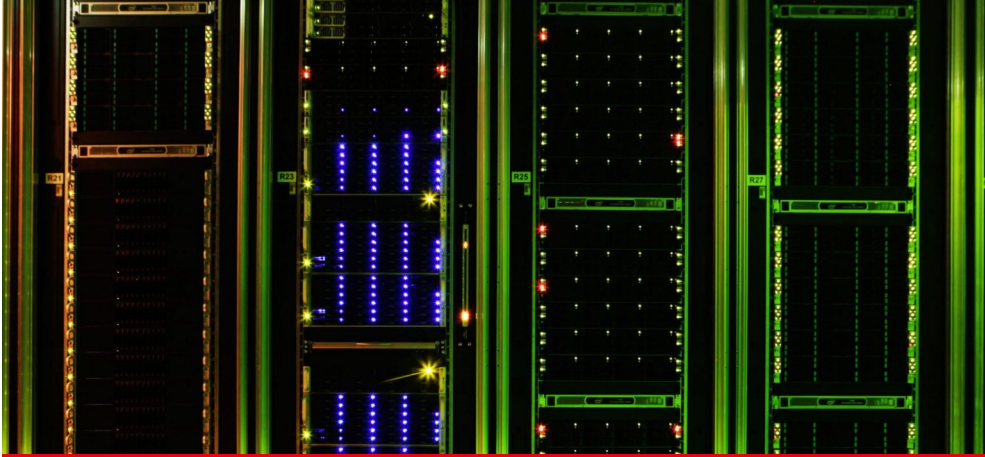


MATH-454 Parallel and High Performance Computing

Lecture 0: Course Introduction

Pablo Antolin

Slides of N. Richart, E. Lanti, V. Keller's lecture notes

February 20 2025

The team

Pablo Antolín

Philipp Weder

Today's program

Today

- Course organization and description
- Some definitions and basic concepts
- Parallelism
- The Top500

Moodle page

- https://moodle.epfl.ch/course/view.php?id=13817
- Slides, exercises, and solutions
- Calendar
- Please, join the ED Discussion Forum (link on Moodle)
- Instructions for accessing the computing facilities

GIT repositories

First, register in https://gitlab.epfl.ch with your epfl account!

- Example codes from the course
  - https://gitlab.epfl.ch/math454-phpc/course-examples-2025.git
- Course exercises
  - https://gitlab.epfl.ch/math454-phpc/exercises-2025.git

Course organization

- Theory on Thursdays, 13h15, in room DIA 003
- Exercises on Thursdays, 15h15 in room DIA 003
- Theory lectures will be recorded and posted on https://mediaspace.epfl.ch (for EUROfusion: Fusion Education and Learning Hub)
- Evaluation will be:
  - Two graded assignments: 25% + 25%
  - Course's project: 25%
  - Oral exam: 25%
- The oral exam will be a presentation of your project + questions about the project and the course (5+5 minutes)
- You will be given access to several computing facilities during this course. Follow the instructions on Moodle. Please respect the usage policies
- Visit to EPFL supercomputers (to be precised)

Calendar

- **Week 0**: today $\Rightarrow$ course introduction + Theory homework
- **Weeks 1–8**: (Feb. 27 – Apr. 17 2025) $\Rightarrow$ ex cathedra lectures + exercise sessions
- **Weeks 9–13**: (May. 1 – May. 22 2025) $\Rightarrow$ work on project
- **Final project proposal**: To be discussed on April 10 2025
- **Final project submission**: Deadline June 8 2025
- **Oral exam** $\Rightarrow$ June/July $\Rightarrow$ to be set by SAC

What this course is about

- To be able to:
  - sustain a conversation about general HPC topics
  - understand the concepts behind data and instruction parallelism
  - learn how to use shared memory, distributed memory, and accelerators paradigm efficiently
  - get a feeling of what algorithms are suited for parallelism
  - choose a target architecture based on the needs of a (real) application
  - learn to use common HPC machines
  - get your hands dirty with the most common programming techniques
- You will learn OpenMP, MPI, and CUDA using C++ (and some Python)

What this course is NOT about

- Bleeding-edge programming languages as Julia, Rust, Go, or alike
- PGAS languages as UPC, CAF, Fortress, HPF, or alike
- High-level parallel programming concepts or tools as SYCL or similar
- Auto parallelization tools or flags as yucca, par4all, intel `-parallel`, or alike

Calendar

0. Introduction
1. Hands-on on servers
2. Performance measurement
3. OpenMP
4. Intro to MPI
5. Advanced MPI (graded exercise)
6. Hybrid MPI/OpenMP and MPI with Python
7. Intro to CUDA
8. Advanced CUDA (graded exercise)
9. Work on project I (?)
10. Work on project II
11. Work on project III
12. Work on project IV

What we expect from you

- Follow the lectures. Do not hesitate to interrupt the lecturer if you may have a question that could interest your mates
- The exercises are key! Without getting your hands dirty you'll make no progress. TA is there to answer your questions
- Exercises are exercises. Project is project. You will have at least 3 times 4 hours to ask questions about the project plus the possibility of posting questions on Ed Forum

Prerequisites

- Moderate C++ (and Python) knowledge
- Basic git
- Basic bash and linux knowledge

For the practical lessons

For the practical lessons we will use EPFL's supercomputers.

- Please, check that you're registered in (link on Moodle) https://groups.epfl.ch/#/home/S30353

For accessing them you have to use your laptop.

- Linux or macOS: just use a terminal + `ssh`
- Windows:
  - Likely the easiest alternative is to use `Git Bash` from https://git-scm.com/
  - Windows Subsystem for Linux (WSL2)
- From any OS: connect to the VDI and select virtual machine `SB-MATH-LINUX-2023`

Get to know each other

1. Name
2. Background: bachelor, EPFL?
3. Master? PhD? Which program?
4. Programming experience
   - C/C++?
   - Parallel programming?
5. Is this course directly related to your work?

