Introduction to HPC for ML

Dr. Anne Weill – Zrahia

January 2020

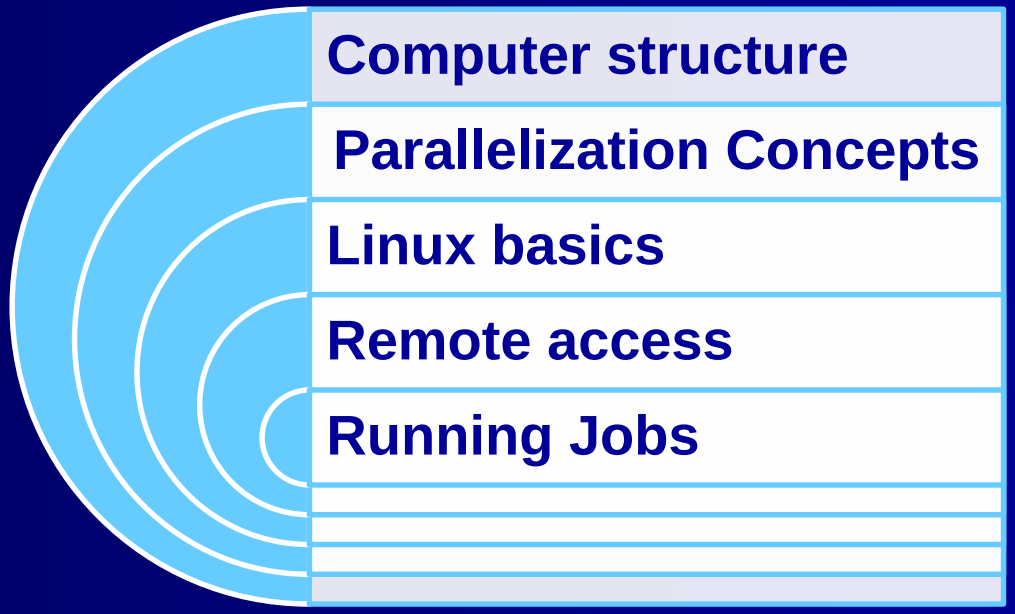
What is HPC?

- High-performance computing (HPC) is the use of parallel processing for running advanced application programs efficiently, reliably and quickly.

HPC in biomedical engineering

- Support and facilitate modeling and simulation activities
- Development and sustainability of software tools and services
- Enhance industries in the healthcare sector

