High Performance Python Lab

Term 2 2020/2021

Lecture 3. Architecture (briefly), parallel intuition, profiling

Outline

- Memory hierarchy
- Building intuition: parallel processes
- Amdahl‘s law, Gustafson's law
- Strong and weak scallabilities
- Flynn‘s taxonomy
- Shared memory and distributed memory systems
- Processes and threads
- Start with profiling in python

Factors influencing the performance

- **Technology of semiconductors**
  - CMOS technology;
  - Decreasing the size of the semiconductor structures leads to better performance:

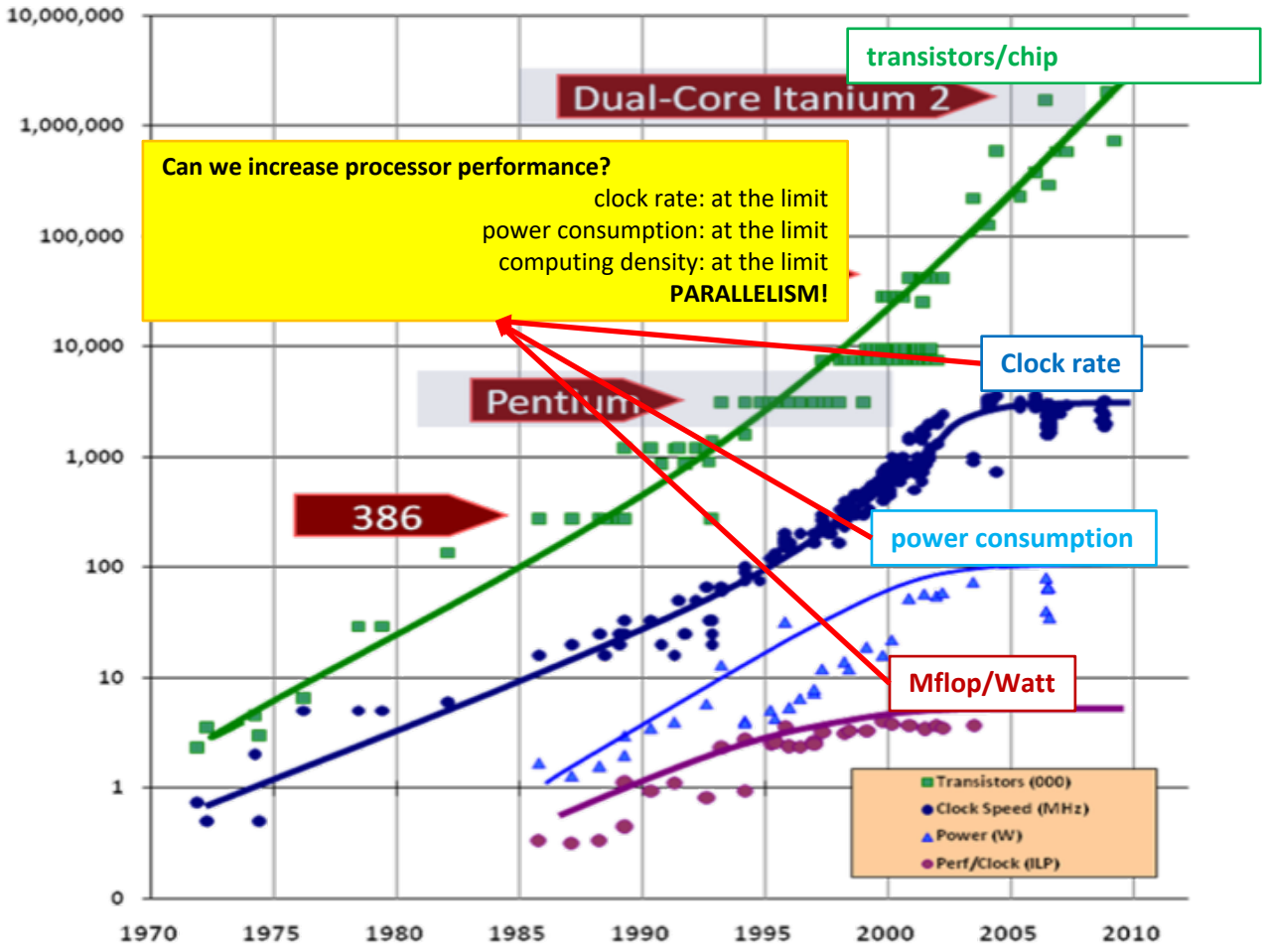
    size, energy consumption, clock rate, price

