard:

even for moderate models and data: → computational limits

same program, but different instances: - eg. reading different files, and perform the same analysis
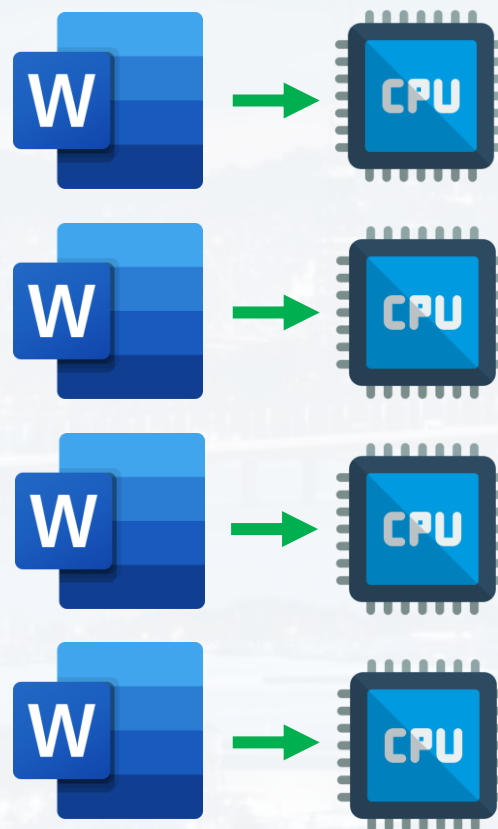- frequency grid search (different sub intervals)

even for moderate models and data: → computational limits

same program, but different instances: - eg. reading different files, and perform the same analysis

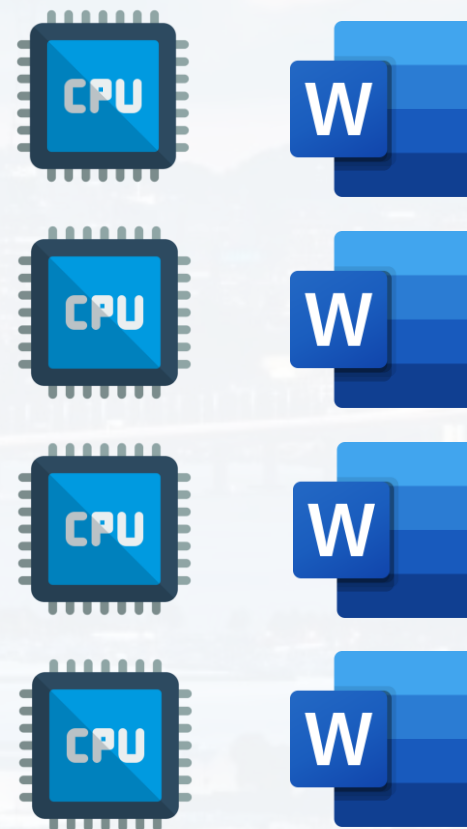   - frequency grid search (different sub intervals)

even for moderate models and data: → computational limits

same program, but different instances: - eg. reading different files, and perform the same analysis

                                                 - frequency grid search (different sub intervals)

even for moderate models and data: → computational limits

same program, but different instances: - eg. reading different files, and perform the same analysis

                                                        - frequency grid search (different sub intervals)
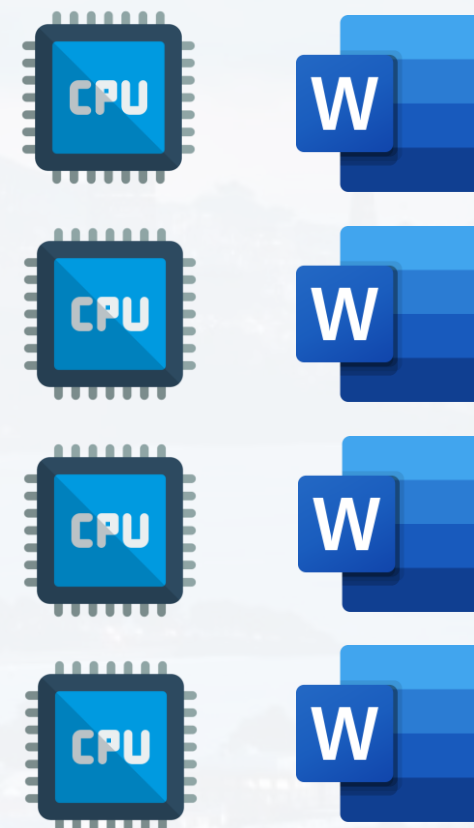
```python
import multiprocessing as mp

Processes = [mp.Process(target = MyFun, args = (List,))\
             for List in list_filenames]

for p in Processes:
    p.start()
```