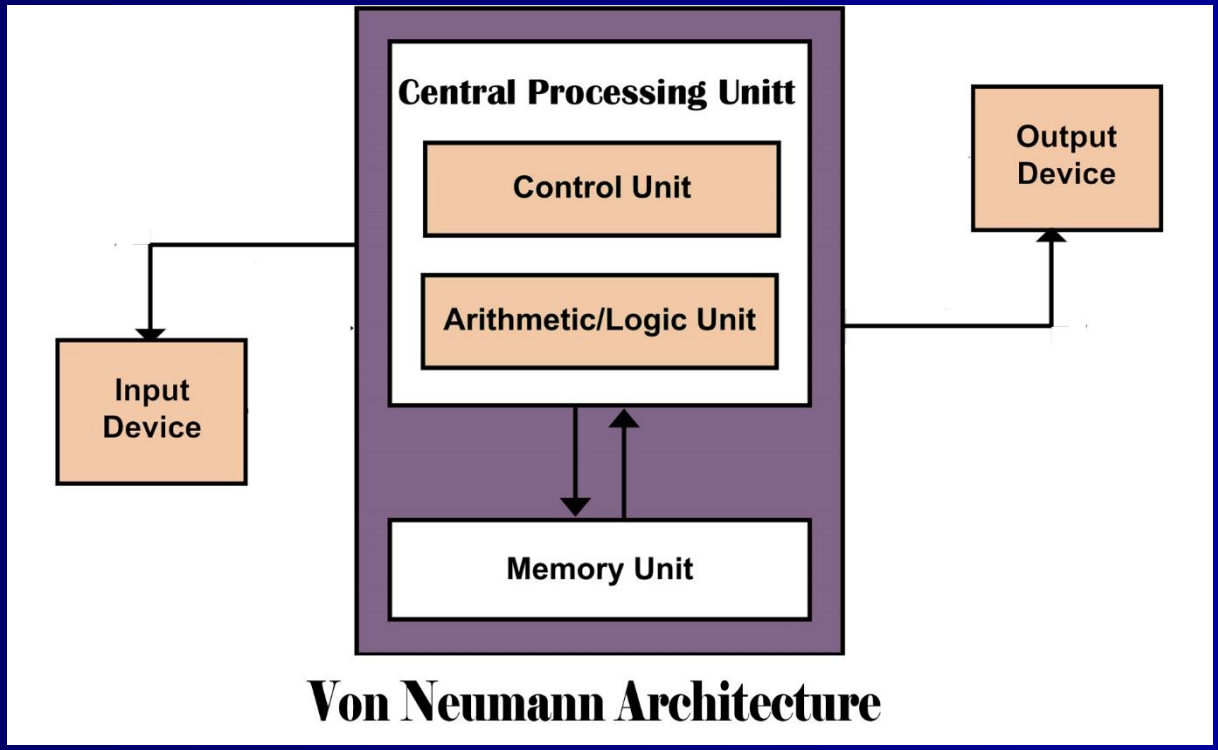
Outline of talk



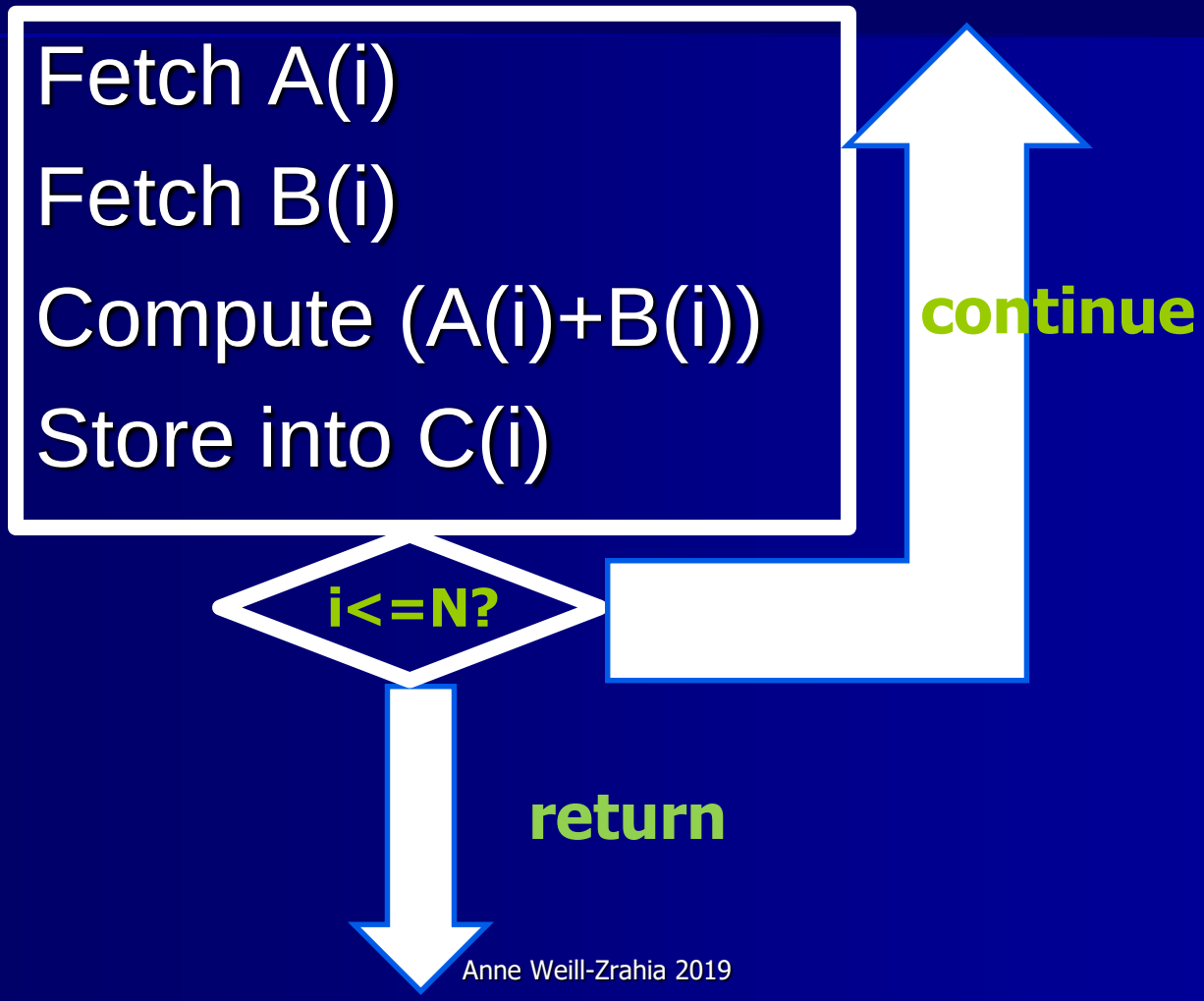
Main parts of a computer

- Central Processing Unit (CPU)
- RAM memory
- Hard drive – stores data at all times
- Video card (GPU) provides the image seen on the screen
- Motherboard – to bind all the components together

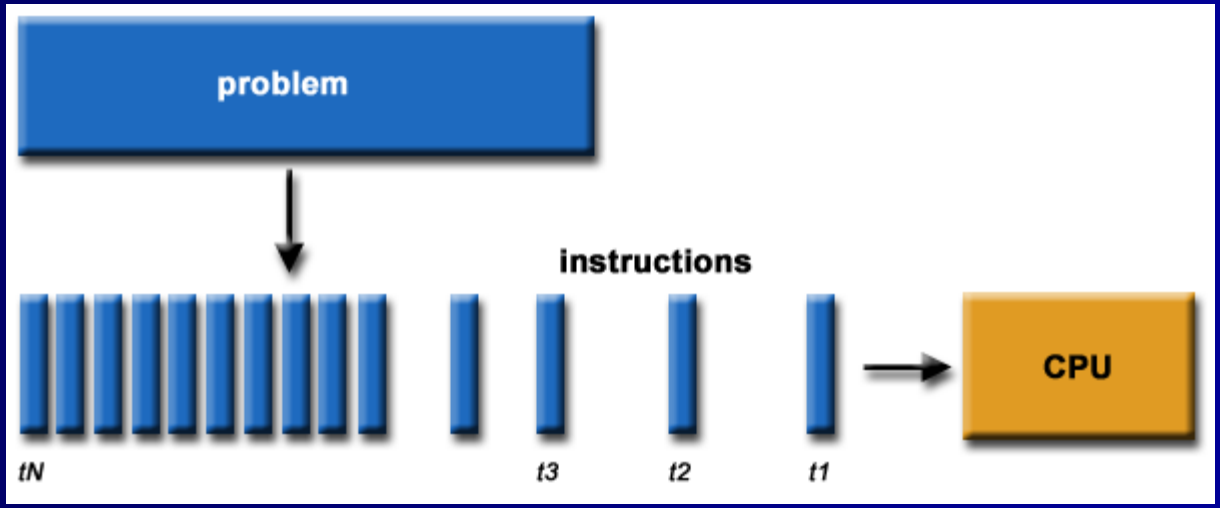
Serial computer architecture



Simple computation



Sequential execution



Memory resources

Variables types :

Type	Size in bytes
Double	8
Float	4
Int	1

Memory performance

Latency	Bandwidth
The time between initiating a request for a byte or word in memory	The maximum throughput of memory, in MHz

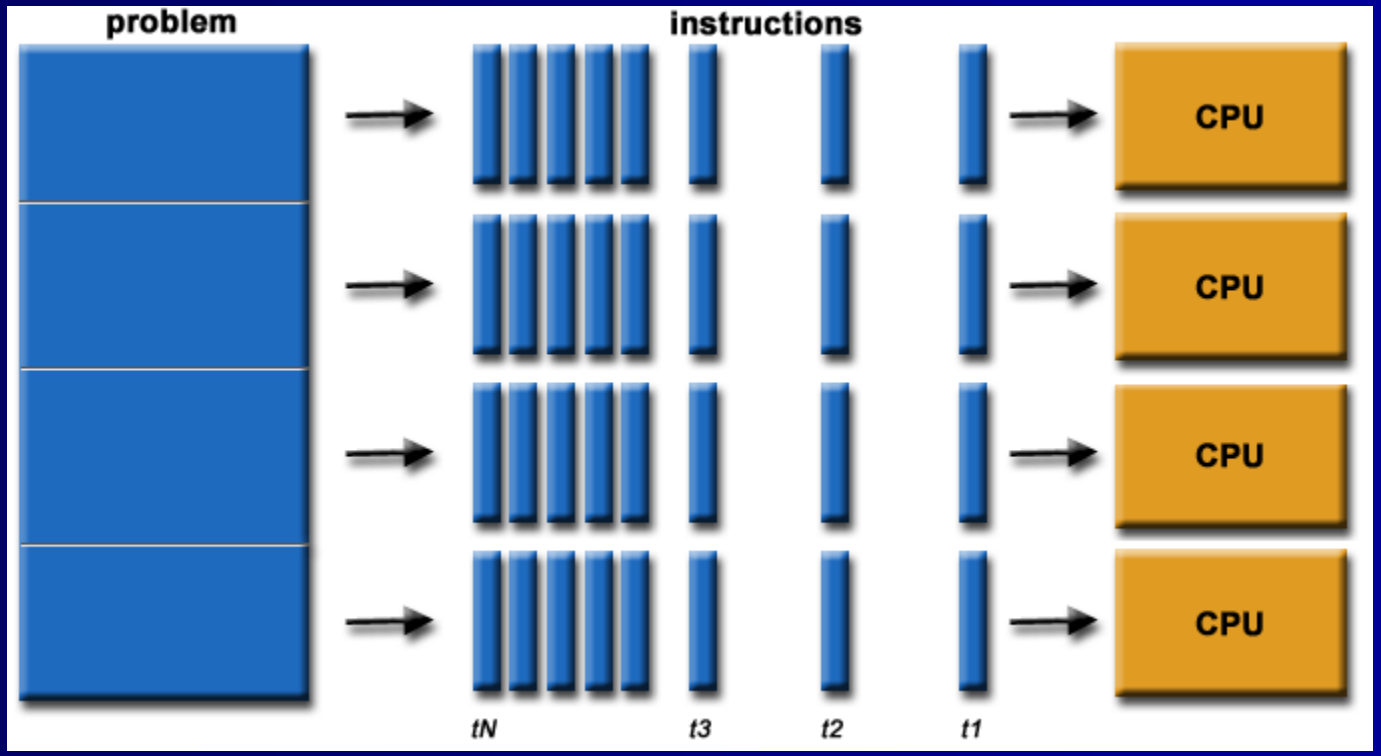
Memory performance

Cache(MB/s)	RAM	GPU	DISK (SSD)	DISK (SATA)
175,000	25,600	45,640	3,500	300

How to speed up an app

- Increase clock speed
- Cache to avoid waiting on memory requests
- Increase no of operations by clock cycle (parallelization)

Parallel execution



Why use parallel computing?

