Pázmány Péter Catholic University
Faculty of Information Technology and Bionics

Parallel Programming

Lecture 1

István Reguly

reguly.istvan@itk.ppke.hu

Structure

- 3 hour lab sessions every week
- 10 classes (Easter break), tests at each, 50%
- Homework 50% (there will be ~ 7-8)
- Textbooks*:
- Introduction to High Performance Computing for Scientists and Engineers, Hager G and Wellein G (2010)
- CUDA by Example: An Introduction to General-Purpose GPU Programming, Sanders and Kandrot (2010)

*Useful, but not required

Course content

- Overview of parallel computing and high performance computing
- Multi-threading on CPUs
- Vectorisation on CPUs
- Distributed Memory computing with MPI
- Introduction to GPU architectures
- Programming with OpenACC
- CUDA basics
- CUDA memory types
- CUDA libraries
- CUDA performance profiling & optimisations

Parallel computing

- So what is parallel computing?
  - Any computing where tasks are executed simultaneously to solve one problem
    - Embarrassing parallelism: no communication between tasks
    - If there is communication between tasks, there are many-many ways to do that
  - Not to be confused with multiprocessing, where multiple unrelated tasks execute

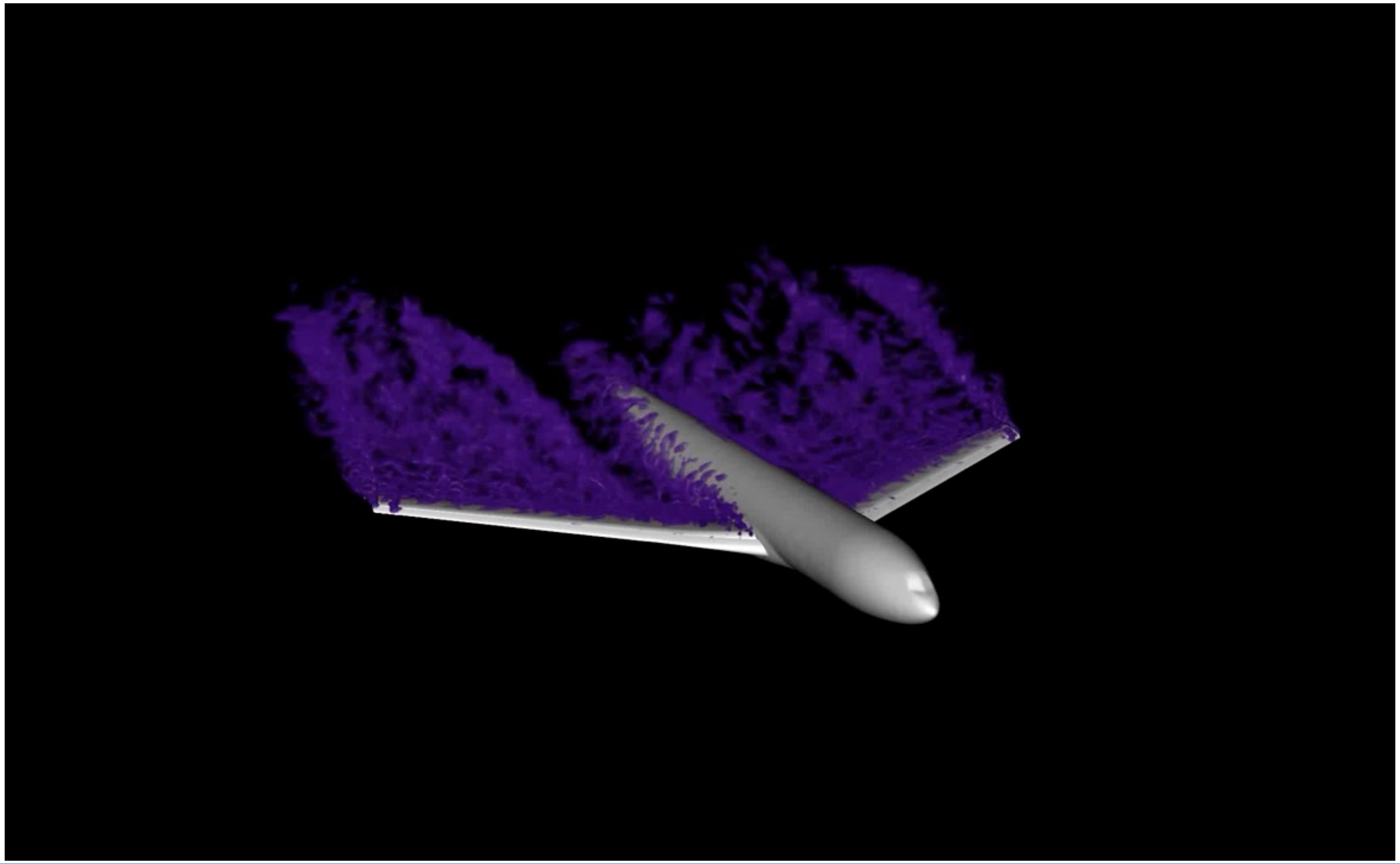
- Why parallel computing?
  - Because all computers are now parallel!
  - How many cores in my laptop?

Parallel computing is everywhere

Computer aided design – optimising, fine-tuning the design is much faster

Parallel computing is everywhere

Computer aided design – improving fuel efficiency & noise



Parallel computing is everywhere

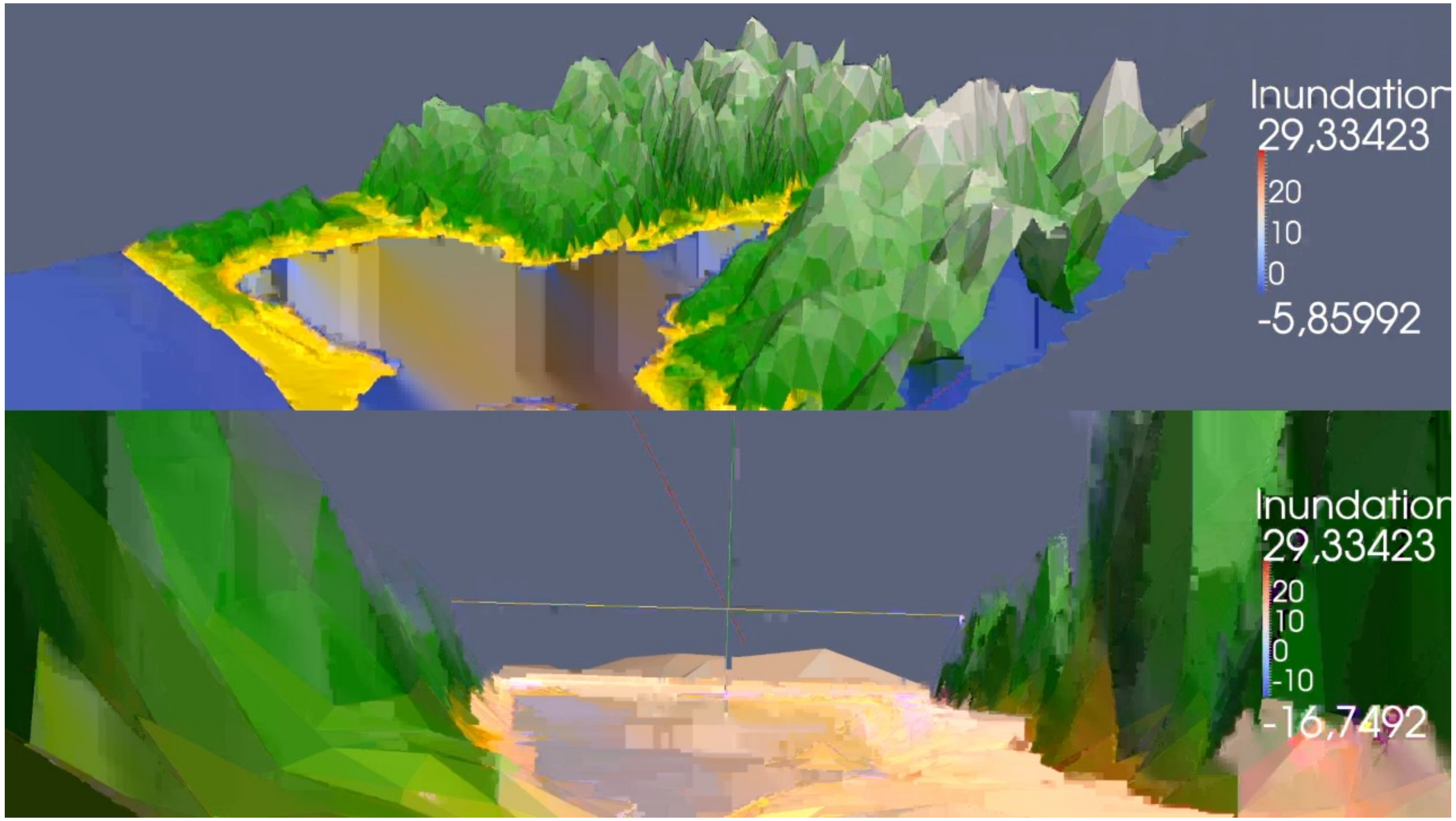
Designing new drugs, understanding diseases