allocating processes to CPUs

```python
for p in Processes:
    p.join()
```

running and finishing processes
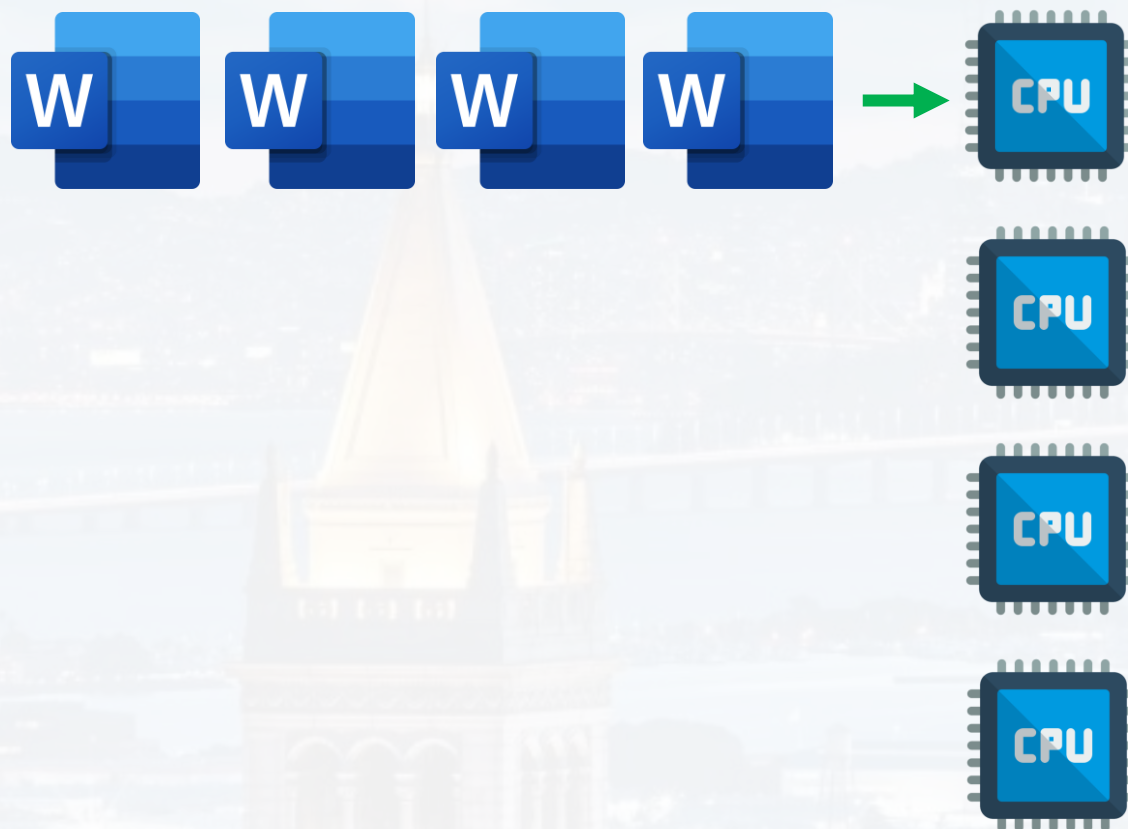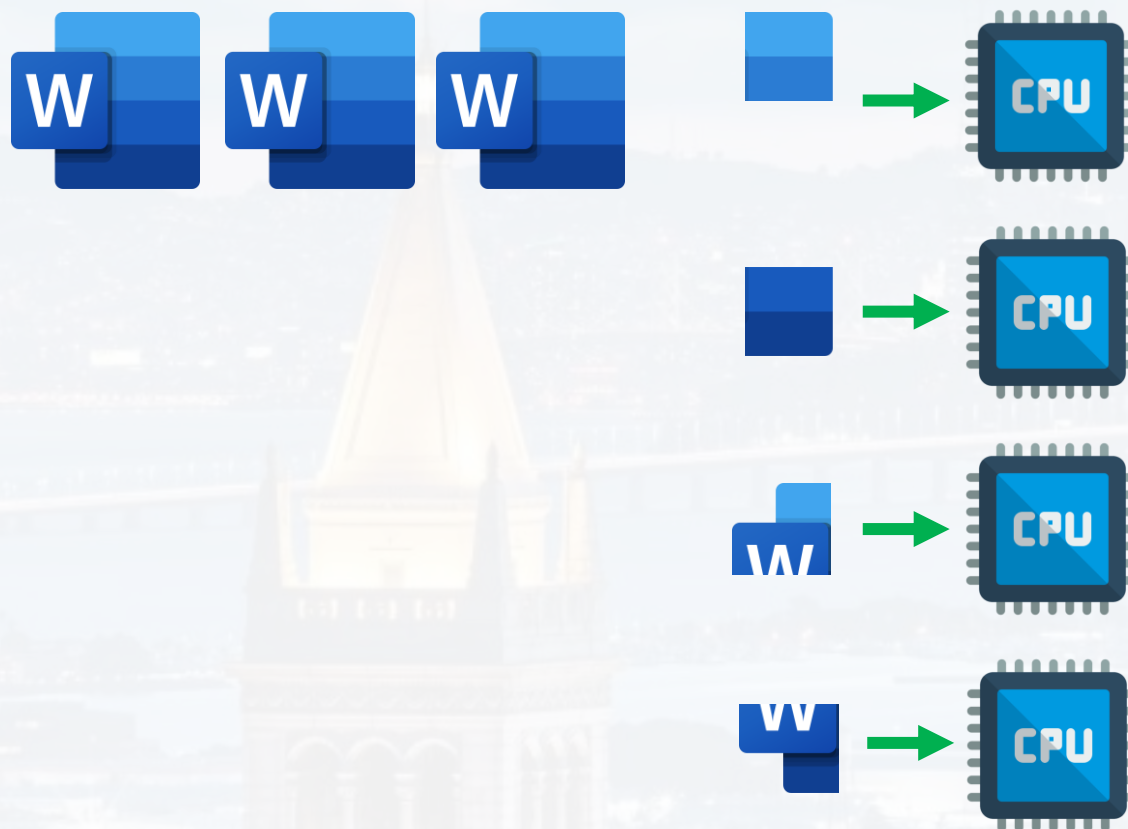
even for moderate models and data: → computational limits

