High-Fidelity Simulations for Turbulent Flows

Luca Sciacovelli

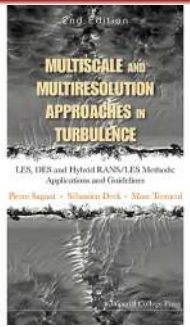
DynFluid Laboratory
Arts et Métiers Institute of Technology
http://savoir.ensam.eu/moodle

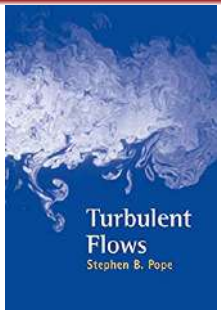
Master Recherche "Aérodynamique et Aéroacoustique"
2021 – 2022

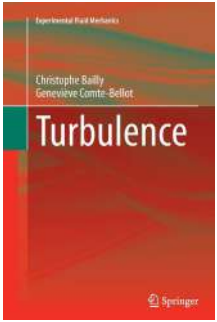
Contact: Luca SCIACOVELLI (`luca.sciacovelli@ensam.eu`)
Simon MARIE (`simon.marie@lecnam.net`)

Course	Topic
1	Introduction - High Performance Computing
2	Hierarchy of Turbulence modeling
3	Large-Eddy Simulations 1
4	Large-Eddy Simulations 2
5	Hybrid RANS/LES Methods
6	TD1: Turbulence Modeling
7	Discretization of the Navier–Stokes equations
8	Time integration schemes for unsteady problems
9	High-order spatial schemes 1
10	High-order spatial schemes 2
11	Lattice Boltzmann Methods 1
12	Lattice Boltzmann Methods 2
13	TP1: Application of LBM Methods
14	TP2: Simulation of unsteady viscous flows
15	TD2: Numerical schemes
16	Final exam

Final mark: $0.5 \times \dfrac{\text{TP1} + \text{TP2}}{2} + 0.5 \times \text{Final Exam}$

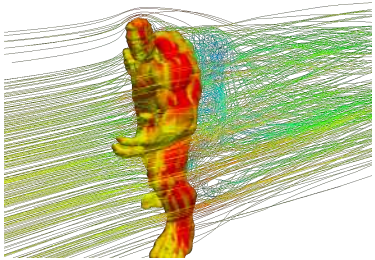


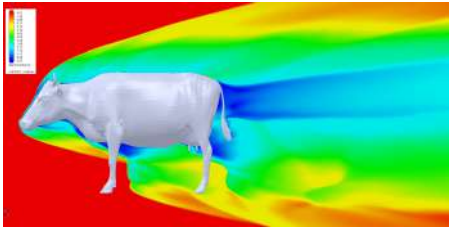


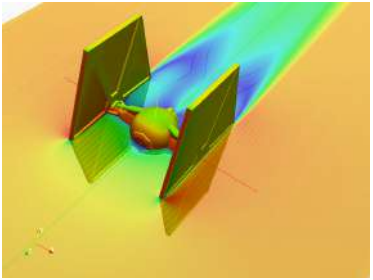


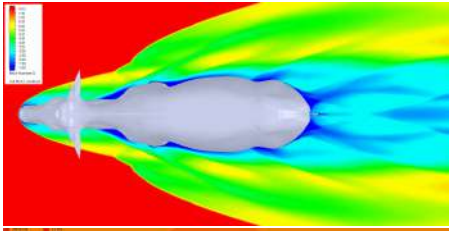

Turbulence
RANS
LES
DNS
DES
0 equation
1 equation
2 equation
Hybrids
Mesh
Mesh generation
Refinement
Dynamic refinement
Mesh mapping
Finite element/difference
Structured/unstructured
Experiments
Uncertainty
Cost
Instrumentation/equipment behavior
Repeatability
Flow Behavior
Laminar
Transitional
Turbulent
Incompressible
Compressible
(sonic, supersonic, hypersonic)
Newtonian/nonnewtonian
Stokesian
Boundary layer
Potential
Flow instabilities
Mathematics
Numerics
Stability
Convergence
PDE behavior
BCs/ICs
Linear algebra
Optimization
Chaos
The Many Faces of CFD
Multiple Physics
Single phase
Multi-phase
Boiling/condensation
Droplet/bubble
Solidification
Candling
Chemistry
Ionization
Shock
Cavitation
Combustion
Aerodynamic heating
V&V
Sensitivity studies
Validation
Verification
Quantification
Materials
EOS
Constitutive
Phase
Composites
Computer science
Computer hardware
Programming
Parallelization
Optimization
Numerical error
Visualization
Parallelization
Contours, arrows, tracers, volume rendering

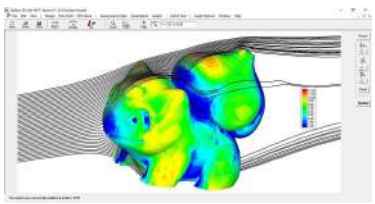
What CFD is not...

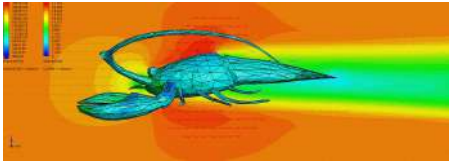


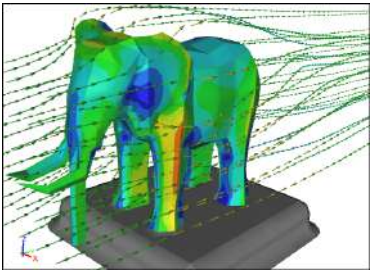












even if.. https://assets.gfm.aps.org/6142a9aa199e4c7029f44dfc/poster/fullsize.png

Part I

Simulation and High-Performance Computing

1. Introduction and Examples of HPC applications
2. HPC in France
3. Architectures
4. Limits and costs
5. Parallel programming models
6. Designing parallel programs
7. Conclusions

Parallel computing: why and how?

Why Parallel computing?

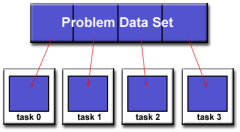
If one CPU executes a program in t hours, then N CPUs can execute the same program in t/N hours!

1. Have access to more memory and more power by combining hardware resources and therefore:
   - Save wall-clock time and/or money
   - Solve bigger/more complex problems
2. Execute several simulations at the same time
3. Overcome limits of sequential computing:
   - Memory size
   - Miniaturization of transistors
   - Cheaper to increase cores number than speed

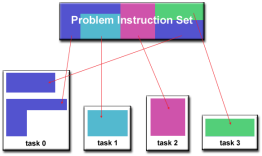
How Parallel computing?

Idea: Break the problem into discrete "chunks" of work that can be distributed to multiple tasks

Domain Decomposition



Functional Decomposition

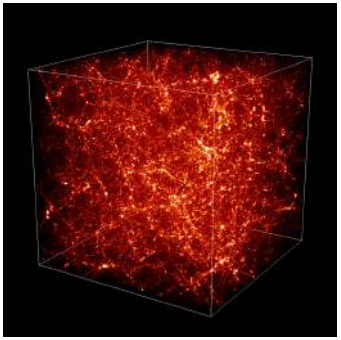


"Parallel computing" may refer to:

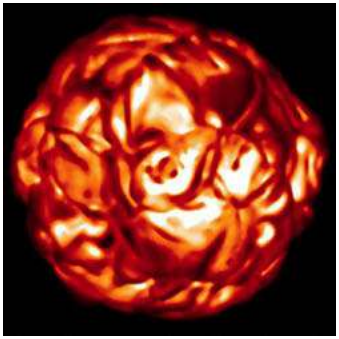
- Supercomputing: multiple machines with one or more processors interconnected by a rapid network
- Cloud computing: delivery of compute resources over the Internet via multiple data centers (e.g., AWS)
- Grid computing: loosely coupled network of compute resources (e.g., Folding@home)

The biggest scales

Simulating the Universe to understand the nature of dark energy, dark matter and black holes



Numerical simulation of structures formation in the Universe

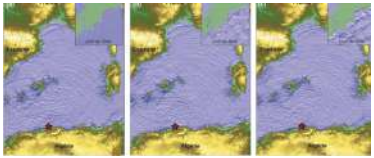


Numerical simulation of a "supergiant" star

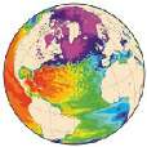
- Distribution of the matter as predicted by the first simulation ever run covering the entire volume of the observable universe. The observer is in the center (here, now) while the first light beams are on the edges, 14 billion light-years away

The scale of the planet

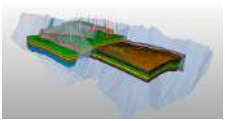
Atmosphere, environment, weather: weather forecast, assessment of natural hazards such as cyclones or tsunamis, climate change studies, geology, seismology, Oil & Gas



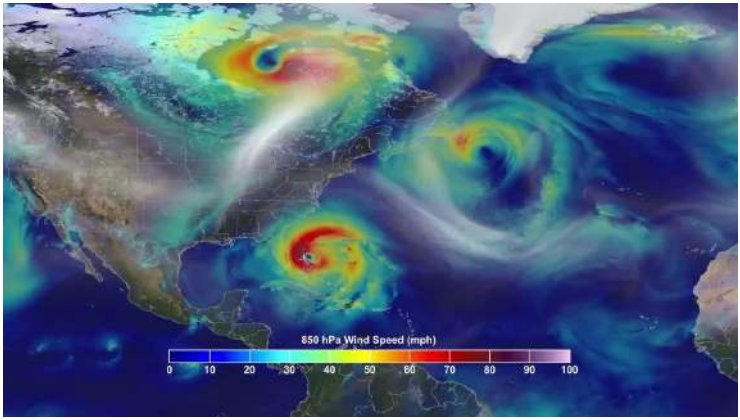
Tsunami arriving in Sète port (2003) after 120 min (a), 130 min (b) et 140 min (c) of propagation from epicenter (red star)



Ocean circulation



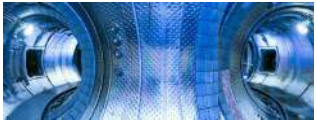
Oil & Gas reservoirs



Simulation of the hurricane Sandy (2012)

Energy

Energy: renewable energy research, optimize hydrocarbon exploration, design next-generation plants, control nuclear technologies (ITER program)



IRFM's tokamak Tore Supra

Turbulence in plasma flows: vortices stretch along the magnetic field lines, visualized through the electric potential. Turbulence experiences sudden bursts of activity for extremely short times, modifying vortices's shapes and locations.

Positioning of wind turbines

Understanding the effect of the tip and root helical vortices of an upstream turbine wake impacting a downstream rotor, plays a key role in understanding the fluctuating nature of the produced power by the downstream turbine.

To understand the turbine-wake interaction we employ an actuator line approach to represent the rotor blades and a large-eddy simulation solver to describe the fluid flow.