Parallel computing is everywhere

Risk assessment – from man-made disasters to natural disasters

Parallel computing is everywhere

Risk assessment – from man-made disasters to natural disasters



Parallel computing is everywhere

Understanding the universe – Milky Way – Andromeda collision

Parallel computing is everywhere

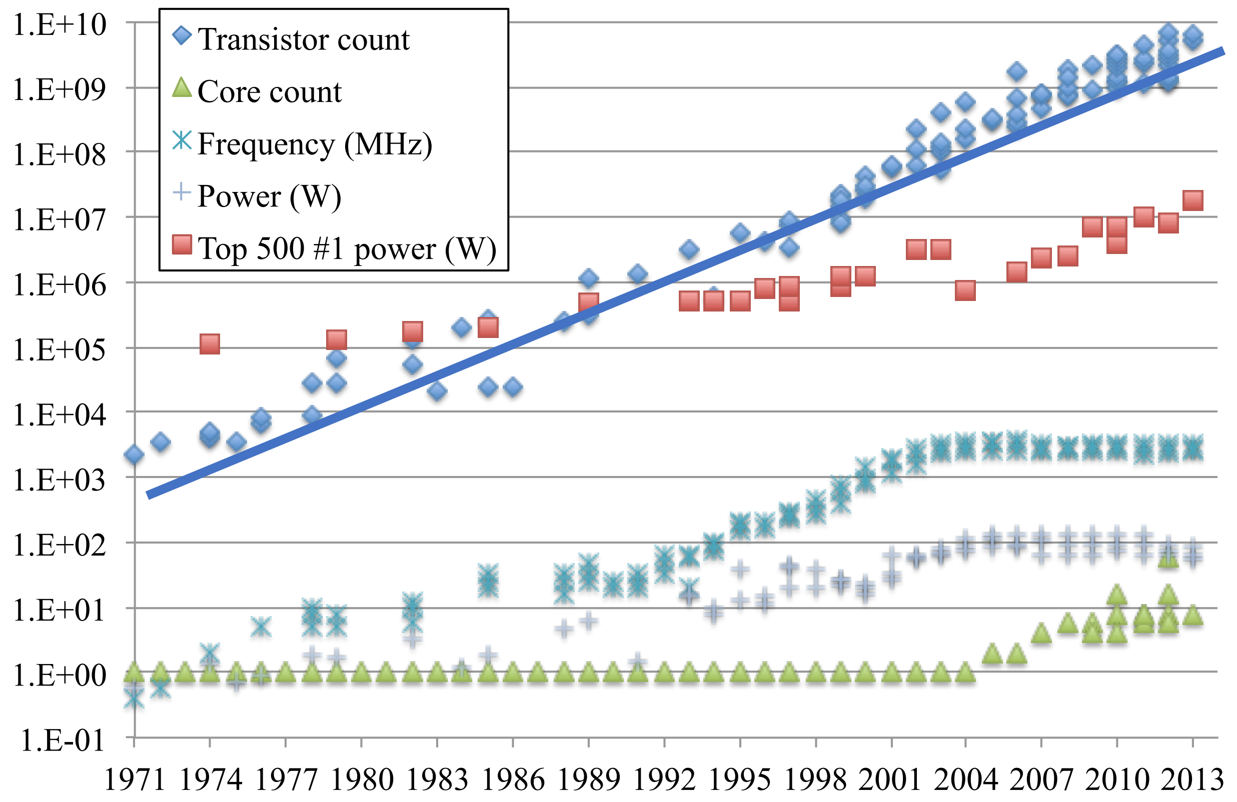
Making movies

Scientific computing

- ACM Gordon Bell Prize
- 2022: 3 supercomputers – plasma simulation
- 2021: 41M cores, quantum circuit simulation- ?? MW
- 2020: 138M cores, 3.6 Pflops, COVID molecule – 13 MW
- 2019: 138M cores, 85 Pflops, quantum simulation – 13 MW
- 2018: 2.4M cores, 2.36 ExaOps, genomics research- 9.7 MW
- 2017: 10.5M cores, 19 Pflops, earthquake simulation – 15 MW
- 2016: 10.5M cores, 8 Pflops, 500 m resolution atmospheric model with 770 billion unknowns, 0.07 years per day – 15 MW
- 2015: 1.5M cores, earth's mantle movement, 602 billion unknowns – 7MW
- 2014: Anton-2 custom machine for molecular dynamics: 23k atoms 85 us/day

Why parallel computing?

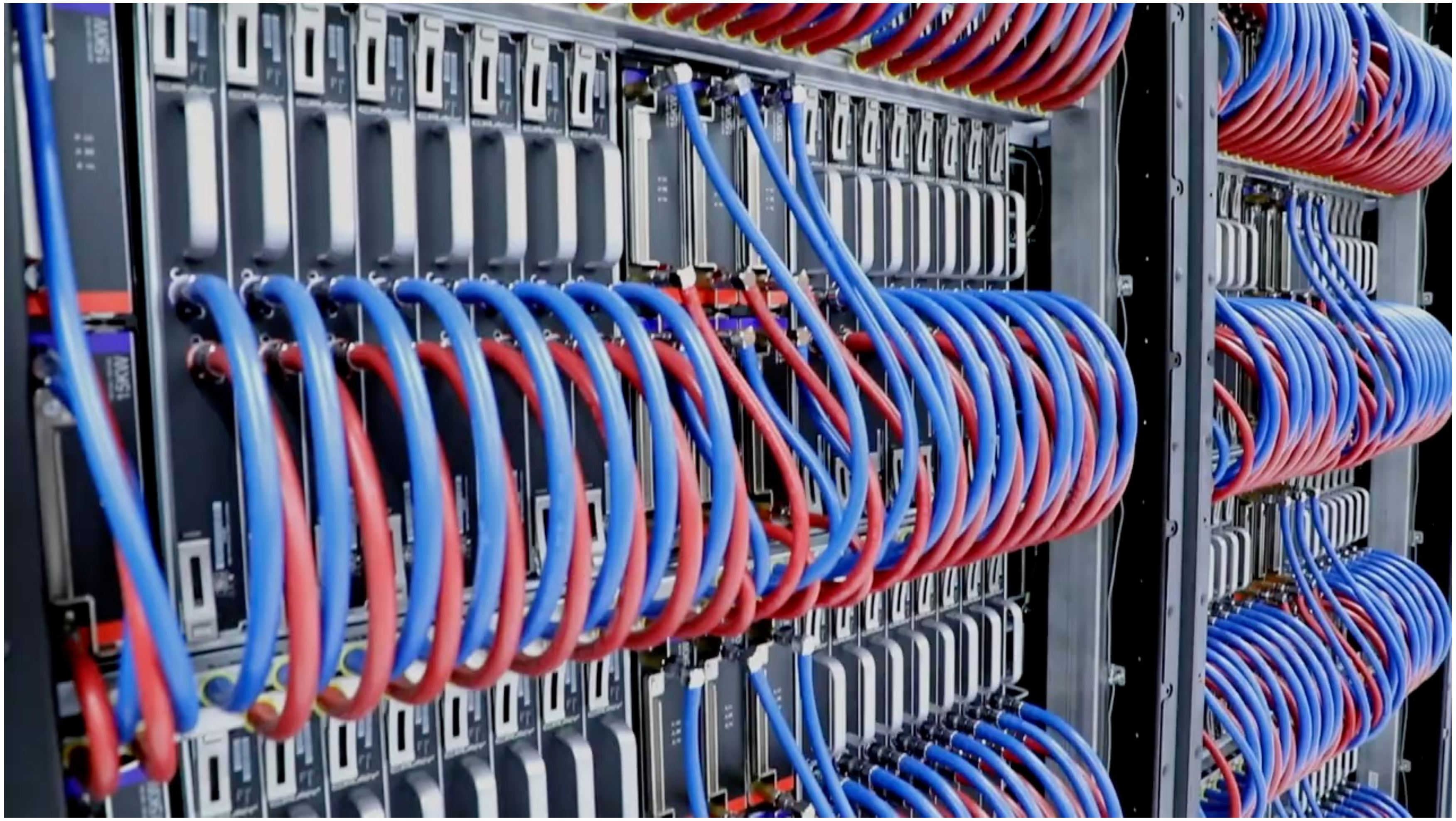
Physical limits



Why parallel computing?