parallelizing the process itself: - eg. a non vectorized function in a for loop

even for moderate models and data: → computational limits

parallelizing the process itself: - eg. a non vectorized function in a for loop
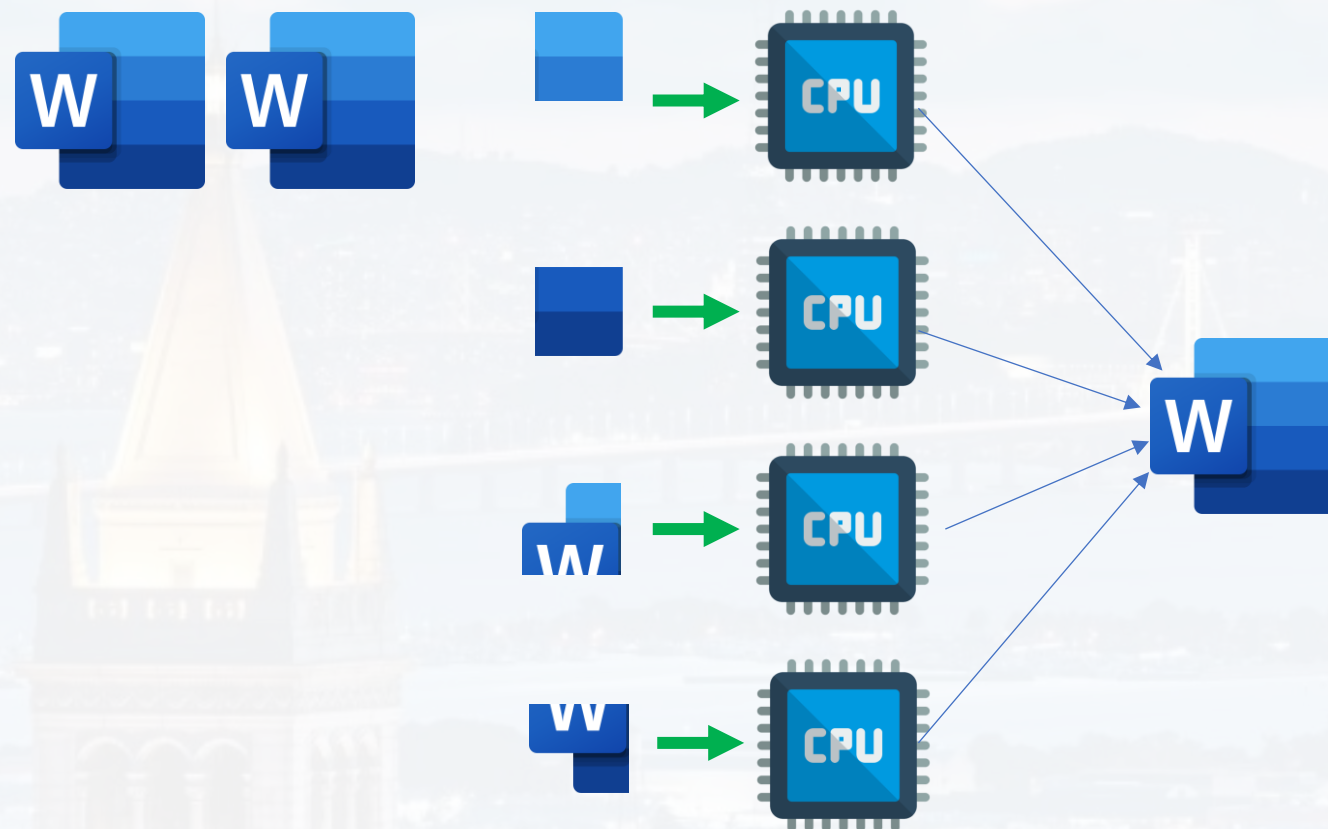
even for moderate models and data: → computational limits

parallelizing the process itself: - eg. a non vectorized function in a for loop
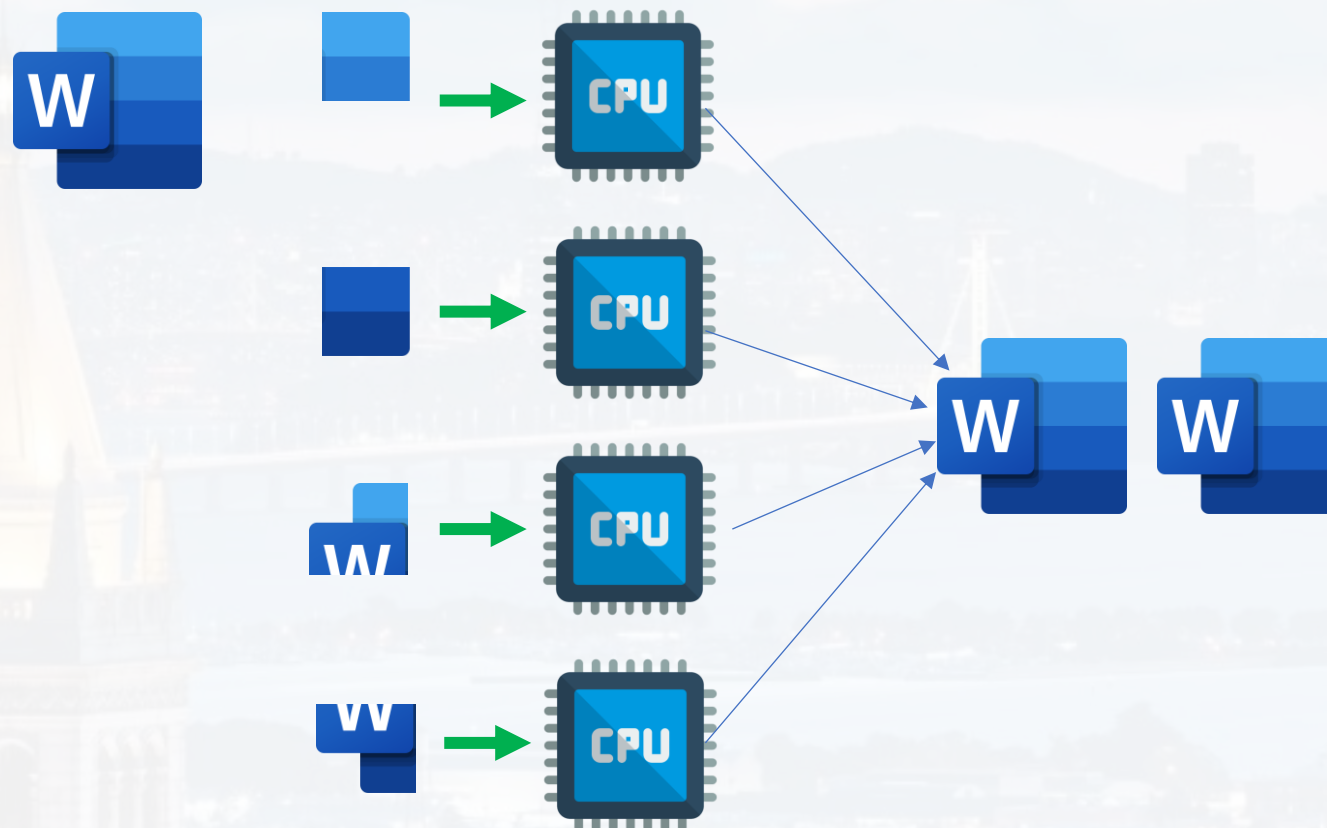
even for moderate models and data: → computational limits

