

# Introduction to Parallel COMPUTING

Luca Tornatore - I.N.A.F.



**“Foundation of HPC - basic” course**



DATA SCIENCE &  
SCIENTIFIC COMPUTING  
2022-2023 @ Università di Trieste



“CRUCIAL PROBLEMS that we can only hope to address computationally REQUIRE US TO DELIVER **EFFECTIVE COMPUTING POWER ORDERS-OF-MAGNITUDE GREATER THAN WE CAN DEPLOY TODAY.**”

DOE’s Office of Science, 2012

..even more valid nowadays, though



# Outline



Intro to Parallel  
Computing      Parallel  
Performance



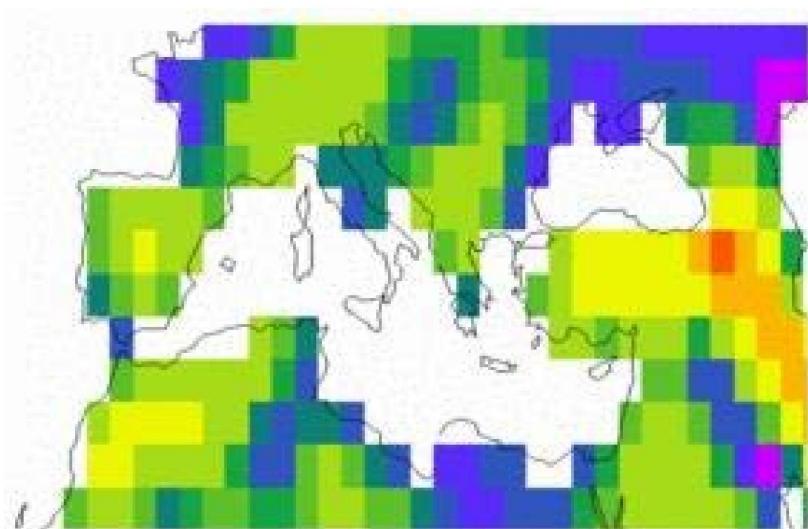
# What is High Performance Computing

High performance computing (HPC), also known as supercomputing, refers to computing systems with extremely high computational power that are able to solve hugely complex and demanding problems.

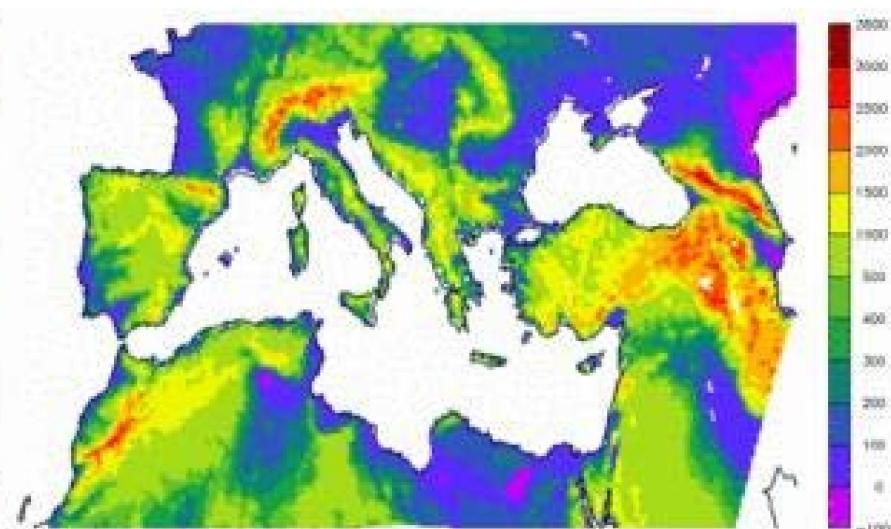
Taken from <https://ec.europa.eu/digital-single-market/en/high-performance-computing>

# Why High Performance Computing ?

Example of a complex problem: climate change over the Mediterranean Sea.  
How the required memory increases if the resolution passes from 200 km (~1GB) to 2km ?



200 km



25 km

# Why High Performance Computing ?

Complex problems solved by simulations

- Simulation has become the way to research and develop new scientific and engineering solutions.
- Used nowadays in leading science domains like aerospace industry, astrophysics, material science, etc.
- Challenges related to the complexity, scalability and data production of the simulators arise.
- Impact on the relaying IT infrastructure.



# Why High Performance Computing ?

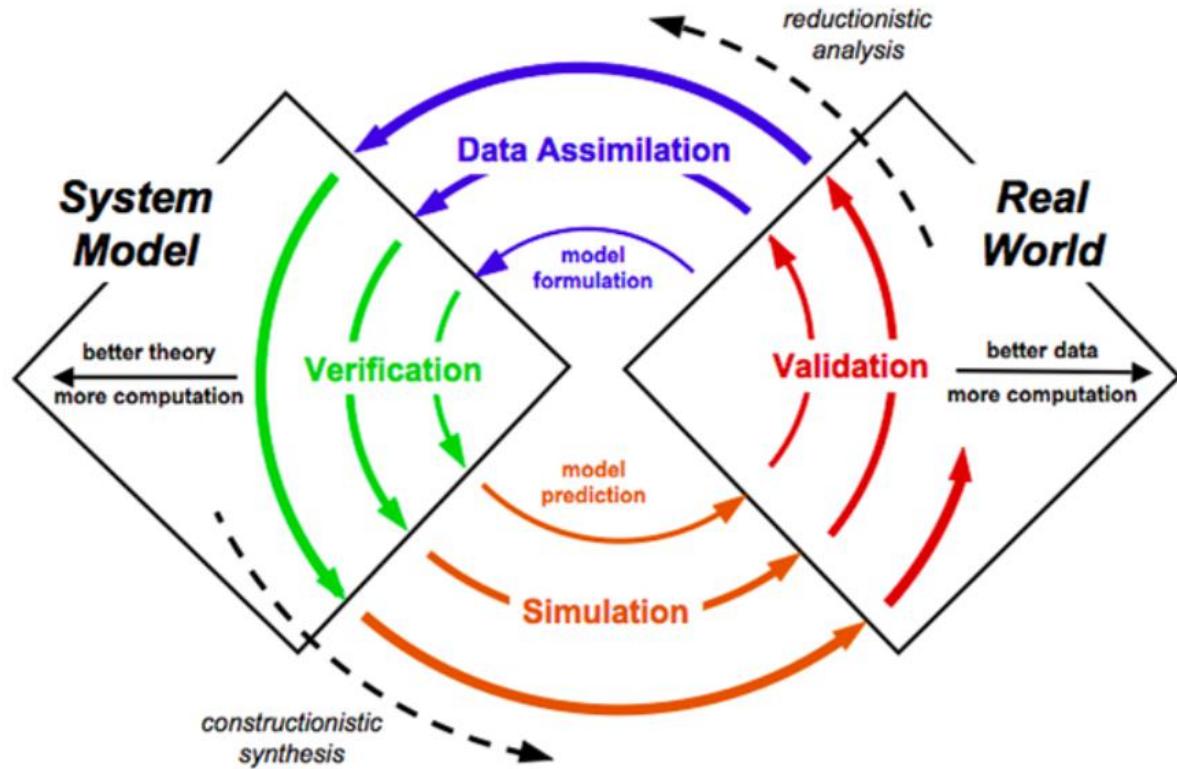
Complex problems solved by simulations

- Simulation has become the way to research and develop new scientific and engineering solutions.
- Used nowadays in leading science domains like aerospace industry, astrophysics, material science, etc.
- Challenges related to the complexity, scalability and data production of the simulators arise.
- Impact on the relaying IT infrastructure.

HPC capabilities are used to solve and address scientific, industrial and societal challenges.

<p> <b>Health</b></p> <ul style="list-style-type: none"><li>• Development of personalised and precision medicine to provide individual and accurate patient treatment.</li><li>• Saving time and money on the development of new drugs from the initial idea phase to the final phase of reaching the market. </li><li>• Early detection of diseases and quicker diagnosis.</li></ul>	<p> <b>Climate change &amp; weather forecast</b></p> <ul style="list-style-type: none"><li>• Europe paid severe weather damage costs between 1970 and 2012<sup>1</sup>.</li><li>• With HPC technology, climate scientists will be able to predict the size and paths of storms and floods with accuracy.</li><li>• This will allow implementation of measures such as alerting or evacuating people, thus saving human lives.</li></ul> <p> <b>Industry</b></p> <ul style="list-style-type: none"><li>• Reducing development time, minimising costs, optimising decision processes and producing higher quality goods and services.</li><li>• E.g. automotive industry will save time and money to develop new vehicle platforms, with improved environmental friendliness and passenger comfort and safety<sup>2</sup>.</li></ul>	<p> <b>Cybersecurity</b></p> <ul style="list-style-type: none"><li>• Enabling complex encryption technologies and better reactions to cyberattacks.</li><li>• Combined with Artificial Intelligence, HPC detects:<ul style="list-style-type: none"><li>◦ strange systems behaviour,</li><li>◦ insider threats and electronic fraud,</li><li>◦ very early cyberattack patterns (few hours instead of a few days)</li></ul>This will allow automated and immediate actions even before a cyberattack happens.</li></ul>	<p> <b>Energy</b></p> <ul style="list-style-type: none"><li>• HPC provides critical tools for example in:<ul style="list-style-type: none"><li>◦ designing renewable energy parks</li><li>◦ designing high-performance photovoltaic materials,</li><li>◦ optimising turbines for electricity production.</li></ul></li><li>• HPC expenditure in the energy sector is projected to grow by 5% in the next years.</li></ul>
--	--	--	--

# Why High Performance Computing ?



## The inference spiral

As models become more complex and increasingly sophisticated experiment experiments and new data bring in more information, an ever increasing computational power is required.

## A flood of data

In today's world, larger and larger amounts of data are constantly being generated, from ~30 zettabytes globally in 2018 to an expected ~200 zettabytes in 2025).

As a result, the nature of computing is changing, with an increasing number of data-intensive critical applications.

Is key to processing and analysing this growing volume of data, and to making the most of it for the benefit of citizens, businesses, researchers and public administrations.

Taken from <https://ec.europa.eu/digital-single-market/en/high-performance-computing>

Name	Symbol	Factor
Yotta	Y	1024
Zetta	Z	1021
Exa	E	1018
Peta	P	1015
Tera	T	1012
Giga	G	109
Mega	M	106
Kilo	K	103

## Refining the definition

High Performance Computing (HPC) is the use of servers, clusters, and supercomputers – plus associated software, tools, components, storage, and services – for scientific, engineering, or analytical tasks that are particularly intensive in computation, memory usage, or data management.

HPC is used by scientists and engineers both in research and in production across industry, government and academia.

# What is High Performance Computing

## Refining the definition

High Performance Computing (HPC) is the use of servers, clusters, and supercomputers – plus associated software, tools, components, storage, and services – for scientific, engineering, or analytical tasks that are particularly intensive in computation, memory usage, or data management.

HPC is used by scientists and engineers both in research and in production across industry, government and academia.