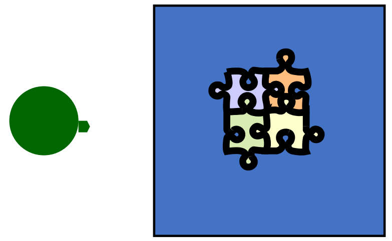
- Do it faster
  - In theory, using more resources will give you the solution faster
- Solve larger or more complex problems
  - Many problems are too complex to be solved on a single computer (e.g. can’t fit in memory, would take too long)
- Provide concurrency
  - Single computer can only do one thing at a time, multiple resources can do many things simultaneously

What does a supercomputer look like?

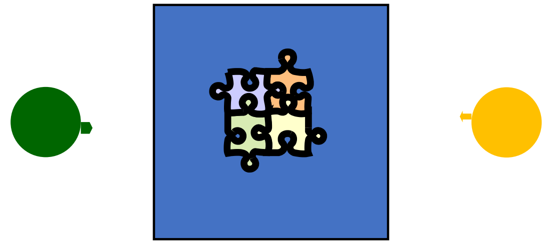


Parallel computing

- Jigsaw puzzle analogy – Henry Neeman @ Oklahoma University
- Serial computing: suppose you want to do it yourself – 1000 pieces takes about an hour

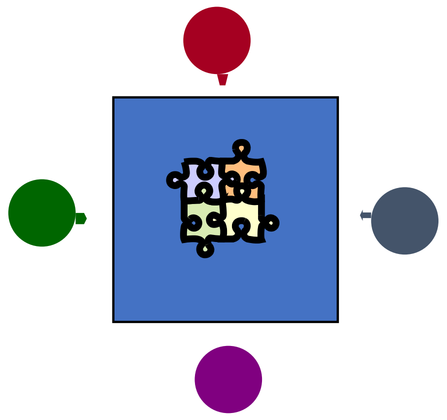


- Suppose your friend sits down and tries to help you: now you need to communicate when working on close by parts, and sometimes you reach into the pile at the same time: contend for resources. You finish the puzzle in 35 minutes instead of 30



Parallel computing

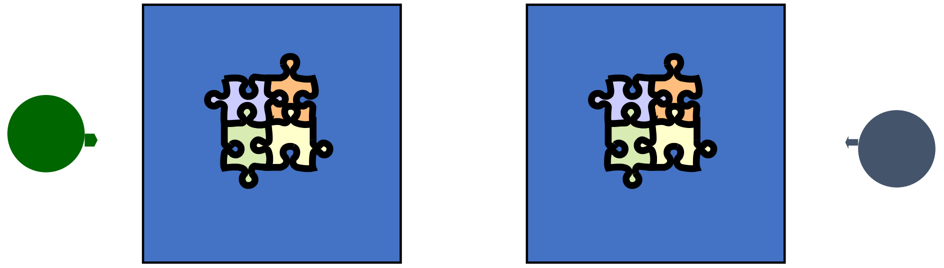
- The more the merrier? Suppose two more friends sit down and try to help: speedup is not significantly less that 4x – say 3x and you finish in 20 minutes



- Diminishing returns: as more and more friends join, there will be a lot more contention and need for communication – adding more resources to solve the same problem leads to diminishing returns

Distributed parallelism

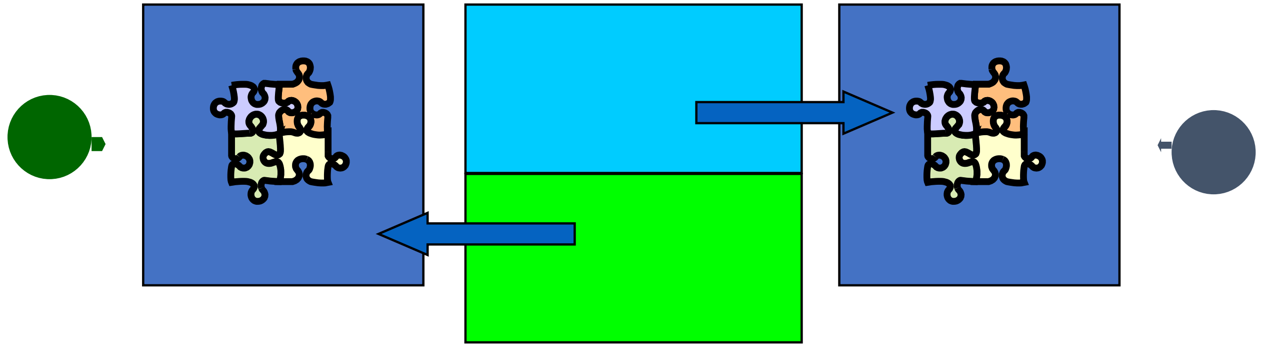
- Let’s try to split the puzzle pieces between two tables and solve it that way: there is now no contention, but the cost of communication is a lot higher – you will have to combine your results at the end



- Additionally, you have to split up your puzzle pieces (decompose) in the beginning. Depending on how much time you spent on a ”good split”, the communication later on may be more or less
  - Perfect split with pieces on the two sides of the final puzzle, vs. random split: it’s a lot easier at the end to put the two halves together, but a lot more work in the beginning