- Current architectures have more parallelism (multi-core CPU's , many-core GPU)
- Save time & money (commodity components)
- Provide concurrency
- Solve larger problems (in a multi node setting)

Parallel performance

Speedup

$$S = \frac{\text{Execution time for 1 computing unit}}{\text{Execution time for p computing units}}$$

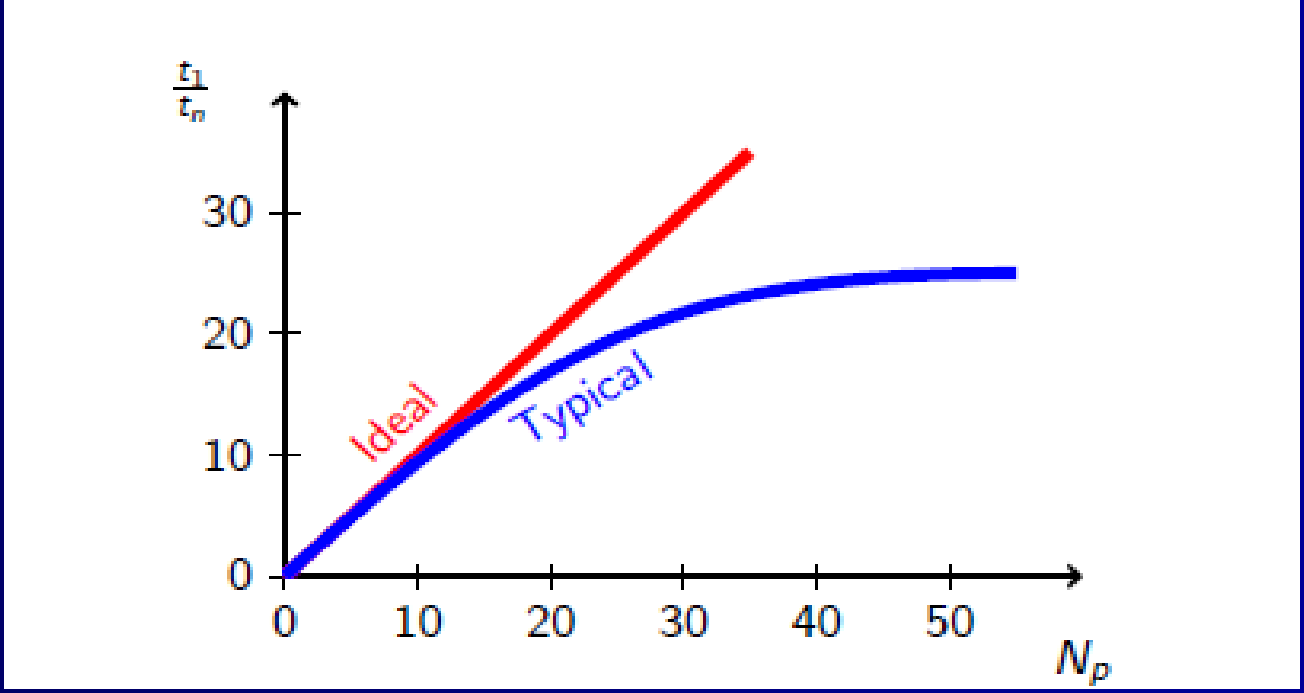
Amdahl's law

$$S = 1/(fs + fp/p)$$

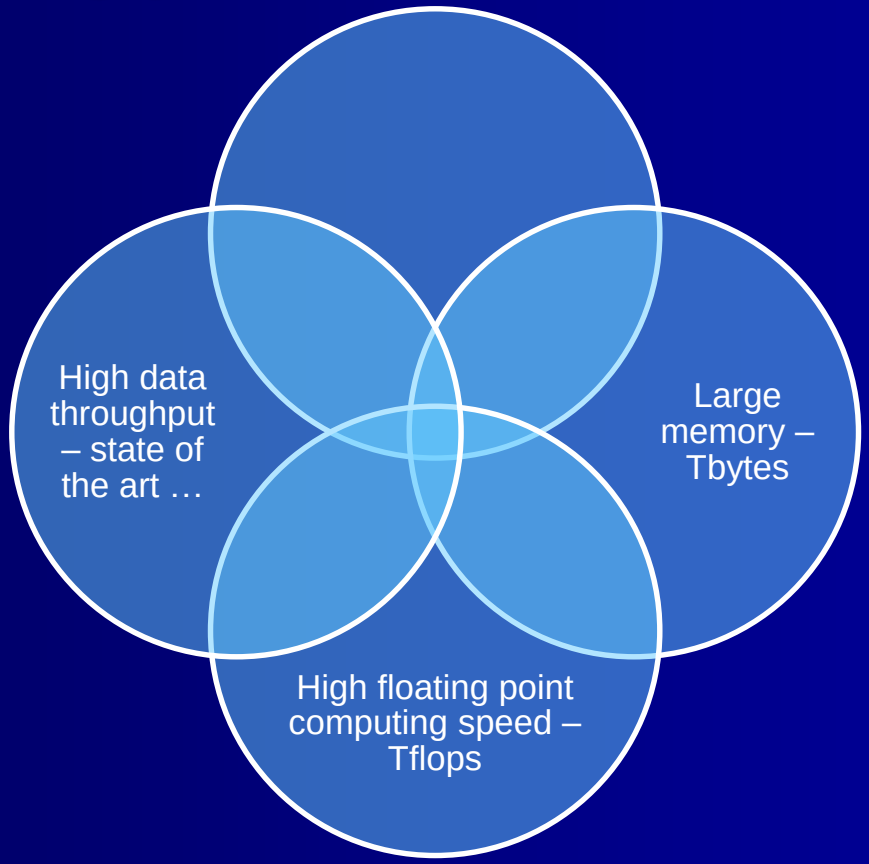
Fs – serial part of computation

Fp -parallel part of computation

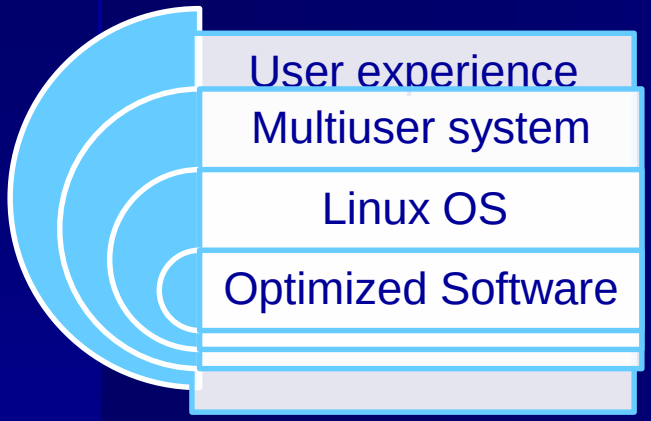
What we expect from parallel application