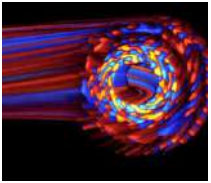
The simulations correspond to wind tunnel tests for which data is available for the time-averaged quantities of the wake field and the generated power.

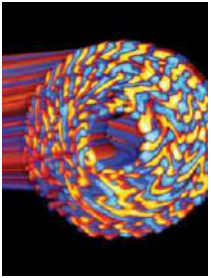
The two cases presented herein are:

I: Two turbines operating in-line
II: Two turbines operating in-line with a spanwise off-set

To resolve the high-Reynolds number fluid flow, we use sixth-order finite-difference compact schemes and a Spectral Vanishing Viscosity-like approach for the subgrid scales.

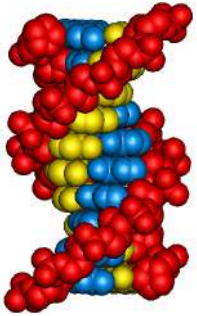
The wind tunnel domain of dimensions 2.0 m x 2.7 m x 11.76 m is discretised using 241 x 241 x 1281 nodes.



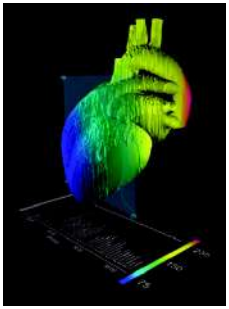


Health-related domains

Chemistry, Medicine, Biology: molecules design, drugs development, medical imaging and diagnosis.
COVID-19 Projects: `https://www.genci.fr/fr/content/projets-contre-le-covid-19`

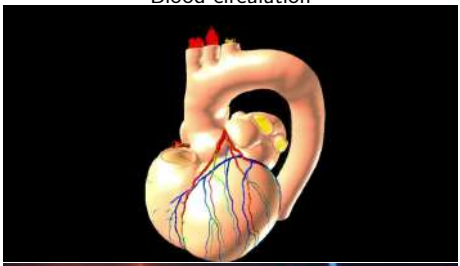


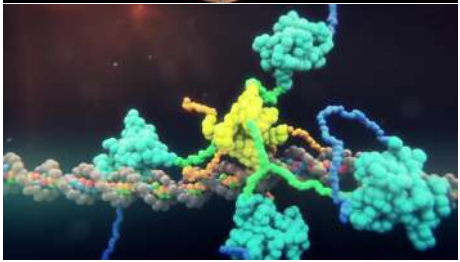
CAO of new molecules



Signal processing and pattern recognition

Blood circulation

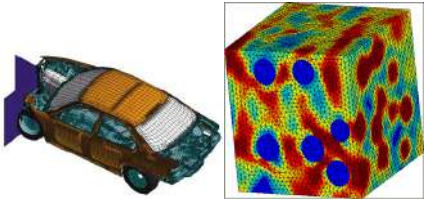




Study of Chromatin (RNA) displacement

The list goes on..

Material Physics: design new materials and measure their strength

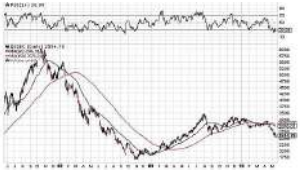


Left: simulation of vehicles crash tests
Right: conception of composite micro-structure with elastic inclusions in an elastoplastic metal matrix

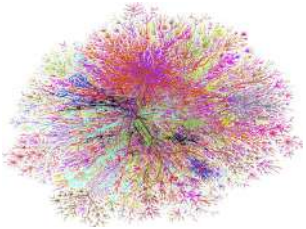
New domains of application:

- High Performance Data Analytics (HDPA, Big Data + HPC)
- Artificial Intelligence, Machine Learning
- Autonomous Driving
- Web search engines, Web-based services

Finance: risk assessment on complex products



Evolution of stock prices



Modeling of the Internet network

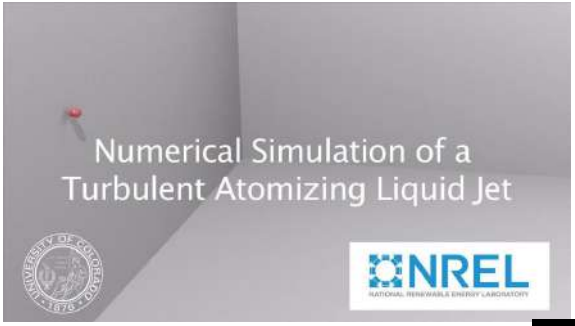
Physics - applied, nuclear, particle, condensed matter, high pressure, fusion, photonics ..

Robotics, Informatics, Virtual Reality..

"Horse" Robot

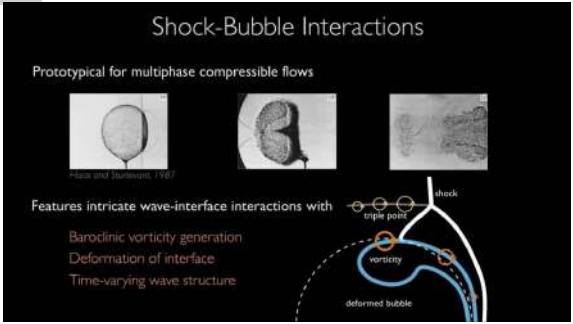
Digital animations

Dynamics of complex fluid flows



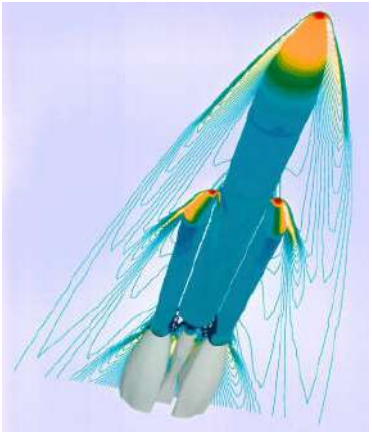
Numerical simulation of a turbulent atomizing liquid jet

Study of Shock-Bubble interactions



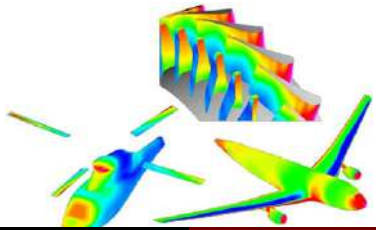
Transportation

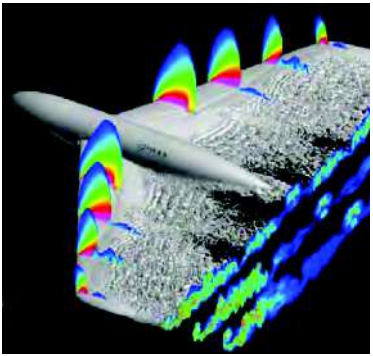
- **Automotive**: aerodynamic design of new vehicles, study of engine combustion to reduce consumption and pollution, modeling of crash test, ..
- **Aerospace**: speed up design and validation of certain components, studies of performance and acoustic noise, ..



Numerical simulation of the flow around Ariane 5

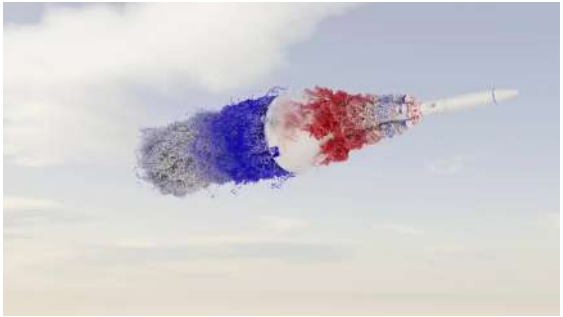
Numerical simulation of the flow around a F1 car



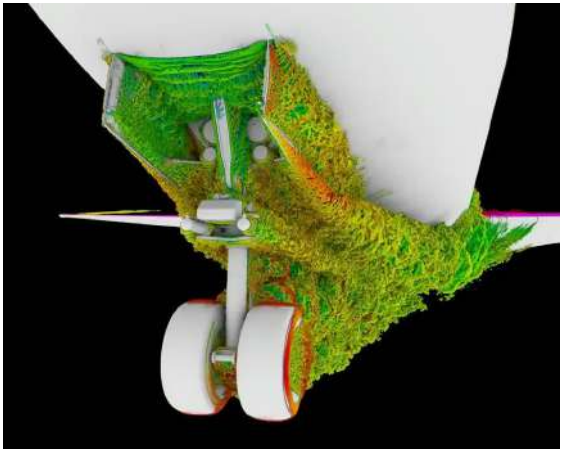


Numerical simulations of the flow around aircraft, helicopters, rotors, ..

And also..

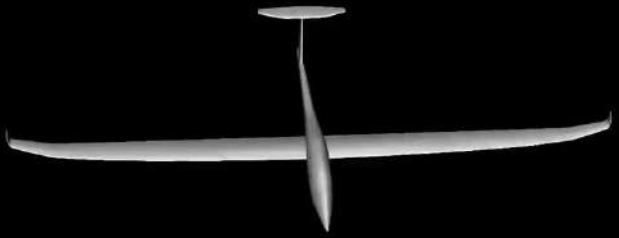


Left, top: Study of bumblebee motion
Left, bottom: Orion's abort scenario triggered as the vehicle is traveling close to the speed of sound
Right, bottom: Study of the noise of a landing gear



And also..

The spectral element code Nek5000 is used to simulate the turbulent flow around a NACA-4412 wing profile.

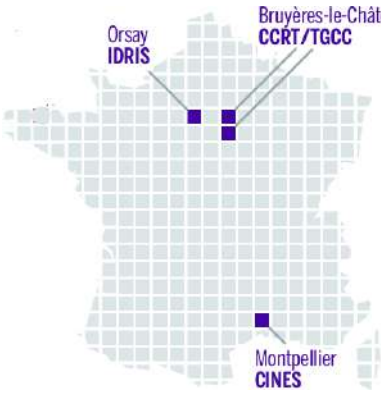


AoA = 5° & Re = 400,000

(e.g. Flow around a small glider wing)

Grand Équipement National de Calcul Intensif

- Created in 2007 by the government, GENCI aims to democratize the use of numerical simulation and supercomputing to support French competitiveness in all areas of science and industry

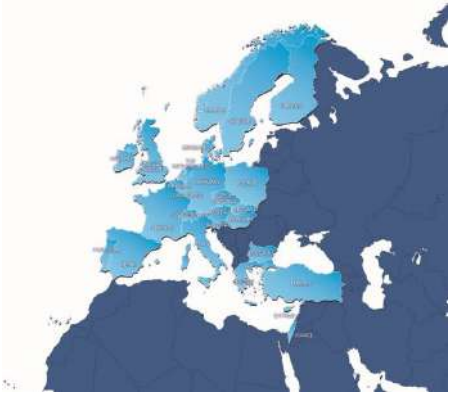


600 projects et $\approx$ 2 billion CPU hours granted each year

Supercomputing centres in Europe

Partnership for Advance Computing in Europe

https://prace-ri.eu



- " .. 97% of industrial enterprises using HPC consider it essential for their competitiveness and their survival.."
- " .. supercomputers made in the United States account for 95% of the EU market .."
- " .. guarantee to the EU independent access to HPC's technologies, systems and services .."
- " .. establish the position of the European Union as a global player in the field .."

