Introduction to High-Performance Computing

Giorgio Amati
Alessandro Ceci

Corso di dottorato in Ingegneria Aeronautica e Spaziale 2025

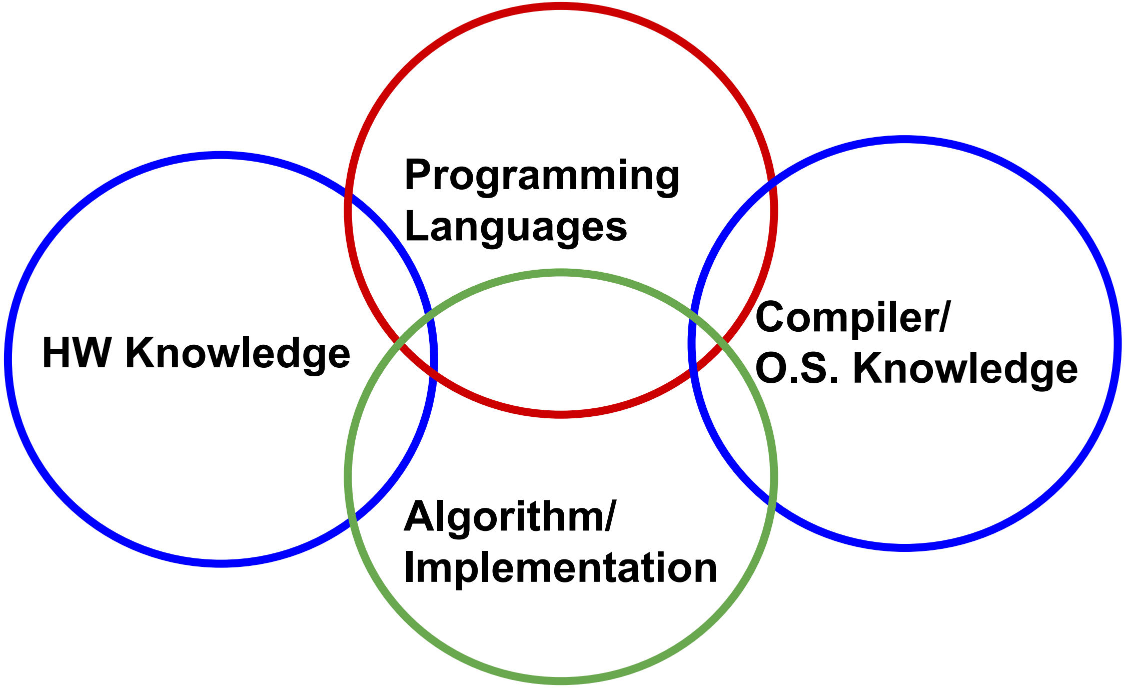
g.amati@cineca.it/g.amaticode@gmail.com
alessandro.ceci@uniroma1.it

Agenda

- ✓ HPC: What it is?
- ✓ Hardware: how it works
- ✓ Algorithm vs. Implementation
- ✓ Compiler + Floating point + I/O
- ✓ HW & Parallel Paradigm
- ✓ Conclusions & Comments

HPC: what it is?

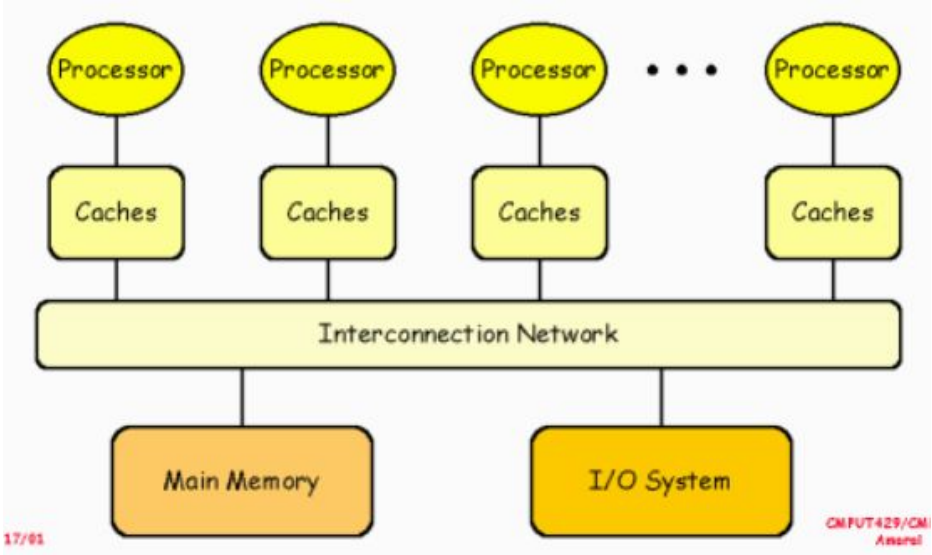
✓ These are the main skills for an efficient HPC