Distributed parallelism

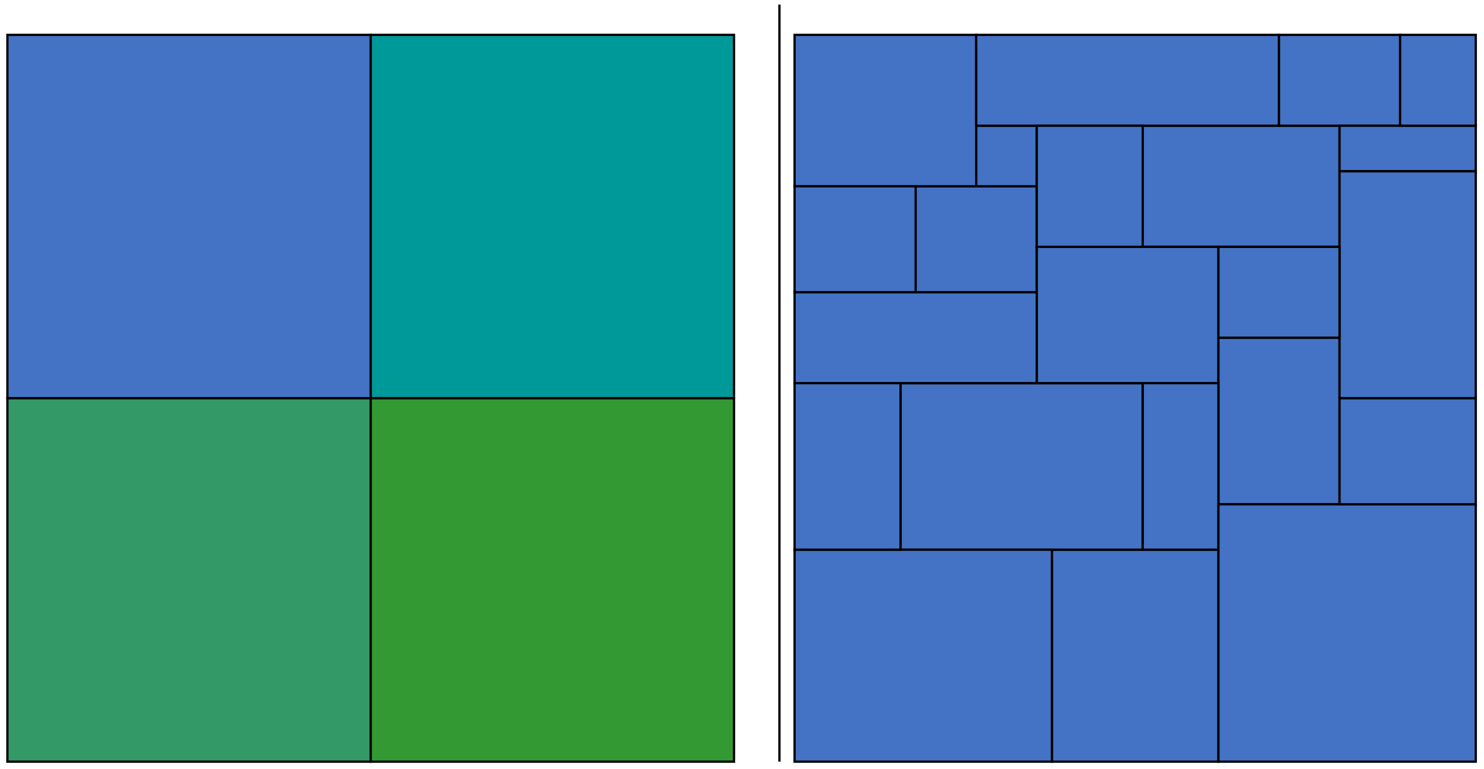
- The initial decomposition determines the load balancing: you try to give everyone the same amount of work to do. If the picture is half grass, half sky, it’s easy, and you’ll have to communicate only on the grass-sky boundary



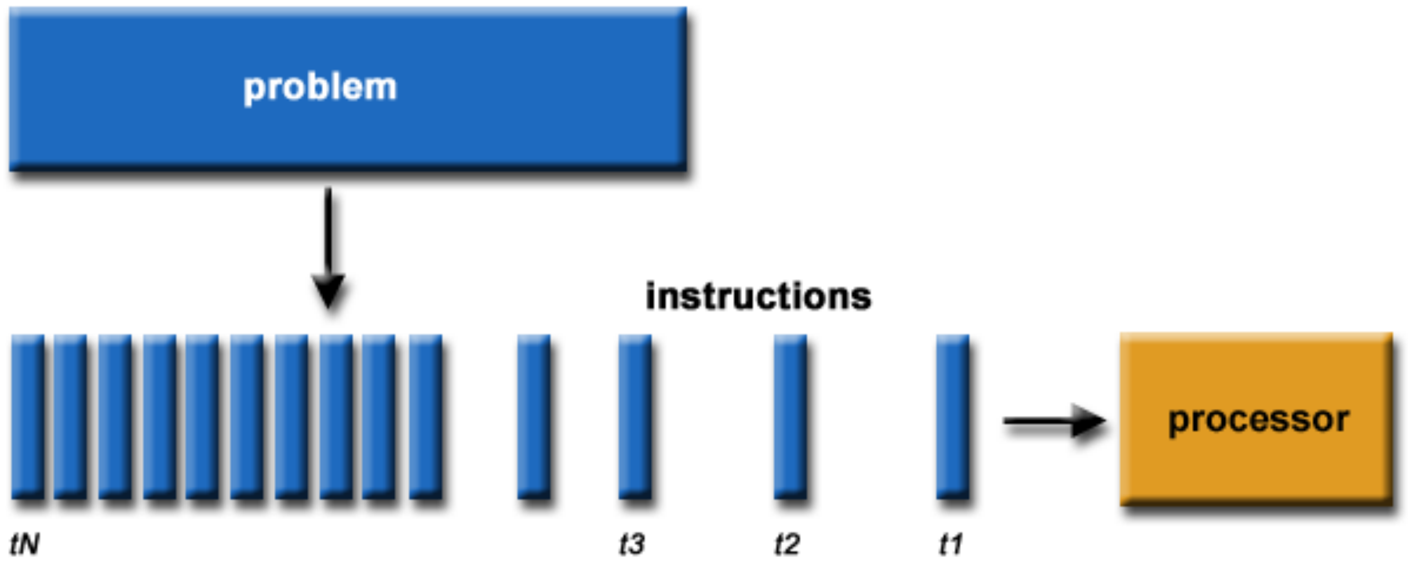
- Some problems lend themselves to good decompositions and a good load balance.

Distributed parallelism

- Load balancing can get difficult, particularly if Jane is better at doing puzzles than Joe

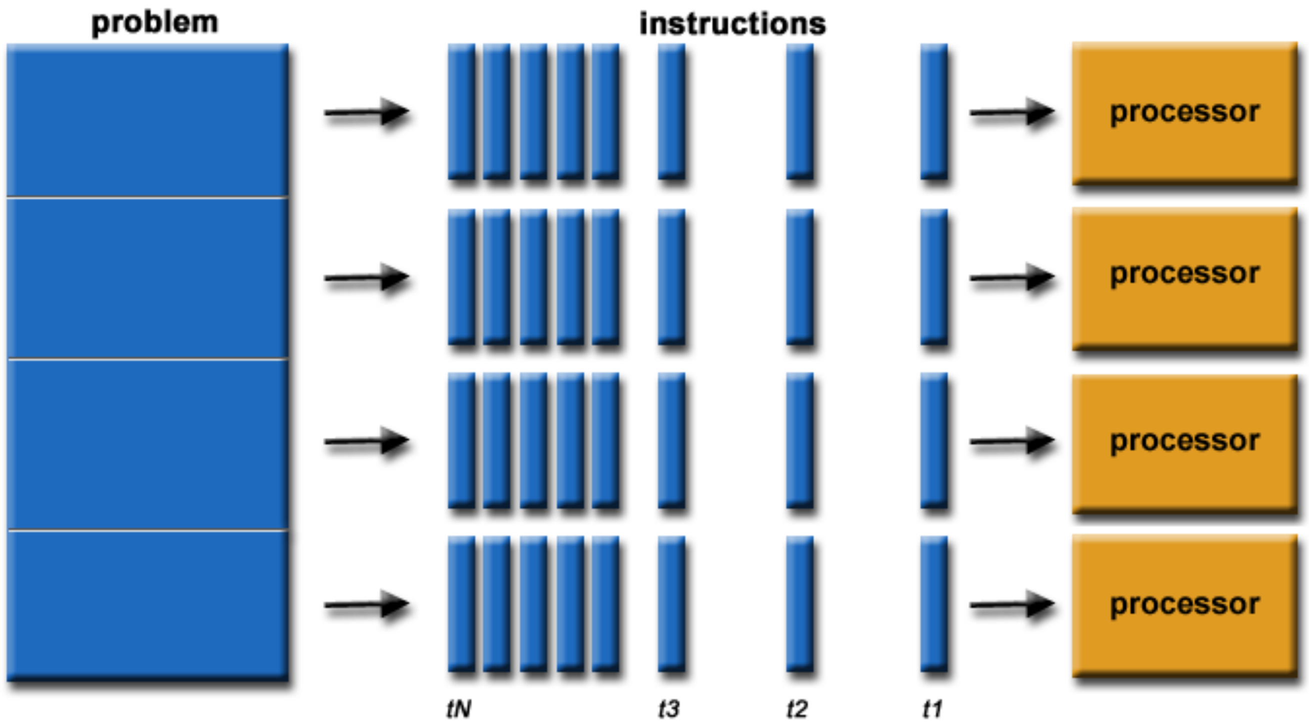


Serial computing



Traditionally: problem broken down into a series of instructions, executed serially and in-order on a single processor. Reality has been different for a long time...