Parallel computing

- form of computation in which calculations are carried out simultaneously

Distributed (or throughput) computing

- Distributed computing is the method of making multiple computers work together to solve a common problem. It makes a computer network appear as a powerful single computer that provides large-scale resources to deal with complex challenges
- Grid computing (e.g., SETI@home)

Embarassingly parallel workload

- Problem where little or no effort is needed to split the problem into a number of parallel tasks. This is due to minimal or no dependency upon communication between the parallel tasks, or for results between them.
- Example: parameter sweeps. Opposite: inherently sequential

Basic computing Thesaurus

Supercomputing

- Use of the fastest biggest machines to solve large problems
- Solving problems on the Top500 machines

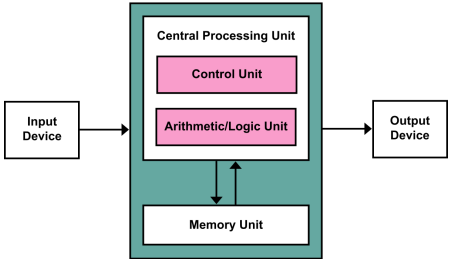
High Performance Computing

- Mostly a buzzword
- Supercomputing + large data + post-processing

Other cool topics around HPC

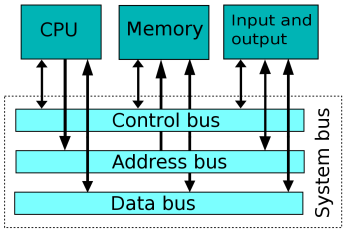
- Cloud Computing (e.g., AWS, Microsoft Azure, Google Cloud)

Von Neumann architecture (stored program computer)



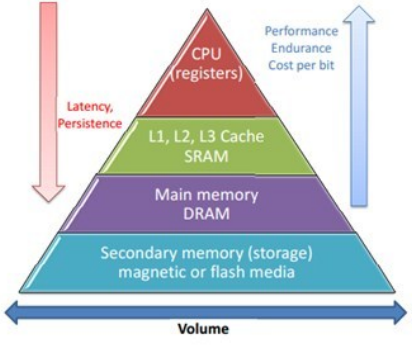
1. **ALU**: Arithmetic and Logical Unit
2. **Control Unit**
3. **Memory**
4. **Input/Output**

System bus

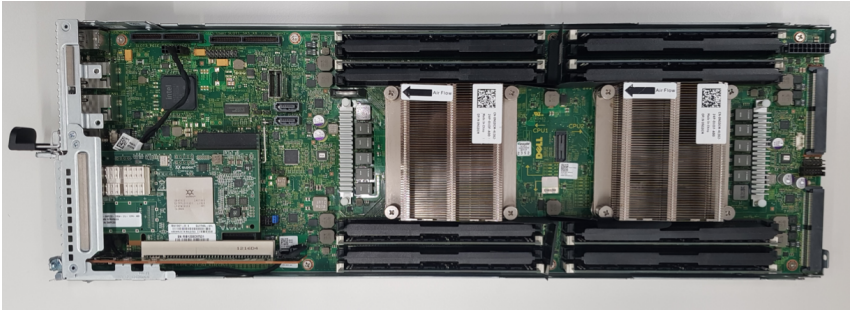


- **Address bus**: instructions on *where to find* the data
- **Data bus**: instructions on *what to find*
- **Control bus**: instructions on *what to do with* (e.g., `read`, `write`, `fetch`, etc. . .)

Memory pyramid



How does it look like in reality?



Some important values

- **CPU performance** measured in FLOPS (Floating Point Operations Per Second)
- **Memory bandwidth** in Bytes per second
- **Memory latency** in seconds

It is usual to give the **peak** values (never reachable) defined as:

- **Peak CPU performance** CPU Frequency $\times$ Number of operations per clock cycle $\times$ size of the largest vector $\times$ number of cores
- **Memory bandwidth** RAM Frequency $\times$ Number of transfers per clock cycle $\times$ Bus width $\times$ number of interfaces
- **Memory latency** depends on the size of the data. Usually given by the constructor

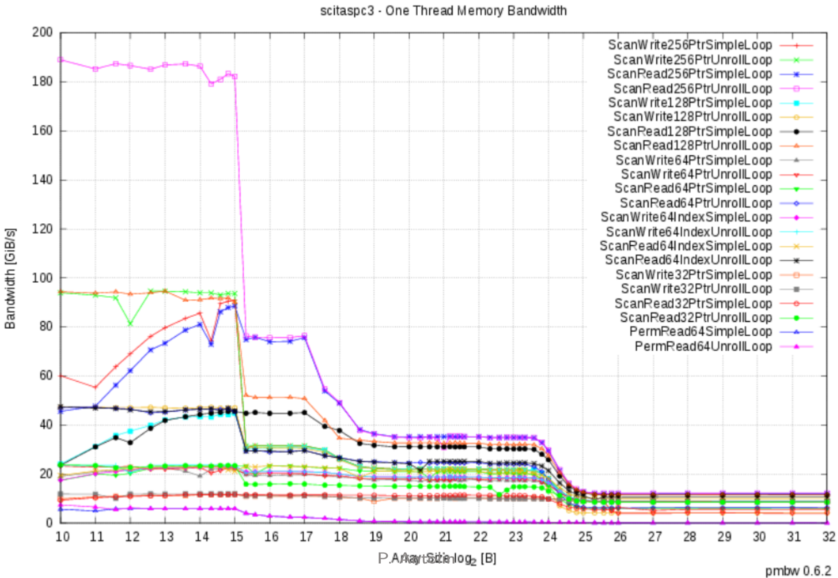
Benchmarks to measure the values

- **CPU performance**: HPL (LINPACK)
- **Memory bandwidth**: STREAM or PMBW
- **Memory latency** Intel Memory Latency Checker

HPL results on one Fidis node (f104)

```
================================================================================
T/V                N    NB     P     Q               Time                 Gflops
--------------------------------------------------------------------------------
WR11C2R4       81792   192     4     4             622.06              5.864e+02
HPL_pdgesv() start time Thu Oct 18 16:32:50 2018

HPL_pdgesv() end time   Thu Oct 18 16:43:12 2018

--VVV--VVV--VVV--VVV--VVV--VVV--VVV--VVV--VVV--VVV--VVV--VVV--VVV--VVV--VVV-
Max aggregated wall time rfact . . . :               2.51
+ Max aggregated wall time pfact . . :               0.64
+ Max aggregated wall time mxswp . . :               0.27
Max aggregated wall time update  . . :             619.19
+ Max aggregated wall time laswp . . :              27.47
Max aggregated wall time up tr sv  . :               0.24
--------------------------------------------------------------------------------
||Ax-b||_oo/(eps*(||A||_oo*||x||_oo+||b||_oo)*N)=        0.0017884 ..... PASSED
================================================================================
```