Large scale applications arising from Nanotechnology or Biomed

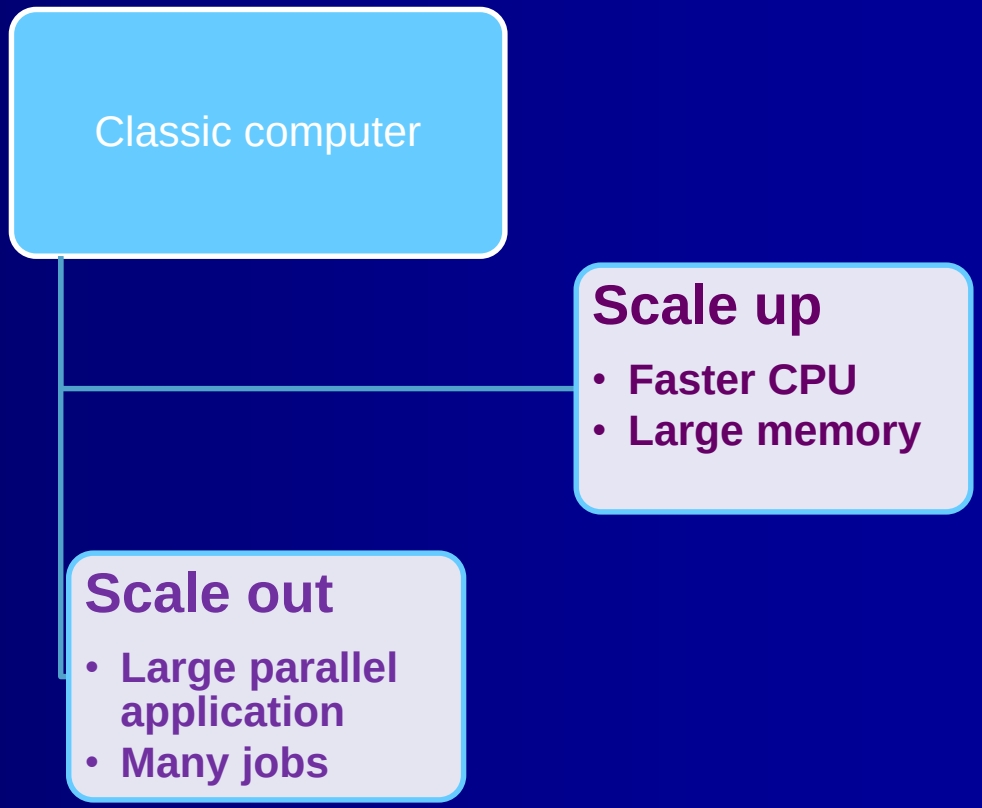


What is a supercomputer

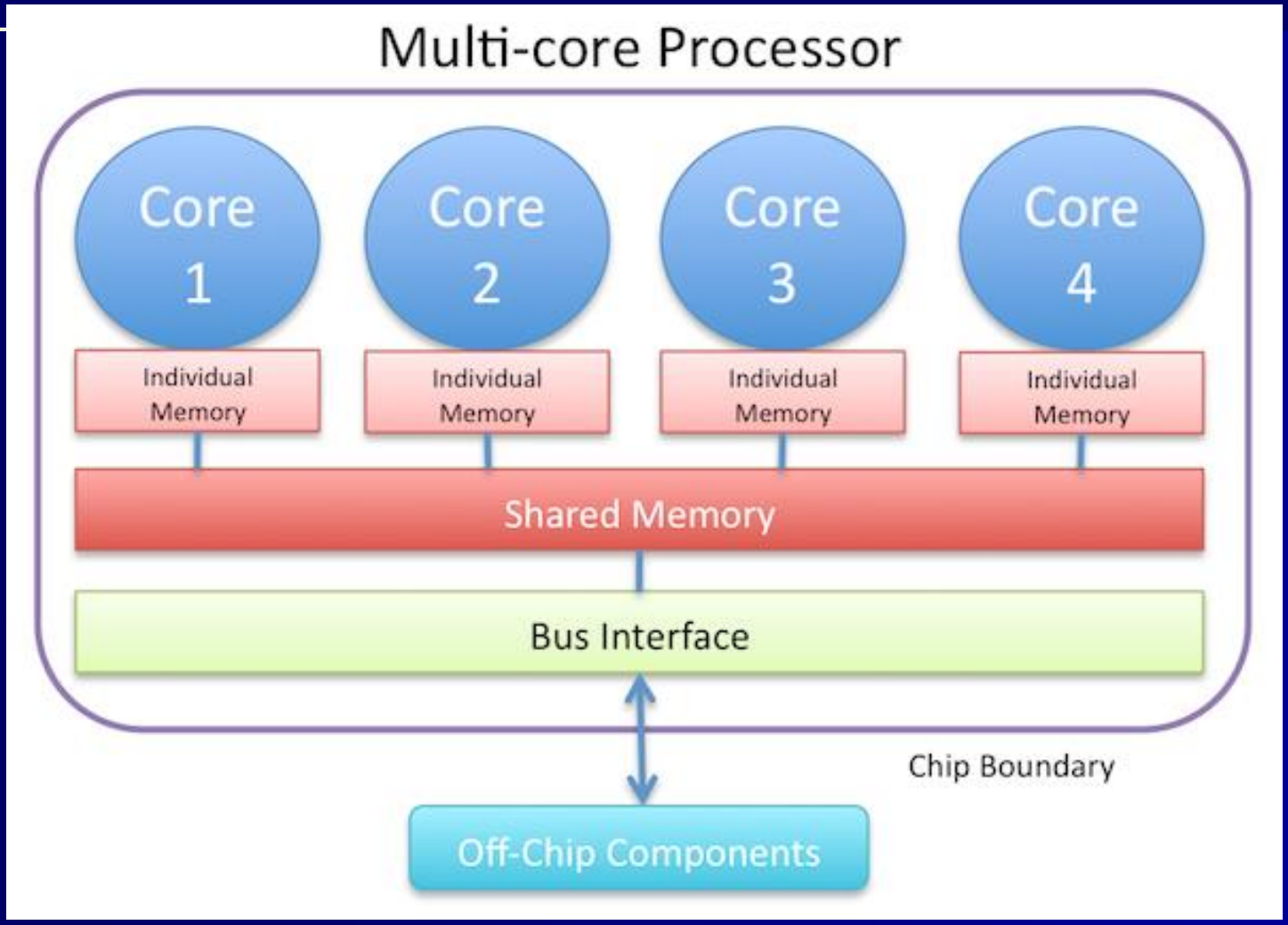


- Compute Power
- Many CPU system
- Specialized Hardware
- Low Latency\high bandwidth