Parallel computing



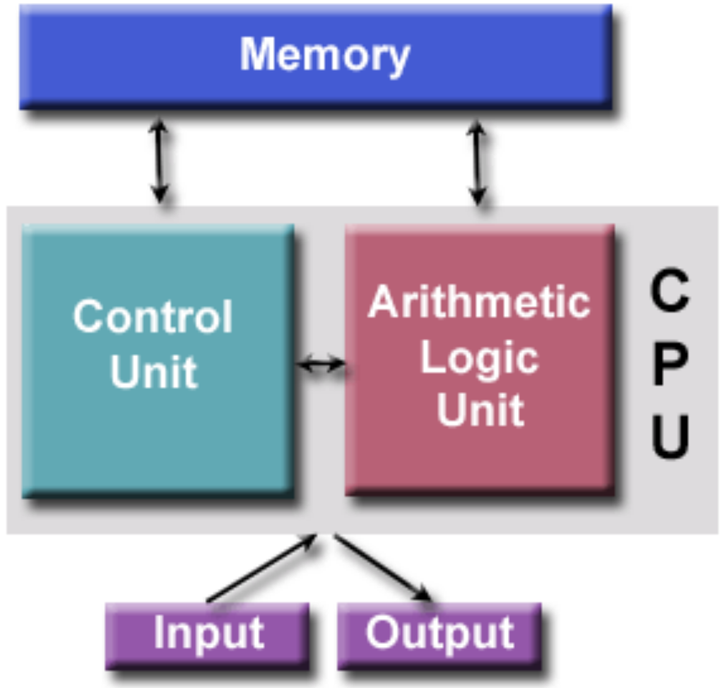
Simultaneous use of multiple resources. Problem broken into parts that can be solved concurrently, then each into a sequence of instructions. Central coordination.

Parallel computing

- The computational problem should be able to:
  - Be broken into discrete pieces of work that can be solved simultaneously (independently?);
  - Execute multiple instructions at any moment in time;
  - Be solved in less time using multiple resources than with a single compute resource.
- What are the “compute resources”:
  - Single computer with multiple processors/cores.
  - A network of these computers, connected.

Neumann architecture

- Stored program computer – program and data are kept in the same memory.
- Memory stores instructions and data
- Control fetches instructions or data from memory, decodes instructions and **sequentially** coordinates operations
- ALU
- These are the three components you will have to think about w.r.t. performance



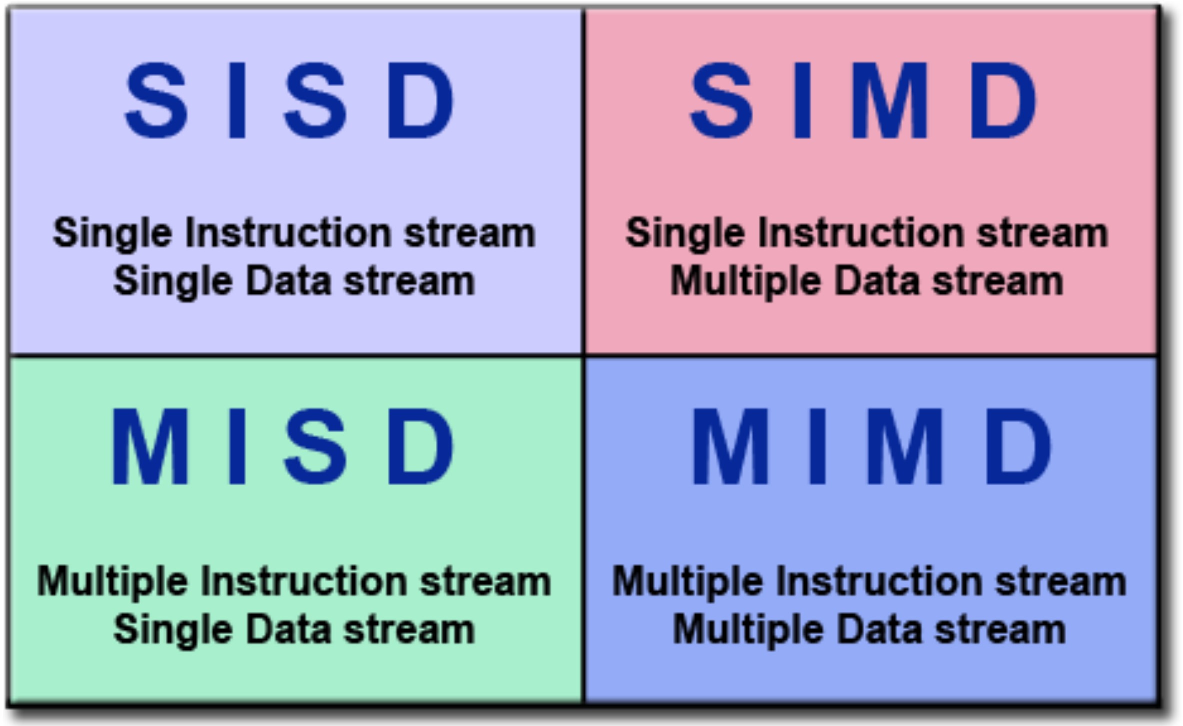
Parallel computers still follow this design, just multiplied in units

How to make parallel Neumann architectures?

- Traditional single core CPUs no longer scalable due to frequency limit, but consumers still expect performance growth
- With process scaling (32-16-12nm) you can put more transistors on the chip – how are you going to use them?
  - Put multiple CPU cores onto the same chip – these can now function independently, but they still see the same memory
  - But CPU cores have a lot of logical circuits – a large control control unit - that help them run fast
    - Out of order execution, branch prediction, etc.
    - You have a lot of overhead to feed the arithmetic logic unit
  - Mitigate the overhead of the control unit, by having it control many ALUs
    - Compute the same things (same instruction), just on different data
  - Make larger and faster caches

Classification of parallel computers

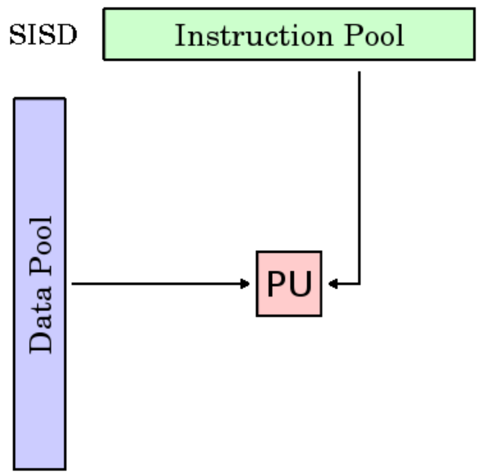
Flynn’s taxonomy, two dimensions: instruction stream and data stream, each may be single or multiple.

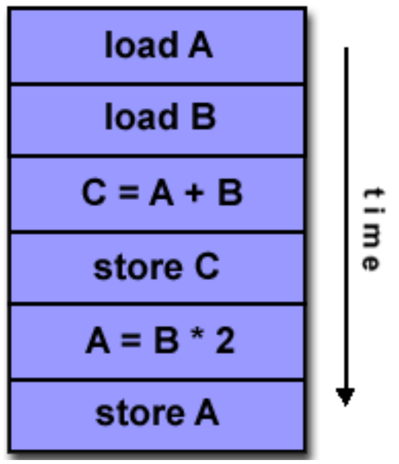


SISD Single Instruction stream Single Data stream	SIMD Single Instruction stream Multiple Data stream
MISD Multiple Instruction stream Single Data stream	MIMD Multiple Instruction stream Multiple Data stream

Single Instruction Single Data (SISD)

- Classical, serial computer
- Single instruction: only one instruction stream is being acted on by the CPU during any one clock cycle
- Single data: Only one data stream is being used as input during any one clock cycle





Single Instruction Single Data (SISD)