Memory bandwidth example



Intel Memory Latency Checker

```
Intel(R) Memory Latency Checker - v3.5
Measuring idle latencies (in ns)...
Numa node
Numa node       0       1
       0   76.2  122.9
       1  123.3   75.7

Measuring Peak Injection Memory Bandwidths for the system
Bandwidths are in MB/sec (1 MB/sec = 1,000,000 Bytes/sec)
Using all the threads from each core if Hyper-threading is enabled
Using traffic with the following read-write ratios
ALL Reads        : 101136.4
3:1 Reads-Writes : 93459.2
2:1 Reads-Writes : 93216.4
1:1 Reads-Writes : 90454.1
Stream-triad like: 81659.8

Measuring Memory Bandwidths between nodes within system
Bandwidths are in MB/sec (1 MB/sec = 1,000,000 Bytes/sec)
Using all the threads from each core if Hyper-threading is enabled
Using Read-only traffic type
```

Intel Memory Latency Checker

```
Measuring Loaded Latencies for the system
Using all the threads from each core if Hyper-threading is enabled
Using Read-only traffic type
Inject Latency Bandwidth
Delay (ns) MB/sec
==========================
 00000 170.22  102119.1
 00002 172.92  102090.6
 00008 176.54  101964.0
 00015 199.00  100029.4
 00050 174.13   94163.3
 00100 135.42   60156.9
 00200 119.19   37658.6
 00300 132.01   27635.2
 00400 122.57   21903.4
 00500 100.41   18037.9
 00700  95.57   13667.4
 01000  93.79    9701.4
 01300  90.08    7806.2
 01700  94.09    6234.9
 02500  90.04    4575.8
```

Intel Memory Latency Checker

```
Measuring cache-to-cache transfer latency (in ns)...
Local Socket L2->L2 HIT  latency 30.8
Local Socket L2->L2 HITM latency 35.5
Remote Socket L2->L2 HITM latency (data address homed in writer socket)
Reader Numa Node
Writer Numa Node     0      1
            0      -  86.4
            1   86.3      -
Remote Socket L2->L2 HITM latency (data address homed in reader socket)
Reader Numa Node
Writer Numa Node     0      1
            0      -  86.1
            1   86.4      -
```

Next Friday afternoon it's Charlotte's 10^{th} birthday party! She decides to invite 20 friends.

. . . Mrs. Smith – her mother – will not cook anything special, just some sandwiches for Charlotte's friends.

Daddy Smith and Charlotte's brother – Mike – will help too. . .

Introductory tale

. . . Mrs. Smith knows well how to make a sandwich. Four steps of approximately the same duration (1 unit of time) are required:

1. **cut the bread in two halves**
2. **spread the butter on the bread**
3. **add a piece of ham/cheese, a few pickles, and some mustard**
4. **close the sandwich carefully and put it on a tray**

Introductory tale

. . . how the family could proceed to produce 20 sandwiches in the more efficient manner. . .

old-school fasion? Mrs. Smith does it all alone while Mr. Smith and the kids watch TV?

- **Pros**:
  - Mrs. Smith does it well without having to explain to others
- **Cons**:
  - We are in the 21st century ...
  - What happens if Charlotte decides to invite the whole school to her birthday?

Time to make 20 sandwiches = 80 units of time

. . . 1923 Ford T mode? Mrs. Smith cuts the bread, then Mr. Smith spread the butter while Mrs. Smith cuts a new bread, Charlotte adds ham and pickles while Mrs. Smiths cut bread and finally Mike closes the sandwiwch and put it in a tray

- **Pros**:
  - $4\times$ more sandwiches produced when the pipeline is full
  - specialized tasks, very few communications
- **Cons**:
  - each step must have the same duration
  - No place for Shirley (Charlotte's best friend). Only 4 persons are required to produce $8\times$ more sandwiches

Time to make 20 sandwiches = 23 units of time

. . . totally distributed? Mrs. & Mr. Smith, Charlotte, and Mike produce 5 sandwiches each. . .

- **Pros**:
  - Most efficient solution: number of sandwiches to produce / number of people
- **Cons**:
  - resources problem: need for 4 knives, 4 packets of butter, 4 portions of ham, etc. . .
  - deadlocks: if only 1 knife is available, one person works when other waits

Time to make 20 sandwiches = 20 units of time

Performance analysis

Before starting parallelization: insight of the application

- Is the algorithm/application a good candidate for parallelization?
- Which parallelization strategy should I choose?
- How should I partition the problem?

Helps understanding and improving existing implementations

- Do the benchmark match the performance predictions?
- Which factors have most influence on the performance?
- Which hardware fits most to the application needs?

Speedup and efficiency

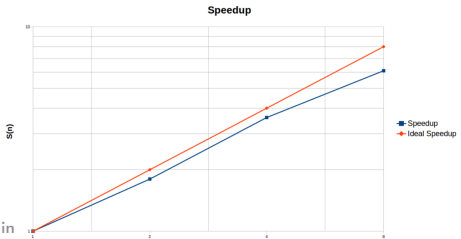
$$S(p) = \frac{T_1}{T_p} \quad E(p) = \frac{S(p)}{p}$$

where

- T_1: Execution time using 1 process
- T_p: Execution time using p processes
- $S(p)$: Speedup of p processes
- $E(p)$: Parallel Efficiency

Example:

p	Walltime [s]	S(p)	E(p)
1	24.5	1.0	1.0
2	13.4	1.8	0.9
4	6.8	3.6	0.9
8	4.0	6.1	0.75



Complexity analysis

Observation:
The speedup $S(p)$ is determined by the **computation time** and the **communication time**, function of the problem size N

Communication time:
It is an overhead to the parallel execution time. It can be partially hidden by computation.

Complexity:
Computing the **complexity** $\mathcal{O}(N, p)$ provides and insight into the parallelization potential.