Shared Memory Machine

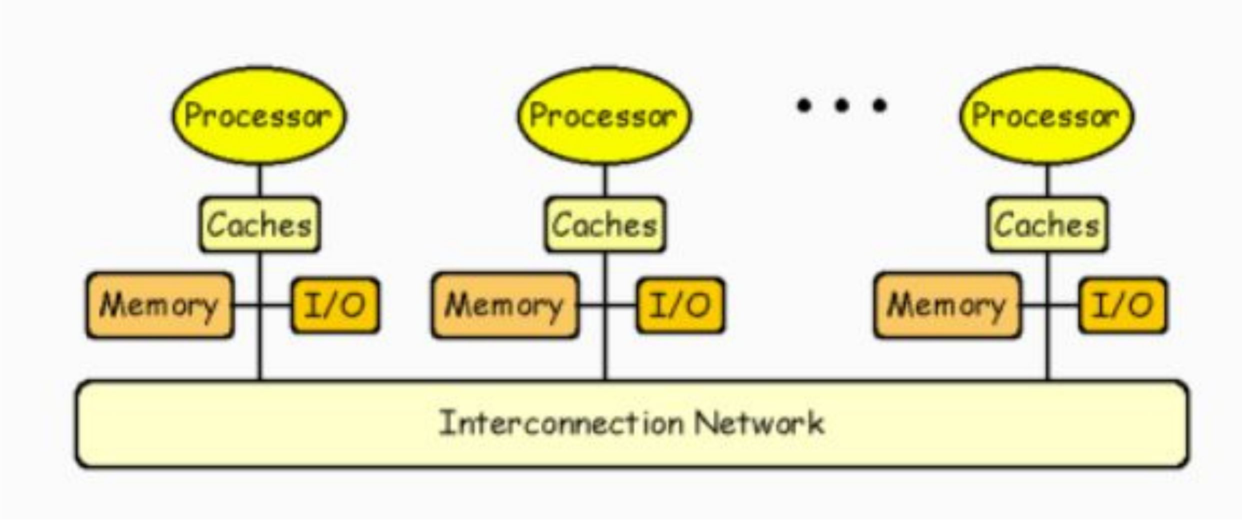
✓ A shared-memory system is an architecture consisting of a number of processors, all of which have direct (i.e. hardware) access to all the main memory in the system. This permits **any** of the system processors to access data that **any** of the other processors has created or will use

- **UMA**: Uniform Memory Access
- **NUMA**: Non Uniform Memory Access



Distributed Memory Machine

- ✓ Distributed memory refers to a computing system in which each processor (or a node) has its memory. Computational tasks efficiently operate with local data, but when remote data is required, the task must communicate (using explicit messages) with remote processors to transfer the right data.



GPU vs. CPU

- ✓ General-purpose computing on graphics processing units (GPGPU) is the use of a graphics processing unit (GPU), which typically handles computation only for computer graphics, to perform computation in applications traditionally handled by the central processing unit (CPU). The use of multiple video cards in one computer, or large numbers of graphics chips, further parallelism the already parallel nature of graphics processing.
- ✓ In brief:
  - GPU are less “flexible” respect CPU
  - GPU could be really more performing respect CPU
  - Classical example:
    - **GPU → BUS**
    - **CPU → sport car**

Why GPUs?