## The HPC ecosystem

### 1. HARDWARE LEVEL

servers, cluster, supercomputers, performant I/O and networks

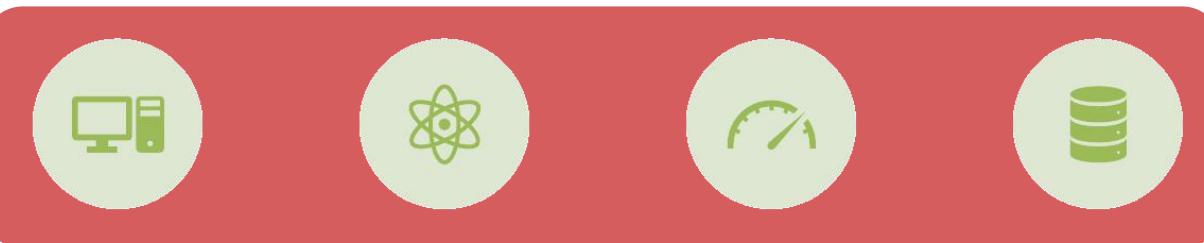
### 2. SOFTWARE LEVEL

Software, tools, programming paradigms

### 3. PROBLEMS to be solved, and theoretically sounded basis

scientific, engineering, analytical tasks

# What is High Performance Computing

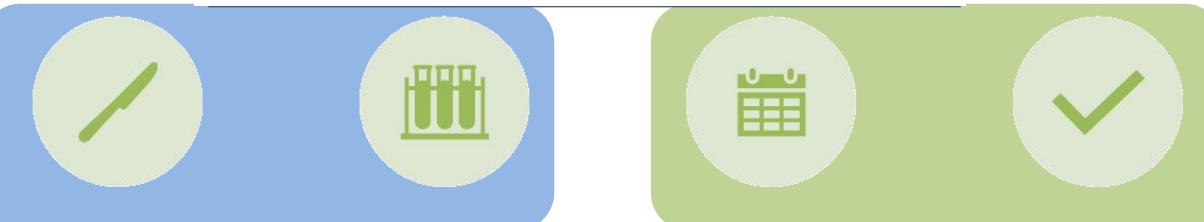


COMPUTATIONAL SERVERS

ACCELERATORS

HIGH SPEED NETWORKS

HIGH END PARALLEL STORAGE



MIDDLEWARE

SCIENTIFIC/TECHNICAL/  
DATA ANALYSIS SOFTWARE

RESEARCH/TECHNICAL  
DATA

PROBLEMS TO BE  
SOLVED

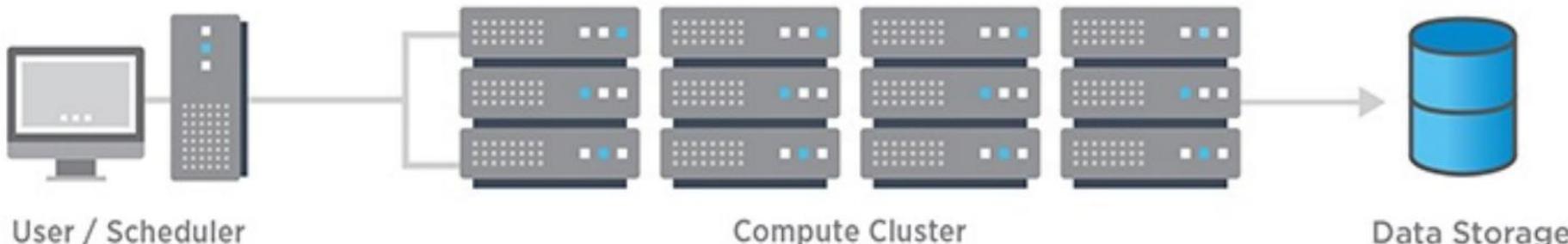
The HPC ecosystem

- HARDWARE LEVEL**  
servers, cluster, supercomputers, performant I/O and networks
- SOFTWARE LEVEL**  
Software, tools, programming paradigms
- PROBLEMS to be solved, and theoretically sounded basis**  
scientific, engineering, analytical tasks

# The hardware behind HPC

The essential elements of a supercomputer are the following:

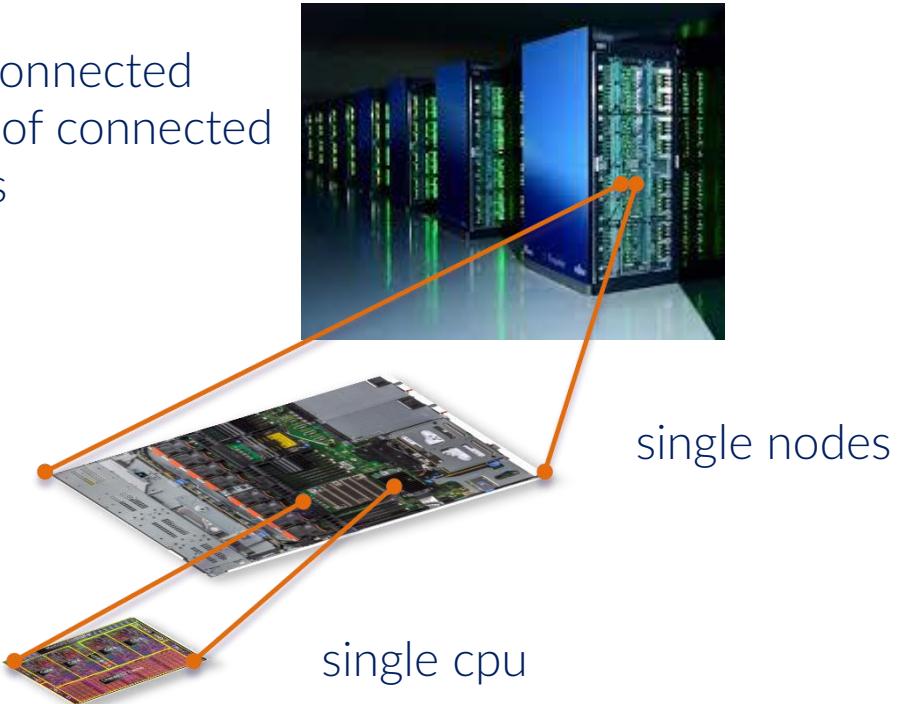
- a (large) number of computational “nodes” (i.e. “objects” that have processors, ram, accelerators, network ports, possibly non-volatile memories for fast storage)
- one or more switch-based networks that interconnect the nodes together; there is large variety of topologies for the networks;
- storage: usually you have 3 storage areas: home, a huge long-term resident storage (usually RAID-based) and a fast parallel file-system for production
- some login nodes to connect
- some parallel scheduler

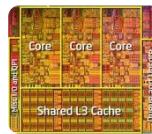


# The hardware behind HPC

Let's now inspect in some detail the complex architecture that is behind the login node of a supercomputer, starting from the most basic element, the single CPU, whose elemental component, "the core", we analyzed in detail in the single-core part of the course.