- Consortium born in 2010, grouping 26 EU countries
- 25 Call To Projects, $\approx$ 30 billion hours

CURIE Supercomputer

Curie, property of **Genci** and hosted at TGCC (CEA), has been the first French supercomputer opened to european researcher through PRACE projects

- **Specifications Curie thin nodes**: 5040 nodes B510 bullx. For each node:
  - 2 octa-cores processors Intel Sandy Bridge (E5-2680) 2.7 GHz, 64 Go RAM, 1 SSD. Total: 10080 processors, 80640 cores.
    Nodes dedicated to MPI applications
- **Specifications Curie fat nodes**: 360 nodes S6010 bullx. For each node:
  - 4 octa-cores processors Intel Nehalem-EX X7560, 2.26 GHz, 128 Go RAM, 1 HD 2 TB. Total: 1440 processors, 11520 cores.
    Nodes dedicated to hybrid MPI-OpenMP applications
- **Specifications Curie hybrids nodes**: 16 chassis Bullx w/ 9 hybrid nodes:
  - 2 Intel Westmere 2.66 GHz / 2 Nvidia M2090 T20A. Total: 288 processors Intel + 288 processors Nvidia
- **Common characteristics**:
  - Interconnection network: InfiniBand QDR Full Fat Tree
  - Global file system: 5 PB of HD (Bandwidth 100 GB/s), 10 PB magnetic tapes, 1 PB cache

TOP500: Ranking in 2012

https://top500.org

#	Name	Site / Country		System	Number of Cores	Rmax (PFLOPs)	Power (kW)
1	**Sequoia**	DOE/LL NL	US	IBM	1572864	16.32	7890
2	**K computer**	RIKEN	Japan	Fujitsu	705024	10.51	12660
3	**Mira**	DOE/Argonne NL	US	IBM	786432	8.16	3945
4	**Supermuc**	Leibniz Rechenzentrum	Germany	IBM	147456	2.9	3423
5	**Tianhe-1A**	Nat. Supercom. Center	China	NUDT	186368	2.57	4040
6	**Jaguar**	DOE/Oak Ridge NL	US	Cray	298592	1.94	5142
7	**Fermi**	Cineca	Italy	IBM	163840	1.72	822
8	**Juqueen**	Juelich	Germany	IBM	131072	1.38	658
9	**Curie**	CEA/TGCC-GENCI	France	Bull	77184	1.36	2132
10	**Nebulae**	Nat. Supercomp. Center	China	Sugon	120640	1.27	2580

TOP500: Ranking in 2014

#	Name	Site / Country	System	Number of Cores	Rmax (PFLOPs)	Power (kW)
1	**Tianhe-2**	Nat. Supercomp. Center China	NUDT	3120000	33.86	17808
2	**Titan**	DOE/SC/Oak Ridge NL US	Cray	560640	17.59	8209
3	**Sequoia**	DOE/NNSA/LLNL US	IBM	1572864	17.17	7890
4	**K computer**	RIKEN Japan	Fujitsu	705024	10.51	12660
5	**Mira**	DOE/SC/Argonne NL US	IBM	786432	8.59	3945
6	**Piz Daint**	CSCS Switzerland	Cray	115984	6.27	2325
7	**Stampede**	Texas Adv. Comp. Center US	Dell	462462	5.17	4510
8	**Juqueen**	FZJ Germany	IBM	458752	5.01	2301
16	**Pangea**	Total E&P France	HPE	110400	2.10	2118
26	**Curie**	CEA/TGCC-GENCI France	Bull	77184	1.36	2132
35	**Tera-100**	CEA France	Bull	138368	1.05	4590
53	**Turing**	CNRS/IDRIS-GENCI France	IBM	65536	0.72	329
55	**Zumbrota**	EDF R&D France	IBM	65536	0.72	329

TOP500: Ranking in 2016

#	Name	Site / Country		System	Number of Cores	Rmax (PFLOPs)	Power (kW)
1	**Sunway**	Nat. Supercom. Center	China	NRCPC	10649600	93.01	15371
2	**Tianhe-2**	Nat. Supercom. Center	China	NUDT	3120000	33.86	17808
3	**Titan**	DOE/SC/Oak Ridge NL	US	Cray	560640	17.59	8209
4	**Sequoia**	DOE/NNSA/LLNL	US	IBM	1572864	17.17	7890
5	**K computer**	RIKEN	Japan	Fujitsu	705024	10.51	12660
6	**Mira**	DOE/SC/Argonne NL	US	IBM	786432	8.59	3945
7	**Trinity**	DOE/NNSA/LANL	US	Cray	301056	8.10	4233
8	**Piz Daint**	CSCS	Switzerland	Cray	115984	6.27	1754
11	**Pangea**	Total E&P	France	HPE	220800	5.28	4150
40	**Prolix**	Meteo France	France	Bull	72000	2.17	2534
44	**Tera-1000-1**	CEA	France	Bull	70272	1.87	1042
53	**Occigen**	GENCI-CINES	France	Bull	50544	1.63	935
62	**Curie**	CEA/TGCC-GENCI	France	Bull	77184	1.36	2132

TOP500: Ranking in 2018

#	Name	Site / Country		System	Number of Cores	Rmax (PFLOPs)	Power (kW)
1	**Summit**	DOE/SC/Oak Ridge NL	US	IBM	2282544	122.30	8806
2	**Sunway**	Nat. Supercom. Center	China	NRCPC	10649600	93.01	15371
3	**Sierra**	DOE/NNSA/LLNL	US	IBM	1572480	71.61	
4	**Tianhe-2A**	Nat. Supercom. Center	China	NUDT	4981760	61.44	18482
5	**ABCI**	AIST	Japan	Fujitsu	391680	19.88	1649
6	**Piz Daint**	CSCS	Switzerland	Cray	361760	19.59	2272
7	**Titan**	DOE/SC/Oak Ridge NL	US	Cray	560640	17.59	8209
8	**Sequoia**	DOE/NNSA/LLNL	US	IBM	1572864	17.17	7890
14	**Tera-1000-2**	CEA	France	Bull	561408	11.96	3178
30	**Pangea**	Total E&P	France	HPE	220800	5.28	4150
34	**Joliot-Curie**	CEA/TGCC-GENCI	France	Bull	79488	4.06	917
70	**Occigen2**	GENCI-CINES	France	Bull	85824	2.50	1430
145	**Curie**	CEA/TGCC-GENCI	France	Bull	77184	1.36	2132

TOP500: Ranking in 2019

#	Name	Site / Country	System	Number of Cores	Rmax (PFLOPs)	Power (kW)
1	**Summit**	DOE/SC/Oak Ridge NL US	IBM	2414592	148.60	10096
2	**Sierra**	DOE/NNSA/LLNL US	IBM	1572480	94.64	7438
3	**Sunway**	Nat. Supercom. Center China	NRCPC	10649600	93.01	15371
4	**Tianhe-2A**	Nat. Supercom. Center China	NUDT	4981760	61.44	18482
5	**Frontera**	Texas Adv. Comp. Center US	DELL	448448	23.52	
6	**Piz Daint**	CSCS Switzerland	Cray	387872	21.23	2384
7	**Trinity**	DOE/NNSA/LANL/SNL US	Cray	979082	20.16	7578
11	**Pangea III**	Total E&P France	IBM	291024	17.86	1367
18	**Tera-1000-2**	CEA France	Bull	561408	11.96	3178
38	**Pangea**	Total E&P France	HPE	220800	5.28	4150
42	**Jean Zay**	CNRS/IDRIS-GENCI France	HPE	93960	4.48	
47	**Joliot-curie**	CEA/TGCC-GENCI France	Bull	79488	4.06	917
90	**Occigen2**	GENCI-CINES France	Bull	85824	2.50	1430
305	**Curie**	CEA/TGCC-GENCI France	Bull	77184	1.36	2132

TOP500: Ranking in 2021

#	Name	Site / Country		System	Number of Cores	Rmax (PFLOPs)	Power (kW)
1	**Fugaku**	RIKEN	Japan	Fujitsu	7630848	442.01	29899
2	**Summit**	DOE/SC/Oak Ridge NL	US	IBM	2414592	148.60	10096
3	**Sierra**	DOE/NNSA/LLNL	US	IBM	1572480	94.64	7438
4	**Sunway**	Nat. Supercom. Center	China	NRCPC	10649600	93.01	15371
5	**Perlmutter**	DOE/SC/LBNL/NERSC	US	HPE	761856	70.87	2589
14	**CEA-HF**	CEA	France	Atos	810240	23.24	4959
29	**Pangea III**	Total E&P	France	IBM	291024	17.86	1367
42	**Tera-1000-2**	CEA	France	Bull	561408	11.96	3178
58	**Taranis**	Meteo France	France	Bull	294912	8.19	1672
36	**Joliot-curie rome**	CEA/TGCC-GENCI	France	Bull	197120	6.99	1436
93	**Pangea**	Total E&P	France	HPE	220800	5.28	4150
105	**Jean Zay hybrid**	CNRS/IDRIS-GENCI	France	HPE	93960	4.48	
109	**CRONOS**	EDF	France	Atos	81600	4.30	1226
113	**Joliot-curie skl**	CEA/TGCC-GENCI	France	Bull	79488	4.06	917
163	**Jean Zay**	CNRS/IDRIS-GENCI	France	HPE	61120	3.05	

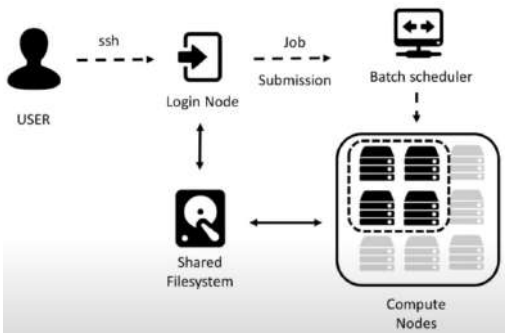
First supercomputer of history:
CRAY-1 (1976). 1 core @ 80 MHz, 0.00025 PFLOPS, 200 kW, 8 MB RAM, 5.5 tons..