Why do you need a supercomputer



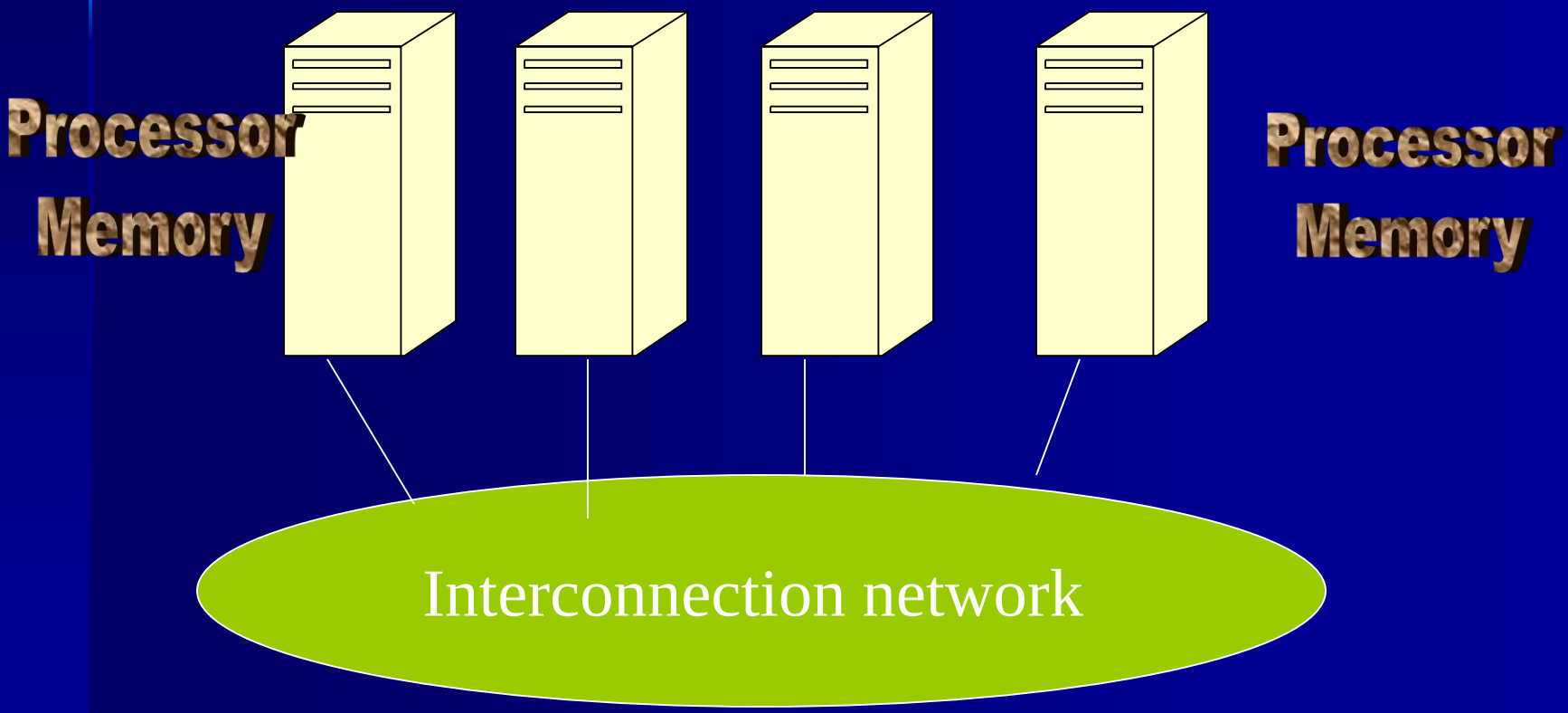
Multicore architecture



Shared Memory

- Each computing unit (core) can access any part of the memory
- Access times are uniform (in principle)
- Easier to program (no explicit message passing)
- Bottleneck when several tasks access same location

Distributed Memory



Distributed Memory

- Processor can only access local memory
- Access times depend on location
- Processors must communicate via explicit message passing

Parallel paradigms

At a low level, three basic programming models are in common use:

- Shared memory model (threads)
- Distributed memory model (MPI)
- GPU model (CUDA) – a combination of distributed and shared memory models (+ vector computing)

Data Parallel Programming

- Same set of instructions runs on each processor (core) simultaneously
- Each processor inputs different data into those instructions
- Shared memory
- Synchronization by the OS

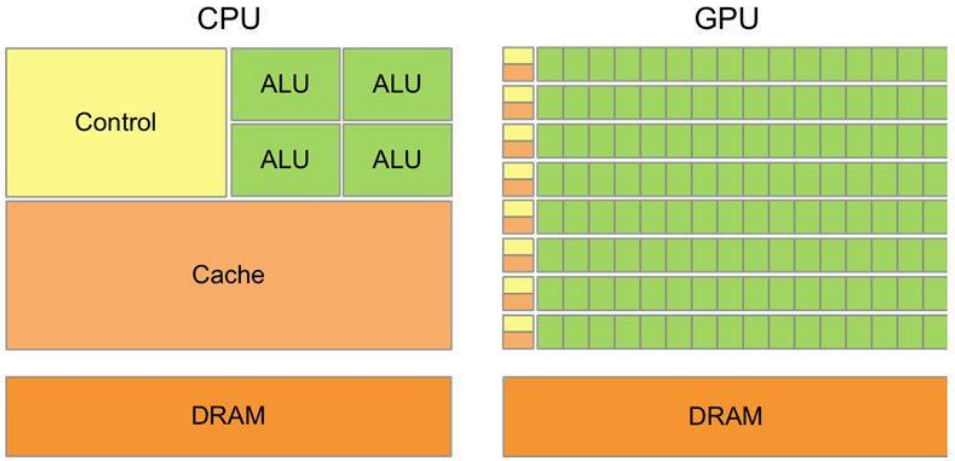
Message Passing Programming

- Separate program on each processor
- Local Memory
- Control over distribution and transfer of data
- Additional complexity of debugging due to communications

CPU vs GPU

CPU vs. GPU

- Different design philosophies:
  - CPU
    - A few out-of-order cores with huge caches
    - Sequential computation
  - GPU
    - Many in-order cores
    - Massively parallel computation