Complexity analysis: example $c = A\,b$

Full $N \times N$ matrix vector multiplication with the naïve method $\mathcal{O}(N^2)$

```
struct timeval t;
const double t1 = gettimeoftheday(&t,0);
for (i=0;i<N;i++){
   for (j=0;j<N;j++){
      c[i] = c[i] + A[i][j]*b[j];
   }
}
const double t2 = gettimeoftheday(&t,0);
const double mflops = 2.*(double)N*(double)N / (t2-t1) / 1.0E6;
```

Complexity analysis: example $c = A\,b$

Sequential algorithm:

- **Time complexity**: $\mathcal{O}(N^2)$
- **Size complexity**: $\mathcal{O}(N^2)$

Complexity analysis: example $c = A\,b$

Parallel version

```
t1 = MPI_Wtime();
MPI_Bcast(b, ndiag, MPI_FLOAT, 0, MPI_COMM_WORLD);
MPI_Scatter(A,(nn_loc*ndiag),MPI_FLOAT,A_loc,(nn_loc*ndiag),MPI_FLOAT,0,
            MPI_COMM_WORLD);
t2 = MPI_Wtime();
for (j=0;j<ndiag;j++){
   for (i=0;i<nn_loc;i++){
      c[i] = c[i] + A_loc[i+j*nn_loc]*b[i];
   }
}
t3 = MPI_Wtime();
MPI_Gather(c,nn_loc,MPI_FLOAT,b,nn_loc,MPI_FLOAT,0,MPI_COMM_WORLD);
t4 = MPI_Wtime();
```

Complexity analysis: example $c = A\,b$

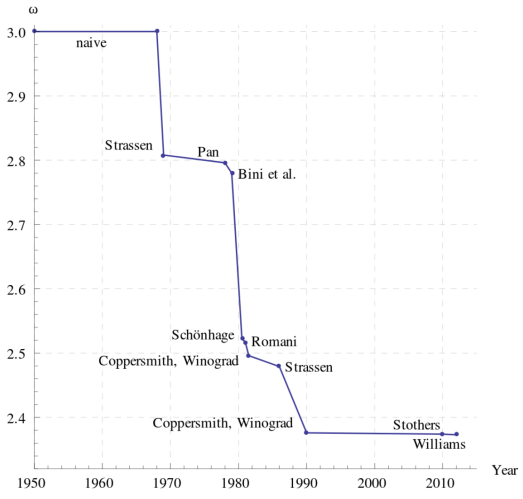
Parallel algorithm over p processes:

- **Computational complexity**: $\mathcal{O}(\frac{N^2}{p})$
- **Communication complexity**: $\mathcal{O}(\log p + N)$
- **Overall complexity**: $\mathcal{O}(\frac{N^2}{p} + \log p + N)$
- for *reasonably* large N, latency is negligible compared to bandwidth
- The algorithm is **not scalable** (as the overall complexity scales with the problem size N independently of the number of processes p)

Full $N \times N$ matrix matrix multiplication with the naïve method $\mathcal{O}(N^3)$

```
struct timeval t;
const double t1 = gettimeoftheday(&t,0);
for (i=0;i<N;i++){
   for (j=0;j<N;j++){
      for (k=0;k<N;k++){
         C[i][j]=C[i][j] + A[i][k]*B[k][j];
      }
   }
}
const double t2 = gettimeoftheday(&t,0);
const double mflops = 2.0*pow(N,3) / (t2-t1) / 1.0E6 ;
```

On the importance of algorithmic analysis



Flynn's taxonomy

Classic model, simple, gives a good first approximation

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

most modern machines are hybrids of these categories

Flynn's taxonomy

Classic model, simple, gives a good first approximation

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

most modern machines are hybrids of these categories

- SISD: entirely sequential program
- SIMD: same operation is done over a large data set
- MISD: rare. Only used for fault tolerance purpose
- MIMD: several processors that function asynchronously and independently

Flynn's taxonomy: notable examples

	Single Instruction	Multiple Instruction
Single Data		
Multiple Data		

Flynn's taxonomy: notable examples

	Single Instruction	Multiple Instruction
Single Data	■ Any sequential code	
Multiple Data		

Flynn's taxonomy: notable examples

	Single Instruction	Multiple Instruction
Single Data	■ Any sequential code	
Multiple Data	■ Intel SSE (Streaming SIMD extensions, 4 float operations per cycle) ■ GPUs	

Flynn's taxonomy: notable examples

	Single Instruction	Multiple Instruction
Single Data	■ Any sequential code	
Multiple Data	■ Intel SSE (Streaming SIMD extensions, 4 float operations per cycle) ■ GPUs	■ A cluster (distributed memory architecture) ■ A multicore processor

Types of parallelism

- Bit-level parallelism: SIMD in the register (like SSE)
- Task parallelism or pipelining (Instruction-level parallelism)
- Data parallelism (loop parallelism)

Bit-level parallelism

- Parallelism based on the size of the data the processor can process at once
- SIMD within a register (SWAR)

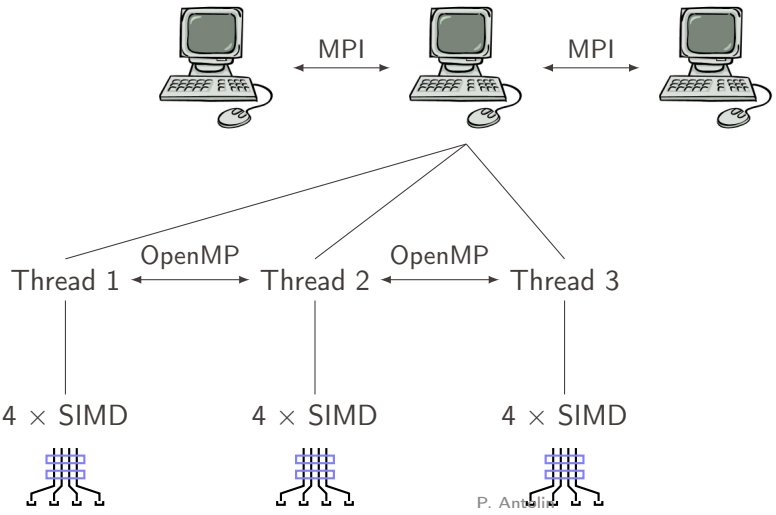
Task parallelism

- often called pipelining
- think of it as an assembly chain
  - a task is divided into a sequence of smaller tasks
  - each task is executed on a specialized piece of hardware
  - each piece of hardware operates concurrently with the other stages of the pipeline
  - each of the step is performed during a clock period of the machine
  - filling the pipeline takes several clock cycles (latency)
  - once the pipeline is filled, a result appears every clock cycle