Working with a supercomputer

- Remote machine
  - Login via ssh (command line), transfers via scp/ftp
- Login node:
  - Editing and transferring files, compile programs, prepare simulations, submit and monitor jobs
- Compute nodes:
  - Multicore nodes, large memories, high-speed interconnections
- Batch scheduler:
  - Resource allocation, job queueing, accounting
- File system:
  - Parallel FS, efficient I/O, node local disks

How to log in to an HPC system?
Install UNIX tools on your local machine

- Windows: Putty, MobaXterm
- Mac OSX: Terminal (pre-installed), XQuartz
- Linux: well, you already know..




```
sciacovelli@pa-dyf-ciocco:~$ ssh zay
Warning: remote port forwarding failed for listen port 52698
Last login: Fri Nov 13 10:45:36 2020 from 195.221.202.87
********************************************************************************
*                                                                              *
* Ceci est un serveur de calcul de l'IDRIS. Tout acces au systeme              *
* doit etre specifiquement autorise par l'IDRIS. Si vous tentez de             *
* de continuer a acceder cette machine alors que vous n'y etes pas             *
* autorise, vous vous exposez a des poursuites judiciaires.                    *
*                                                                              *
*                                 ---                                          *
*                                                                              *
* This is an IDRIS compute node.  Each access to this system must be           *
* properly authorized by IDRIS. If you go on accessing this machine            *
* without authorization, then you are liable to prosecution.                   *
*                                                                              *
********************************************************************************
*                                                                              *
* Orsay       CNRS / IDRIS - Frontale  -   jean-zay.idris.fr    France         *
*                                                                              *
********************************************************************************
rcx1001@jean-zay1:~$
```

Knowledge of architecture is necessary

Architectures more and more complex, with different levels of parallelism

First steps:

- **Decode** the relationship between architecture/application
- **Adapt** algorithms to modern architectures
- **Understand** the logic of a program
- **Choose** arch. and programming language depending on your needs

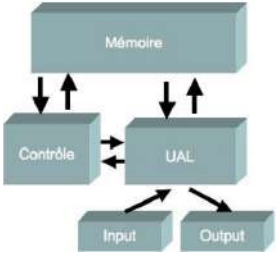
The ingredients of a supercomputer:

- Processors
  - Deliver the computing power
  - Multi-core CPUs, GPUs, coprocessors, ..
- Memory/Storage
  - Efficient I/O, large memories
  - As of today, very hierarchical (caches, RAM, SSD, HD, tapes, ..)
- Networks
  - Low-latency/high-bandwidth internode connections
  - Several types often present (MPI, I/O, ..)
- Softwares
  - Actual programs
  - Reserve resources to users, process management, ...

The balance between the different components is paramount!

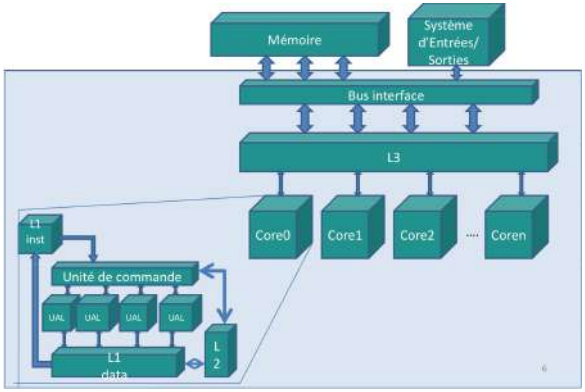
Architecture of a processor

The von Neumann Model (1945)



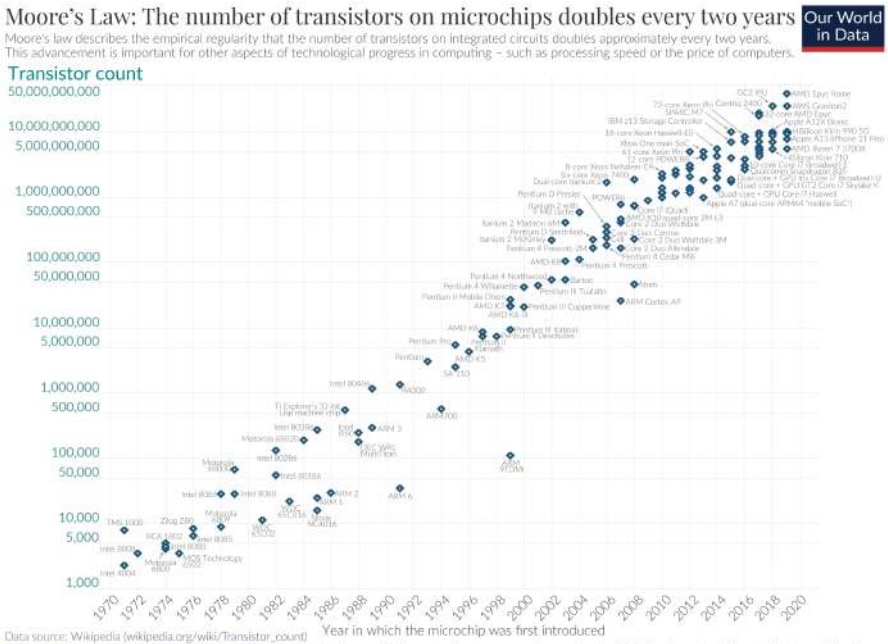
- Memory: contains program (instructions) and data
- Arithmetic and Logic Unit: performs the operations
- Control Unit: operation sequencing (read data, decode and send to execution units)
- Input/Output Unit

A more recent processor..



- Bus Management Unit (I/O Unit) connected to RAM memory
- Execution Units perform the tasks given by CU; composed by:
  - 1 or more Arithmetic and Logic Units (ALU)
  - 1 or more Floating-Point Units (FPU)
- Hierarchical memory levels

Architecture evolution: Moore's Law



- More transistors $\Longrightarrow$ more operations!
- In 10 years, the semiconductor manufacturing process passed from 0.25 μm (2000, Pentium III) to 45 nm (2010, Nehalem)
- Today, 7 nm (2020, AMD Ryzen)
- In 2022, 3 nm!

Characteristics of a processor

Clock Frequency

- Determines duration of a cycle
  - Each operation uses a given number of cycles
  - If max. frequency ↑, instructions per second ↑
- **Technological limits:**
  - Power and heat ↑↑ with f
  - Interconnection and memory are much slower!

Peak Performance (PP)

- Measured in **floating point operations/second** (FLOPs)
- Depends on:
  - The size of the registers
  - The desired accuracy:
    - ↪ Single Precision (SP): 4 bytes
    - ↪ Double Precision (DP): 8 bytes

Example: PP of Intel Core i7 9th–gen 8700K (Coffee Lake): 6 cores with a frequency of 3.7 GHz

- 2 Floating Point Units (FPU) per core, capable of executing 2 64-bit operations
- 2 8-wide Fused–Multiply–Add (FMU) instructions $\Longrightarrow$ 32 DP FLOPs/cycle

$$6 \text{ (cores)} \times 2 \text{ (FPU)} \times 2 \text{ (FMA)} \times 8 \text{ (instructions)} \times 3.7 \text{ (Ghz)} = \textbf{710 GFLOPs en DP}$$

How can we increase performances (reduce time-to-solution)?

1. Increase clock frequency (technical limits, expensive solution)
2. Improve the memory accesses: multiple cache levels, with low latency and high bandwidth
3. Use multiple computing cores: multi-core architectures
4. Allow simultaneous execution of multiple instructions

What is an instruction?

Each operation consists of a number of independent steps involving different processor components:

1. IF: Instruction Fetch
2. ID: Instruction Decode / Register Fetch
3. EX: Execution / Effective Address
4. MEM: Memory Access / Cache Access
5. WB: Write-Back

Examples:

- Load value from main memory address into register
- Store value from register into main memory address
- Perform an op. and put the result in a register
- Jump from one sequence of instructions to another

How to execute multiple instructions simultaneously?

Thread Level Parallelism (TLP)

Mix two streams of instructions to optimize the use of all resources. Examples:

- Multi-threading: process divided in mult. threads
- Hyper-threading: a processor divided into two virtual proc. (good for OS, bad for HPC!)

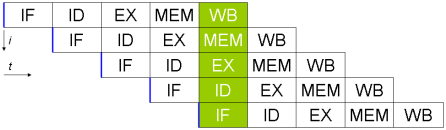
Instruction Level Parallelism (ILP)

Aggregate independent short instructions and execute them in parallel. Examples:

1. Superscalar: multiple ops. at the same time
2. Pipeline: different steps executed simultaneously on different data
3. Superpipeline: multiple pipelined operations at the same time
4. Vectorization: load multiple data into special registers and perform the same op. on all of them
5. VLIW, EPIC, ..

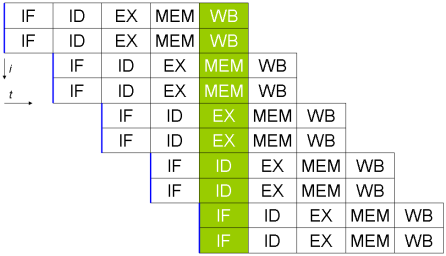
Instruction Level Parallelism: basics

- Pipelining: different steps executed simultaneously on different data



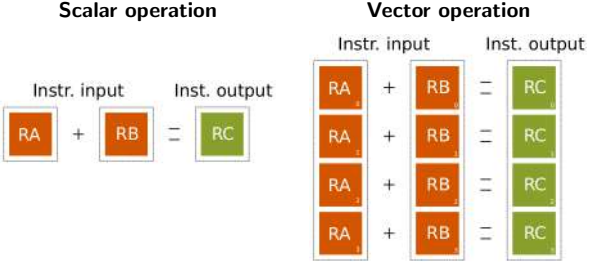
9 cycles instead of 25!

- Superpipeline: duplication of components (FMA, FPU) and simultaneous execution of multiple instructions



10 instructions in 9 cycles!

- Vectorization: load multiple data into special registers and perform the same op. on all of them

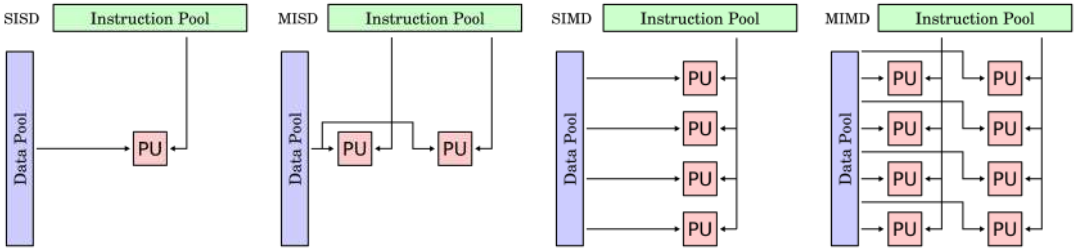


- Static (in-order) or Dynamic (out-of-order) instruction managing
- Speculative execution: hypotheses about instruction continuation after a conditional branch (the condition not being calculated yet)
- Problems of dependencies: sharing of resources, data dependencies, control dependencies, ..