interconnected  
racks of connected  
nodes





Race to  
Multicore

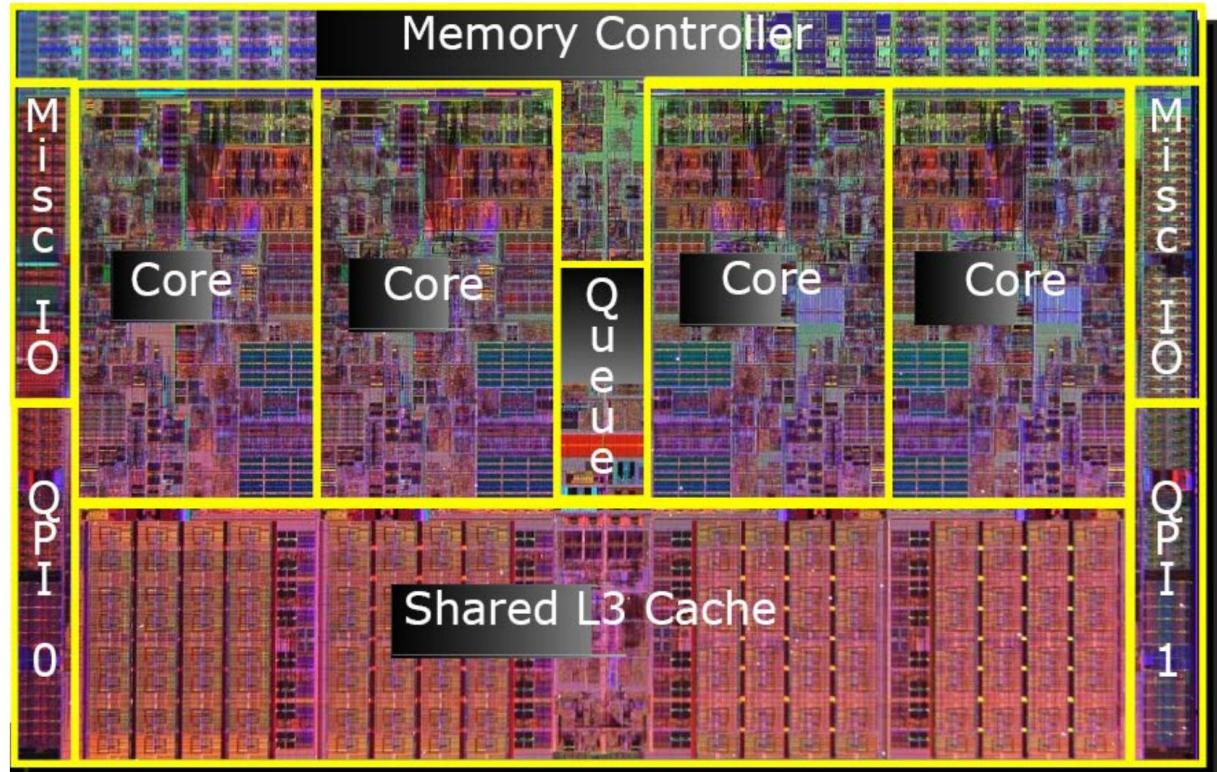
hardware  
level

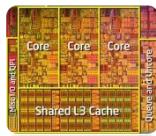
# Single-CPU topology

Introduction



Modern  
CPUs  
are multi-  
(or many-)  
cores





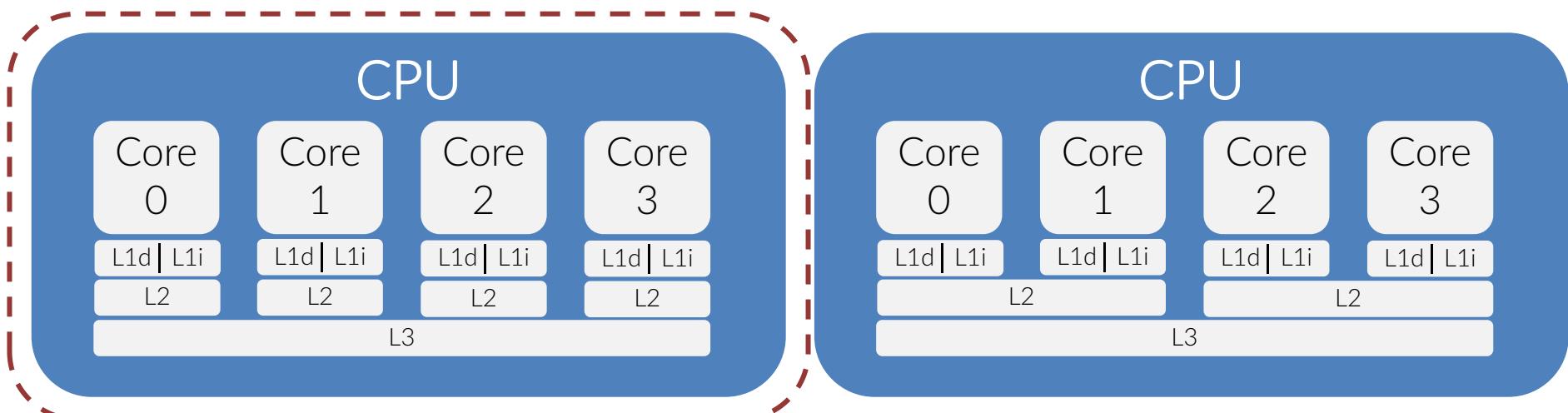
Race to  
Multicore

hardware  
level

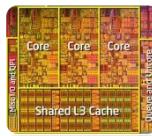
# Single-CPU topology



Cache hierarchy can have different topologies



the standard one, L3 shared among all the cores



Race to Multicore

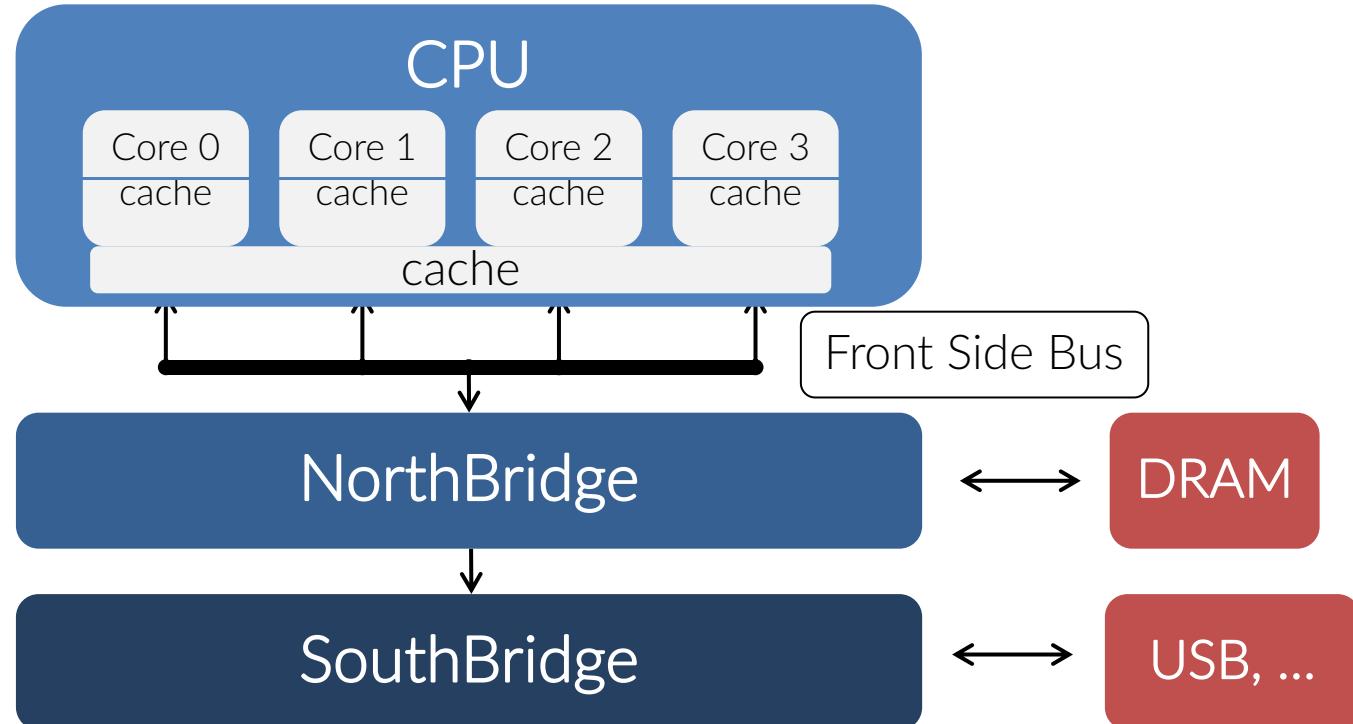
hardware level

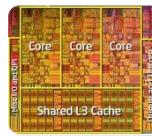


# Single-CPU topology

In old times, this was the typical way in which a cpu connected to the outer world.

The FSB managed all communications, quickly becoming a bottleneck when an increasing nr. of cores shared its bandwidth.