parallelizing the process itself: - eg. a non vectorized function in a for loop

even for moderate models and data: → computational limits

parallelizing the process itself: - eg. a non vectorized function in a for loop

even for moderate models and data: → computational limits

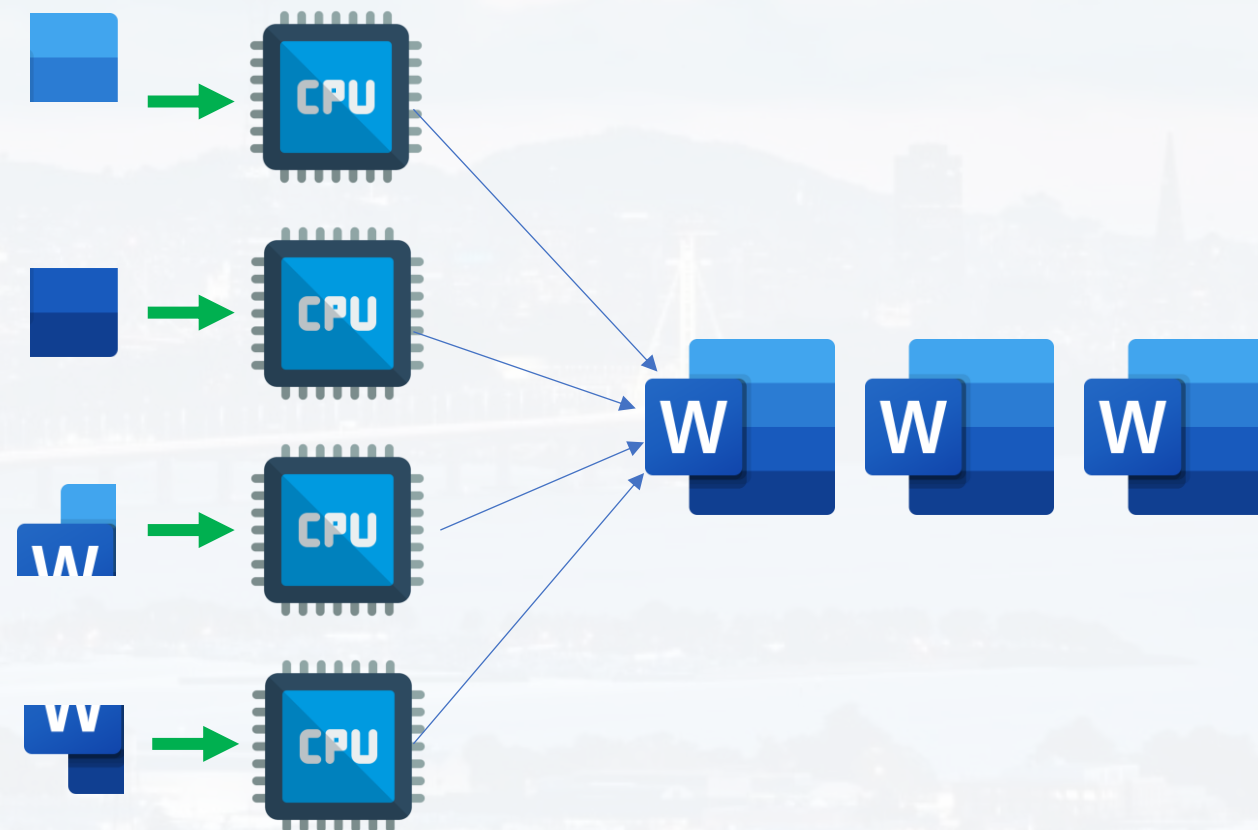**parallelizing the process itself:**   - eg. a non vectorized function in a for loop
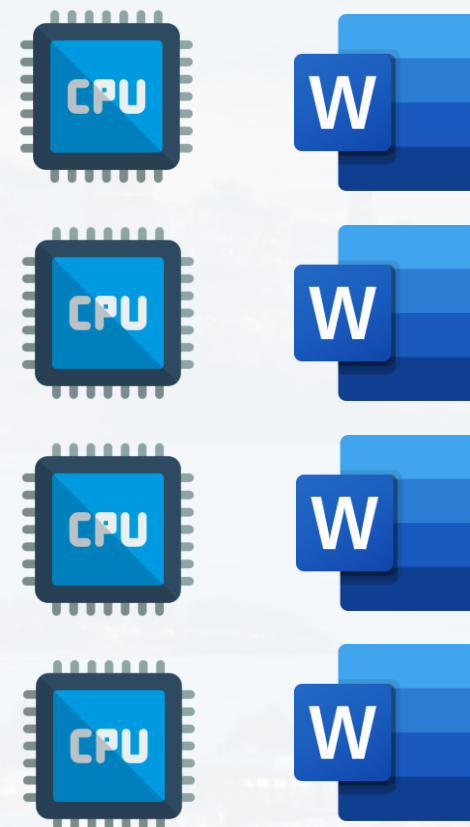
```python
import multiprocessing as mp


with mp.Pool(processes = len(list_filenames)) as pool:
        All = pool.map(MyFun, list_filenames)

return All
```
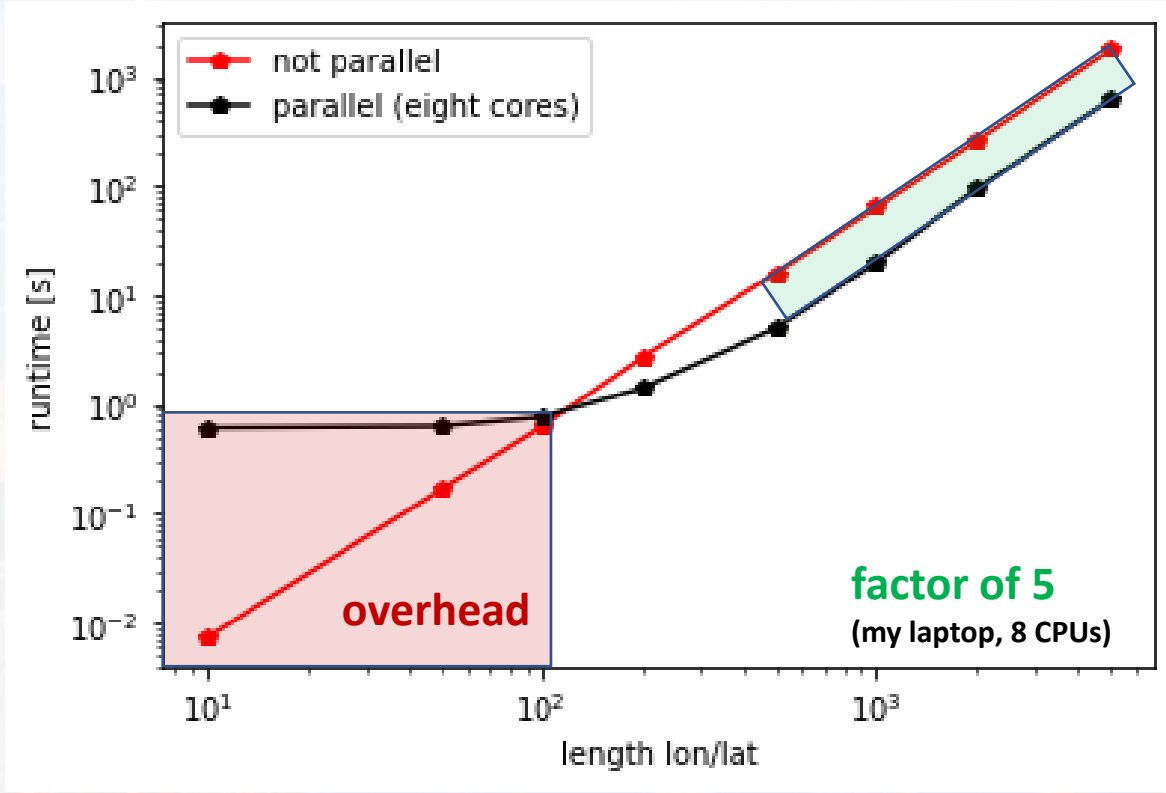
combined output