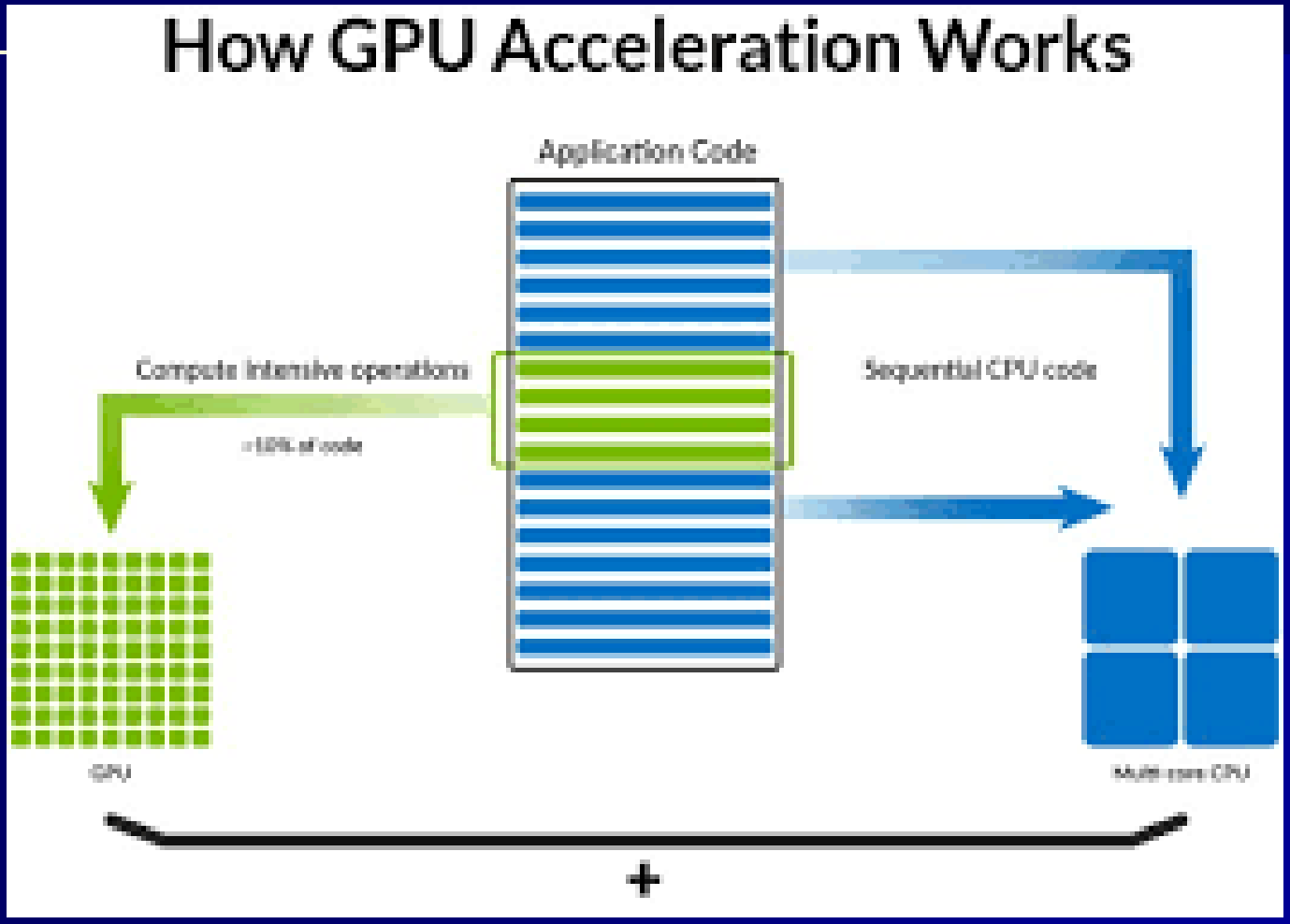


CPU vs GPU -concept

CPU vs GPU

CPU	**Up to 32 cores**	**Latency-optimized cores**	**Fast serial processing**
GPU	Thousands of cores	Throughput optimized cores	Scalable parallel processing

GPU Acceleration

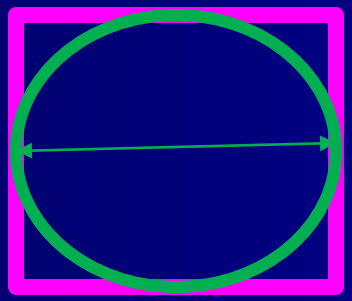


General concepts

- Transfering data between normal RAM and graphics RAM takes time
- “Loading/starting GPU programs takes some time
- “Embarrassingly parallel” algorithms can profit from GPU computing

Monte Carlo example

- We can use Monte Carlo approximation for PI
- We start to toss random points between coordinates (0,0) and(1,1) and count how many are in the circle



- Pi=(numbers of points in circle)/(overall number)*4

Python code (PI, serial)

```
import random
def monte_carlo_pi(n):
    count=0
    for i in range(n):
        x=random.random()
        y=random.random()

        # if it is within the unit circle
        if x*x + y*y <= 1:
            count=count+1

    return count/(n*1.0)*4

if __name__=='__main__':
    print (monte_carlo_pi(10000000))
```

Python code (PI, multiproc)

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-"
Created on Tue Jan  7 16:09:25 2020

import random
import multiprocessing
from multiprocessing import Pool
#def monte_carlo_pi_part(n):
    count = 0
    for i in range(n):
        x=random.random()
        y=random.random()


        # if it is within the unit circle
        if x*x + y*y <= 1:
            count=count+1

    return count
```

Python code (PI, multiproc,cont.)

```
if __name__=='__main__':
    maxnp = multiprocessing.cpu_count()
    np=int(input("Enter number of threads "))

    print  ("you have {1} CPUs",np)

 # Number of points to use for the Pi estimation
    n = 100000000

    # iterable with a list of points to generate in each worker
    # each worker process gets n/np number of points to calculate Pi from

    part_count=[int(n/np) for i in range(np)]
    print(part_count)
#Create the worker pool
        pool = Pool(processes=np)

 # parallel map
    count=pool.map(monte_carlo_pi_part, part_count)


    print (sum(count)/(n*1.0)*4)
```

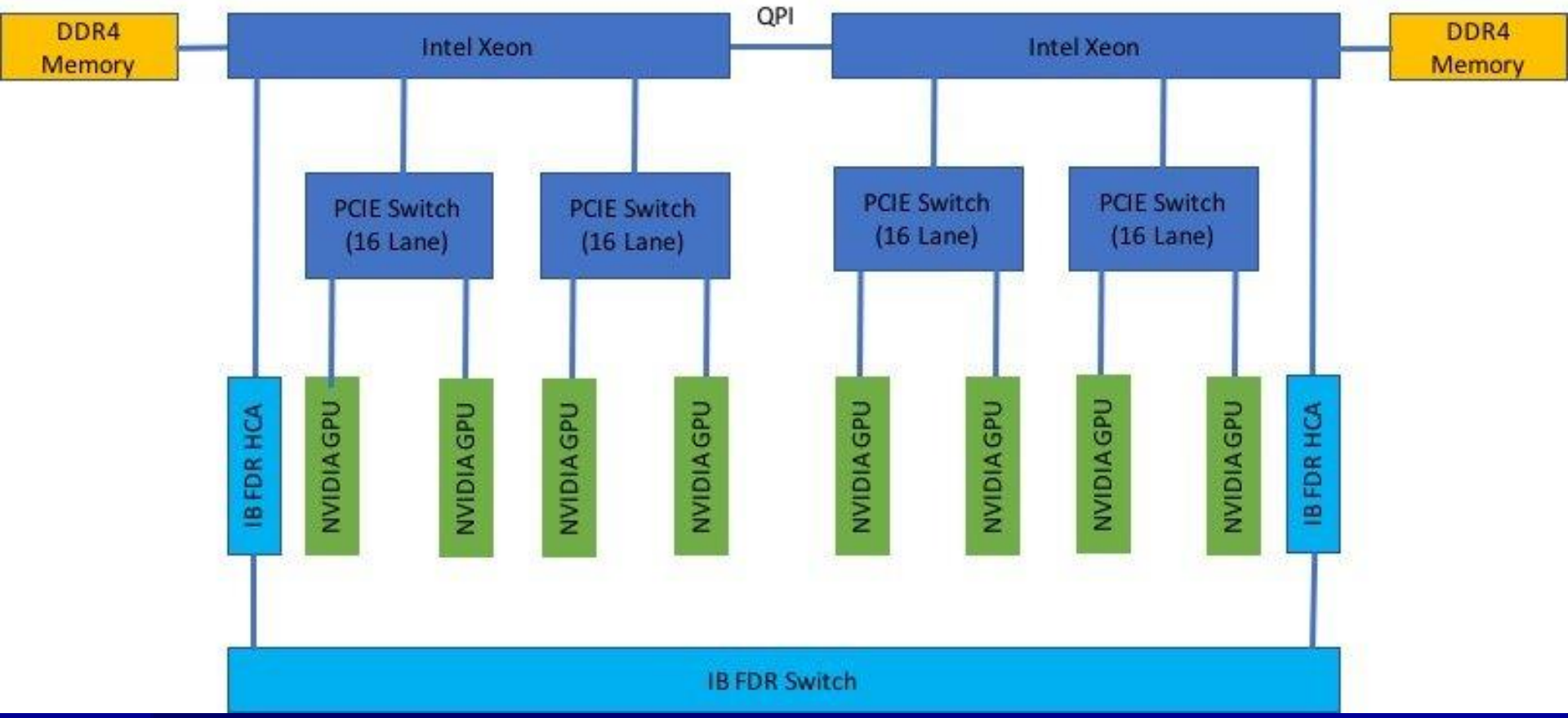
PI example with PYCUDA