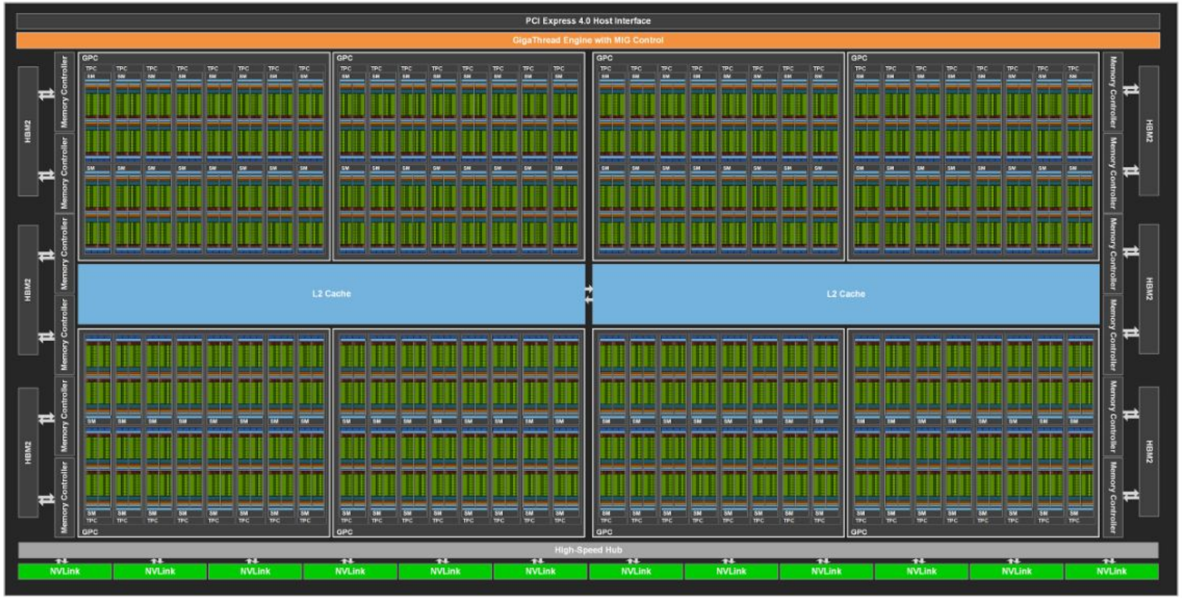
✓ Pro

- GPU more powerful: 1 GPU ~ 10x CPU (Peak Mflops)
- GPU ask for less space: for same performance CPU ask for **~3x racks**
- GPU are less expensive: for same peak performance CPU are **~2x expensive**
- GPU asks for (relative) less power: for the same peak performance CPU **~4x energy**

✓ Cons

- GPU are less flexible respect CPU
- Some algorithm are not GPU-friendly
- There's no a common programming model between different vendors
- Porting to GPU is expensive and error-prone procedure

GPU



NVIDIA GA100

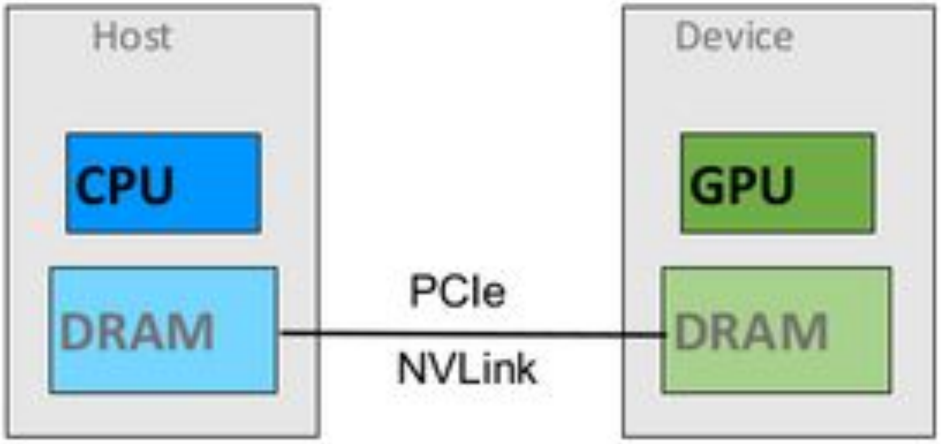
- ✓ up to 128 Streaming multiprocessor (SM)
- ✓ Each SM has
  - 64 FPU@32bit
  - 32 FPU@64bit
  - 64 INT@32bit