parallelizing the process itself: - eg. a non vectorized function in a for loop

same program, but different instances: - eg. reading different files, and perform the same analysis
- frequency grid search (different sub intervals)

note: preparing and coordinating the different processes takes time → **overhead**

parallelizing two
nested for loops

"What was initially thought to be a simple process is in fact an incredibly complicated, intricate, and complex system that I've codified and organized into a few easy-to-follow rules that are more difficult to implement than you'd think."

Outline

- Parallel Processing

- Using Map

- Using Process

- CUDA & PyTorch

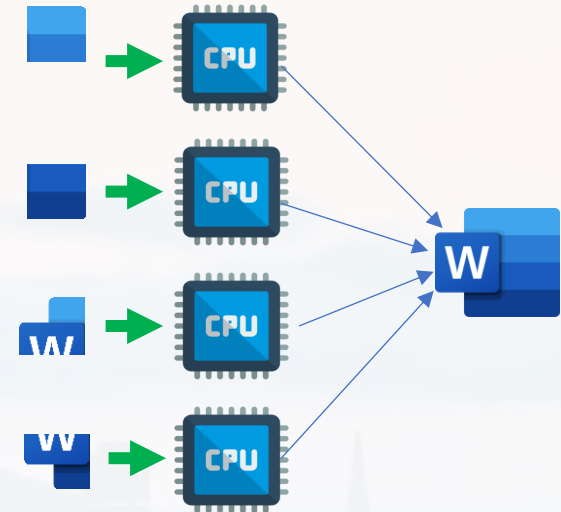example1: reading excel files, extracting some content and combing it.

```python
import multiprocessing as mp
```

a) writing a function that does one thing at the time

```python
def MyFun(filename: str) -> np.array:
    #reads huge data set
    data = pd.read_excel(filename)
    return np.array(data.time)
```

b) writing a **main** function that calls the first function using **map**

```python
def Parallel(self, list_filenames: list) -> list:

        with mp.Pool(processes = len(list_filenames)) as pool:
            All = pool.map(MyFun, list_filenames)

        return All
```

example1: reading excel files, extracting some content and combing it.

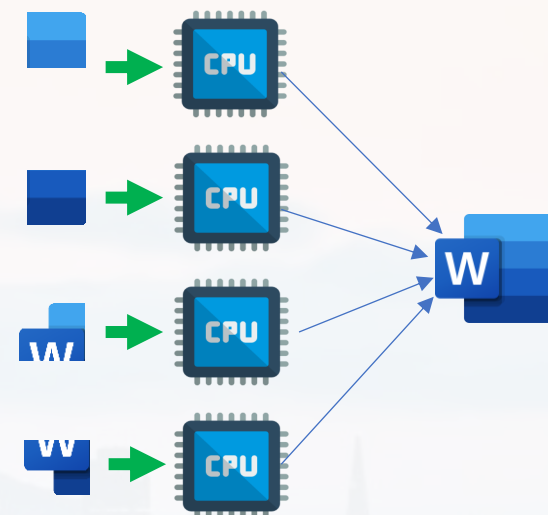check out the function `Map.py`

```python
from Map import *

globals()['MyFun'] = my_timer(MyFun)

MAP             = Parallel_MAP()
```

```
In [7]: out  = MyFun(list_filenames[0])
Total runtime: 26.9369999997625 seconds

In [8]: out1 = MAP.Serial(list_filenames)
Total runtime: 151.76500000001397 seconds

In [9]: out2 = MAP.Parallel(list_filenames)
Total runtime: 49.14100000000326 seconds
```