```
import pycuda.gpuarray as gpuarray
import pycuda.driver as cuda
import pycuda.autoinit
import numpy
 from pycuda.curandom import XORWOWRandomNumberGenerator
from pycuda.reduction import ReductionKernel
rng = XORWOWRandomNumberGenerator()
N = 10000000
x_gpu = rng.gen_uniform((N,), dtype=numpy.float32)
y_gpu = rng.gen_uniform((N,), dtype=numpy.float32)
circle = ReductionKernel(numpy.dtype(numpy.float32), neutral="0", reduce_expr="a+b",
map_expr="float((x[i]*x[i]+y[i]*y[i])<=1.0f)", arguments="float *x, float *y")
result = 4.0 * circle(x_gpu, y_gpu).get() / N
print 'Estimate for PI on GPU: {}'.format(result)
```

HPC platforms 2020

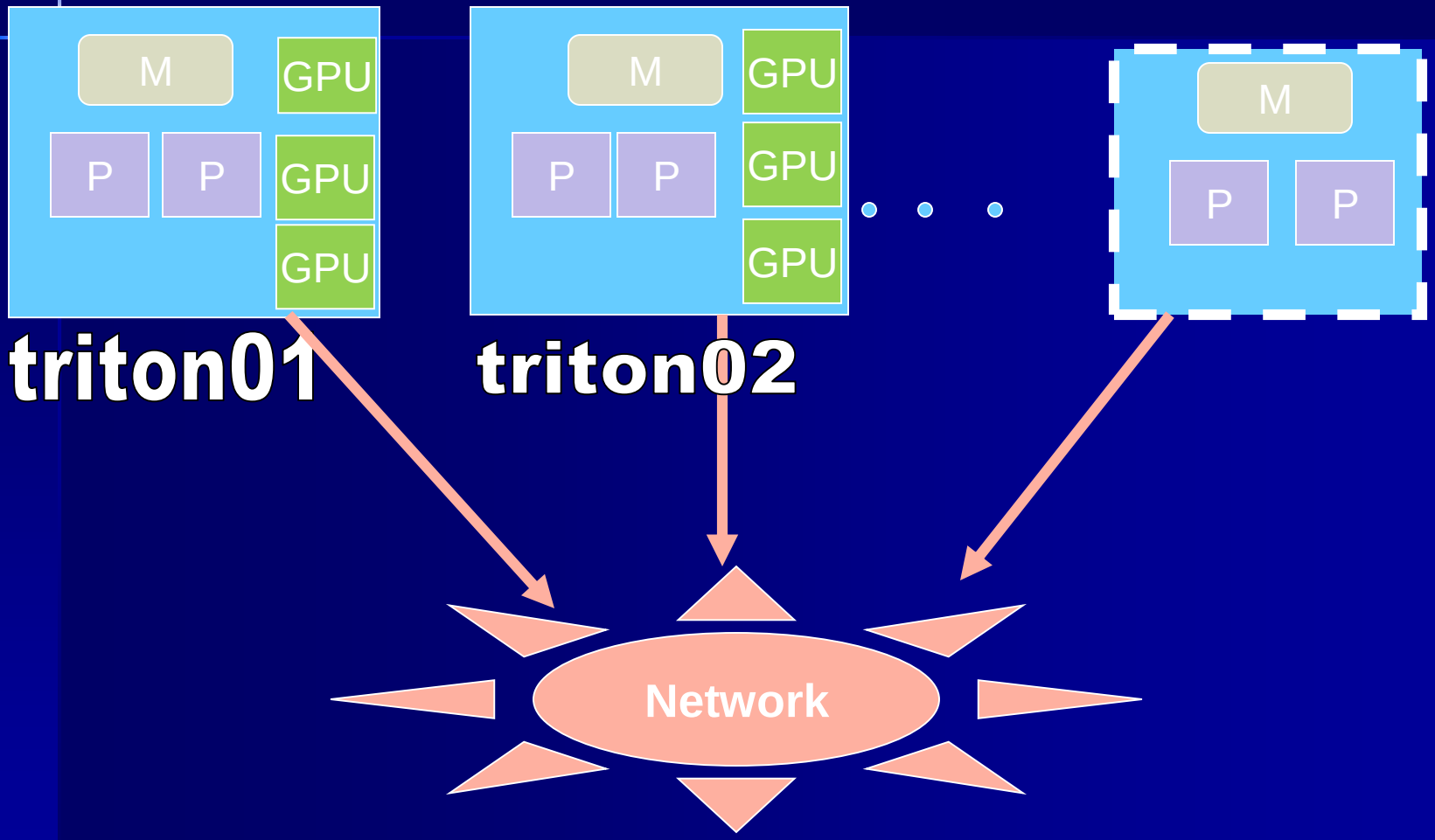
- Multicore workstations
- Hybrid architectures (CPU+GPU)
- Clusters
- Grid (not for all applications)

Triton Configuration



There are 16 cores per CPU, 32 threads

Configuration



Components

- 2 Multicore CPU's

Intel(R) Xeon(R) Gold 6130 -16 cores

- 3 2080 RTX GPU

2944 cuda cores

Getting started

- Security
- Logging in
- Shell environment
- Transferring files

System access-security

- Secure access
- X-tunelling (for graphics
- Can use ssh –X for tunnelling

Accessing triton0x from Windows

First we must use a client to access Linux machines from a Windows PC

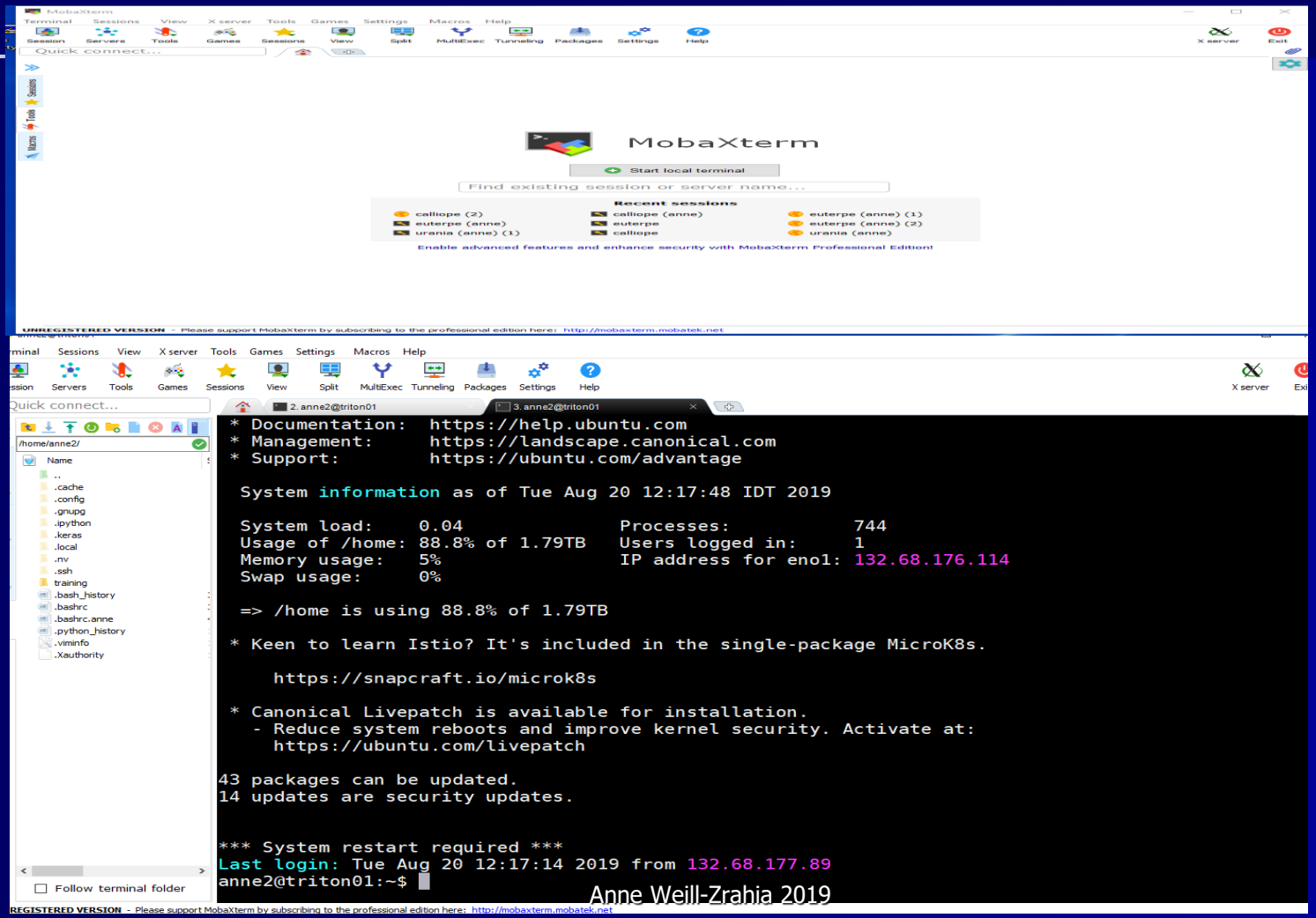
Choices are :

- mobaxterm
- putty (fast, no X apps)
- Command line from Win10 (no X apps)

Logging in and transferring files to triton from Windows

- **ssh angel@triton01.bm.technion.ac.il**
- **WINSCP – to transfer files**

Logging in via mobaxterm



Some generalities about Linux

- On LINUX machines shared by several users a user called **root** is the superuser; some things have to be done by root, but you can do a lot in your own account too

- Linux is **case-sensitive**

Login Environment

- Paths and environment variables have been setup (change things with care)
- bash is the default (can transfer to tcsh if you like)
- User modifiable environment variables are in .bashrc in home directory
- Home directory is in /home/angel

BASH commands

- Bash is the shell for Linux
- .bashrc - a file read at login for each user
- Contains definitions for system variables settings, aliases etc.

Compilers and Software

- Options are **gcc,fortran,g++**
- **Matlab 2019**
- **Python 3.6**
- **Anaconda**
- **Spyder**
- **pycharm**

Useful commands

- top - to see your processes – memory and cpu

- **ps –u <username> - to see processes**
- **nvidia-smi - to see GPU usage**

Getting around

Command	Meaning	Usage
>pwd	which directory	
>ls	List files	
>cd	Change directory	>cd mydir >cd .. (one level up)
>mkdir	Make directory	>mkdir mydir
>cp >mv	Copy file 1 onto file2 Move file1 to file2	>cp file1 file2 >mv file1 file2
>rm	Remove file ,directory	>rm file1 >rm -r mydir

Working on files

Command	Meaning	Usage
>cat	displays contents of a file	Cat file1
>echo	Moves text into a file	>echo hello >>hello.txt
>nano,vi	Editors built-in	>nano hello.txt >vi hello.txt
>tar	Make archive of file1.txt, file2.txt into arch.tar	>tar –cvf arch.tar file*.txt
>tar	Extract archive from arch.tar	>tar –xvf arch.tar
>gzip,gunzip	Compress and uncompress files	>gzip arch.tar >gunzip arch.tar.gz

Archiving files

- *Archiving files* : tar –cvf allmyfiles.tar

packs them into a binary file called allmyfiles.tar which can then be moved around with scp or downloaded from a web or http server

- ***Unpacking the files :*** tar –xvf allmyfiles.tar

I t preserves also subdirectories but will write over a file you have with the same name

Important libraries

- **CUDA** – a parallel computing language for using GPU capabilities
- **CUDNN**- *CUDA Deep Neural Network library that sits on-top of CUDA*
- ***Anaconda***— *A good package and environment manager that comes with a lot of data scientific computing tools such as Numpy, Panda and Matplotlib. The other benefit of Anaconda is it makes it easy to create custom python environments.*

.bashrc on triton01

export PATH=/usr/local/cuda-10.0/bin:$PATH

export LD_LIBRARY_PATH=/usr/local/cuda-10.0/lib64:$LD_LIBRARY_PATH

export CUDA_HOME=/usr/local/cuda-10.0/

alias lt=‘ls –lt’

Python programming

Invoke python from command line