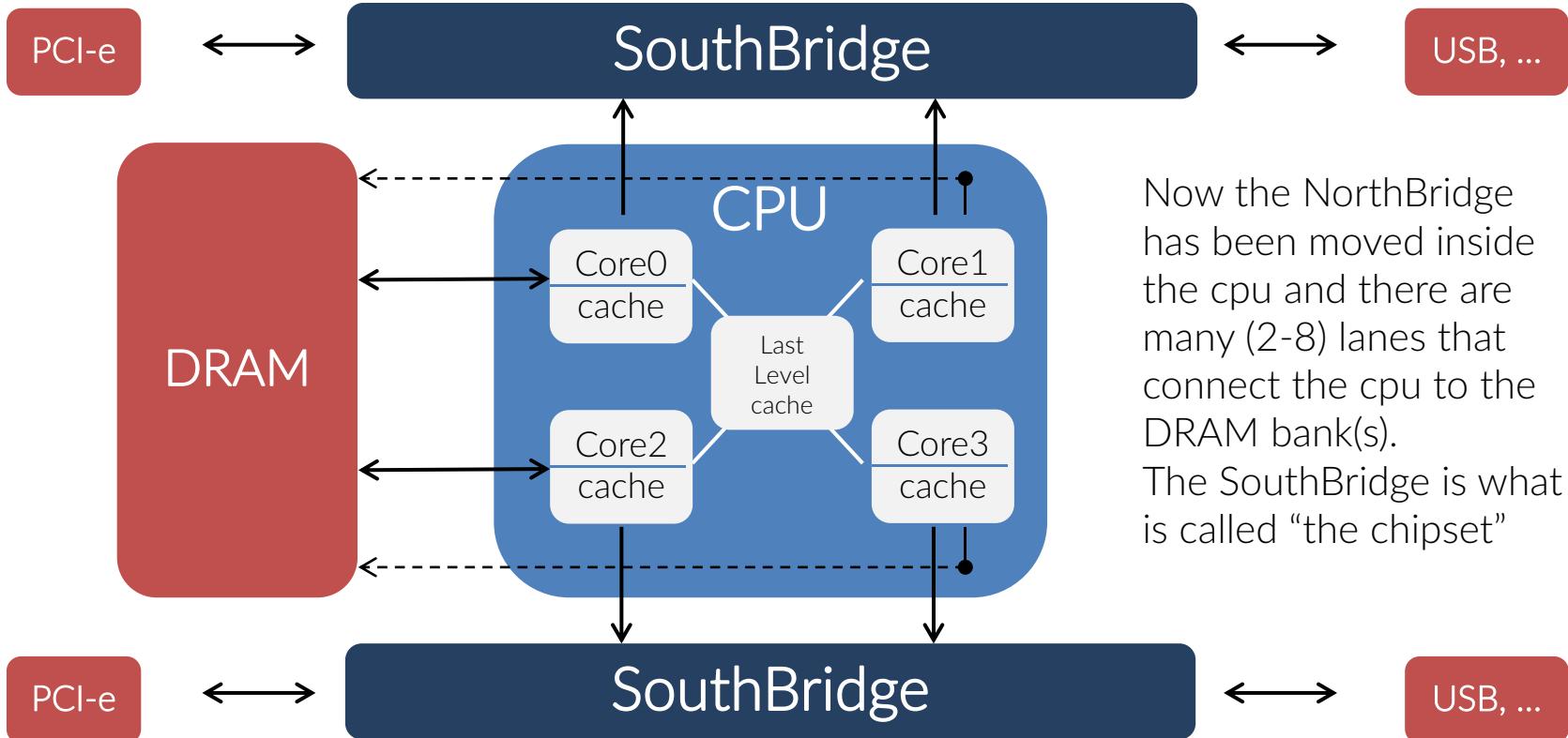


Race to  
Multicore

hardware  
level



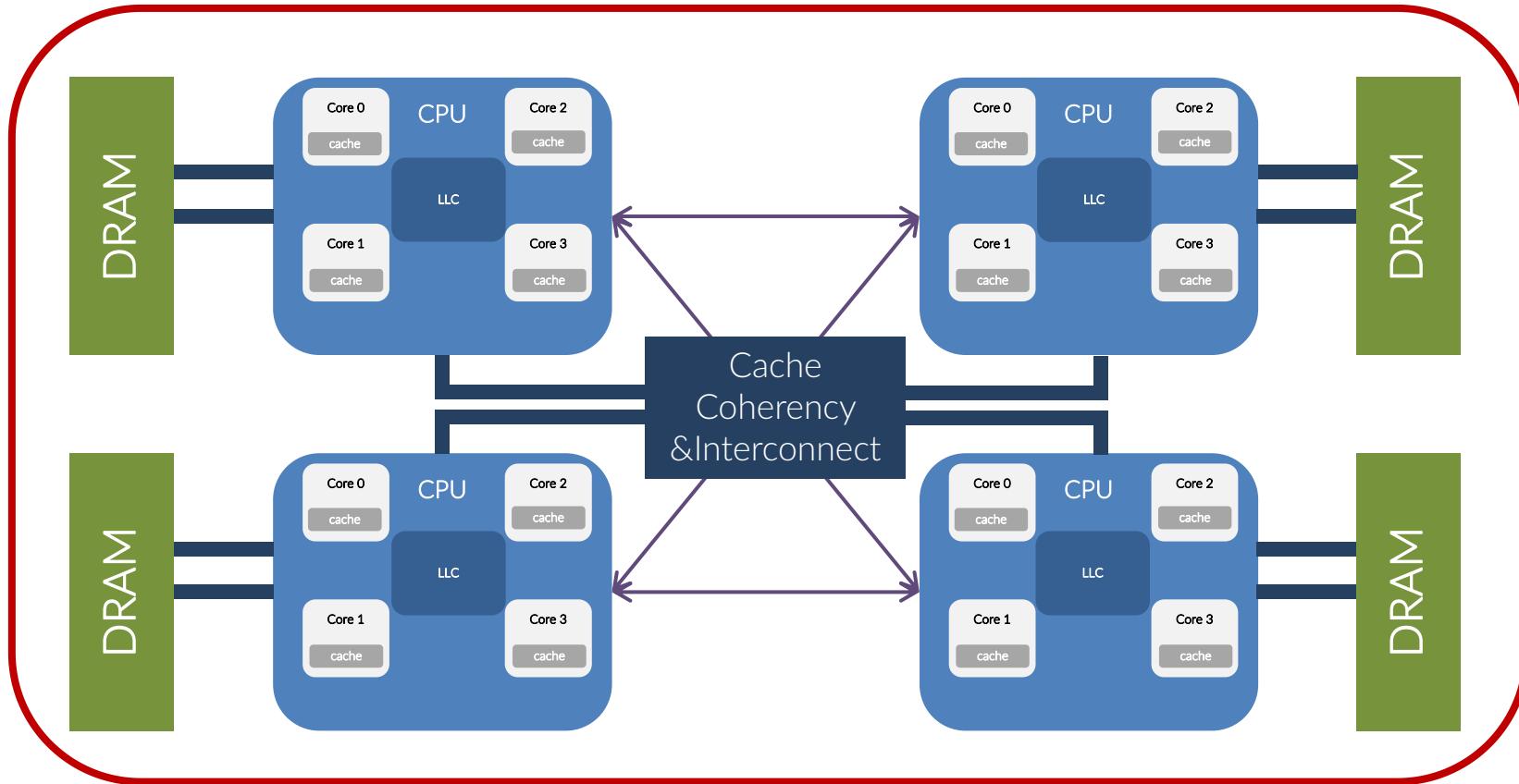
# Single-CPU topology





# The NODE topology

**COMPUTING NODE**  
there may be more or less sockets  
(usually an even number and  $\geq 2$ )

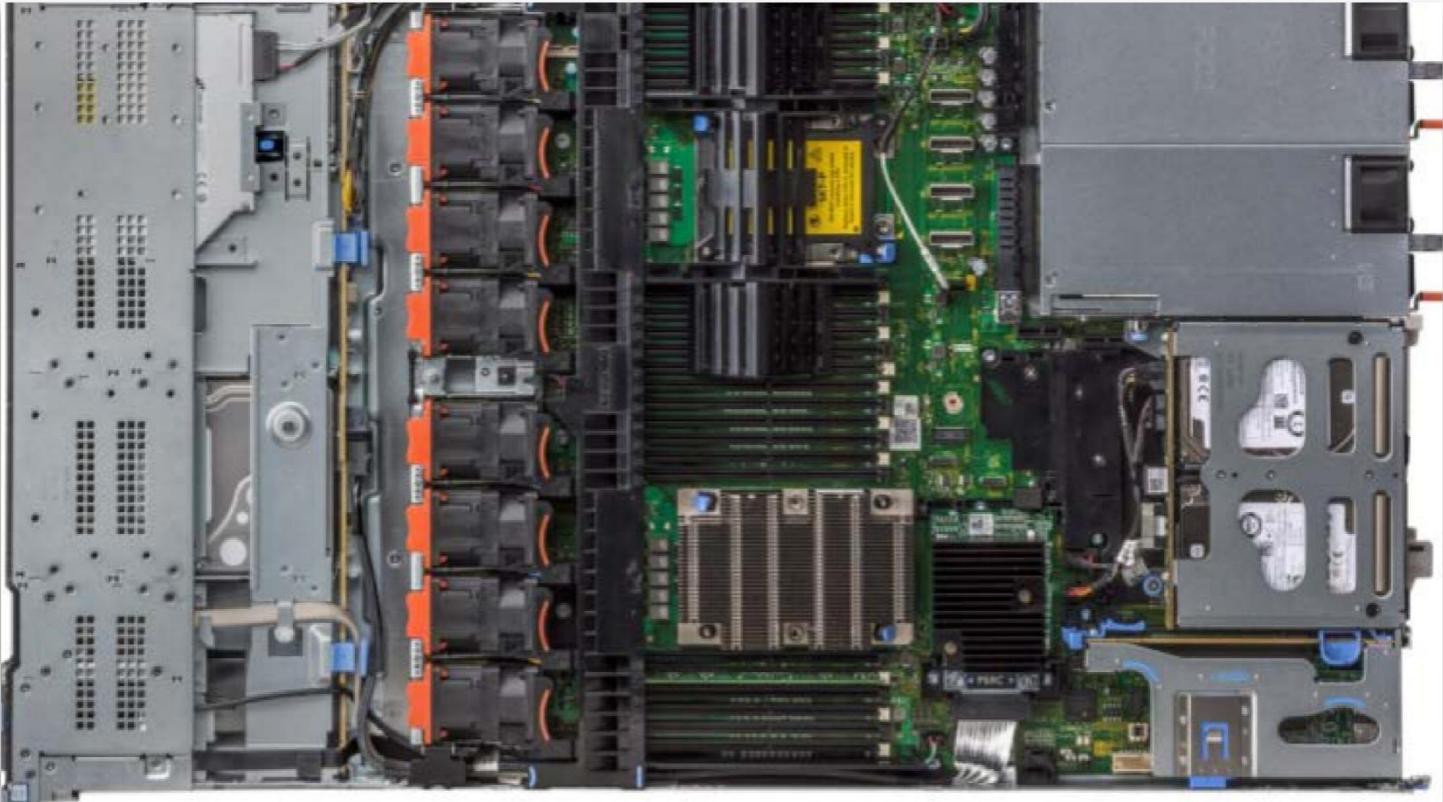




# The NODE topology

## COMPUTING NODE

there may be more or less sockets  
(usually an even number and  $\geq 1$ )

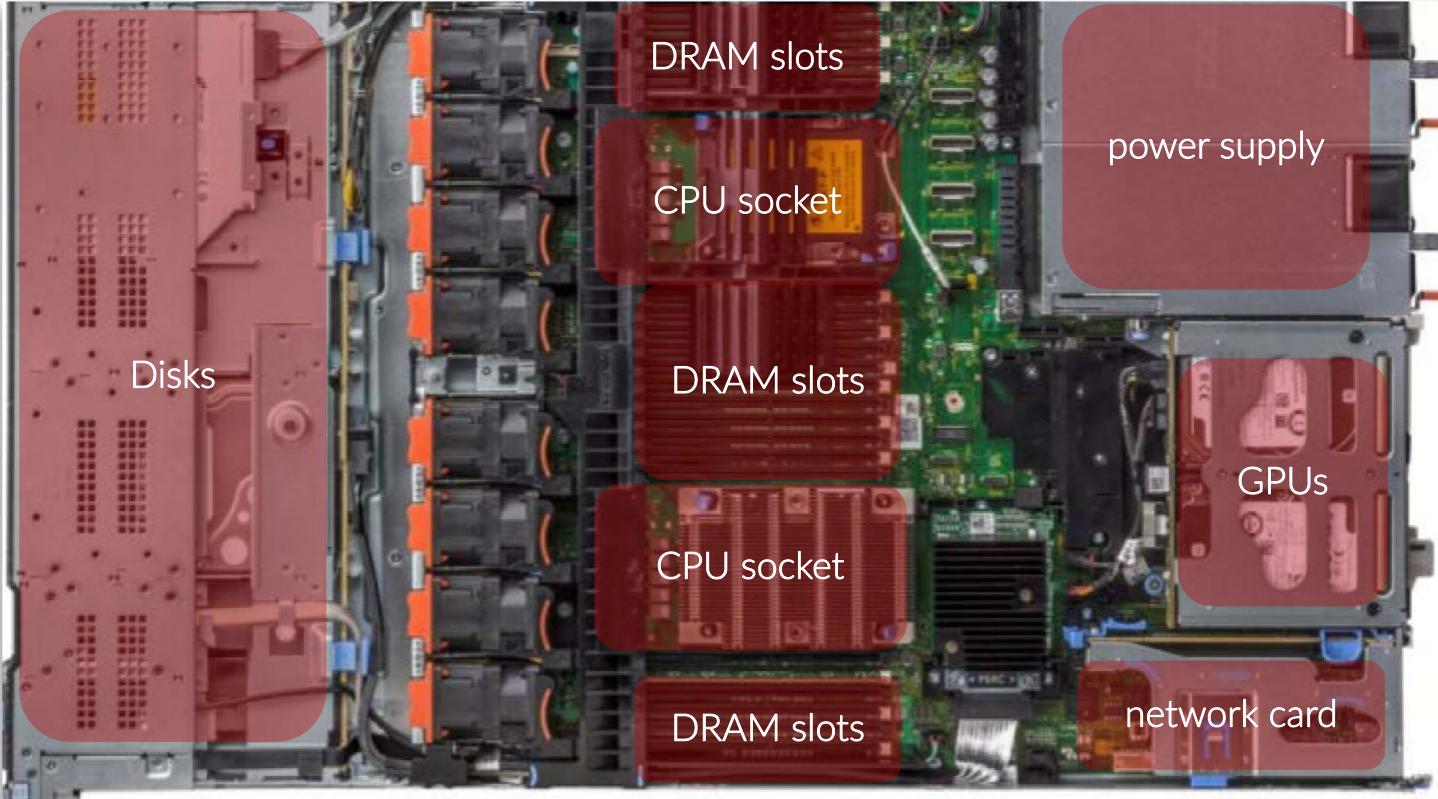




# The NODE topology

## COMPUTING NODE

there may be more or less sockets  
(usually an even number and  $\geq 1$ )