example2: parallelizing a non vectorized function.
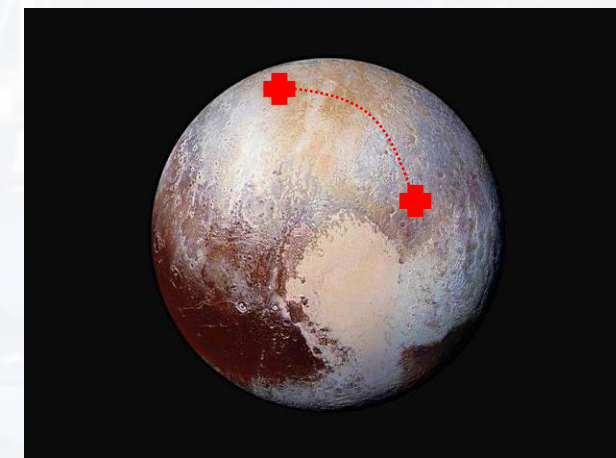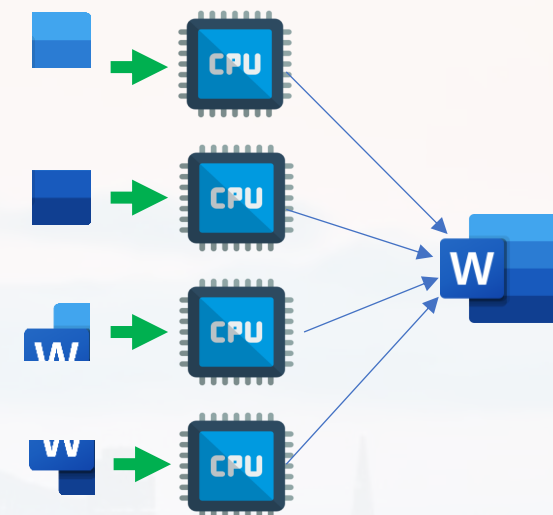
geodesic from GeoPy can take only **one pair of coordinates** at the time


```
coords1 = (lon[i],lat[i])
coords2 = (lon[j],lat[j])

d        = geodesic(coords1, coords2).m

distance[i,j] = d
distance[j,i] = d
```

an example of a non vectorized method

$$d^2s = R^2 dv^2 + R^2 sin^2v \, d\phi^2$$

example2: parallelizing a non vectorized function.

a) writing a function that does one thing at the time

```python
from geopy.distance import geodesic

def calculate_distance(args):

    i, j, lat, lon = args

    coords1 = (lon[i], lat[i])
    coords2 = (lon[j], lat[j])

    d = geodesic(coords1, coords2).m

    return i, j, d
```
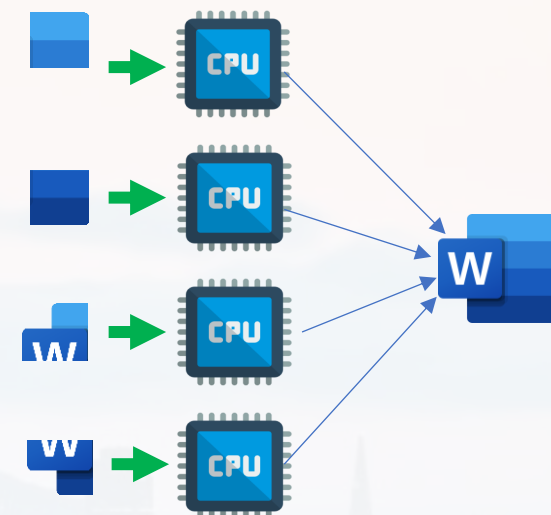
example2: parallelizing a non vectorized function.

b) writing a **main** function that calls the first function using **map**

```python
import numpy as np
from multiprocessing import Pool
from calculate_distance import calculate_distance
```

calling subroutine

```python
def DistanceGeoParallel(lat, lon):

        n        = len(lat)
        distance = np.zeros((n, n))
```
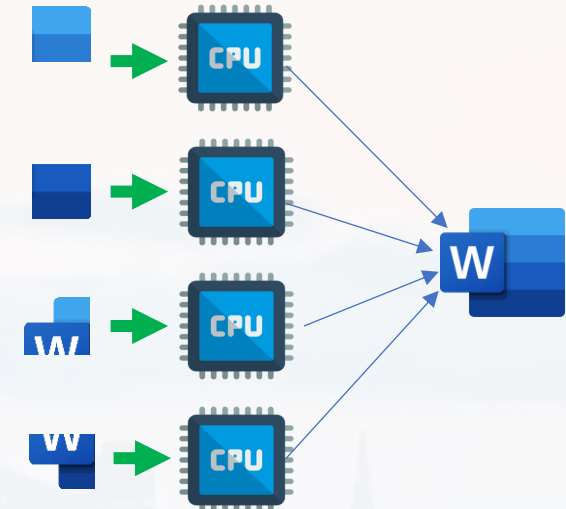
creating list of all variables and indices **(be careful: j starts at i+1 for symmetry reasons!)**

```python
args_list = [(i, j, lat, lon) for i in range(n) for j in range(i+1, n)]

with Pool() as pool:
        results = pool.map(calculate_distance, args_list)

for i, j, d in results:
        distance[i, j] = d
        distance[j, i] = d
```

actual parallelization

```python
return distance
```
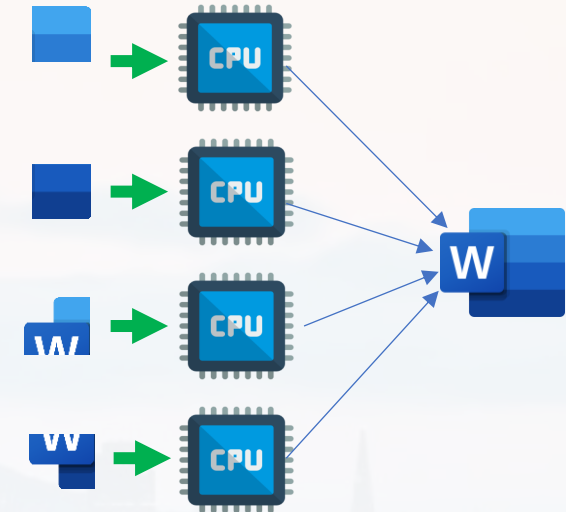
check out:

```
from multiprocess import cpu_count
cpu_count()
```