```
>python3
>>> import tensorflow as tf
>>> tf.__version__
'1.14.0'
```

To install python library for current user

```
pip3 install --user numpy scipy matplotlib keras
```

Tensorflow example

- Check if running on a GPU

tf_test.is_gpu_available

- If true, then it is running on a GPU. The lowest number GPU (0) will be chosen

```
a = tf.constant([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
b = tf.constant([[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]])
c = tf.matmul(a, b)
print(c)
```

Tensorflow example

- Choose on which device operations will run

```
# Place tensors on the GPU
with tf.device('/GPU:2'):
    my2 = [[2. for x in range(2000)] for x in range(2000)]
    my1 = [[1. for x in range(2000)] for x in range(2000)]
    product = tf.matmul(my1, my2)
```

Tensorflow matmul example

```
import time
import tensorflow as tf

def measure(x, steps):
  # TensorFlow initializes a GPU the first time it's used, exclude from timing.
  tf.matmul(x, x)
  start = time.time()
  for i in range(steps):
    x = tf.matmul(x, x)
_ = x.numpy()
  end = time.time()
  return end - start

shape = (4000, 4000)
steps = 200
print("Time to multiply a {} matrix by itself {} times:".format(shape, steps))

# Run on CPU:
```

Tensorflow matmul example (cont)

```
# Run on CPU:
with tf.device("/cpu:0"):
  print("CPU: {} secs".format(measure(tf.random.normal(shape), steps)))

# Run on GPU, if available:
if tf.config.experimental.list_physical_devices("GPU"):
  with tf.device("/gpu:0"):
    print("GPU: {} secs".format(measure(tf.random.normal(shape), steps)))
else:
  print("GPU: not found")
```

Timings of tensorflow matmul example

4000x4000	time
CPU	12.7 sec
GPU	2.59 sec

IDE’s for python

- Spyder

>spyder

- Jupyter

>python3 -m pip install --upgrade --user pip
>python3 -m pip install --user jupyterlab

>jupyter notebook mynote.ipynb

Using jupyter remotely

- Login to triton, then fire a jupyter notebook:

Jupyter notebook –-no-browser –-port=2001

(you get an URL)

- Ssh into triton and do :

ssh –NL 2001:localhost:2001 user@triton01

- Paste the URL into a browser on your computer

Working with a supercomputer

- Batch systems - to distribute computational tasks on the available nodes
- Job script : file containing the commands to execute

- The batch system is responsible for allocating cores, processors or nodes to a job

Batch scheduler advantages

- It allows to run many jobs at the same time
- Multiusers

- System load balance

We are using a system called :

SLURM

Interacting with slurm

Submit a job script :

```
sbatch   <job-script>
```

Show submitted jobs and their status :

```
squeue
```

Remove job from queue system :

```
scancel  <jobid>
```

Run a command

```
srun <command>
```

Batch file

```
#!/bin/bash
#SBATCH -N 1  # number of  minimum nodes
#SBATCH -c 2 # number of cores
#SBATCH –gres=gpu:1   # Request 1 gpu
#SBATCH –job-name="just_a_test"
#SBATCH -o slurm.%N.%j.out    # stdout goes here
#SBATCH -e slurm.%N.%j.out   # stderr goes here
Hostname      #to see which node I am on
Conda init myenv
python3 main.py
```

Happy programming on Triton !