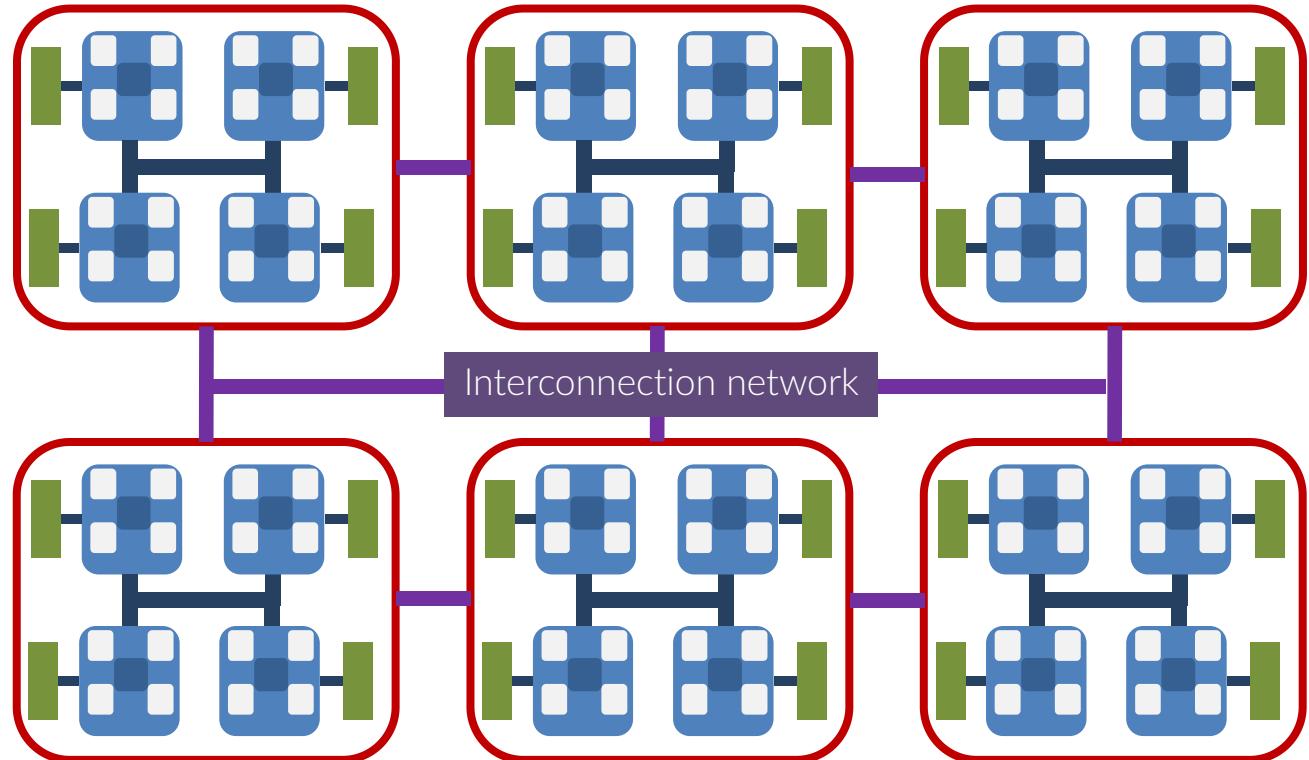




# The overall topology

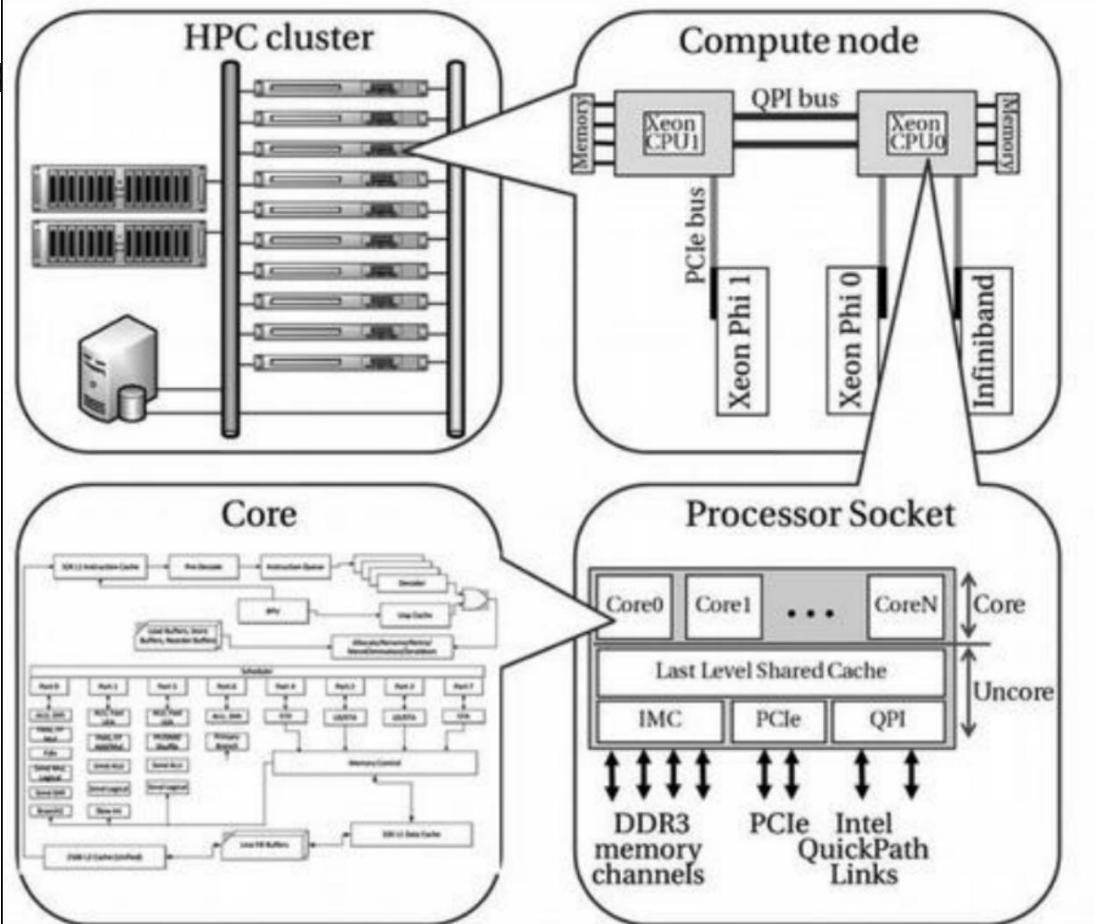
## CLUSTER OF COMPUTING NODES

Note: there are many different topologies for the interconnection network.





# The overall topology



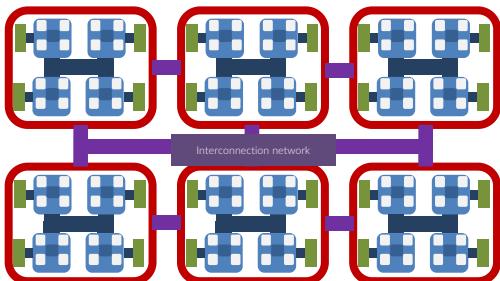
Many ( $\sim 10\text{-}10^5$ ) nodes are connected by a switch-based network, whose topology may vary a lot. The details of that may severely affect the overall performance.

Typical figures for latency and bandwidth are  $\sim 1\mu\text{s}$  and  $\sim 100\text{Gbit/s}$  respectively.

The most common standard are InfiniBand and OmniPath.

# The overall topology

Note that on a supercomputer there is a hybrid approach as for the memory placement:



- the memory on a single nodes can be accessed directly by all the cores on that node, meaning that memory access is a “read/write” instructions irrespectively of what exact memory bank it refers to. This is called **shared-memory**.
- when you use many nodes at a time, a process can not directly access the memory on a different node. It need to issue a request for that, not a read/write instruction. That is named **distributed memory**.

These are *hardware concepts*, i.e. they describe how the memory is physically accessible. However, they do also refer to *programming paradigms*, as we'll see in a while.

# Shared vs Distributed Memory

Then, in a multiprocessor system (i.e. a one having multiple cpus)

When some cpus can directly access the same DRAM memory, they are in **shared-memory**:

- the processes running on those cpu share a unique physical address space that is mapped on memory physically distributed on the memory banks
- read and write are simple memory accesses, and the process communicate using the shared memory

When some cpus can not access each other's DRAM memory because they are on different nodes, they are in **distributed-memory**

- the physical address space is separated and private to the processes that run on a given cpu
- the processes running on different cpu communicate through *messages* that travel on a linking network.

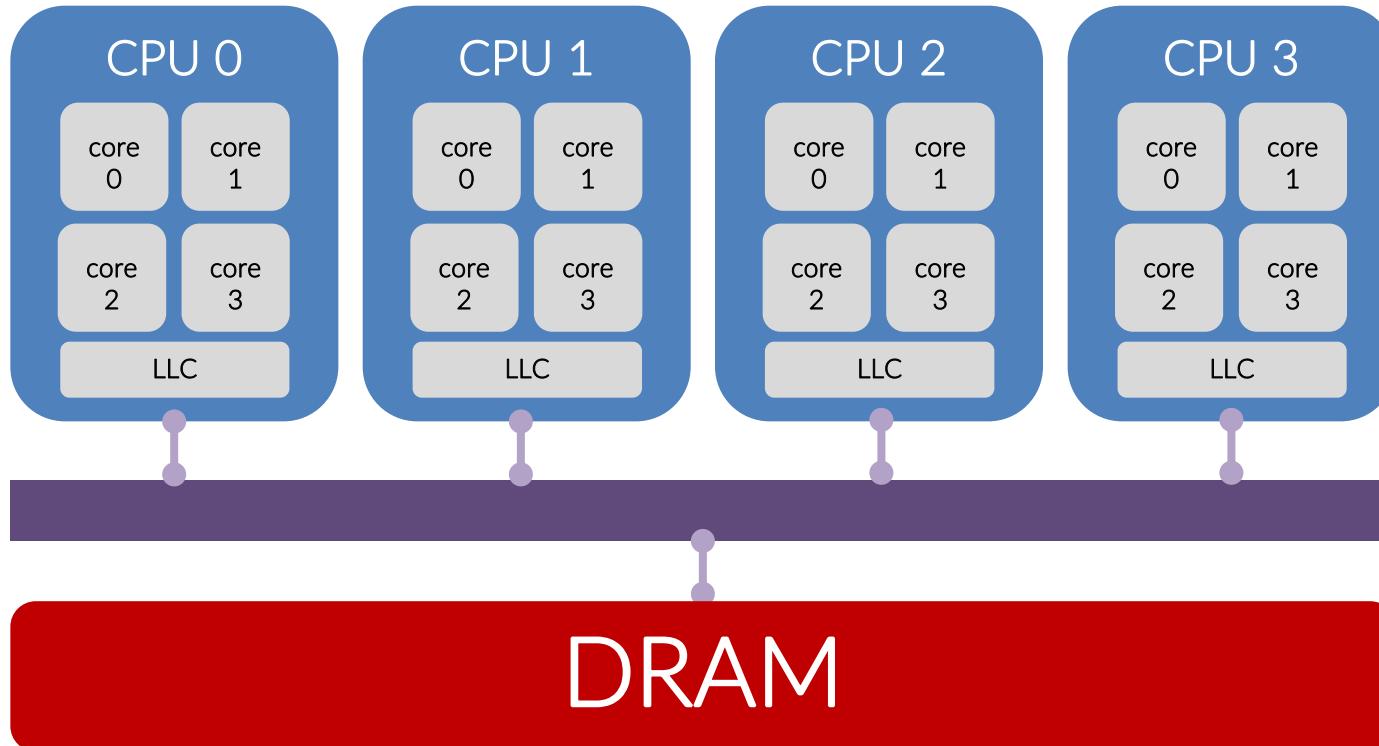
# Shared Memory

In turn, there are two fundamental types of shared-memory:

- if the access to every RAM location is equally costly for all cpus, the system is **UMA, Uniform Memory Access**
- if the cost of RAM memory access depends on the location, then the system is said to be **NUMA, Non-Uniform Memory Access**