Heterogeneous programming/1

Host-device model

- Host: CPU and its memory
- Device: GPU (or other) and its memory

1. Allocate memory on host
2. Data transfer from host to devic
3. Execute on device
4. Data transfer from device to ho
5. ….



Host-Device data movement

- ✓ Host-device BW is usually lower with respect to device-device BW
- ✓ Try to minimize data transfer
  - use intermediate results (take care of arrays)
  - compute as possible “on the fly”

Model	PCIe BW	Gen	D-D BW
V100	12 GB/s	Gen3	700 GB/s
A100	24 GB/s	Gen4	1700 GB/s
B200	48 GB/s	Gen5	7000 GB/s

Heterogeneous programming/2

Host-device model

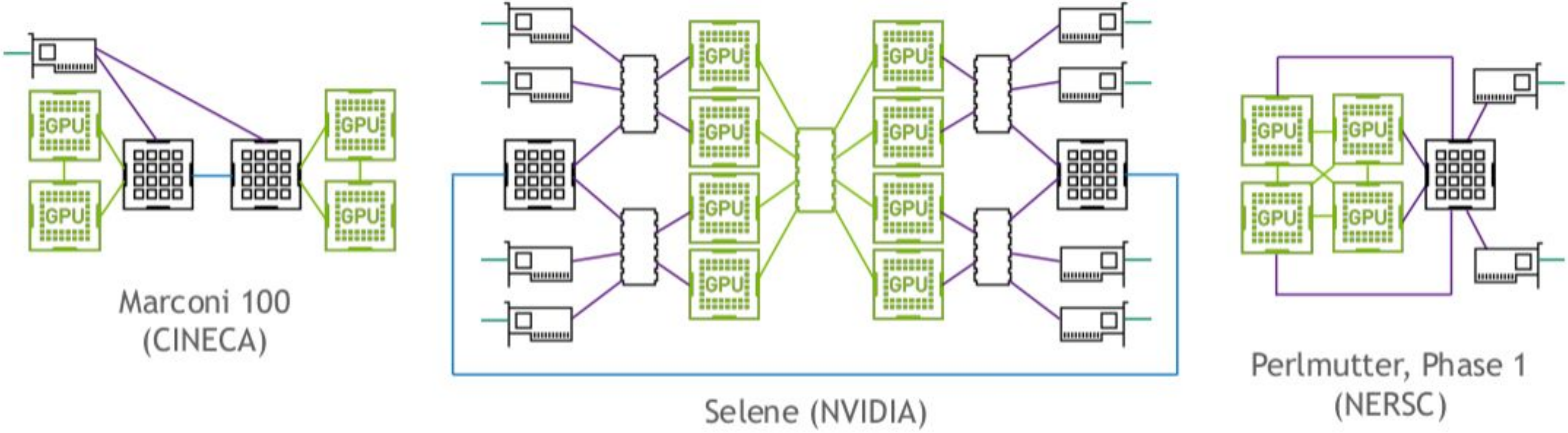
- Host: CPU and its memory
- Device: GPU (or other) and its memory

1. Allocate memory on host
2. Data transfer from host to device
3. Execute on device
4. Data transfer from device to host
5. ….

- ✓ CPU = O(1) TF
- ✓ GPU = O(10) TF
- ✓ BW CPU = O(100) GB/s
- ✓ BW GPU = O(1000) GB/s
- ✓ PCIe = 12-48 GB/s
- ✓ NVLINK = 25-400 GB/s

DIfferent machine….