- calling CPUs via `Pool` **itself takes time** and creates **overhead**
    → only effcient for larger runtimes

- processes have to be **independent** *(i.e. calculating distances, linear algebra, see also AI)*

- functions have to be **independent** *(function A **is not** a subroutine of B)*

- processes should not contain `lambda` or `map`

© MARK ANDERSON, WWW.ANDERTOONS.COM

"What was initially thought to be a simple process is in fact an incredibly complicated, intricate, and complex system that I've codified and organized into a few easy-to-follow rules that are more difficult to implement than you'd think."

Outline

- Parallel Processing

- Using Map

- **Using Process**

- CUDA & PyTorch

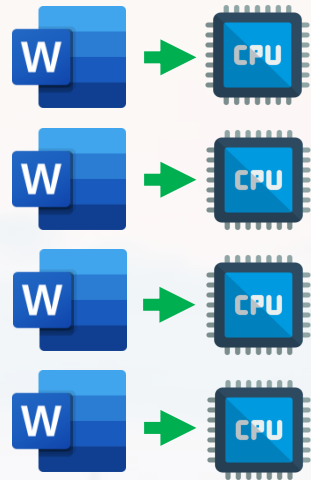example: reading excel files and saving them as csv.

```python
import multiprocessing as mp
```

a) writing a function that does one thing at the time

```python
def MyFun(filename: str):
    #reads & saves huge data set
    data = pd.read_excel(filename)
    data.to_csv('Data_set_' + str(time.monotonic()).replace('.','-') + '.csv'))
```

adding a time stamp to the output file name

b) writing a **main** function that calls the first function using Process

example: reading excel files and saving them as csv.

b) writing a **main** function that calls the first function using Process

```python
def Parallel(self, list_filenames: list) -> list:

    Processes = [mp.Process(target = MyFun, args = (List,))\
                    for List in list_filenames]
```
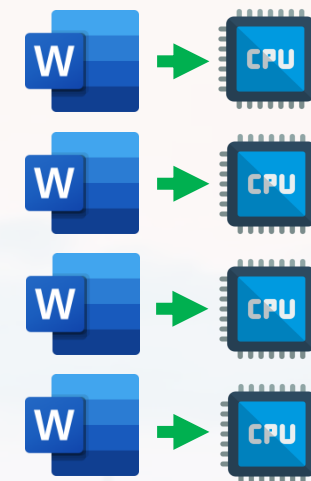
```python
for p in Processes:
    p.start()
```
allocating processes to CPUs

```python
for p in Processes:
    p.join()
```
running and finishing processes

example: reading excel files and saving them as csv.

check out the function Process.py

```python
from Process import *

globals()['MyFun'] = my_timer(MyFun)

Pro                = Parallel_Process()
```

```python
out  = MyFun(list_filenames[0])
Pro.Serial(list_filenames)
Pro.Parallel(list_filenames)
```

```
Total runtime: 27.70300000003772 seconds
Total runtime: 172.03099999995902 seconds
Total runtime: 57.40700000000652 seconds
```

example: reading excel files and saving them as csv.

every 30 seconds
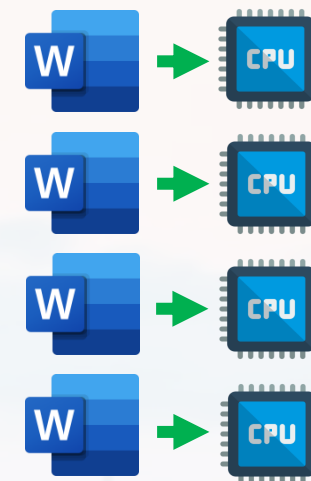
⊞	Data_set_46714-39.csv	20/11/2024 02:57
⊞	Data_set_46742-39.csv	20/11/2024 02:58
⊞	Data_set_46771-984.csv	20/11/2024 02:58
⊞	Data_set_46800-75.csv	20/11/2024 02:59
⊞	Data_set_46829-625.csv	20/11/2024 02:59
⊞	Data_set_46858-187.csv	20/11/2024 03:00

every 0.3 seconds

⊞	Data_set_46912-031.csv	20/11/2024 03:00
⊞	Data_set_46912-39.csv	20/11/2024 03:00
⊞	Data_set_46912-765.csv	20/11/2024 03:00
⊞	Data_set_46912-828.csv	20/11/2024 03:00
⊞	Data_set_46913-281.csv	20/11/2024 03:00
⊞	Data_set_46914-14.csv	20/11/2024 03:00

```
out  = MyFun(list_filenames[0])
Pro.Serial(list_filenames)
Pro.Parallel(list_filenames)
```
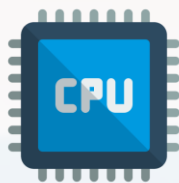
```
Total runtime: 27.70300000003772 seconds
Total runtime: 172.03099999995902 seconds
Total runtime: 57.40700000000652 seconds
```

"What was initially thought to be a simple process is in fact an incredibly complicated, intricate, and complex system that I've codified and organized into a few easy-to-follow rules that are more difficult to implement than you'd think."
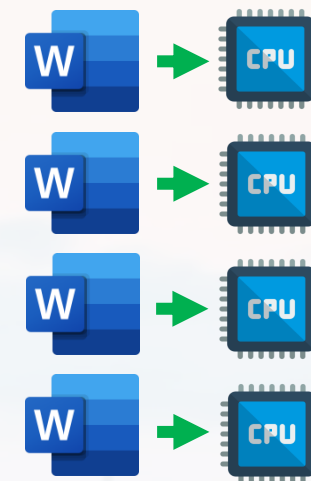
Outline