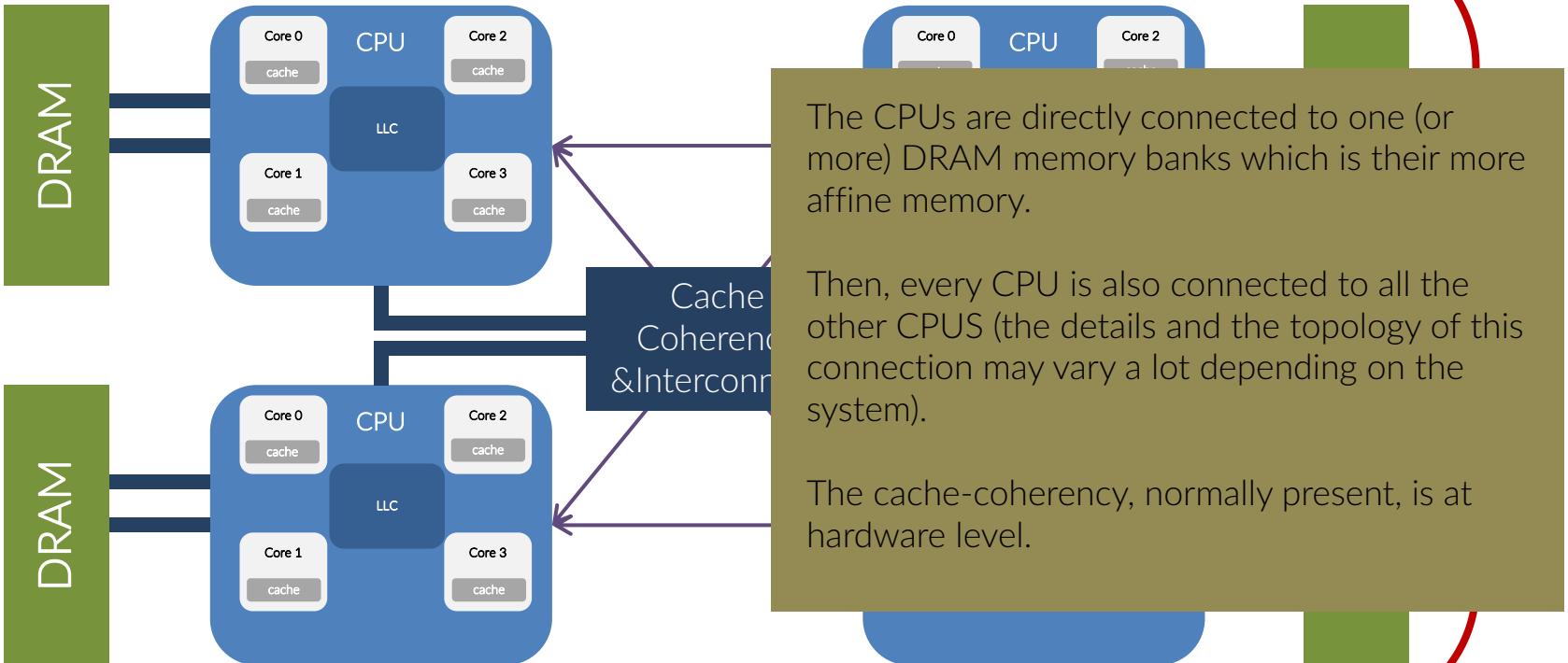


# Shared Memory : UMA



In Uniform Memory Access systems, all the cpus are connected to the DRAM banks through a common connection, and for each CPU the cost for accessing any DRAM memory location is independent on the memory location itself.

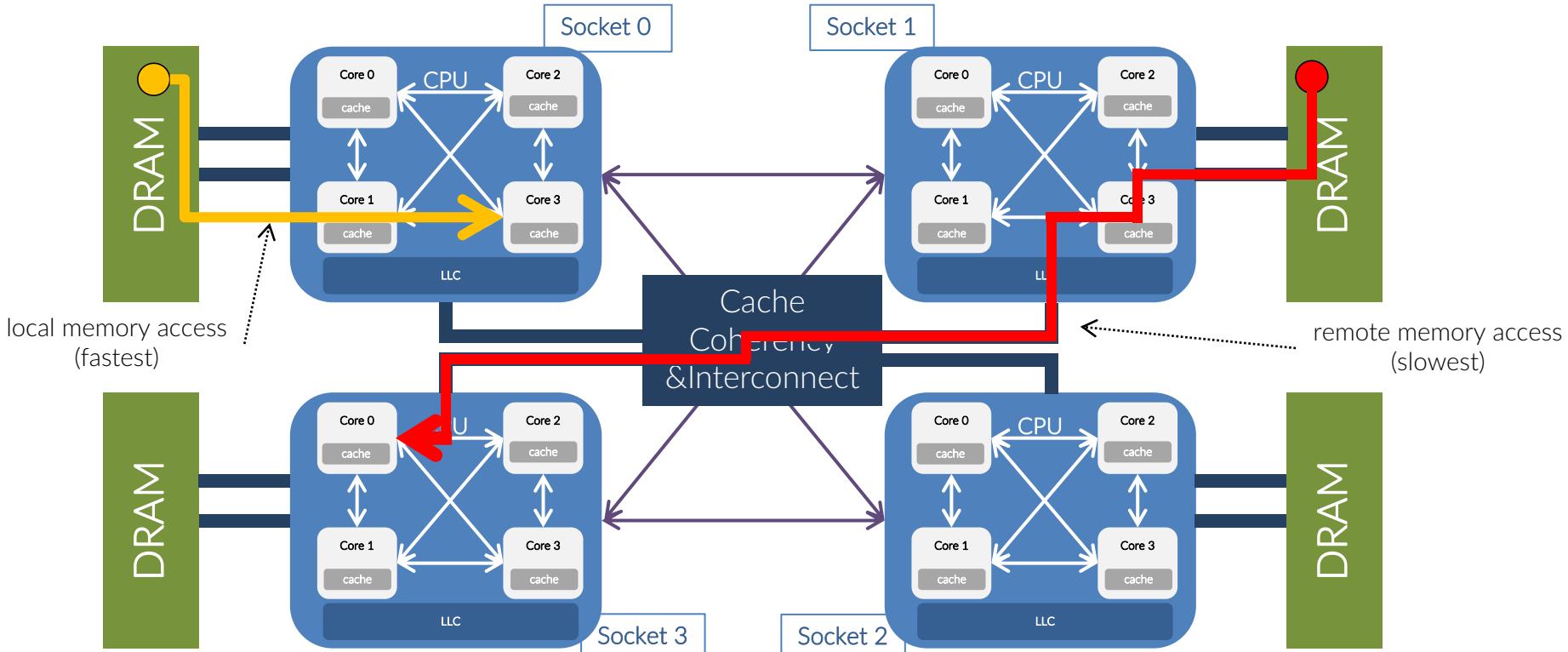
# Shared Memory : NUMA





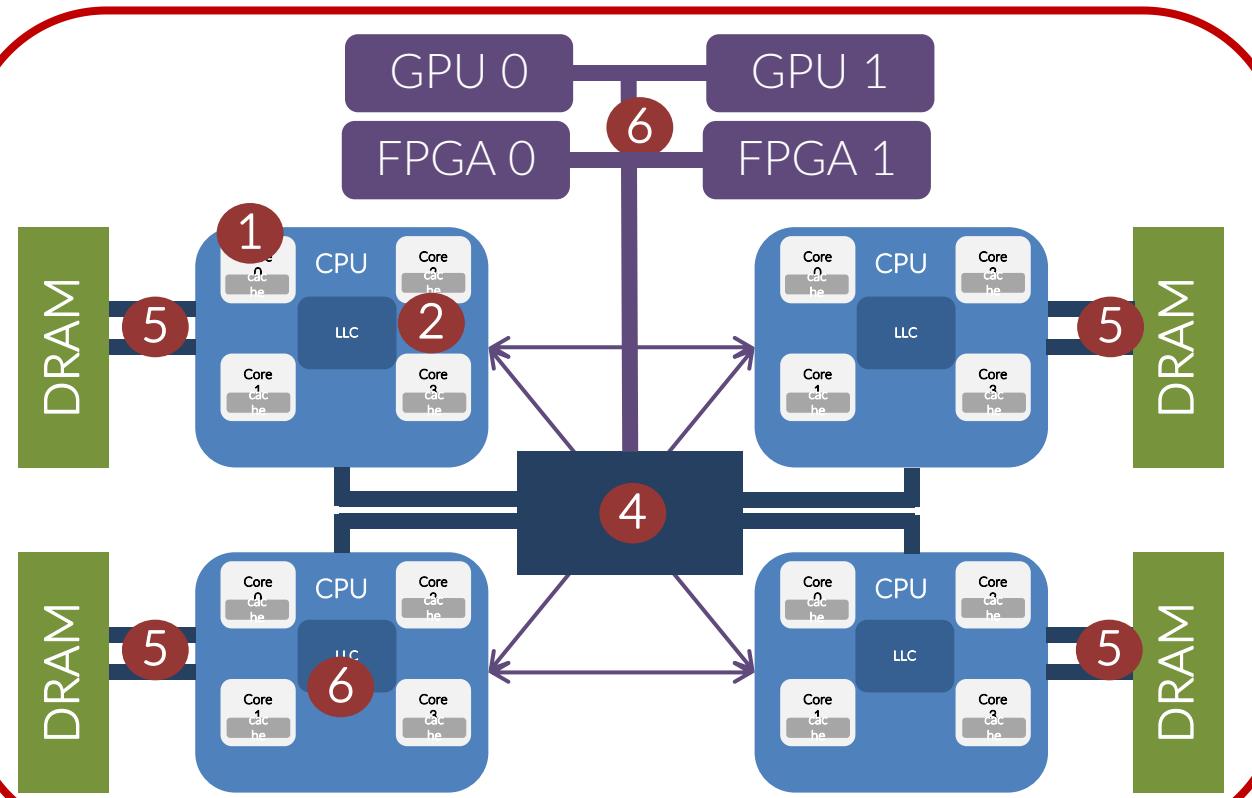
# Shared Memory : NUMA

Zoom-out to a multi-socket node (all the RAM is accessible from every core, i.e. it is shared)





# Summary: node-level parallelism



1. ILP/SIMD at core level
2. many cores per each socket
3. last-level cache per socket
4. inter-socket links
5. multiple memory links
6. heterogeneous accelerators

# Shared Memory : NUMA



Two examples, both of nodes with 4 sockets each

```
[ltornatore@hp10 ~]$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                40
On-line CPU(s) list:  0-39
Thread(s) per core:   1
Core(s) per socket:   10
Socket(s):             4
NUMA node(s):          4
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 63
Model name:            Intel(R) Xeon(R) CPU E5-4627 v3 @ 2.60GHz
Stepping:               2
CPU MHz:               1200.000
CPU max MHz:           2600.000
CPU min MHz:           1200.000
BogoMIPS:              5194.05
Virtualization:        VT-x
L1d cache:              32K
L1i cache:              32K
L2 cache:               256K
L3 cache:               25600K
NUMA node0 CPU(s):     0-4,20-24
NUMA node1 CPU(s):     5-9,25-29
NUMA node2 CPU(s):     10-14,30-34
NUMA node3 CPU(s):     15-19,35-39
```

hyperthreading off

node	0	1	2	3
0:	10	21	21	21
1:	21	10	21	21
2:	21	21	10	21
3:	21	21	21	10

non-uniform access to memory

```
[ltornatore@gen10-01 ~]$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                96
On-line CPU(s) list:  0-95
Thread(s) per core:   2
Core(s) per socket:   12
Socket(s):             4
NUMA node(s):          4
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 85
Model name:            Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz
Stepping:               4
CPU MHz:               2663.305
CPU max MHz:           3200.0000
CPU min MHz:           1000.0000
BogoMIPS:              4600.00
Virtualization:        VT-x
L1d cache:              32K
L1i cache:              32K
L2 cache:               1024K
L3 cache:               16896K
NUMA node0 CPU(s):     0-11,48-59
NUMA node1 CPU(s):     12-23,60-71
NUMA node2 CPU(s):     24-35,72-83
NUMA node3 CPU(s):     36-47,84-95
```

hyperthreading on

node	0	1	2	3
0:	10	21	21	21
1:	21	10	21	21
2:	21	21	10	21
3:	21	21	21	10



# Discovering a NUMA topology

How to discover the topology of your node:

- **numactl** tool  
it also controls the Linux NUMA policy
- **cpuinfo** tool (by Intel)
- **hwloc** (by OpenMPI)

# Shared Memory : NUMA

How to discover the topology of your node (examples):

- **numactl** tool

**numactl -H**

- **cpuinfo** tool (by Intel)