- **Compute node architecture**
  - Connection of CPUs and GPUs to memory, memory bandwidth, interconnect bandwidth

- **Infrastructure architecture**
  - Energy efficiency;
  - Cooling efficiency (influences the clock rate)
  - Computational density (less path for the signal to take)

Processor technology overview

Scale	Year Intro
130 nm	2001
90 nm	2004
65 nm	2006
45 nm	2008
32 nm	2010
22 nm	2012
14 nm	2014
10 nm	2017 (-19 Intel)
7 nm	2019 AMD (Roma)
5 nm	*~2020 (???)*

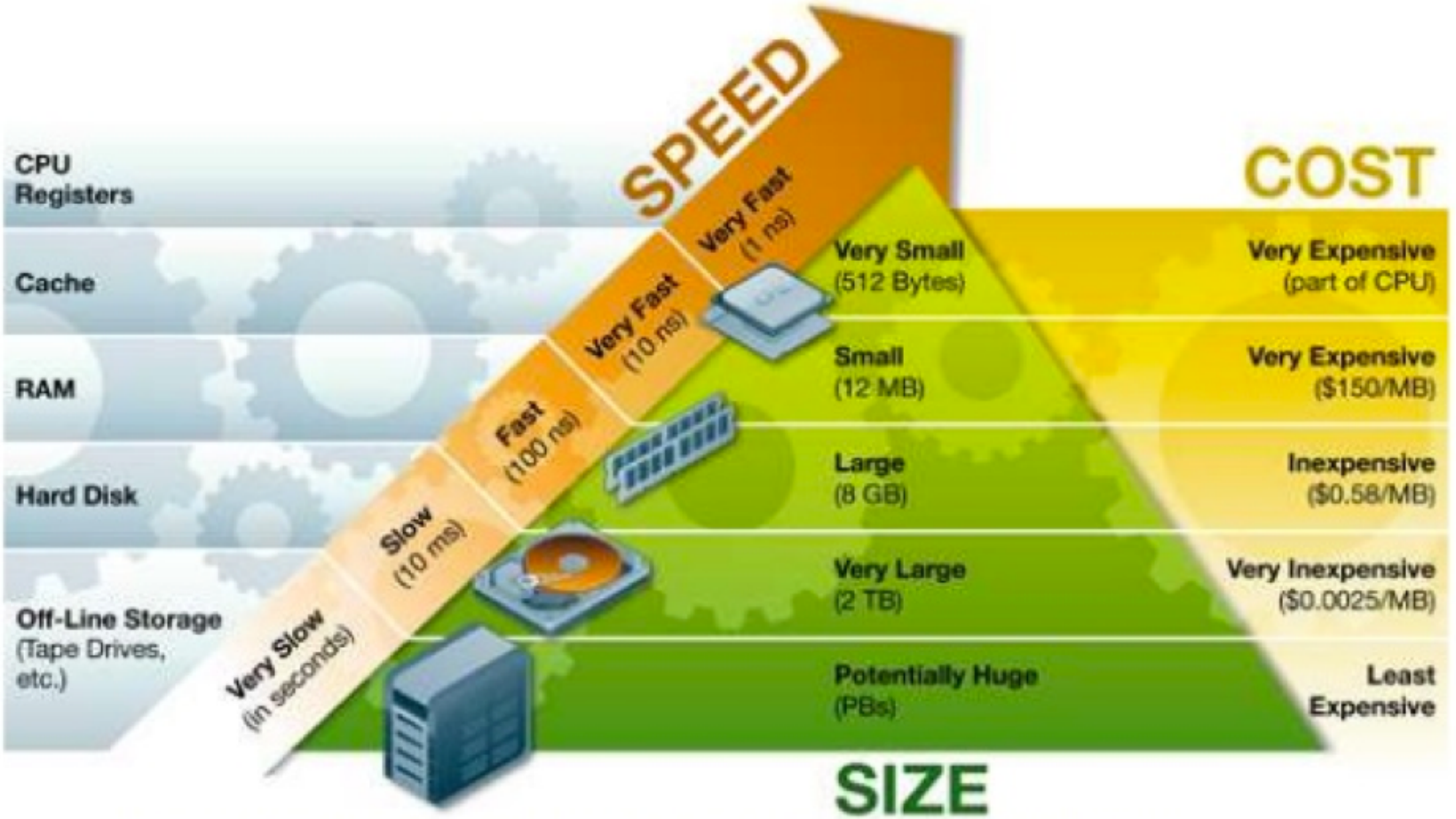


Why the ~~Fastest~~ all (since 2005) Computers are Parallel Computers