Flynn's Taxonomy

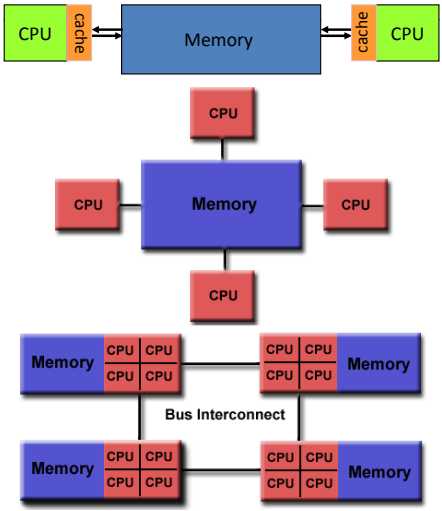
Classification of architectures according to instructions and data streams

- SISD (Single Instruction Single Data): deterministic execution, von Neumann standard model
- MISD (Multiple Instruction Single Data): same data used from multiple PU in parallel. Few practical implementations (e.g., parallel execution of several cryptography algorithms for decoding a message)
- SIMD (Single Instruction Multiple Data): memory parallelism, e.g., vector processors, pipelining
- MIMD (Multiple Instruction Multiple Data): the most common parallel architecture, whose main typologies are 1) with shared memory, 2) with distributed memory



MIMD Architectures: Shared memory

- Hardware POV: all CPUs have direct access to a common memory
- Programmer POV: all tasks have the same memory image and can access the same memory location, no matter where it is
- 2 types according to memory access times:
  - UMA (Uniform Memory Access): identical CPUs having the same memory access time
  - NUMA (Non-Uniform Memory Access): designed to overcome problems related to concurring memory accesses via the same bus. But: larger access times, need of cache-coherence management



✔ Global address space (programmer's job easier)

✔ Memory close to CPUs: fast data sharing

✘ Scalability problems: increasing the number of CPUs increases the "traffic" on the path to memory

✘ The programmer must manage the synchronization for proper memory access

✘ It's expensive to build shared memory architectures with lot of cores

MIMD Architectures: Distributed memory

- Hardware POV: based on access to network connection for non-shared physical memory
- Programmer POV: tasks can only see the local memory and must perform communications to access the memory of a remote machine on which other tasks are running
- No global address space: each PU operates independently and has its own memory.
  If a processor needs data in the memory of another proc., the programmer must explicitly define the communication.

✔ Number of procs and memory can be increased proportionally

✔ Quick access to local memory on each proc

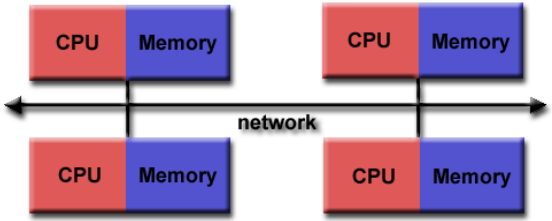
✔ No problem of cache coherence

✔ Reasonable cost (e.g., network of PCs)

✘ The programmer must handle all communications

✘ Can be difficult to match a data structure based on global memory to this physical organization of memory

✘ Large times for non-local memory access

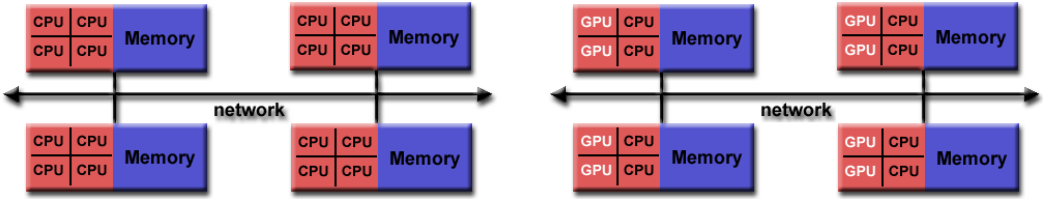


MIMD Architectures: Hybrid distributed-shared memory

- Architecture of all modern supercomputers
- **Pros and Cons**: Whatever is common to both shared and distributed memory architectures

✔ Increased scalability is an important advantage

✘ Increased programmer complexity is an important disadvantage

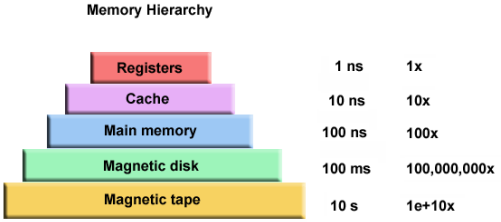


Memory vs Processor: how to feed the beast?

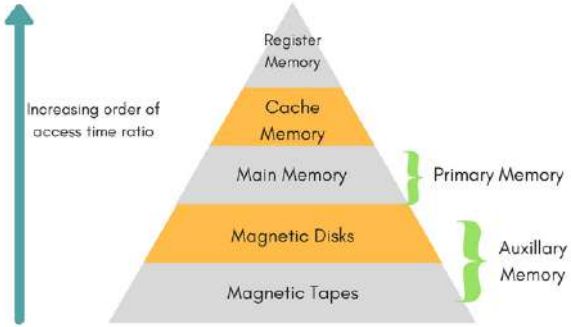
Ideal Case: memory as big, fast and close to CPU as possible!

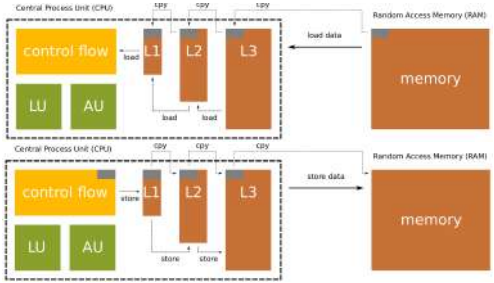
In reality, performances are limited by:

- Latency: time for a single memory access (Memory access time $\gg$ clock cycle)
- Bandwidth: $\frac{\text{Data transferred}}{\text{time}}$



- I/O operations are inhibitors to parallelism. Reduce them as much as possible!
- Fewer, larger files performs better than many small files





Cache Properties

Cache organization

- Cache is divided in lines of n words
- 2 levels of granularity:
  - CPU works on "words" (e.g., 32 or 64 bits)
  - Memory transfers are by block (>256 bits)
- The same data can be present at different memory levels: problem of coherence and propagation of the modifications
- Cache lines are organized into sets of k lines
  - The size of these sets is constant and is called degree of associativity
  - Each address has an alternative of k lines

Notion of memory locality

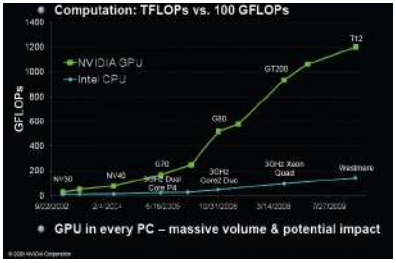
- Temporal locality: if a zone is referenced, it is likely to be referenced again in the near future
- Spatial locality: if a zone is referenced, neighboring regions are likely to be referenced in the near future

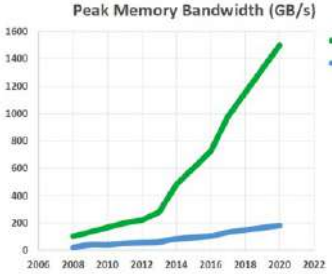
Prefetching: anticipate data and instructions that the processor will need, and thus preload them from memory hierarchy (L2 cache or central memory)

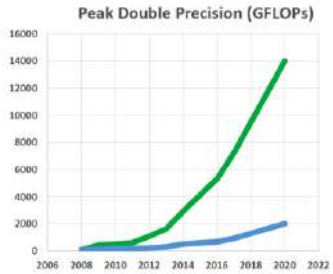
Effect of cache parameters on memory performance

- Larger cache
  - Reduce "cache misses"
  - Increase access time
- Bigger degree of associativity
  - Reduce cache conflicts
  - Can increase access time

CPU vs. GPU performance evolution (1)



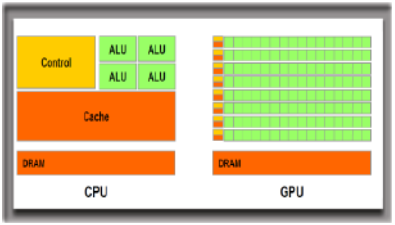




- **Theoretical Performance**

  NVIDIA A100 vs Intel Xeon Platinum 8380 (40 cores):
  9700 GFlops vs 2000 GFlops

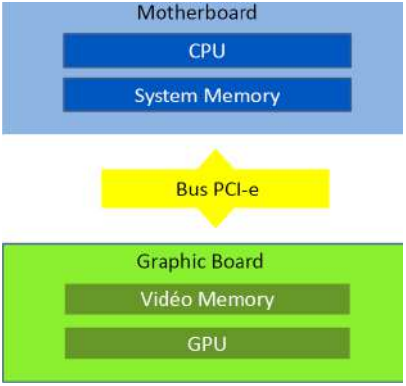
- Memory Bandwidth: 2040 GB/s vs. 200 GB/s
- Present in all PCs: a mass market
- Adapted to massively parallel concepts (thousands of threads)
- Up to a few years, only programmable via graphical APIs
- Many programming models available: CUDA, OpenCL, HMPP



CPU vs. GPU performance evolution (2)

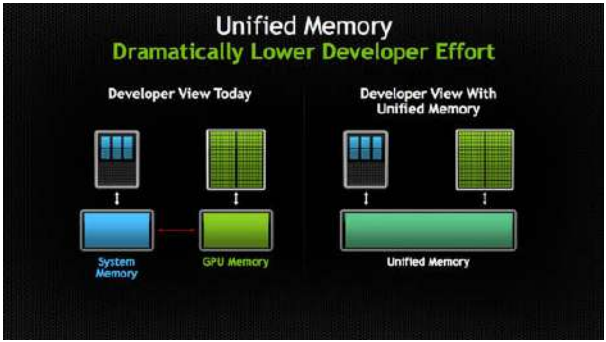
For equal intrinsic performance, GPU-based platforms:

- Take up less space
- Are cheaper
- Consume less power



But..

- Reserved for massively-parallel applications
- Data transfer through PCI-e severely limits performance
- Need to learn new tools
- What is the guarantee of durability of codes and therefore the investment in terms of porting?
  (several codes in FORTRAN77 yet..)



Coprocessors: the MIC

Xeon Phi (KNL) Coprocessor

- In 2012, presentation of the architecture MIC (Many Integrated Core)
- 1 processors: 72 cores at $\sim$ 1.5 GHz
- Developed for supercomputers, servers, high-end workstations
- Already available on most supercomputers
- Paradigm shift: **from multi-cores to many-cores**

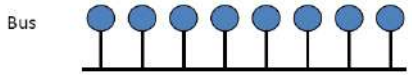
✔ Power / energy consumption ratio very high
✔ Architecture natively massively parallel ($>$ 1 TFLOPS)
✔ Use of standard CPU programming languages

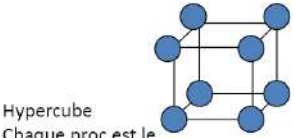
✘ Porting too difficult
✘ Big problems for memory-bounded applications
✘ Decommissioned in 2018

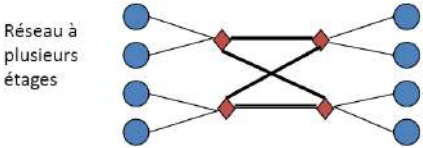
Network technology for HPC

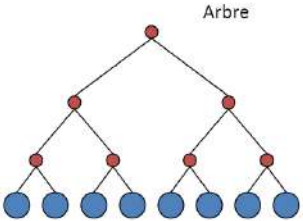
Processors and/or computers may be connected differently, with very different performance levels

- **Important characteristics**:
  - Bandwidth is the maximum bit rate
  - Latency: time needed for initialization of the communication
  - Maximum distance between two nodes
- **Interconnection topologies**:
  - Bus, Switch crossbar, Hypercube, Tree, ..