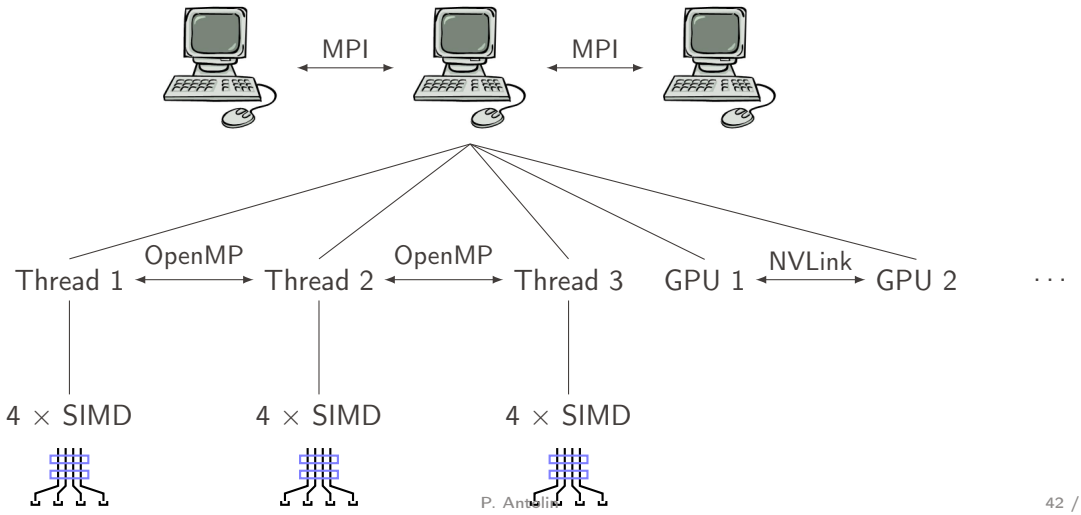
Data parallelism

- often called loop-level parallelism
- data is distributed across different computing units or
- resides on a global memory address space

... more on that on the next lectures


MPI
MPI
OpenMP
OpenMP
Thread 1
Thread 2
Thread 3
4 × SIMD
4 × SIMD
4 × SIMD

Levels of parallelism



Bit-level parallelism

- SIMD parallel processing is applied across sections of a CPU register
- "vector" computing
- on x86 processors, this is achieved with the following instructions sets:
  - MMX, 3DNow!
  - SSE, SSE2, SSSE3, SSE4
  - AVX, AVX2, AVX512
- SSE up to the currently latest version 4.2 can process four single precision (32-bit) floating point numbers or two double precision (64-bit) floating point numbers in vectorized manner.
- AVX512 can process eight double precision floating point numbers or sixteen in single precision.

Processor instructions extensions

```
grep -m1 flags /proc/cpuinfo
flags : fpu vme de pse tsc msr
pae mce cx8 apic sep mtrr pge mca cmov pat pse36
clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe
syscall nx pdpe1gb rdtscp lm constant_tsc art
arch_perfmon pebs bts rep_good nopl xtopology
nonstop_tsc cpuid aperfmperf pni pclmulqdq dtes64
monitor ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16
xtpr pdcm pcid dca sse4_1 sse4_2 x2apic movbe popcnt
tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm
abm 3dnowprefetch cpuid_fault epb cat_l3 cdp_l3
invpcid_single pti intel_ppin ssbd mba ibrs ibpb
stibp tpr_shadow vnmi flexpriority ept vpid fsgsbase
tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm
cqm mpx rdt_a avx512f avx512dq rdseed adx smap clflushopt
clwb intel_pt avx512cd avx512bw avx512vl xsaveopt xsavec
xgetbv1 xsaves cqm_llc cqm_occup_llc cqm_mbm_total
cqm_mbm_local dtherm ida arat pln pts pku ospke md_clear flush_l1d
```

Taking advantage of vectorization

- hope that Matlab/Python/Octave is smart and does it by itself (??)

Taking advantage of vectorization

- hope that Matlab/Python/Octave is smart and does it by itself (??)
- count on the compiler (what most people do)

Taking advantage of vectorization

- hope that Matlab/Python/Octave is smart and does it by itself (??)
- count on the compiler (what most people do)
- help the compiler by manually unrolling loops and adding vectors (that's what most people do when the previous fails)

Taking advantage of vectorization

- hope that Matlab/Python/Octave is smart and does it by itself (??)
- count on the compiler (what most people do)
- help the compiler by manually unrolling loops and adding vectors (that's what most people do when the previous fails)
- use *intrinsics* within your code: manual vectorization with processor instructions (painful but works)

Taking advantage of vectorization

- hope that Matlab/Python/Octave is smart and does it by itself (??)
- count on the compiler (what most people do)
- help the compiler by manually unrolling loops and adding vectors (that's what most people do when the previous fails)
- use *intrinsics* within your code: manual vectorization with processor instructions (painful but works)
- write assembly code and link it with `nasm` (you don't want to do this)

Taking advantage of vectorization

- hope that Matlab/Python/Octave is smart and does it by itself (??)
- count on the compiler (what most people do)
- help the compiler by manually unrolling loops and adding vectors (that's what most people do when the previous fails)
- use *intrinsics* within your code: manual vectorization with processor instructions (painful but works)
- write assembly code and link it with `nasm` (you don't want to do this)
- use efficient vectorized implementations (MKL, OpenBLAS,...) of BLAS, LAPACK, etc.