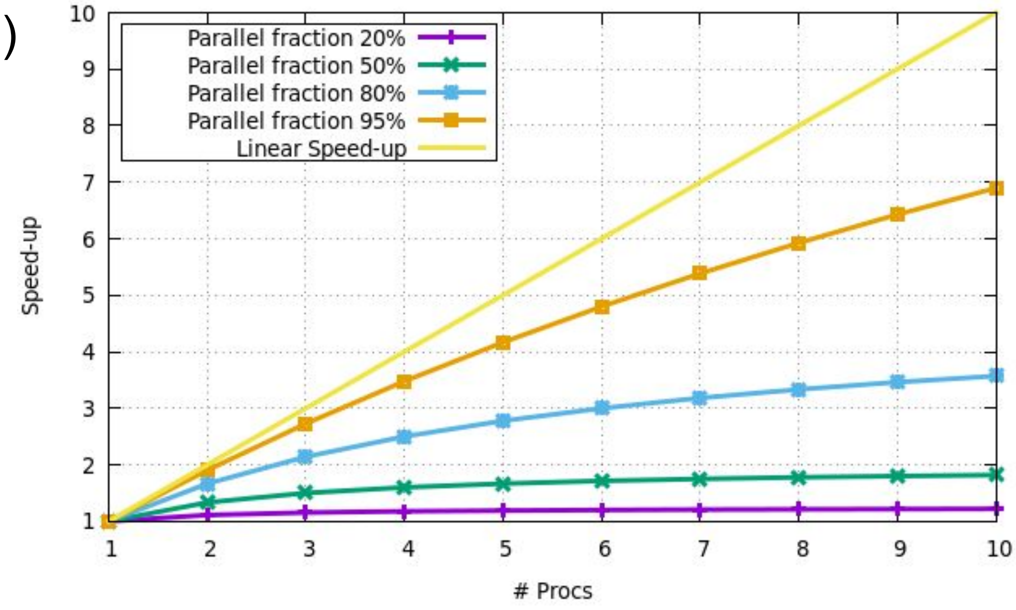
✓ …. different behaviour



Amdhal's law (strong scaling)

- ✓ **Fixed problem size**
- ✓ F = parallel fraction of the code (F < 1,00)
- ✓ N = #of processors
- ✓ S = velocity increment (Speed-up)

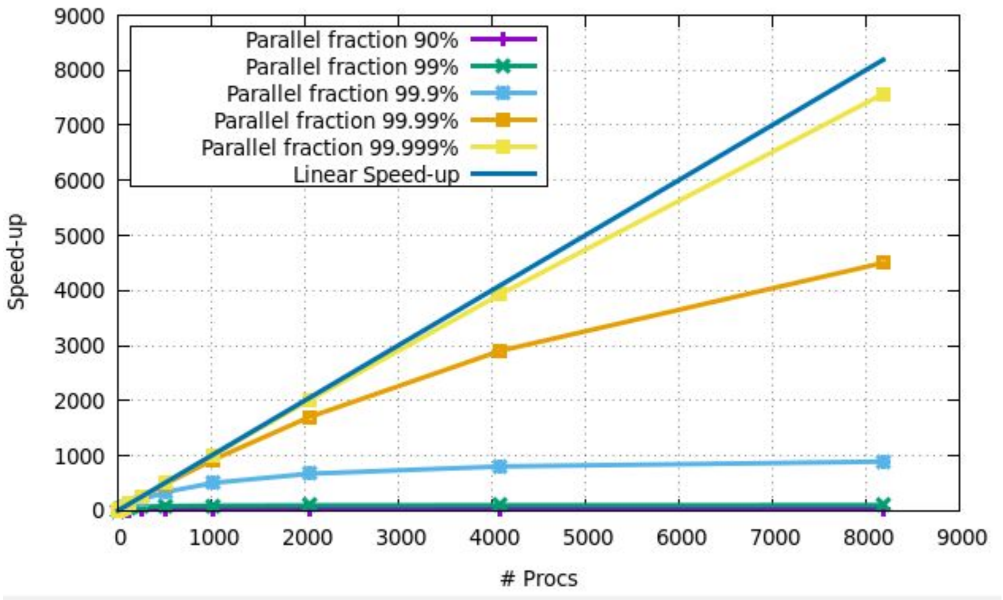
$$S = \frac{1}{(1-F) + \frac{F}{N}}$$



Amdhal's law (strong scaling)