Limits and costs of Parallel Programming: Ahmdal's law

What is the largest speedup achievable for a given program?

It is given by Amdahl's Law. Consider:

- S sequential fraction of the program
- P parallel fraction of the program ($P = 1 - S$)
- N number of tasks

Maximum and potential speedup are then

$$s_{\text{max.}} = \frac{1}{\frac{P}{N} + S} = \frac{1}{\frac{1-S}{N} + S}$$

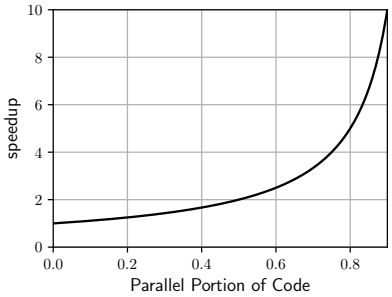
$$s_{\text{pot.}} = \lim_{N \to \infty} s_{\text{max.}} = \frac{1}{S} = \frac{1}{1-P}$$

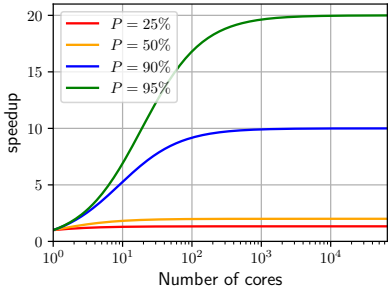
$$P = 0 \implies s_{\text{pot.}} = 1 \quad \text{(no speedup)}$$

$$P = 1 \implies s_{\text{pot.}} \to \infty$$

$$P = 0.5 \implies s_{\text{pot.}} = 2 \quad \text{the code will run twice as fast at best}$$

You can spend a lifetime getting 95% of your code to be parallel, and never achieve better than 20x speedup no matter how many processors you throw at it!





Limits and costs of Parallel Programming

Speedup as a function of P and N

N	P = 50%	P = 90%	P = 95%	P = 99%
10	1.82	5.26	6.89	9.17
100	1.98	9.17	16.80	50.25
1000	1.99	9.91	19.62	90.99
10000	1.99	9.91	19.96	99.02
100000	1.99	9.99	19.99	99.90

- Scalability limits become soon obvious
- In practice, sequential and parallel versions of the algorithms can be very different
- Algorithm specificities not considered: e.g., when the parallel algorithm is more efficient (superlinear accelerations)

- Some problems show increased performance by increasing the problem size, e.g.:

Parallelizable fraction	85 seconds	85%
Serial fraction	15 seconds	15%

- Increasing the problem size by doubling the grid and halving the time step (x4 grid points, x2 time steps):

Parallelizable fraction	680 seconds	97.84%
Serial fraction	15 seconds	2.16%

Problems that increase the percentage of parallel time with their size are more scalable than problems with a fixed percentage of parallel time.

Limits and costs of Parallel Programming

- **Complexity**: measured in programmer time in every aspect of the development cycle (Design, Coding, Debugging, Tuning, Maintenance)
- **Portability**
- **Resource Requirements**:
  - Primary aim is to decrease wall clock time T_{wc}
  - ↪ However in order to accomplish this, more CPU time is required ($t_{CPU} = N \times T_{wc}$)
  - The amount of memory required can be greater for parallel codes than serial codes
  - ↪ Need to replicate some data and overheads for parallel libraries
  - For short running parallel programs, there can actually be a decrease in performance compared to a similar serial implementation
  - ↪ overhead for task creation/termination and communications

The cost of communications over the program acceleration should be considered!

Let's suppose $S = 0$:

$$T_{seq} = T_{calc} \quad \text{and} \quad T_{par} = \frac{T_{calc}}{N} + T_{comm}$$

$$\Longrightarrow \quad \frac{T_{seq}}{T_{par}} = \frac{T_{calc}}{\frac{T_{calc}}{N} + T_{comm}} = \frac{N}{1 + N \times \frac{T_{comm}}{T_{calc}}}$$

$$\Longrightarrow \quad \frac{T_{comm}}{T_{calc}} \text{ is a fundamental parameter!}$$

Definitions:

$$\text{Speedup} = \frac{T_{wc}^{1 \text{ task}}}{T_{wc}^{N \text{ tasks}}} \qquad \text{(ideally, } N\text{)}$$

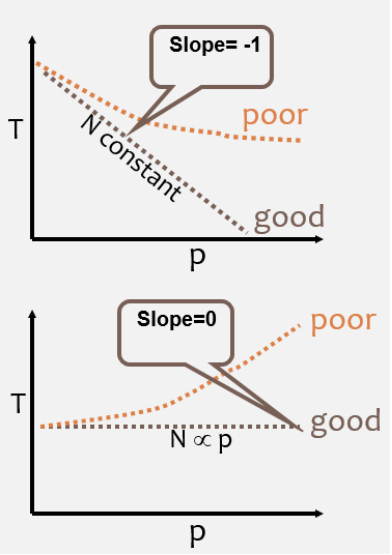
$$\text{Efficiency} = \frac{T_{wc}^{1 \text{ task}}}{T_{wc}^{N \text{ tasks}} \times N} \qquad \text{(ideally, 100\%)}$$

- Generally, there are limits intrinsic to the algorithms used
- Both hardware and software play a fundamental role on scalability properties of a given program

Scalability

Two types of scaling based on time-to-solution:

- **Strong scaling**: increase the number of tasks, keeping constant the total problem size (problem size assigned at each task decreasing)
  - Goal: run the same problem size faster
  - Ideal limit: execution time inversely proportional to the number of tasks used.
- **Weak scaling**: increase the number of tasks, keeping constant the problem size assigned at each task (total problem size increasing)
  - Goal: run larger problem in same amount of time
  - Ideal limit: constant execution time



- The scalability of a parallel program is a result of a number of interrelated factors.
  Simply adding more processors is rarely the answer
- The algorithm may have inherent limits to scalability. At some point, adding more resources causes performance to decrease
- Hardware factors play a significant role in scalability

Parallel Programming Models

- Parallel programming models exist as an abstraction above hardware and memory architectures.
  - These models are NOT specific to a particular type of machine or memory architecture
  - Any of these models can (theoretically) be implemented on any underlying hardware

1. Shared Memory (without threads)
2. **Shared Memory (with threads)**
3. **Distributed Memory / Message Passing**
4. Data Parallel
5. **Hybrid**

From Flynn's taxonomy POV:

1. Single Program Multiple Data (SPMD)
2. Multiple Program Multiple Data (MPMD)

- Models in bold are used in CFD
- Which model to use? This is often a combination of what is available and personal choice.
- There is no "best" model, although there are certainly better implementations of some models over others

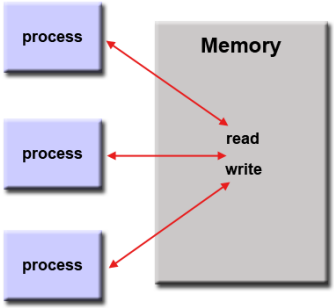
Shared Memory Model (without threads)

- Common address space shared, asynchronous read/write
- Locks / semaphores for memory access control, resolve contentions, prevent race conditions and deadlocks

✔ no need to specify explicitly data communication

✔ All processes see and have equal access to shared memory

✘ More difficult to understand/manage data locality

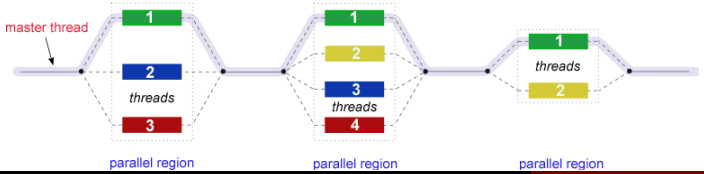


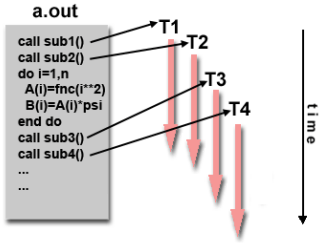
Implementations:

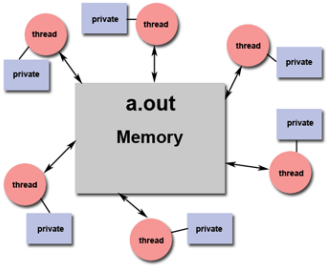
- Stand-alone shared memory machines, native operating systems, compilers and/or hardware provide support for shared memory programming. For example, the POSIX standard provides an API for using shared memory, and UNIX provides shared memory segments (shmget, shmat, shmctl, etc).
- On distributed memory machines, memory is physically distributed across a network of machines, but made global through specialized hardware and software. A variety of SHMEM implementations are available: `http://en.wikipedia.org/wiki/SHMEM`

Shared Memory Model (with threads)

- A single "heavy weight" process have multiple "light weight" concurrent execution paths
- Example: main program `a.out` is scheduled to run by the OS
  - It loads and acquires all of the necessary system and user resources to run; performs some serial work and creates a number of tasks (threads) that can be scheduled and run by the OS concurrently
  - Each thread has local data and shares the entire resources of `a.out`, saving the overhead associated with replicating a program's resources for each thread. Each thread also benefits from a global memory view because it shares the memory space of `a.out`
  - Threads communicate through global memory (synchronization required to ensure two threads don't update the same address concurrently)
  - Threads can come and go, but `a.out` remains present to provide the necessary shared resources until the application has completed







Shared Memory Model (with threads) – Implementations

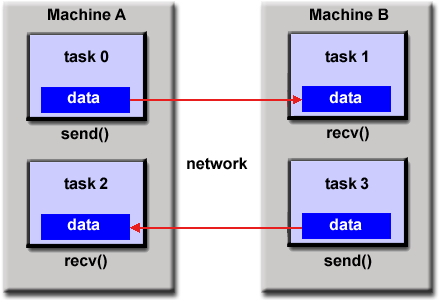
- Threads implementations commonly comprise:
  1. A library of subroutines called from within source code
  2. A set of compiler directives embedded in source code

  The programmer is responsible for determining the parallelism!
- Hardware vendors have implemented their own proprietary versions of threads (portability problems)
- Standardization efforts resulted in two different implementations of threads: POSIX Threads and OpenMP