Exploiting bit parallelism

Bit-level parallelism can be automatically exploited by the compiler

ex1.c

```
7         for (i=0; i<256; ++i){
8             a[i]=i; b[i]=i;
9         }
10        for (i=0; i<256; ++i)
11            c[i] = a[i] + b[i];
```

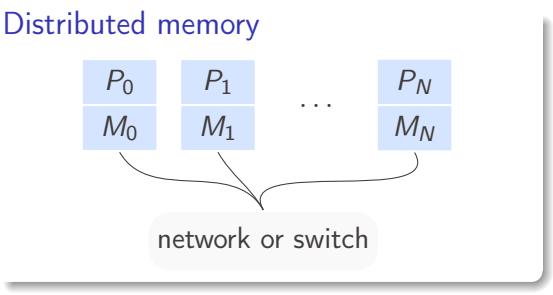
gcc

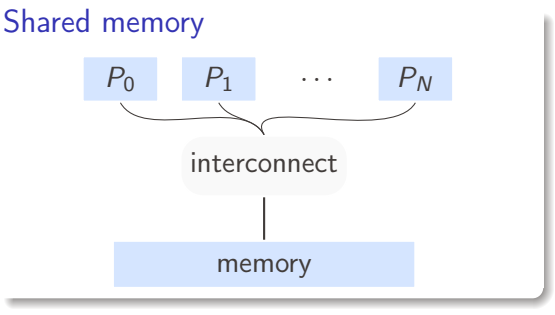
```
gcc -O3 -msse2 -ftree-vectorizer-verbose=2 ex1.c

ex1.c:10: note: LOOP VECTORIZED.
ex1.c:7: note: LOOP VECTORIZED.
ex1.c:6: note: vectorized 2 loops in function.
```

icc

```
icc -O3 -vec-report ex1.c
ex1.c(7): (col. 2) remark: LOOP WAS VECTORIZED.
```

Distributed versus shared memory machines





The Top500

Measuring computers speed

Kilo	k	thousand	10^3
Mega	M	million	10^6
Giga	G	billion	10^9
Tera	T	trillion	10^{12}
Peta	P	quadrillion	10^{15}
Exa	E	quintillion	10^{18}
Zetta	Z	sextillion	10^{21}
Yotta	Y	septillion	10^{24}

Today's units:

Processor speed	GHz
RAM	GB
Hard disk	TB

$\text{Hz} = 1/\text{s} \rightarrow 1\ \text{GHz} = 10^9$ machine cycles per second

What is the Top500

- List of the 500 most powerful machines in the world
- Recognized by:
  - industry
  - academia
  - vendors
  - media
- Idea by Hans Meuer, Jack Dongarra, Erich Strohmaier, Horst Simon
- The list is published twice per year (June and November)
- Computers are ranked by their performance on the LINPACK Benchmark (Rmax, FLOPS)

A great way to keep track of computer evolution!

FLOPS: Floating Point Operations Per Second

R_{Peak}: theoretical peak performance provided by the constructor

A consumer processor has peak performance of the order of hundreds of GigaFLOPS, while top supercomputers are in the order of ExaFLOPS.

FLOPS: Floating Point Operations Per Second

R_{Max}: performance measured with the LINPACK benchmark

- Solve Ax=b with LU decomposition
- Data is partitionned onto a PxQ grid of processes
- In practice:
  - download HPL
  - compile
  - modify `HPL.dat`
  - run!

In practice, we solve a dense linear system

$$Ax = b$$

the following way:

- let P be a permutation matrix, then $\exists P | PA = LU$. Compute P, L, U.
- solve $Ly = Pb$ and $Ux = y$

Memory requirements

- need to store A and b
  $\Rightarrow N \times N + N$ real numbers
- double precision numbers use 8 bytes
  $\Rightarrow 8(N^2 + N)$ bytes of RAM needed
- given a fixed amount of RAM, it is easy to compute the N parameter in `HPL.dat`

Number of operations

- an LU decompostion requires $\mathcal{O}(N^3)$ operations
- solving triangular systems requires $\mathcal{O}(N^2)$ operations

Precisely, the benchmark algorithm requires

$$\frac{2}{3}N^3 + N^2 + \mathcal{O}(N)$$

operations for a matrix of order N.

Orders of magnitude: Watts

- 1 kW: a hairdryer
- EPFL average consumption (Ecublens campus): $\sim$5 MW
- 100 kW: 3 days steady consumption of 100 kW = 1 year power consumption of a typical family
- 1 MWy cost: 2.5 MCHF (average domestic energy price in Switzerland)
- Top supercomputers' consumption is in the order of 30 MW: $\sim$75 MCHF/year (considering domestic energy price)

Highlights of the Top500

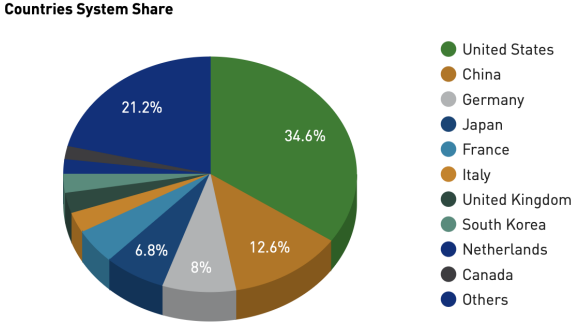
- The HPL benchmark has been criticized by many, for the following reasons:
  - it is not representative of the real applications running on the machines
  - it is suspected that the chip manufacturers optimize their instruction set only to have a higher HPL score
- We have a new candidate: the High Performance Conjugate Gradient (HPCG)
- There is a big turnover in the machines in the list.