Including laptops and cell phones

Think parallel!

Extended Memory Hierarchy



Source: http://www.ts.avnet.com/uk/products_and_solutions/storage/hierarchy.html

Parallel intuition

- Examples of parallelism in real world?

Building a house

Say, we have 4 workers building a house

Construction will be faster than if it would be done by 1 worker

After each floor, workers have to synchronize

Connect their parts - communication

Can‘t build 7th floor before 3rd

Amdahl‘s law

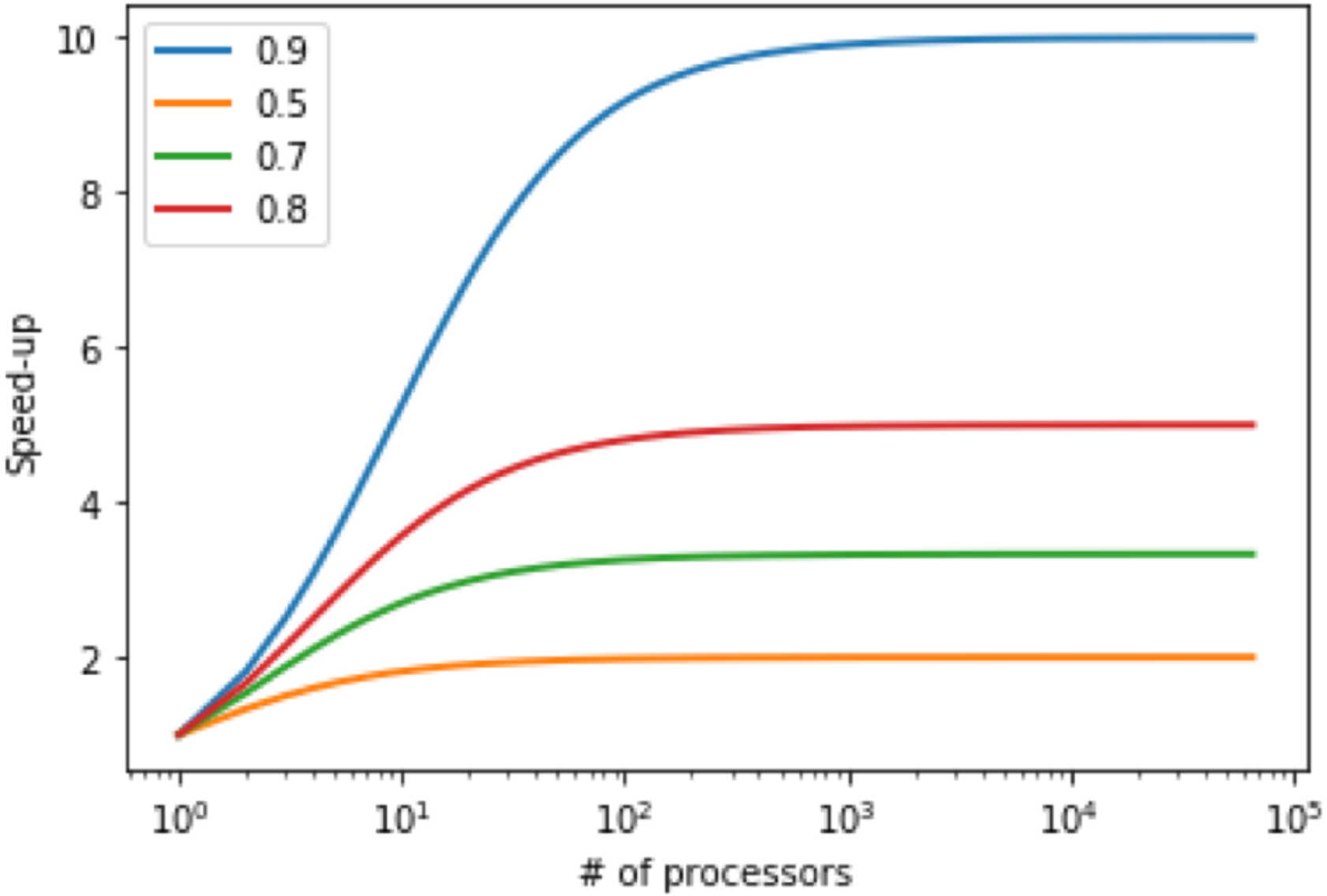
Speed-up of a parallel program/work:

$$S = \frac{1}{(1-f) + \frac{f}{n}}$$

f – is a fraction of the work that CAN be done in parallel

n – number of parallel workers

Amdahl‘s law implication



Weak and strong scalability

- **Scalability** – the ability of hardware and software to deliver greater computational power when the amount of resources is increased.

- **Strong scaling** measures how much the software is scalable when the problem size is fixed. => *Amdahl's law.*

- **Weak scaling** measures how much the software is scalable when the problem size per processor is fixed => *Gustafson's law.*

Gustafson‘s law

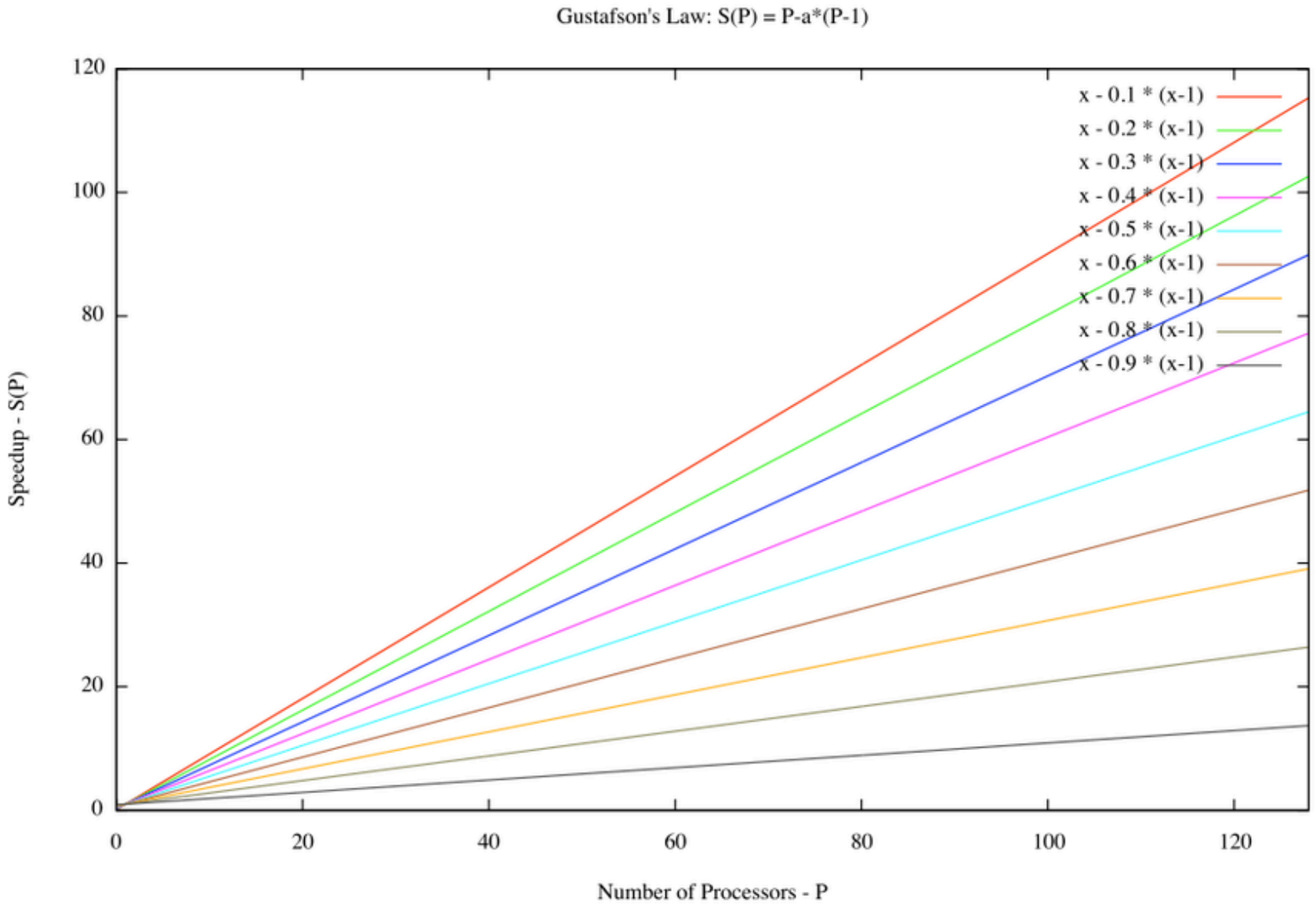
Speed-up of a parallel program/work:

$$S = N + (1 - N)s$$

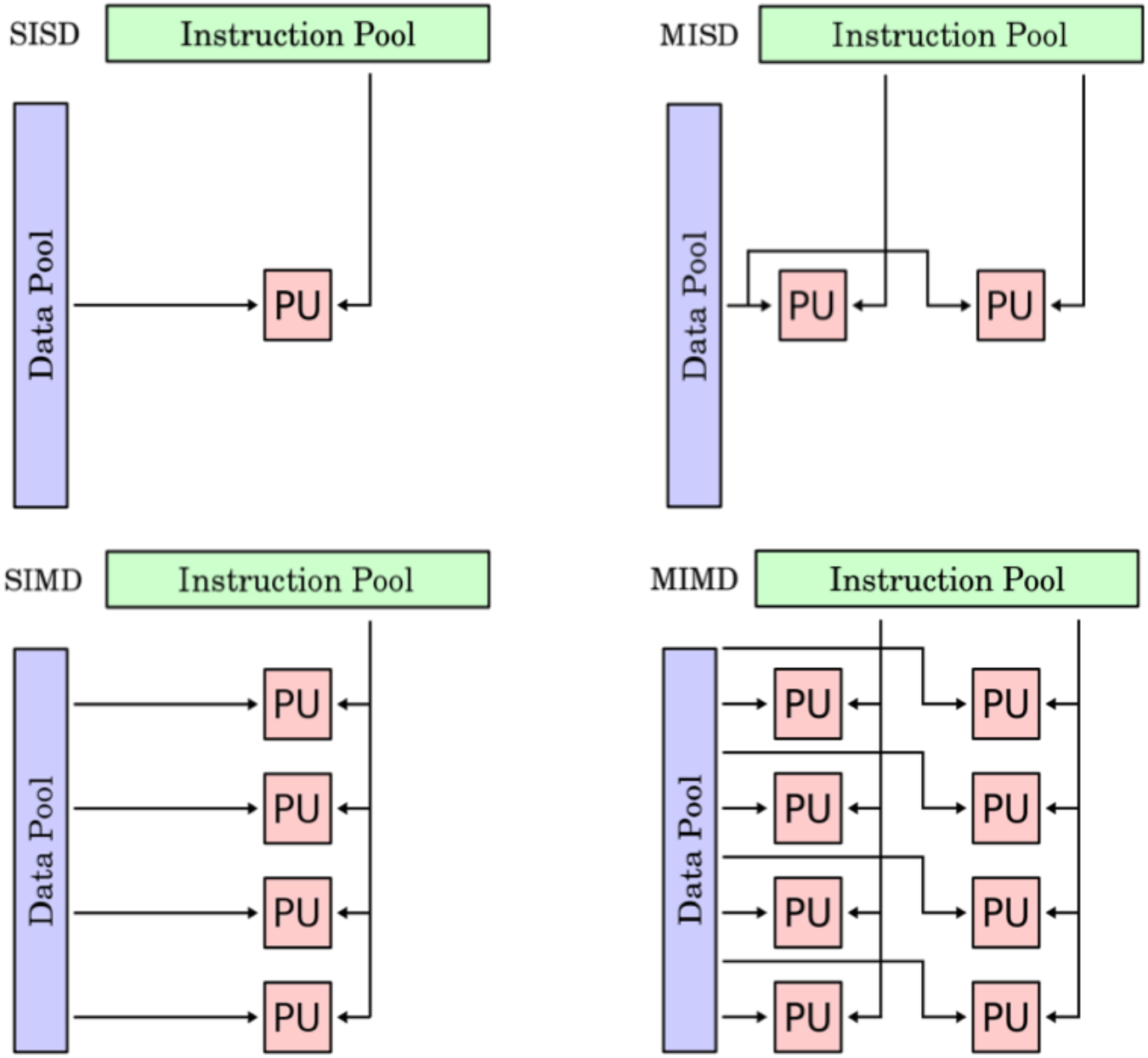
s – is a fraction of the work
that CANNOT be
done in parallel