- **POSIX Threads** (a.k.a. Pthreads): Specified by the IEEE POSIX 1003.1c standard (1995)
  - Library based, C Language only, part of Unix/Linux operating systems
  - Very explicit parallelism; requires significant programmer attention to detail
- **OpenMP**: Industry standard, jointly defined and endorsed by a group of major computer hardware and software vendors, organizations and individuals
  - Compiler directive based, Portable/multi-platform, including Unix and Windows platforms, Available in C/C++ and Fortran implementations
  - Can be very easy and simple to use, provides for "incremental parallelism". Can begin with serial code.
- Other common threaded implementations: Microsoft threads, Java/Python threads, CUDA for GPUs

Distributed Memory / Message Passing Model

- Set of tasks that use their own local memory during computation
- Multiple tasks can reside on the same physical machine and/or across an arbitrary number of machines
- Tasks exchange data through communications by sending and receiving messages
- Data transfer usually requires cooperative operations to be performed by each process (e.g., a `SEND` operation must have a matching `RECEIVE` operation)



Implementations:

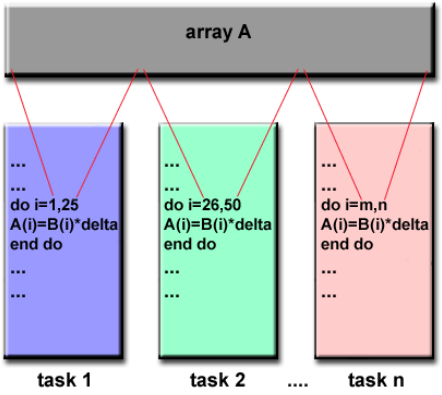
- Message passing implementations comprise:
  - A library of subroutines
  - Calls to these subroutines are embedded in source code

  The programmer is responsible for determining the parallelism!

- Standardization: Message-Passing Interface (MPI) is the "de facto" industry standard for message passing. Specifications at `http://www.mpi-forum.org/docs/`.

Data Parallel Model

- Also referred to as the Partitioned Global Address Space (PGAS) model. Address space is treated globally
- Most of the parallel work focuses on performing operations on a data set, typically organized into a common structure
- A set of tasks work collectively on the same data structure; however, each task works on a different partition
- Tasks perform the same operation on their partition
- On shared memory arch., all tasks may have access to the data structure through global memory
- On distributed memory arch., the global data structure can be split up logically and/or physically across tasks



Implementations:

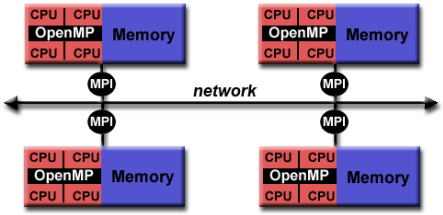
- Coarray Fortran: extensions to Fortran 95 for SPMD programming, compiler dependent
- Unified Parallel C (UPC): extension to C for SPMD programming, compiler dependent
- X10: PGAS-based programming language developed by IBM
- Global Arrays: shared-memory style environment in the context of distributed array data structures. Public domain library with C and Fortran77 bindings
- Chapel: open-source programming language project led by Cray

Hybrid Model

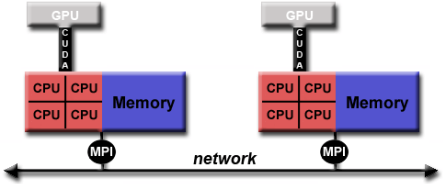
Combines more than one of the previously described programming models. Common examples:

1. Message Passing model (MPI) + threads model (OpenMP)
   - Threads perform computationally intensive kernels using local, on-node data
   - Communications between processes on different nodes occurs over the network using MPI
2. MPI with CPU–GPU programming
   - MPI tasks run on CPUs using local memory and communicating with each other over a network
   - Computationally intensive kernels are off-loaded to GPUs on-node
   - Data exchange between node-local memory and GPUs uses CUDA
3. MPI with Pthreads, MPI with non-GPU accelerators, ...

MPI + OpenMP



MPI + CPU-GPU

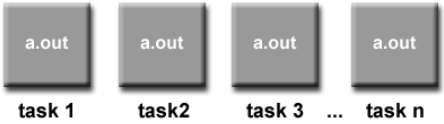


SPMD and MPMD models

"High-level" models that can be built upon any of the previously mentioned programming models.

Single Program Multiple Data (SPMD)

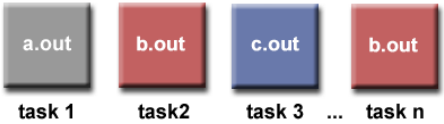
- **Single Program**: All tasks execute their copy of the same program simultaneously. This program can be threads, message passing, data parallel or hybrid.
- **Multiple Data**: All tasks may use different data



Tasks do not necessarily have to execute the entire program – perhaps only a portion of it. The SPMD model using MPI or hybrid programming is the most common parallel programming model for multi-node clusters.

Multiple Program Multiple Data (MPMD)

- **Multiple Program**: Tasks may execute different programs simultaneously. The programs can be threads, message passing, data parallel or hybrid.
- **Multiple Data**: All tasks may use different data



MPMD programs not as common as SPMD, but may be better suited for certain types of problems.

First steps in developing parallel software

- Understand the problem that must be solved
- Determine if it can actually be parallelized
  - ↪ Problem easy to parallelize: calculate the potential energy for each of several thousand independent conformations of a molecule and find the minimum energy conformation
  - ↪ Problem with little-to-no parallelism: calculation of the Fibonacci series by use of the formula $F(n) = F(n-1) + F(n-2)$
- Identify the program's hotspots:
  - Know where most of the real work is being done
  - Profilers and performance analysis tools can help here
  - Focus on parallelizing the hotspots and ignore programs sections accounting for little CPU usage
- Identify bottlenecks:
  - Are there areas that are disproportionately slow, or cause parallelizable work to halt or be deferred? (e.g., I/O)
  - Restructure the program or use a different algorithm
- Identify inhibitors to parallelism: one common class is data dependence (as in the Fibonacci sequence)
- Investigate other algorithms if possible!

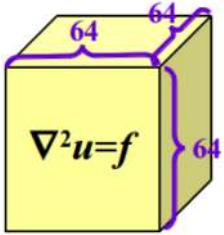
Also, take advantage of highly optimized math libraries available from vendors (IBM ESSL, Intel MKL, AMD AMCL..)

Power of optimal algorithms

Advances in **algorithmic efficiency** as important as hardware architecture!

- Consider Poisson's equation on a cube of size $N = n^3$:

Year	Method	Reference	Storage	FLOPs
1947	GE (Banded)	Von Neumann	n^5	n^7
1950	Optimal SOR	Young	n^3	$n^4 \log n$
1971	CG	Reid	n^3	$n^{3.5} \log n$
1984	Full MG	Brandt	n^3	n^3

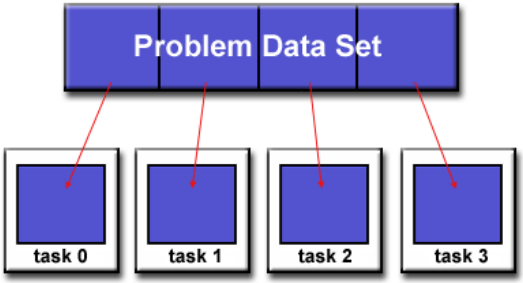


- $n = 64$: overall reduction in FLOPs of ≈ 16 million
- 1 month reduced down to 0.1 seconds...

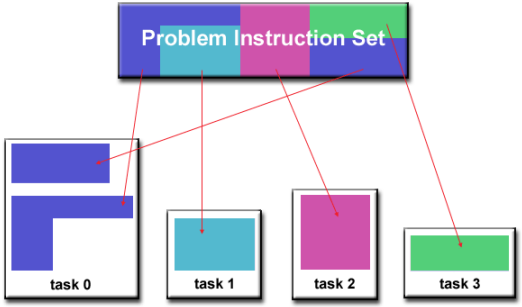
Decomposition / Partitioning

Break the problem into discrete "chunks" of work that can be distributed to multiple tasks

Domain Decomposition

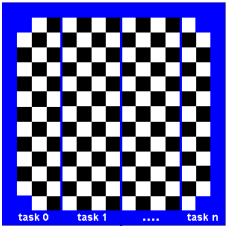


Functional Decomposition



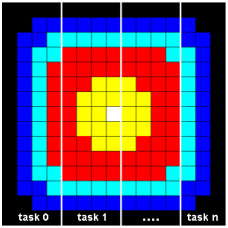
Communications

- Need for communications between tasks depends upon the problem:
  - Little / no comms, ex.: image processing where every pixel in a black and white image needs to have its color reversed (embarrassingly parallel problems)
  - Need for comms:, ex.: 2-D heat diffusion problem (most of parallel applications)



Factors to consider:

1. Communication overhead: resources that could be used for computation are instead used to package and transmit data
   - Comms frequently require sync between tasks $\Longrightarrow$ waiting instead working
   - Competing comms traffic can saturate the network bandwidth $\Longrightarrow$ further performance degradation



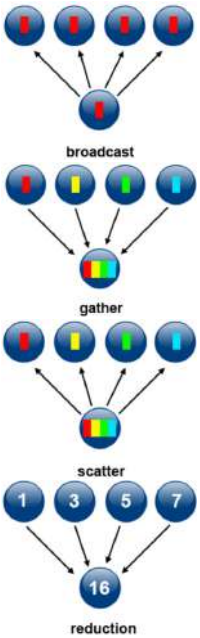
2. Latency vs. Bandwidth
   - Latency (in μs): time to send a minimum message from A to B
   - Bandwidth (in GB/s): data that can be communicated per unit of time
   - Sending many small messages can cause latency to dominate communication overheads. Often it is more efficient to pack small messages into a larger message, thus increasing the effective communications bandwidth

3. Visibility of communications
   - With MPI, comms are explicit and under the programmer's control

Communications (cont'd)

Factors to consider:

4. Synchronous vs. asynchronous communications
   - Synchronous comms require "handshaking" between tasks (explicit or implicit). Often referred to as blocking comms since other work must wait until completion
   - Asynchronous comms allow tasks to transfer data independently from one another. Often referred to as non-blocking comms since other work can be done
   - With asynchronous comms, one can interleave computation with communication
5. Scope of communications
   - Knowing which tasks must communicate with each other is critical during the design stage of a parallel code. Both of the two scopings described below can be implemented synchronously or asynchronously
   - Point-to-point: involves two tasks with one task acting as the sender/producer of data, and the other acting as the receiver/consumer.
   - Collective: involves data sharing between more than two tasks, often specified as members in a common group (e.g., broadcast, gather, scatter, reduction)
6. Efficiency of communications
   - Which implementation for a given model should be used? (e.g., for MPI, Intel MPI, OpenMPI, MPICH, Microsoft MPI..)
   - What type of communication operations should be used?
   - The programmer has choices that can affect performance!