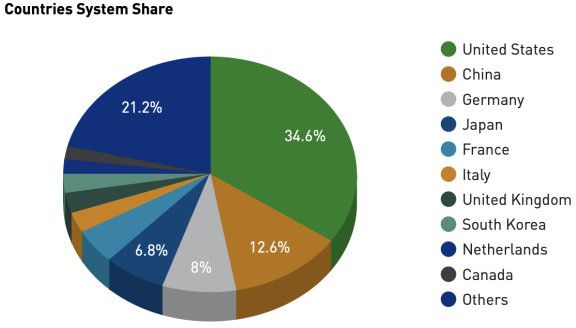
Top500 – November 2024

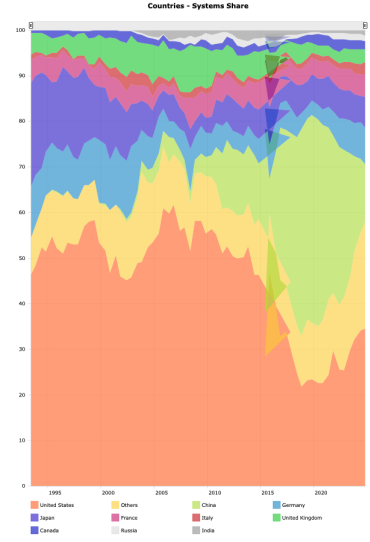
	Machine	# cores	R_{max} [PFlops]	R_{peak} [PFlops]	Power [MW]	Country
1	El Capitan	11,039,616	1,742	2,746	29,6	USA
2	Frontier	9,066,176	1,353	2,056	24,6	USA
3	Aurora	9,264,128	1,012	1,980	38,7	USA
4	Eagle	2,073,600	561	847		USA
5	HPC6	3,143,520	478	607	8,5	Italy
6	Fugaku	7,630,848	442	537	29,9	Japan
7	Alps	2,121,600	435	575	7,1	Switzerland
8	LUMI	2,752,704	380	532	7,1	Finland
9	Leonardo	1,824,768	241	306	7,5	Italy
10	Tuolumne	1,161,216	208	289	3,4	USA
...	...	...	...	...	...	...
102	Kuma	26,240	12	22	0,3	Switzerland (EPFL)

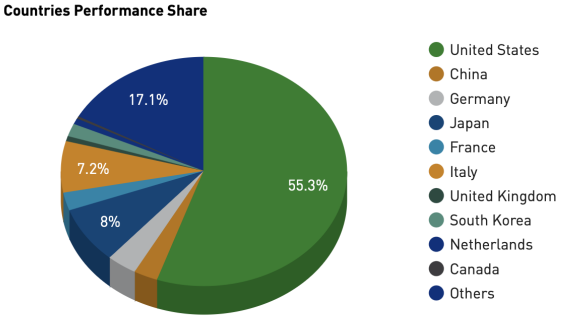
Top500 – November 2024

A few interesting facts



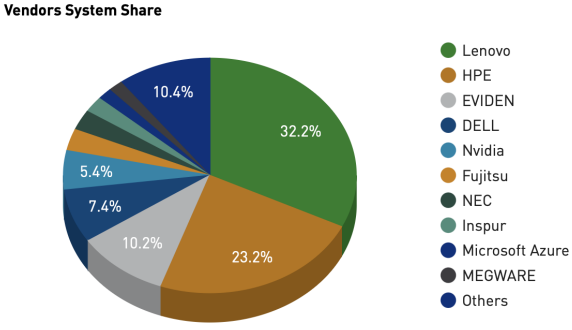

Countries System Share
21.2%
34.6%
12.6%
8%
6.8%
United States
China
Germany
Japan
France
Italy
United Kingdom
South Korea
Netherlands
Canada
Others

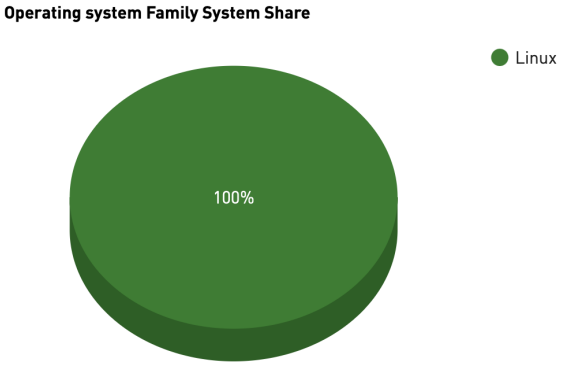

Countries - Systems Share
100
90
80
70
60
50
40
30
20
10
0
1995
2000
2005
2010
2015
2020
United States
Others
China
Germany
Japan
France
Italy
United Kingdom
Canada
Russia
India


Countries Performance Share
17.1%
7.2%
8%
55.3%
United States
China
Germany
Japan
France
Italy
United Kingdom
South Korea
Netherlands
Canada
Others