- ✓ **Fixed problem size**
- ✓ F = parallel fraction of the code (F < 1,00)
- ✓ N = #of processors
- ✓ S = velocity increment (Speed-up)

$$S = \frac{1}{(1-F) + \frac{F}{N}}$$

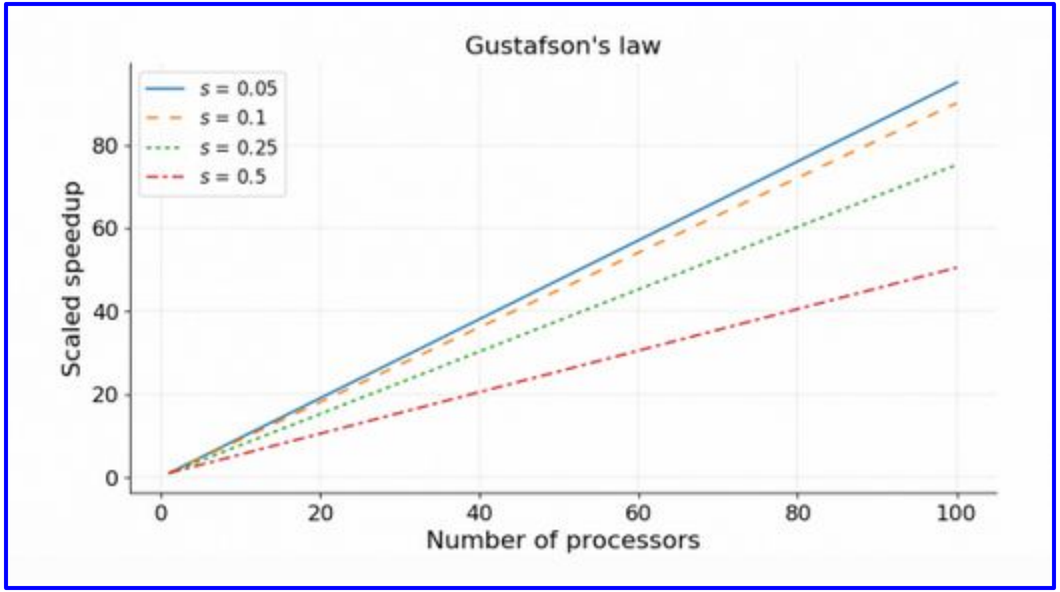


- ✓ Any hint?

(Gustafsson laws): Weak scaling

- ✓ **Scaled problem size**
- ✓ S = velocity increment (Speed-up)
- ✓ s = serial part
- ✓ p = parallel part
- ✓ N = #of processors

$$S = s + p*n$$



Different Level of optimization

- ✓ **Single core optimization**
  - Vectorization
  - Data access
  - Serial Optimized libraries
  - Compiler optimization
- ✓ **Single-node optimizations**
  - Intra-node optimization
  - Data access
  - Shared memory Parallelization/Distributed memory Parallelization
  - Offloading
  - Shared Memory Optimized Libraries
- ✓ **Multi-node optimizations**
  - Distributed memory optimization (i.e. load balancing)
  - Parallel Optimized libraries
- ✓ **I/O issues**

You cannot skip one single level of optimization!!

Optimized Libraries

- ✓ There are many optimized libraries
  - Serial: BLAS, LAPACK,...
  - Parallel: PETSc, Trillinos, SCALAPACK, FFTW…
  - Proprietary: ESSL, MKL, ….
- ✓ Usually could be robust or flexible or fast (but few times all together)
- ✓ Sometimes can help to perform a part of you problem more hard to solve all your problem
- ✓ library maintenance could be a problem, for the open source one

- ✓ **HINT**: do not reinvent the wheel! Look around if you find something that’s fine for you (no one needs to write a parallel FFT from scratch)