N – number of parallel workers

Gustafson's law implication

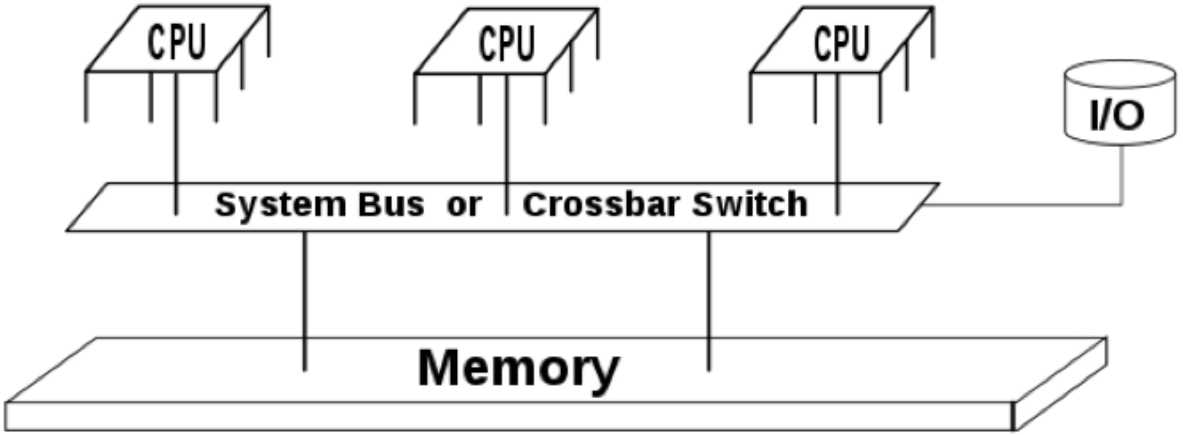


Flynn's taxonomy

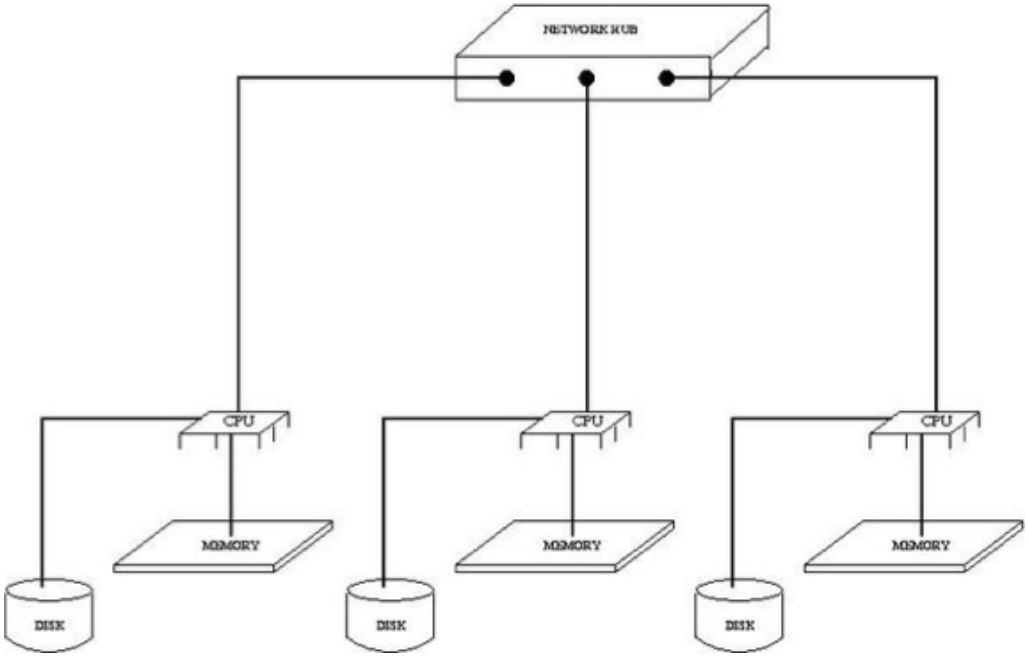


Shared memory and distributed memory

shared memory



distributed memory



Ways to work parallel

1. Scientific software (Matlab, Mathematica, ANSYS, OpenFoam, Tensorflow, PyTorch, etc)

2. Libraries (Math Kernel Library, CuDNN, CuBlas, Thrust, Boost)

3. Parallel instruments (OpenMP, MPI, Cuda, OpenCL)

Types of parallelism in computing

1. Task parallelism (multi-core CPUs) – different tasks or instructions can be done concurrently
2. Data parallelism (GPUs) – same instruction can be done on multiple data concurrently

Types of parallelism in computing

1. Task parallelism (multi-core CPUs) – different tasks or instructions can be done concurrently
2. Data parallelism (GPUs) – same instruction can be done on multiple data concurrently

Types of parallel tasks:

1. Embarrassingly parallel tasks (aka ”farming”) – no communication between workers

   bitcoin mining, solving integral with a parameter for 1e6 parameters, running simulation for different input parameters (different initial conditions)

2. Compute-intensive tasks

   main time is spent on computing, low level of communication, i.e. numerical photonics or plasma physics simulations

3. Data-intensive tasks

   main time is spent on data transfer (big data, operations with file system, databases)

Processes and threads

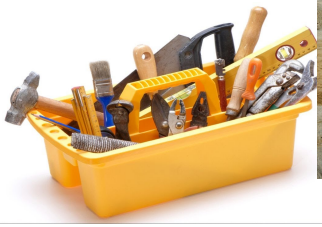
Processes and threads

Thread

Thread

Task

Program



Address space

Profiling

- Analysis of code that shows us show efficient our resource usage is.
- Resource can be: CPU time, memory, network bandwidth, etc.
- Today we focus on CPU time and memory.
- There are a lot of visualizers.