- **hwloc** (by OpenMPI)

```
[ltornatore@hp08 ~]$ numactl -H
available: 4 nodes (0-3)
node 0 cpus: 0 1 2 3 4 20 21 22 23 24
node 0 size: 65411 MB
node 0 free: 40998 MB
node 1 cpus: 5 6 7 8 9 25 26 27 28 29
node 1 size: 65536 MB
node 1 free: 58475 MB
node 2 cpus: 10 11 12 13 14 30 31 32 33 34
node 2 size: 65536 MB
node 2 free: 59344 MB
node 3 cpus: 15 16 17 18 19 35 36 37 38 39
node 3 size: 65536 MB
node 3 free: 59641 MB
node distances:
node   0   1   2   3
 0: 10 21 21 21
 1: 21 10 21 21
 2: 21 21 10 21
 3: 21 21 21 10
```

# Shared Memory : NUMA

How to discover the topology of your node (examples):

- **numactl** tool

- **cpuinfo** tool (by Intel)
- **hwloc** (by OpenMPI)

```
cpuinfo -d
=====
Placement on packages
=====
Package Id.    Core Id.      Processors
0            0,2,4,9,11,1,3,8,10,12      0,1,2,3,4,20,21,22,23,24
1            0,2,4,9,11,1,3,8,10,12      5,6,7,8,9,25,26,27,28,29
2            0,2,4,9,11,1,3,8,10,12      10,11,12,13,14,30,31,32,33,34
3            0,2,4,9,11,1,3,8,10,12      15,16,17,18,19,35,36,37,38,39
```

```
cpuinfo -g
=====
Processor composition
=====
Processor name      : Intel(R) Xeon(R) E5-4627 v3
Packages(sockets) : 4
Cores             : 40
Processors(CPUs)  : 40
Cores per package : 10
Threads per core  : 1
```

```
cpuinfo -c
=====
Cache sharing
=====
Cache  Size      Processors
L1    32 KB     no sharing
L2    256 KB    no sharing
L3    25 MB     (0,1,2,3,4,20,21,22,23,24)(5,6,7,8,9,25,26,27,28,29)(10,11,12,13,14,30,31,32,33,34)(15,16,17,18,19,35,36,37,38,39)
```

# Shared Memory : NUMA

## How to discover the topology of

- **numactl** tool
- **cpuinfo** tool (by Intel)
- **hwloc-ls** (by OpenMPI)  
**hwloc-info**

Machine (256GB total)  
NUMANode L#0 (#P#0 64GB)

Package L#0 + L3 L#0 (25MB)  
 L2 L#0 (256KB) + L1d L#0 (32KB) + L1i L#0 (32KB) + Core L#0 + PU L#0 (P#0)  
 L2 L#1 (256KB) + L1d L#1 (32KB) + L1i L#1 (32KB) + Core L#1 + PU L#1 (P#1)  
 L2 L#2 (256KB) + L1d L#2 (32KB) + L1i L#2 (32KB) + Core L#2 + PU L#2 (P#2)  
 L2 L#3 (256KB) + L1d L#3 (32KB) + L1i L#3 (32KB) + Core L#3 + PU L#3 (P#3)  
 L2 L#4 (256KB) + L1d L#4 (32KB) + L1i L#4 (32KB) + Core L#4 + PU L#4 (P#4)  
 L2 L#5 (256KB) + L1d L#5 (32KB) + L1i L#5 (32KB) + Core L#5 + PU L#5 (P#20)  
 L2 L#6 (256KB) + L1d L#6 (32KB) + L1i L#6 (32KB) + Core L#6 + PU L#6 (P#21)  
 L2 L#7 (256KB) + L1d L#7 (32KB) + L1i L#7 (32KB) + Core L#7 + PU L#7 (P#22)  
 L2 L#8 (256KB) + L1d L#8 (32KB) + L1i L#8 (32KB) + Core L#8 + PU L#8 (P#23)  
 L2 L#9 (256KB) + L1d L#9 (32KB) + L1i L#9 (32KB) + Core L#9 + PU L#9 (P#24)

NUMANode L#1 (#P#1 64GB) + Package L#1 + L3 L#1 (25MB)

L2 L#10 (256KB) + L1d L#10 (32KB) + L1i L#10 (32KB) + Core L#10 + PU L#10 (P#5)  
 L2 L#11 (256KB) + L1d L#11 (32KB) + L1i L#11 (32KB) + Core L#11 + PU L#11 (P#6)  
 L2 L#12 (256KB) + L1d L#12 (32KB) + L1i L#12 (32KB) + Core L#12 + PU L#12 (P#7)  
 L2 L#13 (256KB) + L1d L#13 (32KB) + L1i L#13 (32KB) + Core L#13 + PU L#13 (P#8)  
 L2 L#14 (256KB) + L1d L#14 (32KB) + L1i L#14 (32KB) + Core L#14 + PU L#14 (P#9)  
 L2 L#15 (256KB) + L1d L#15 (32KB) + L1i L#15 (32KB) + Core L#15 + PU L#15 (P#25)  
 L2 L#16 (256KB) + L1d L#16 (32KB) + L1i L#16 (32KB) + Core L#16 + PU L#16 (P#26)  
 L2 L#17 (256KB) + L1d L#17 (32KB) + L1i L#17 (32KB) + Core L#17 + PU L#17 (P#27)  
 L2 L#18 (256KB) + L1d L#18 (32KB) + L1i L#18 (32KB) + Core L#18 + PU L#18 (P#28)  
 L2 L#19 (256KB) + L1d L#19 (32KB) + L1i L#19 (32KB) + Core L#19 + PU L#19 (P#29)

NUMANode L#2 (#P#2 64GB) + Package L#2 + L3 L#2 (25MB)

L2 L#20 (256KB) + L1d L#20 (32KB) + L1i L#20 (32KB) + Core L#20 + PU L#20 (P#10)  
 L2 L#21 (256KB) + L1d L#21 (32KB) + L1i L#21 (32KB) + Core L#21 + PU L#21 (P#11)  
 L2 L#22 (256KB) + L1d L#22 (32KB) + L1i L#22 (32KB) + Core L#22 + PU L#22 (P#12)  
 L2 L#23 (256KB) + L1d L#23 (32KB) + L1i L#23 (32KB) + Core L#23 + PU L#23 (P#13)  
 L2 L#24 (256KB) + L1d L#24 (32KB) + L1i L#24 (32KB) + Core L#24 + PU L#24 (P#14)  
 L2 L#25 (256KB) + L1d L#25 (32KB) + L1i L#25 (32KB) + Core L#25 + PU L#25 (P#30)  
 L2 L#26 (256KB) + L1d L#26 (32KB) + L1i L#26 (32KB) + Core L#26 + PU L#26 (P#31)  
 L2 L#27 (256KB) + L1d L#27 (32KB) + L1i L#27 (32KB) + Core L#27 + PU L#27 (P#32)  
 L2 L#28 (256KB) + L1d L#28 (32KB) + L1i L#28 (32KB) + Core L#28 + PU L#28 (P#33)  
 L2 L#29 (256KB) + L1d L#29 (32KB) + L1i L#29 (32KB) + Core L#29 + PU L#29 (P#34)

NUMANode L#3 (#P#3 64GB) + Package L#3 + L3 L#3 (25MB)

L2 L#30 (256KB) + L1d L#30 (32KB) + L1i L#30 (32KB) + Core L#30 + PU L#30 (P#15)  
 L2 L#31 (256KB) + L1d L#31 (32KB) + L1i L#31 (32KB) + Core L#31 + PU L#31 (P#16)  
 L2 L#32 (256KB) + L1d L#32 (32KB) + L1i L#32 (32KB) + Core L#32 + PU L#32 (P#17)  
 L2 L#33 (256KB) + L1d L#33 (32KB) + L1i L#33 (32KB) + Core L#33 + PU L#33 (P#18)  
 L2 L#34 (256KB) + L1d L#34 (32KB) + L1i L#34 (32KB) + Core L#34 + PU L#34 (P#19)  
 L2 L#35 (256KB) + L1d L#35 (32KB) + L1i L#35 (32KB) + Core L#35 + PU L#35 (P#35)  
 L2 L#36 (256KB) + L1d L#36 (32KB) + L1i L#36 (32KB) + Core L#36 + PU L#36 (P#36)  
 L2 L#37 (256KB) + L1d L#37 (32KB) + L1i L#37 (32KB) + Core L#37 + PU L#37 (P#37)  
 L2 L#38 (256KB) + L1d L#38 (32KB) + L1i L#38 (32KB) + Core L#38 + PU L#38 (P#38)  
 L2 L#39 (256KB) + L1d L#39 (32KB) + L1i L#39 (32KB) + Core L#39 + PU L#39 (P#39)



# Shared Memory : NUMA

How to discover the topology of your node:

- `numactl` tool
- `cpuinfo` tool (by Intel)
- `hwloc` (by OpenMPI)
- directly exploring **/sys/devices/system/**

# Cache coherence

**Data synchronization** is one of the main performance killers for multi-core applications.

- when a memory region is accessed by two cores (i.e. by two different threads running on two different cores), it must be present in both L1/L2, and when one core updates the value stored in the region, the change must be propagated.
- when a thread migrates, the data will still resides on another's core memory.

Memory consistency for the whole system is guaranteed at hardware level, resulting in huge wasting of time if data are not properly handled.

For instance, concurrent access in writing is a main sink of cpu cycles.



# Cache coherence: MESI

Data consistency is maintained by the **MESI** standard.

It is the successor of the MSI protocol and the ancestor of MESOI one

**MODIFIED**

X's values has been modified by this core, and then this is the only valid copy in the system

**EXCLUSIVE**

X is used by this core only; changes do not need to be signalled

**SHARED**

X is used by multiple cores; changes need to be signalled

**INVALID**

X's value has been modified by another core (or X is not used)

see:  
MD64 Architecture Programmer's Manual  
Volume 2: System Programming

In the above, "X" stands for any given memory location

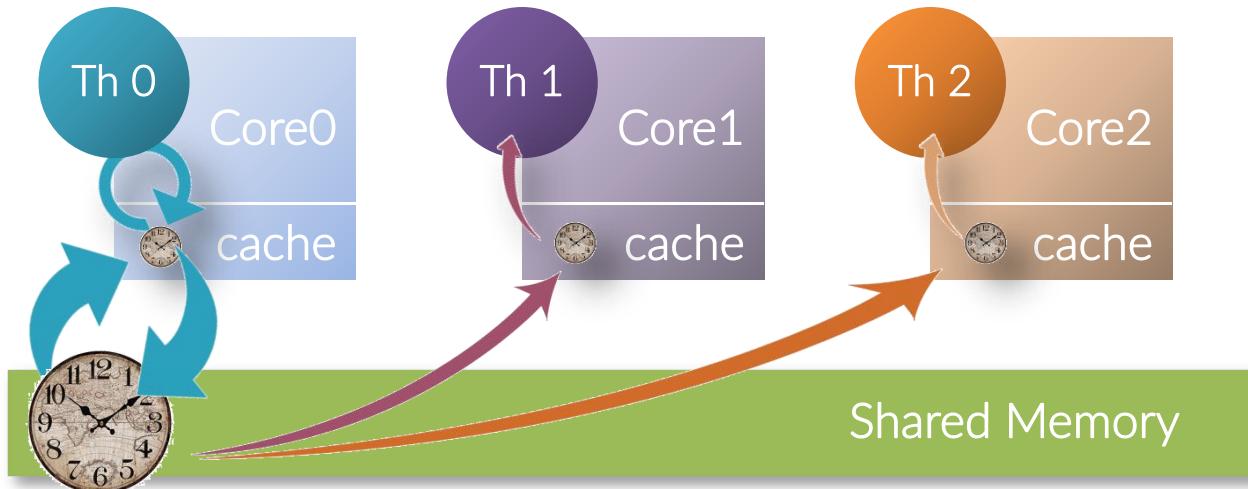


# MESI – an example



Let's clarify with an example. Let's say that there are 3 threads, running on separate cores, accessing some shared-memory.