UNIVAC1

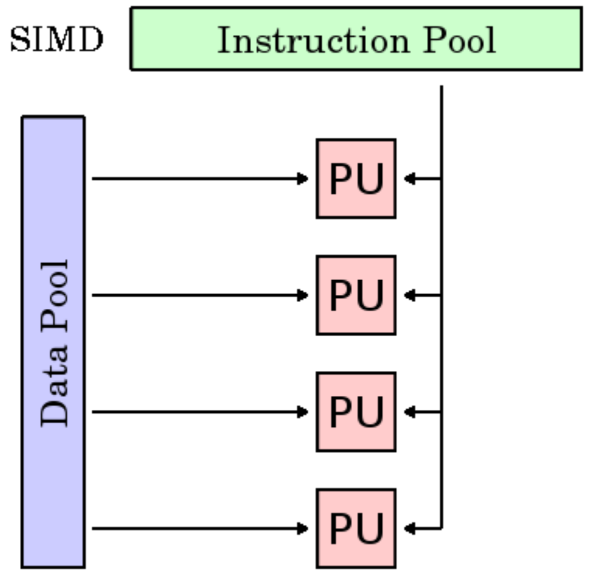
CRAY1

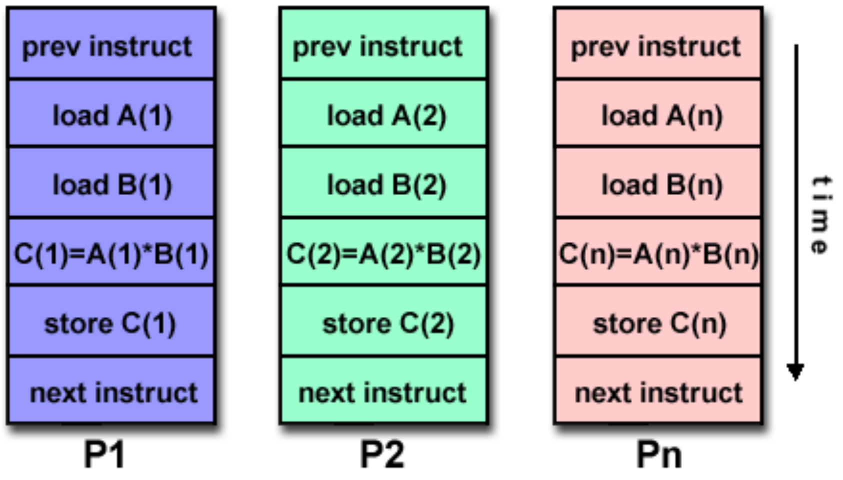
PDP1

Dell Laptop, late 90s'

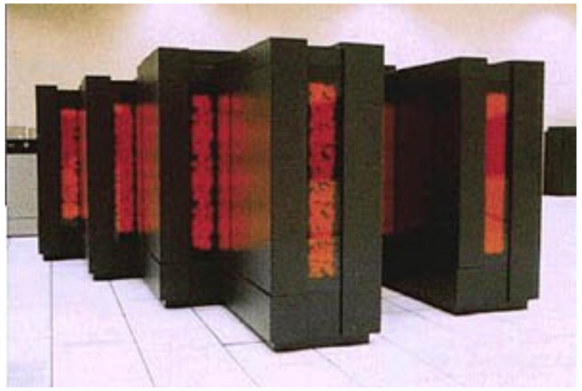
Single Instruction Multiple Data (SIMD)

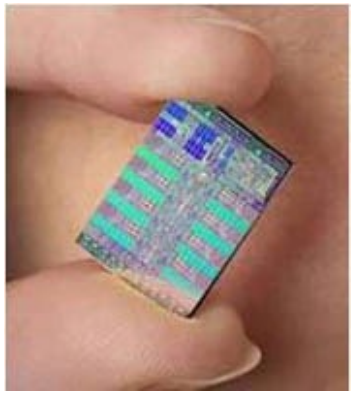
- Single instruction: all processing units execute the same instruction at any given clock cycle
- Multiple data: each processing unit can operate on a different data element
- Synchronous (lockstep) execution, best for highly regular computations



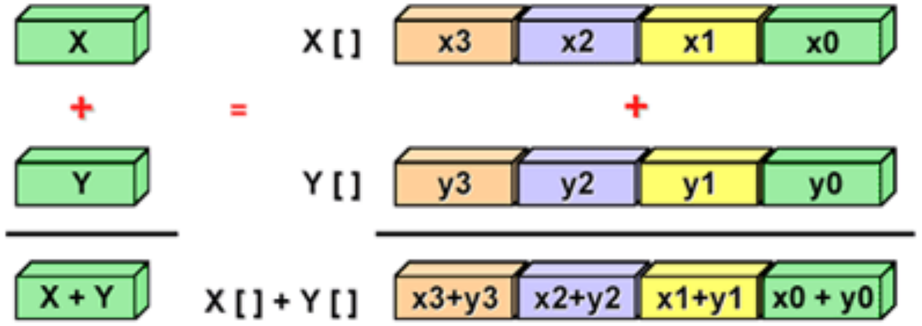


Single Instruction Multiple Data (SIMD)


Thinking Machines CM-2

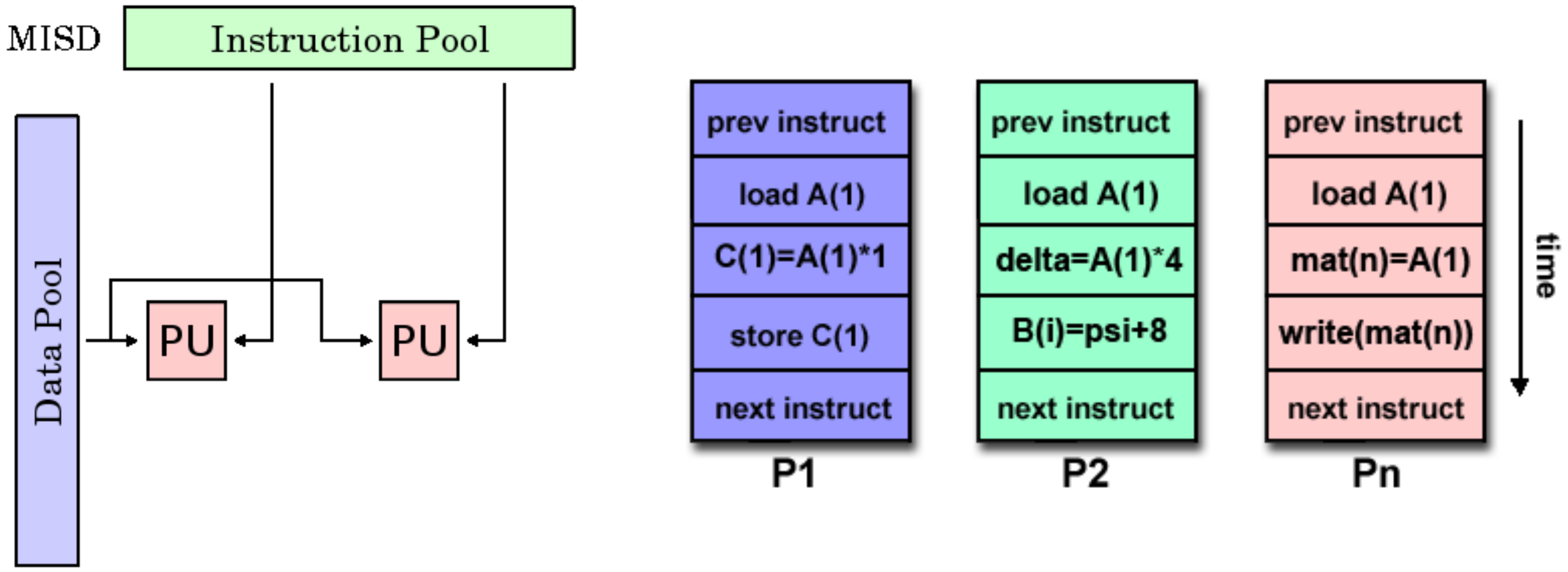

IBM Cell

GPUs



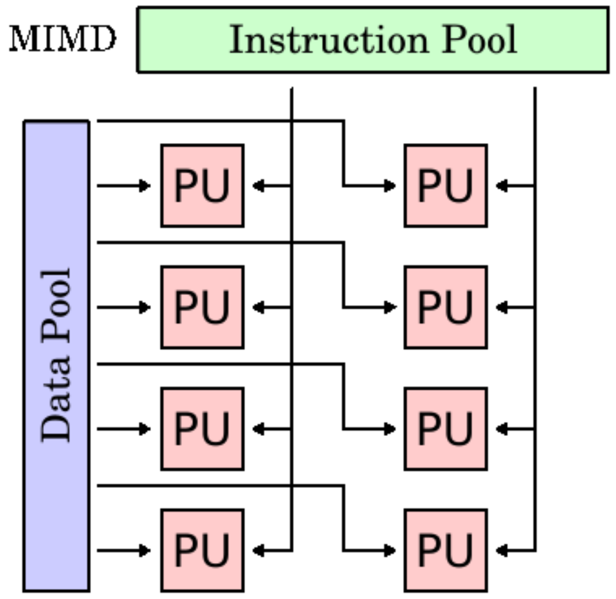
Multiple Instruction, Single Data (MISD)