Memory profiling

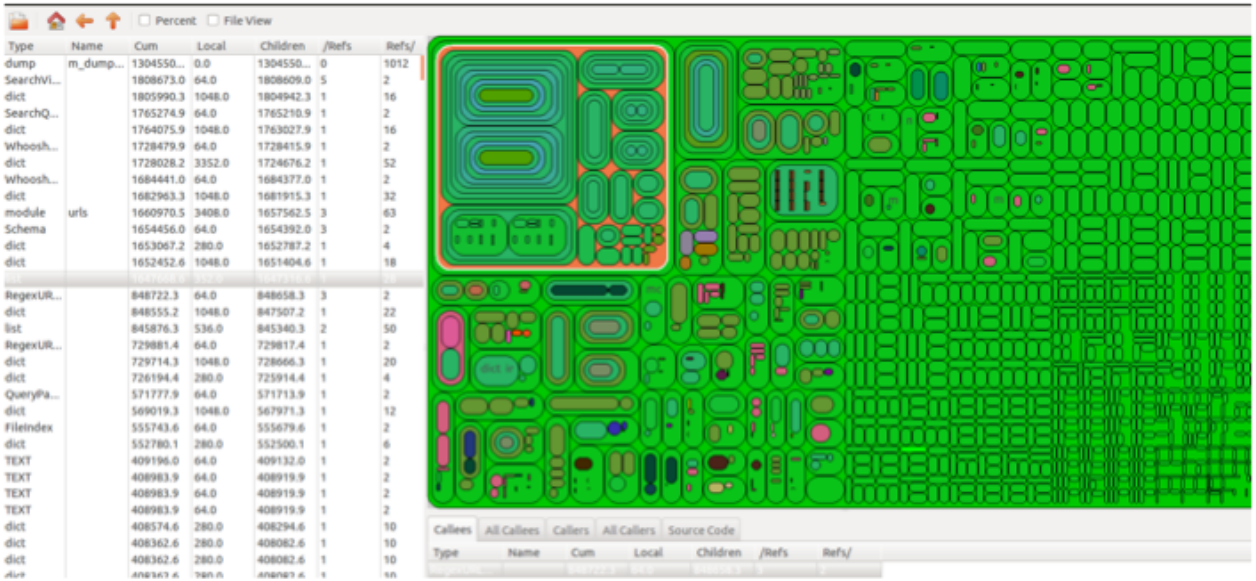
```
> $ python -m memory_profiler memory_profiler_demo.py
Filename: memory_profiler_demo.py

Line #    Mem usage    Increment   Line Contents
================================================
     8     51.9 MiB     51.9 MiB   @profile
     9                             def test1():
    10     51.9 MiB      0.0 MiB       c = []
    11     75.9 MiB     24.0 MiB       a = [1, 2, 3] * (2 ** 20)
    12     83.9 MiB      8.0 MiB       b = [1] * (2 ** 20)
    13    107.9 MiB     24.0 MiB       c.extend(a)
    14    115.9 MiB      8.0 MiB       c.extend(b)
    15    115.9 MiB      0.0 MiB       del b
    16    115.9 MiB      0.0 MiB       del c
```

Measures memory usage of the program. Understanding the memory usage characteristics of your code allows you to ask yourself two questions:

- Could we use less RAM by rewriting this function to work more efficiently?
- Could we use more RAM and save CPU cycles by caching?

Time profiling



Measures time distribution among function calls of a program.
Understanding the CPU time usage characteristics of your code allows you to ask yourself the following questions:

- Can we identify suspicious functions that take too much computation time?
- Can we optimize functions so that they would use less CPU computation time?

Profiling tools for python

1. CPU time profiling tools:
    1. timeit
    2. cProfile
    3. line_profiler

2. Memory profiling tools:
    1. memory_profiler
    2. heapy

Exercises

1. Profile bifurcation map script from lecture 1:
   - use any of the disscussed profiling tools to see how much time each part of the script takes, also do memory profiling;
   - try varying different parameters ("n" and "m")
   - which parameters influence change the profiling configuration most?

2. Profile example script that performs spectogram:
   - profile computation time of "get_spectogram" function
   - can you find a way to increase the speed of the function?

3. Profile your own python code.