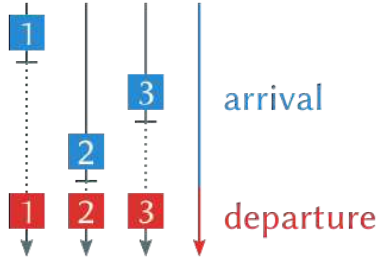


Synchronization

- Managing the sequence of work and the tasks performing it is a critical design consideration for most parallel programs
- Can be a significant factor in program performance (or lack of it)
- Often requires "serialization" of segments of the program

Types of Synchronization

1. Barrier: involves all tasks
   - Each task performs its work until it reaches the barrier, then stops. When the last task reaches the barrier, all tasks are synchronized
2. Lock / semaphore: can involve any number of tasks
   - Typically used to serialize (protect) access to global data
   - First task to acquire the lock "sets" it. Other tasks can attempt to acquire the lock but must wait until the first task releases it
3. Synchronous communication operations: Involves only those tasks executing a communication operation
   - Some form of coordination is required with the other tasks participating in the communication



Data Dependencies

- A dependence between statements exists when the order of statement execution affects the results
- Results from multiple use of the same location in storage by different tasks
- One of the primary inhibitors to parallelism. Examples:

Loop carried data dependence

```
DO J = 1,N
   A(J) = A(J-1) + 2
END DO
```

`A(J-1)` must be computed before `A(J)`. If Task 2 has `A(J)` and task 1 has `A(J-1)`, computing the correct value of `A(J)` necessitates:

- Distributed memory arch.: task 2 must obtain the value of `A(J-1)` from task 1 after task 1 finishes its computation
- Shared memory arch.: task 2 must read `A(J-1)` after task 1 updates it

Loop independent data dependence

```
task 1        task 2
------        ------
X = 2         X = 4
  .             .
  .             .
Y = X**2      Y = X**3
```

The value of `Y` is dependent on:

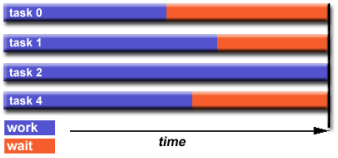
- Distributed memory arch.: if or when the value of X is communicated between the tasks
- Shared memory arch.: which task last stores the value of X

How to Handle Data Dependencies:

- Distributed memory arch.: communicate required data at synchronization points
- Shared memory arch.: synchronize read/write operations between tasks

Load Balancing

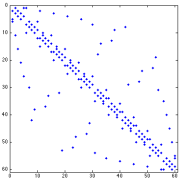
- Distributing approximately equal amounts of work among tasks so that all tasks are kept busy all of the time
- It can be considered a minimization of task idle time
- Important for performance: e.g., if all tasks are subject to a barrier, the slowest task will determine the overall performance

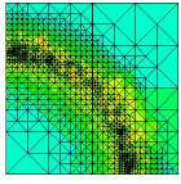


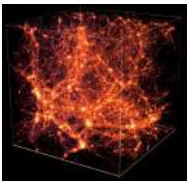
How to achieve Load Balance

- Equally partition the work each task receives. If heterogeneous mix of machines in use, adjust work accordingly
- Use dynamic work assignment

Certain classes of problems result in load imbalances even if data is evenly distributed among tasks:







1. Sparse arrays: some tasks will have actual data to work on while others have mostly "zeros"
2. Adaptive grid methods: some tasks may need to refine their mesh while others don't
3. N-body simulations: particles may migrate across task domains requiring more work for some tasks

Granularity

Qualitative measure of the ratio of computation to communication

Computation is typically separated from communication by synchronization events

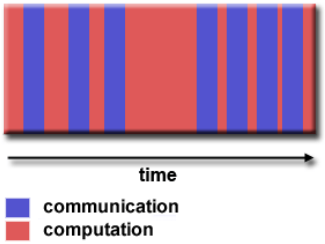
Fine-grain Parallelism

Small amounts of computational work done between comms

- Low computation to communication ratio

✔ Facilitates load balancing

✘ High communication overhead for comms/syncs between tasks: if granularity too fine, it can takes longer than the computation



Coarse-grain Parallelism

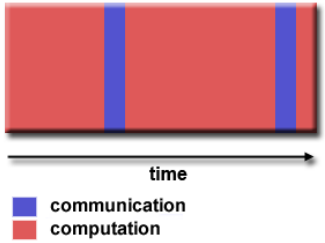
Large amounts of computational work done between comms/syncs

- High computation to communication ratio

✔ More opportunity for performance increase

✘ Harder to load balance efficiently

Which is best? Depends on the algorithm and the hardware!
In most cases, coarse granularity is advantageous



HPC: High Performance Computing

"Computational science has become the third pillar of the scientific enterprise, a peer alongside theory and physical experiment"

- Develop products/services or carry out research without going through experimentation
  - Suppression of material prototypes: costs and time ↓, quality and innovation ↑
- **A strategic bet**:
  - Essential tool for competitiveness and innovation for all industrial sectors and research laboratories
  - Huge increase in the amount of digital information available and increasing complexity of systems to design $\Longrightarrow$ mastering modeling techniques and HPC is the key to success
  - Matter of sovereignty and security of States (energy independence, risk prevention, defense..)

Source of performance degradation

- Latency: waiting for access to memory
- Overhead: extra work for managing concurrency and parallel resources
- Starvation: not enough work to do due to insufficient parallelism or poor load balancing
- Contention: delays due to fighting over what task gets to use a shared resource next

The biggest challenges for HPC:

- Visualization/exploitation of large volumes of data
- Keep up / adapt to rapidly-evolving technologies
- Need for multi-disciplinary teams, develop partnerships, sharing competences, ..
- Control energy consumption and avoid breakdowns

HPC vs Computer Science

Most people in HPC are no computer scientists!

- Software has to be correct first and (then) efficient; packages can be over 30 years old
- Technology is a mix of "high-end" and "stone age"

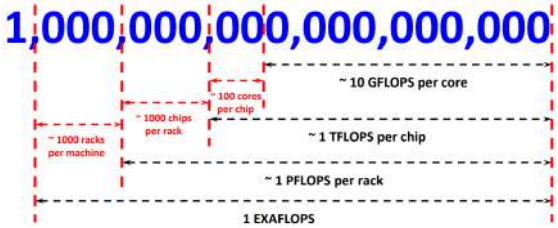
So what skills do I need to for HPC?

- Common sense, cross-discipline perspective
- Good understanding of calculus and (some) physics
- Patience and creativity, ability to deal with "jargon"

HPC is more like a craft than a science...

- Practical experience is most important
- Leveraging existing solutions is preferred over inventing new ones requiring rewrites
- A good solution today is worth more than a better solution tomorrow
- A readable and maintainable solution is better than a complicated one

Future trends



Short to mid term: race to Exascale

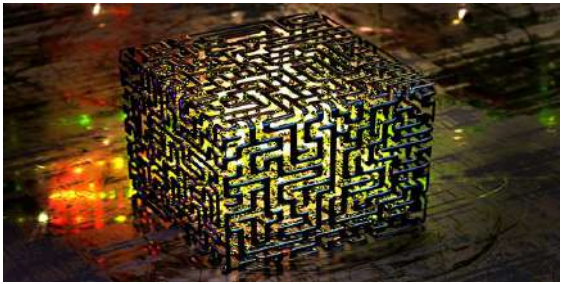
- What: 1 EXAFLOP = 10^{18} FLOPS
- When: 2022, probably
- Where: US or China

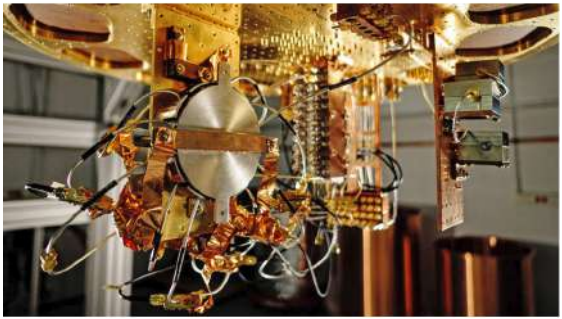

NEWS QUANTUM PHYSICS

Rumors hint that Google has accomplished quantum supremacy

Reports suggest a quantum computer has surpassed standard computers on a specific type of calculation

Mid to long term: Quantum Computing





Useful jargon (1)

- **High-Performance Computing**: Using world's largest computers to solve large problems
- **Node**: a standalone "computer in a box". Usually comprised of multiple CPUs/processors/cores, memory, network interfaces, etc. Nodes are networked together to comprise a supercomputer
- **CPU / Socket / Processor / Core**: in the past, a CPU (Central Processing Unit) was a singular execution component for a computer. Then, multiple CPUs were incorporated into a node. Then, individual CPUs were subdivided into multiple "cores", each being a unique execution unit. CPUs with multiple cores are sometimes called "sockets" - vendor dependent. The result is a node with multiple CPUs, each containing multiple cores
- **Task**: a logically discrete section of computational work.
  A task is typically a program or program-like set of instructions that is executed by a processor.
  A parallel program consists of multiple tasks running on multiple processors
- **Pipelining**: breaking a task into steps performed by different processor units, with inputs streaming through, much like an assembly line; a type of parallel computing
- **Shared Memory**: from hardware point of view, describes a computer architecture where all processors have direct (usually bus based) access to common physical memory. In a programming sense, it describes a model where parallel tasks all have the same "picture" of memory and can directly address and access the same logical memory locations regardless of where the physical memory actually exists
- **Symmetric Multi-Processor (SMP)**: shared memory hardware architecture where multiple processors share a single address space and have equal access to all resources.

Useful jargon (2)

- **Distributed Memory**: in hardware, refers to network based memory access for physical memory that is not common. As a programming model, tasks can only logically "see" local machine memory and must use communications to access memory on other machines where other tasks are executing
- **Communications**: parallel tasks typically need to exchange data. There are several ways this can be accomplished, such as through a shared memory bus or over a network
- **Synchronization**: the coordination of parallel tasks in real time, very often associated with communications. Usually involves waiting by at least one task, and can therefore cause an increase of wall clock execution time
- **Granularity**: qualitative measure of the ratio of computation to communication.
  Coarse: relatively large amounts of computational work are done between communication events
  Fine: relatively small amounts of computational work are done between communication events
- **Observed Speedup**: defined as the ratio of serial execution to parallel execution wall-clock time, the simplest indicator for a parallel program's performance
- **Parallel Overhead**: amount of time required to coordinate parallel tasks, as opposed to doing useful work. Includes factors such as task start-up and termination time, synchronizations, communications, software overhead imposed by parallel languages, libraries, ...
- **Massively Parallel**: refers to the hardware that comprises a given parallel system - having many processing elements. The meaning of "many" keeps increasing; currently numbering in the millions
- **Embarrassingly Parallel**: solving many similar, but independent tasks simultaneously; little to no need for coordination between the tasks
- **Scalability**: refers to a parallel system's (hardware and/or software) ability to demonstrate a proportionate increase in parallel speedup with the addition of more resources.