Top500 – November 2024

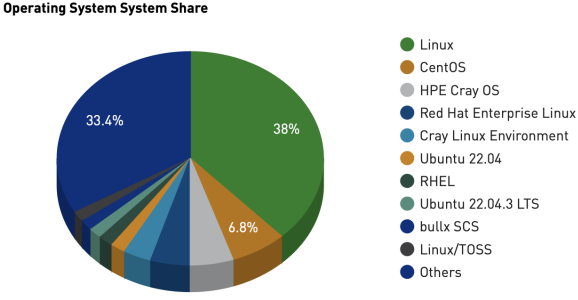
A few interesting facts




Operating system Family System Share
Linux
100%

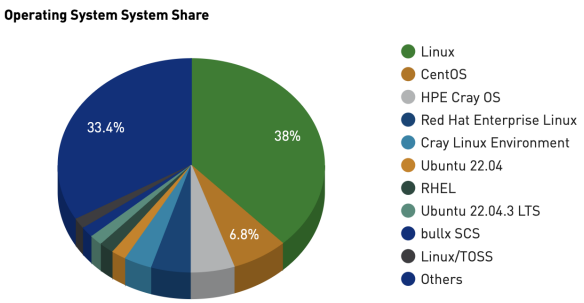
Top500 – November 2024

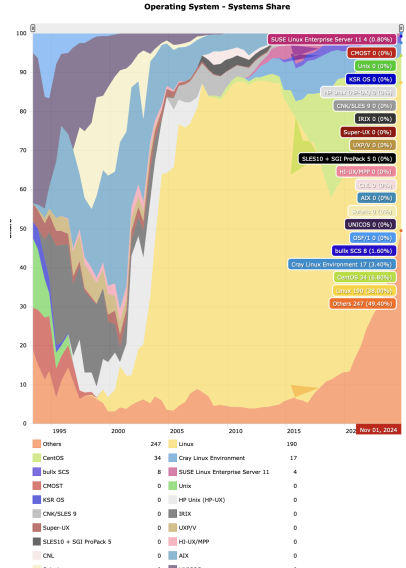
A few interesting facts

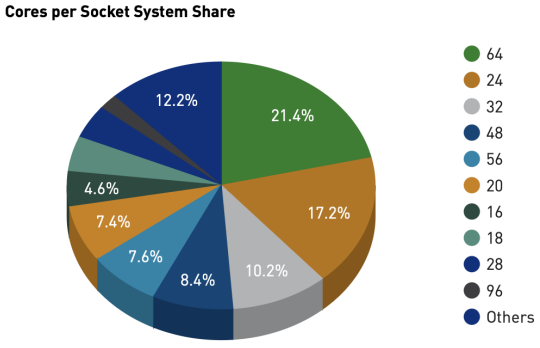


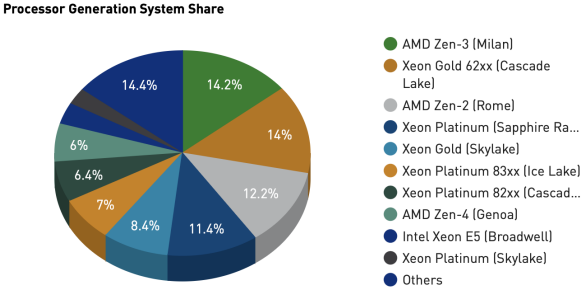
Top500 – November 2024

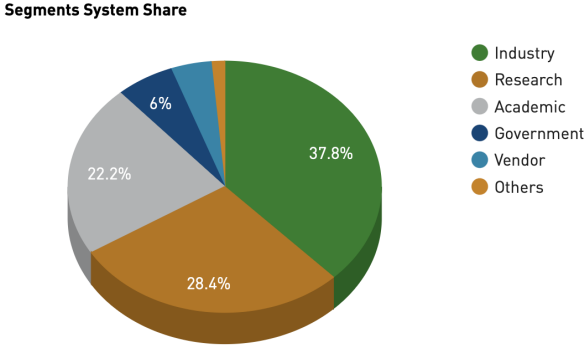
A few interesting facts

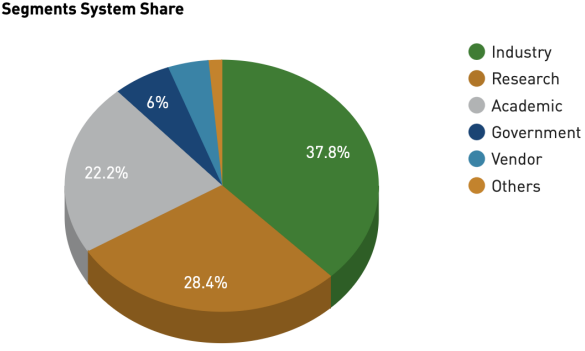


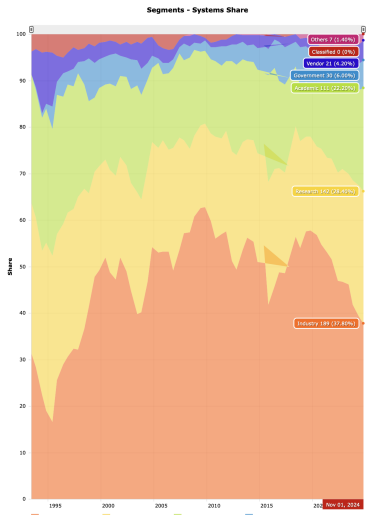


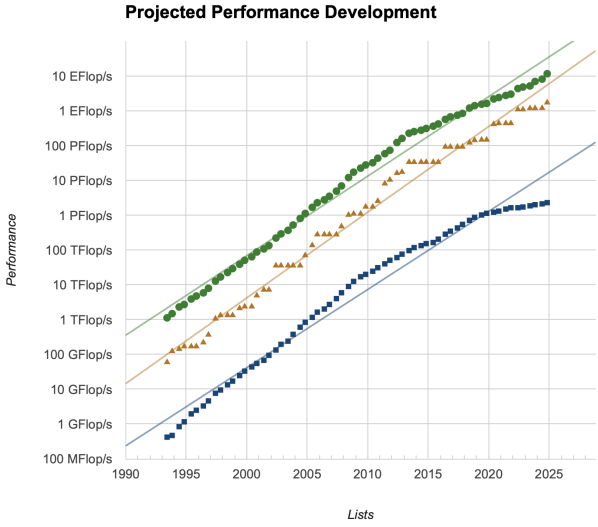

Cores per Socket System Share
21.4%
17.2%
10.2%
8.4%
7.6%
7.4%
4.6%
12.2%
64
24
32
48
56
20
16
18
28
96
Others


Processor Generation System Share
14.2%
14%
12.2%
11.4%
8.4%
7%
6.4%
6%
14.4%
AMD Zen-3 (Milan)
Xeon Gold 62xx (Cascade Lake)
AMD Zen-2 (Rome)
Xeon Platinum (Sapphire Ra...
Xeon Gold (Skylake)
Xeon Platinum 83xx (Ice Lake)
Xeon Platinum 82xx (Cascad...
AMD Zen-4 (Genoa)
Intel Xeon E5 (Broadwell)
Xeon Platinum (Skylake)
Others


Segments System Share
37.8%
28.4%
22.2%
6%
Industry
Research
Academic
Government
Vendor
Others


Segments System Share
37.8%
28.4%
22.2%
6%
Industry
Research
Academic
Government
Vendor
Others


Segments - Systems Share
Others 7 (1.40%)
Classified 0 (0%)
Vendor 21 (4.20%)
Government 30 (6.00%)
Academic 111 (22.20%)
Research 142 (28.40%)
Industry 189 (37.80%)
Share
Nov 01, 2024


Projected Performance Development
Performance
10 EFlop/s
1 EFlop/s
100 PFlop/s
10 PFlop/s
1 PFlop/s
100 TFlop/s
10 TFlop/s
1 TFlop/s
100 GFlop/s
10 GFlop/s
1 GFlop/s
100 MFlop/s
1990
1995
2000
2005
2010
2015
2020
2025
Lists
Sum
#1
#500

The Green 500

http://www.green500.org

- provides a ranking of the most energy-efficient supercomputers in the world
- MFLOPs/Watts

EPFL NEWS EPFL ENAC SB STI IC SV CDM CDH All EN

/ News

New supercomputer enables cutting-edge and sustainable research

EPFL's new Kuma supercomputer, which ranks 23rd in the Green500 ranking, illustrates EPFL's efforts to support cutting-edge research with a low environmental impact. With Kuma, EPFL is helping anchor Switzerland's position at the forefront of sustainable computing.

20.11.24

IMAGES TO DOWNLOAD

The SCITAS team - 2024 - CC-BY-SA 4.0

Welcome to the course!