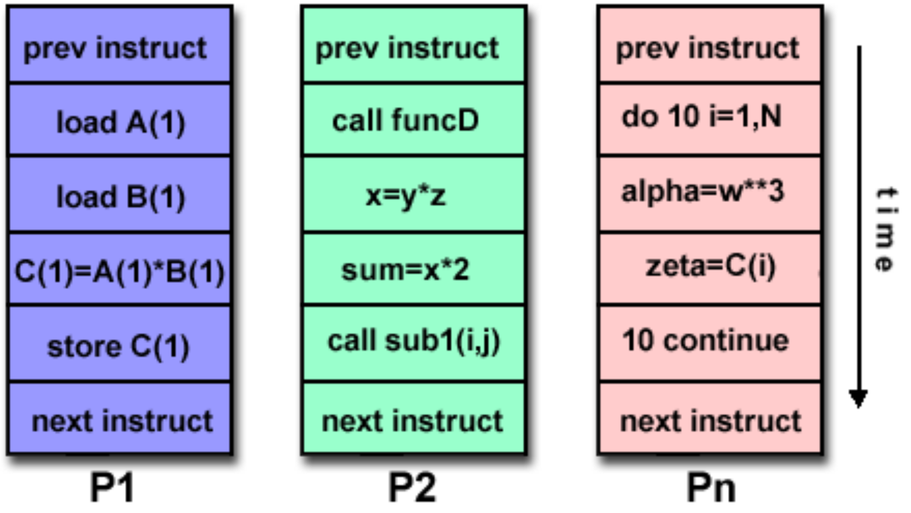
- Multiple Instruction: each processing unit operates on the data independently
- Single Data: single data stream is fed into multiple processing units
- Not really used



Multiple Instruction, Multiple Data (MIMD)

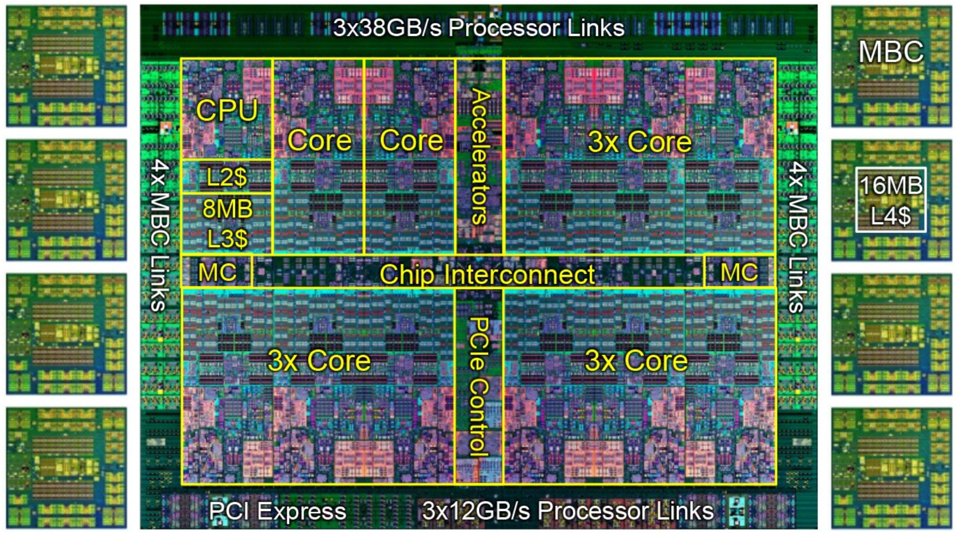
- Multiple instruction: each processor executing a different instruction stream
- Multiple data: each processor works with a different data stream
- Synchronous/asynchronous
- Most supercomputers, multicore processors



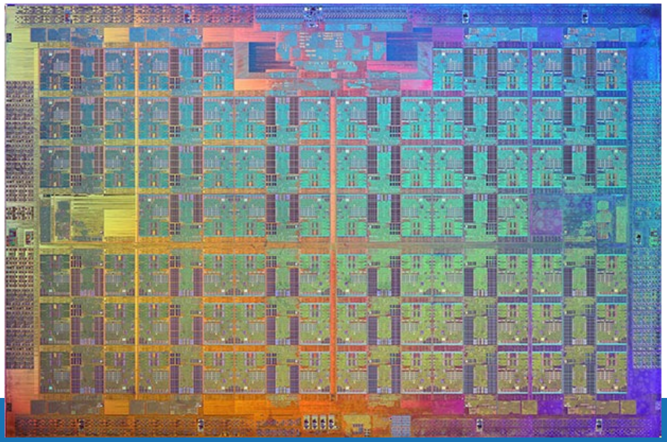


Multiple Instruction, Multiple Data (MIMD)

IBM BlueGene/Q



IBM Power8 (12 cores)



Intel Xeon Phi (72 cores)

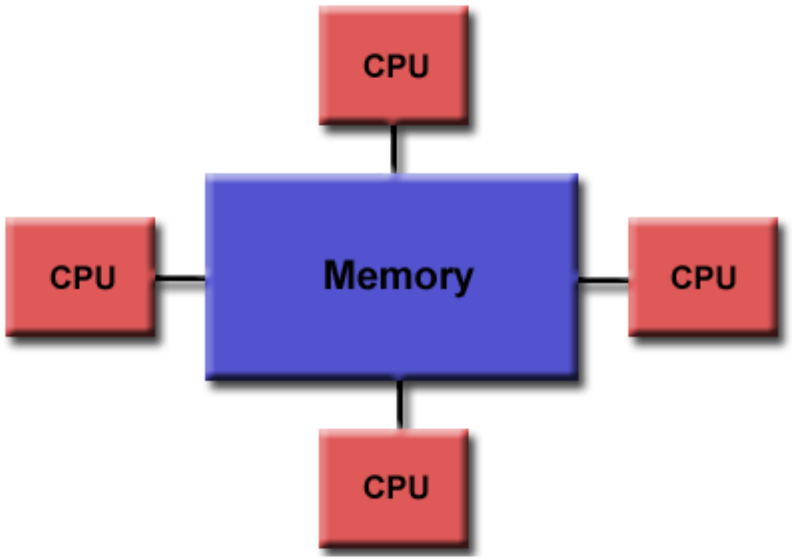
Programming parallel hardware

- Now we have some idea about how parallel computers work, and what are some of the ways in which parallelism can happen. How do we write code to make use of them?
- Turns out parallel programming models are very close to how parallel hardware work
  - Which makes sense, the better they match, the easier it is to map code to stream(s) of instructions – the better performance you can expect
  - SISD – all the classical languages, C/C++, Fortran, Java, etc.
  - SIMD – compiler methods (transparent) and languages like CUDA, OpenCL
  - MIMD – multi-threading with OpenMP/TBB, etc, multiple processes with MPI
- However, a specifying parallel computations is just one side of the coin, specifying communications is perhaps even more important
  - We distinguish between shared memory models and distributed memory models

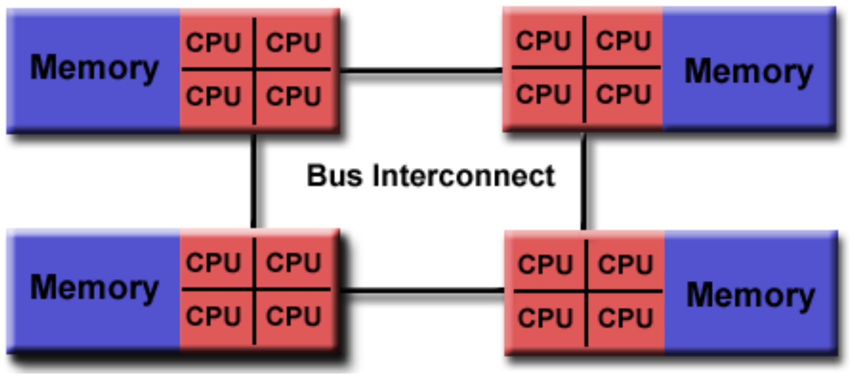
OpenMP

- Open Multi-Processing
- Three components:
  - Compiler directives
  - Runtime library routines
  - Environment variables
- High-level model
  - Implicit mapping and load balancing of work
- Standard
- Portable