- Parallel Processing

- Using Map

- Using Process

- **CUDA & PyTorch**

training AI   → mainly matrix operations
                     → GPUs are a lot better at it!
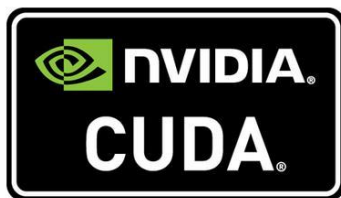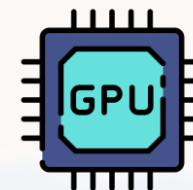
CUDA is the link of your GPU to Python (PyTorch)

check, if graphic card is on [list](#)

check your graphics device:

→ Windows command shell prompt

→ type `nvidia-smi`

→ press *Enter*

Installing CUDA

Installing CUDA

```
(CUDAenv) PS C:\Users\MMH_user> conda install -c anaconda cudatoolkit
```

check libraries

→ type: `conda list`

```
(CUDAenv) PS C:\Users\MMH_user> conda list
# packages in environment at C:\Users\MMH_user\anaconda3\envs\CUDAenv:
#
# Name                    Version                   Build  Channel
blas                      1.0                         mkl
bzip2                     1.0.8                 h2bbff1b_6
ca-certificates           2024.7.2              haa95532_0
cudatoolkit               11.8.0                hd77b12b_0
expat                     2.6.2                 hd77b12b_0
filelock                  3.13.1            py312haa95532_0
intel-openmp              2023.1.0           h59b6b97_46320
```

Installing CUDA

usually, a few libraries are missing

check again graphics card:	type in anaconda	`nvidia-smi`

check libraries:	type in anaconda	`conda list cudnn`
		`conda list cudatoolkit`
		`conda list torch`

	if not:	`conda install <library>`

check Python:	type in anaconda	`python`
		`import torch`
		`torch.cuda.is_available()`

open Spyder	run in Spyder	`pip install` (see the commented line in CheckMyCuda.py)

Installing CUDA

usually, a few libraries are missing   CheckMyCuda.py

```python
import torch

def test_cuda():
    print("PyTorch version: ", torch.__version__)
    print("CUDA version: ", torch.version.cuda)
    print("CUDA Available: ", torch.cuda.is_available())
    if torch.cuda.is_available():
        print("Number of GPUs: ", torch.cuda.device_count())
        print("GPU Name: ", torch.cuda.get_device_name(0))

if __name__ == "__main__":
    test_cuda()
```

```
PyTorch version:  2.3.1+cu118
CUDA version:  11.8
CUDA Available:  True
Number of GPUs:  1
GPU Name:  NVIDIA GeForce MX450
```

The key part in PyTorch is to set all matrices and the model **to the device (CPU or GPU)**

```python
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device:", device)
```

```
In [13]: device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    ...: print("Using device:", device)
Using device: cuda
```

Congratulation! If you see this, you are ready to go!

The key part in PyTorch is to set all matrices and the model **to the device (CPU or GPU)**

```
In [13]: device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    ...: print("Using device:", device)
Using device: cuda
```

Torch objects like **model** or **torch.tensor** have the property **.to**

```
TrainX = torch.tensor(TrainX, dtype = torch.float32)
TrainY = torch.tensor(TrainY, dtype = torch.float32)
```

```
TrainX = TrainX.to(device)
TrainY = TrainY.to(device)
```

```
model  = model.to(device)
```

turning numpy array into torch.tensor

allocating objects to the device

The key part in PyTorch is to set all matrices and the model **to the device (CPU or GPU)**

```
In [13]: device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    ...: print("Using device:", device)
Using device: cuda
```

When running the training, we need to **synchronize** between GPU (for training the model) and CPU (for everything else)…

```
torch.cuda.synchronize()

        #training loop
        for epoch in range(n_epochs):
                outputs = model(TrainX)

                ... #do stuff...

torch.cuda.synchronize()
```

The key part in PyTorch is to set all matrices and the model **to the device (CPU or GPU)**

```
In [13]: device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    ...: print("Using device:", device)
Using device: cuda
```

When running the training, we need to **synchronize** between GPU (for training the model) and CPU (for everything else)…

…and later detach the model from the GPU

```
PredY = model(TestX).detach().to('cpu').numpy()
```

check out script **HowToRun.py**

that runs the **same LSTM model** for Keras and PyTorch

```
n_neurons   = 100
n_epochs    = 200
dt_past     = 30
dt_futu     = 10
n_features = 1
n_sample    = 1

n_stack     = 3
```
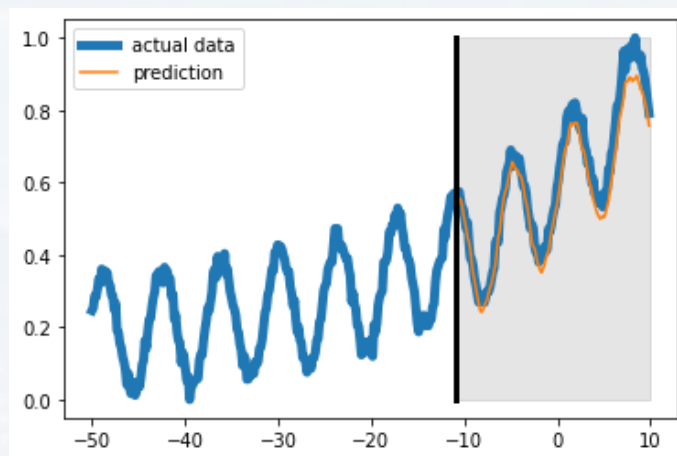
number of stacked LSTM

check out script **HowToRun.py**

that runs the **same LSTM model** for Keras and PyTorch


```
n_neurons   = 100
n_epochs    = 200
dt_past     = 30
dt_futu     = 10
n_features  = 1
n_sample    = 1

n_stack     = 3
```

Lenovo T14, NVIDIA GeForce MX450:

Keras LSTM (CPU):	**300 sec**
PyTorch (CPU):	11 sec
PyTorch (GPU):	**3 sec**

Thank you very much for your attention!