OpenMP view of memory



Uniform Memory Access



Reality may be:
Uniform Memory Access

Compiler directives

- Appear as comments in the code and ignored by the compiler unless instructed to (compiler flag)
- OpenMP directives are used for various purposes:
  - Spawning a parallel region
  - Dividing blocks of code among threads
  - Distributing loop iterations between threads
  - Serializing sections of code
  - Synchronization of work among threads
- Syntax:

```
Sentinel    directive-name      [clause, ...]
#pragma omp parallel default(shared) private(beta,pi)
```

Environment variables

- OpenMP provides several environment variables to control the execution of parallel code
- Can be used to:
  - Set number of threads
  - Specify how loop iterations are divided
  - Binding threads to physical processors
  - Controlling nested parallelism, dynamic threads
  - etc.

```
export OMP_NUM_THREADS=4
```

Vector add

```
#include <omp.h>
#include <iostream>
#include <chrono>
#include <vector>
#define N     100000000
int main (int argc, const char **argv)
{
  std::vector<float> a(N), b(N), c(N);
  /* Some initialisation */
  for (int i=0; i < N; i++)
    a[i] = b[i] = i * 1.0;
  auto t1 = std::chrono::high_resolution_clock::now();
#pragma omp parallel for
  for (int i=0; i < N; i++)
    c[i] = a[i] + b[i];
  auto t2 = std::chrono::high_resolution_clock::now();
  std::cout << "took "
    << std::chrono::duration_cast<std::chrono::milliseconds>(t2-t1).count()
    << " milliseconds\n";
}
```

Timing your code

- In previous courses we rarely looked at how much time it takes to run something
  - During this course, that is one of the most important metrics of how well we are doing
- C++11 chrono library

```
#include <iostream>
#include <chrono>
auto t1 = std::chrono::high_resolution_clock::now();
//Do domething…
auto t2 = std::chrono::high_resolution_clock::now();
std::cout << "took "
    << std::chrono::duration_cast<std::chrono::milliseconds>(t2-t1).count()
    << " milliseconds\n";
```

Compiling your code

- In previous courses, we use IDEs (integrated development environments) to compile & run our code
  - Green “Play” button manages everything
    - Debug & Release mode
  - We need to better understand what compilation does, and what will affect the performance of our code
  - We’ll usually compile by hand
- Compiling:
  - CC your_file.cpp   -o your.exe
- Running:
  - ./your.exe
- Compiling faster code:
  - Add the -Ofast flag
- Compiling with OpenMP support:
  - Add the -fopenmp flag
- Try and see the difference on the vector add code!

Connecting

- We’ll be using Komondor – Hungarian supercomputer
  - Based in Debrecen, 128 CPU server, 58 GPU servers (4x NVIDIA A100)
  - https://docs.hpc.kifu.hu/
  - Running stuff with SLURM: https://docs.hpc.kifu.hu/first-steps/slurm.html
  - Interactive jobs & batch jobs
- Registration – https://portal.hpc.kifu.hu
- SSH connection via VS Code
  - Two-factor authentication – click on details tab

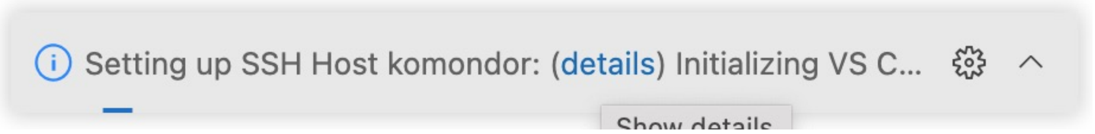
Setting up VS Code with Komondor

- Add the following to the ssh config

```
Host komondor
  User p_cov
  HostName komondor.hpc.kifu.hu
```

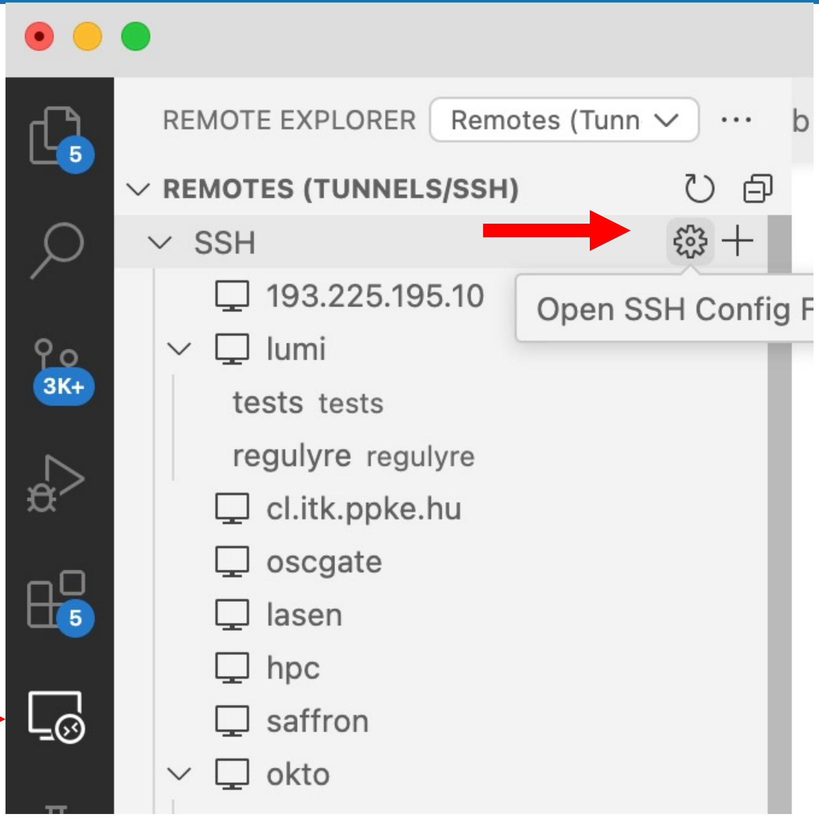
Your username

- Save, then refresh

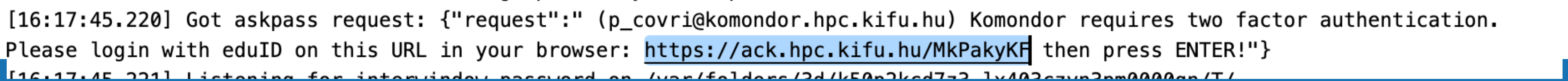


Setting up SSH Host komondor: (details) Initializing VS C...

Show details



- Find this line after clicking details – open in browser, authenticate, back to VS code, hit enter in top dialog



```
[16:17:45.220] Got askpass request: {"request":" (p_covri@komondor.hpc.kifu.hu) Komondor requires two factor authentication.
Please login with eduID on this URL in your browser: https://ack.hpc.kifu.hu/MkPakyKF then press ENTER!"}
[16:17:45.221] Listening for interwindow password on /var/folders/3d/k50n2kcd7z3_1x403czyn3pm0000gn/T/
```

Using Komondor

- Create new file/directory
- To run, get an interactive session:
  - srun --reservation=p_covidpre_115 -p cpu -c 4 --mem-per-cpu=2000 --time=1:00:00 --pty bash

1 hour

Reservation different every week

4 cores

2GB/core

- Reservation during class
- Without --reservation for assignments (may have to wait)

Github classroom

- Helps manage lab material, assignments
- Create a user on github.com
- https://classroom.github.com/a/Tm3Nlqsb

- Introduction to Github assignment, due march 10 – will not be graded

Exercise

- Take the vector add example, compile it with/without OpenMP
- Test at various sizes (1K-1G), plot results (excel/matlab/python)
  - Run multiple times at each size
- If you can, try on your own machine too

- https://classroom.github.com/a/vWJFtycl