`Thread0` is running the application `clock()`, which ticks a shared-memory variable that contains the wall-clock `time`. In time to time, both `Thread1` and `Thread2` want to know what `time` it is.





# MESI – an example

	Time (in secs)	Action	cache status		
M	Modified		Core0	Core1	Core2
E	Exclusive	0	-	I	I
S	Shared	1	Th0 reads	E	I
I	Invalid	2	Th0 writes <sup>0</sup>	E	I
		2.3	Th1 reads <sup>1</sup>	S	I
		2.7	Th2 reads	S	S
		3	Th0 writes <sup>2</sup>	M	I
		4	Th0 writes	M	I
		4.4	Th2 reads <sup>1</sup>	S	I
		5	Th0 writes	M	I
		...	...	...	...

0 Core0 is the only one using the value, that is then “Exclusive”. No signal needs to be sent around.

1 A signal is issued to “the memory”, which recognizes that the only valid copy is in the Core0 cache.

Hence, that value is copied back into the shared memory, and from there it is copied in the cache. At that point, everybody has a valid copy, which is then “Shared” (\*).

2 A signal about the change is issued to all the interested actors (those who have a copy) because their values are now “Invalid”. O’s copy is instead “Modified”.

(\*) In the MESOI protocol, in this case the copy can be sent directly to the other caches, without having to transit by the DRAM

# Great powers, great responsibility

A

Variables used by a single core  
They should reside in  
a single cache

B

Read-only variables  
No issues in being shared  
among many cores

C

Modified variables  
Variables modified by many  
cores, or read by many cores

If possible, a cache line should contain only one type of data (A, B or C on the left).

Put variables in the order they will be used.

Type C, modified variables, should stay together, they are the bottlenecks.

The **false sharing** happens when variables of type A or B resides in the same cache line of a type C. Or when two type C variables, modified by two different cores, reside in the same cache line.

# What is parallel computing ?

1. A **parallel computer** is a computational system that offers *simultaneous access* to *many computational units* fed by memory units.  
The computational units are required to be able to *co-operate* in some way, meaning *exchanging data and instructions* with the other computational units.
  
2. **Parallel processing** is the ensemble of techniques and algorithms that makes you able to actually use a parallel computer to successfully and *efficiently* solve a problem.

# What is parallel computing ?

The parallel processing is expressed by **software entities** that have an increasing level of granularity:  
processes, threads, routines, loops, instructions..

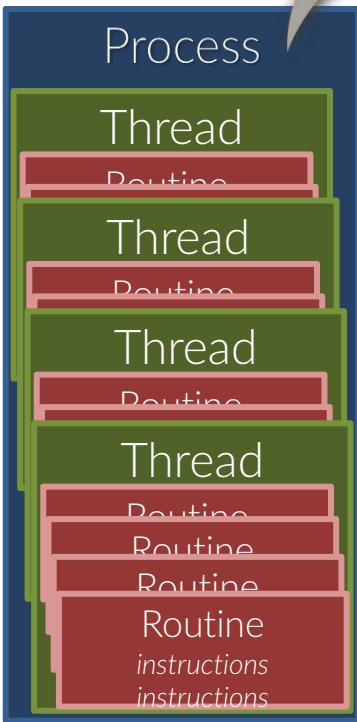
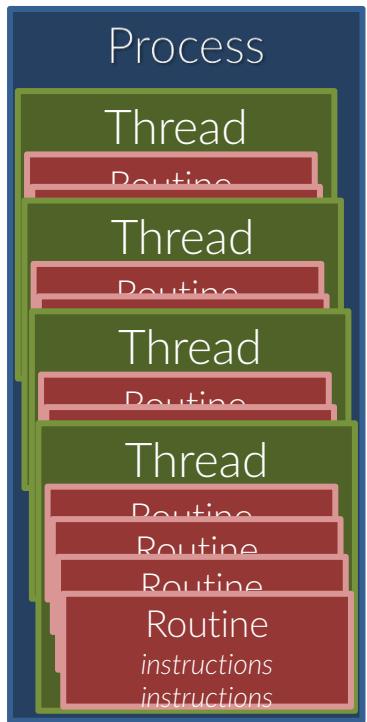
The software entities run on underlying **computational hardware entities** as processors, cores, accelerators

The data to be processed/created live and travel in **storage hardware entities** as  
Memory, caches, NVM, networks, DMA

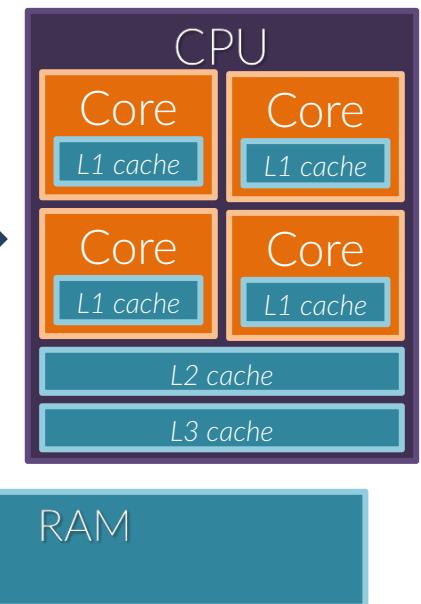
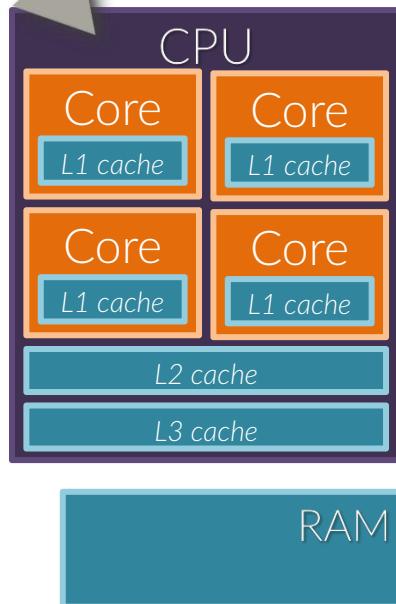
The *exploitation/access* of hardware resources (computational and storage) is **concurrent** among software entities

# What is parallel computing ?

Software level



Hardware level



# Shared vs Distributed Memory

We have discussed before the shared- and distributed- memory concepts. As we have seen, they refer to the physical placement of memory.

However, they may also refer to two distinct **programming paradigms**, named after them, i.e. exactly the shared-memory and distributed-memory paradigm.

In the sense of programming paradigms, the two terms are not linked to the *physical* access to the memory but only to the way you consider it at the level of coding techniques and algorithm implementation.

# Shared vs Distributed Memory

## DISTRIBUTED MEMORY

A typical programming paradigm is the message-passing, i.e. the processes communicate via “messages” while each process has its own memory space. The actual communication may happen either over the network (different protocols are possible at hardware/middleware level) or via shared memory techniques if the communicating processes can directly access the same memory (the user still does not see it as shared memory but uses messages in its coding, the actual communication is managed by the middle-ware). A very diffuse standard is **MPI**, Message-Passing Interface.

A message-passing standard may expose (as MPI does since version 2.0) interfaces for direct memory access, mimicking shared-memory mechanisms.

# Shared vs Distributed Memory

## SHARED MEMORY

A typical programming paradigm is multi-threading, i.e. multiple threads access concurrently the same virtual address space.

No such a thing as “messages” exists, communications and synchronizations must be managed in shared-memory.

A very diffuse high-level standard is **OpenMP** ([openmp.org](http://openmp.org)), Open Multi-Processing. On all platforms, a very low-level library for threading is present. On POSIX system it is named *pthread*.

On some systems, a software middleware may hide the actual physical details from you and expose the entire machine’s memory of all nodes as a shared memory, even if underneath the remote memory access happens via network.  
That is called PGAS, partitioned global address space.

# Shared vs Distributed Memory

The two programming paradigms, shared- and distributed-memory, can easily be fused and used together.

That typically happens as the following.

An MPI process running on a node spawns a number of threads that by definition run on shared-memory among themselves. The same MPI process will communicate with other MPI processes via messages, irrespectively of where they run.

Actually, MPI v2.0 and v3.0 offer handy mechanisms to have RDMA (Remote Direct Memory Access) and even shared-memory access among MPI processes.

# What is parallel computing ?

The Flynn's taxonomy (~60s)

		Instructions	
		single	multiple
Data	multiple	SISD	MISD
	single	SIMD	MIMD

The Flynn's taxonomy helps in understanding the logical subdivision of parallel systems, but it is no more up-to date; it mainly refers to HW capabilities but in 60yrs the HW evolved a lot and today we can imagine it refers to a mix of HW and SW

SISD - Single Instruction on Single Data

MISD - Multiple Instructions on Single Data

SIMD - Single Instruction on Multiple Data

MIMD - Multiple Instructions on Multiple Data



# What is parallel computing ?



	HW level	SW level
SISD	A Von Neumann CPU	no parallelism at all
MISD	On a superscalar CPU, different ports executing different <i>read</i> on the same data	<ul style="list-style-type: none"><li>• ILP on same data;</li><li>• Multiple tasks or threads operating on the same data</li></ul>
SIMD	Any vector-capable hardware, the vector registers on a core, a GPU, a vector processor, an FPGA, ...	data parallelism through vector instructions and operations
MIMD	Every multi-core/processor system; on a superscalar CPUs, different ports executing different ops on different data	<ul style="list-style-type: none"><li>• ILP on different data;</li><li>• Multiple tasks or threads executing different code on different data.</li></ul>



# Why parallelism ?

For two main reasons:

## 1. Time-to-solution

- To solve the same problem in a smaller time
- To solve a larger problem in the same time

## 2. Problem size ( ~ data size)

To solve a problem that could *not* fit on the memory addressable by a single computational units (or that could fit in the space around a single computational units without serious performance loss)

# How do you do parallelism ?



A good reference is the Part 1 of “Designing and building parallel programs” by Ian Foster, [publicly available](#) (pdf is easily findable).

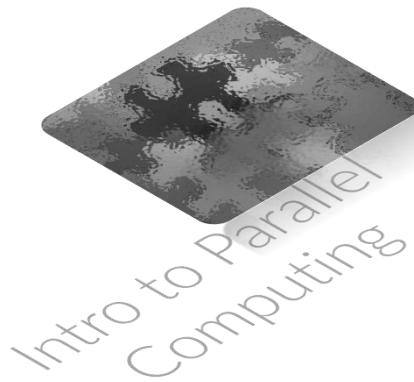
It is outdated in some respects, but the concepts about parallelism, parallel design and parallel performance are explained in a clear and clean way.

While the **whole Part 1** is a useful reading, I suggest  
Chapter 2, 0 → 2.5  
Chapter 3, 0 → 3.8

Note that also the case studies in Chap 2 and 3 are interesting, of course.  
Chap 4 is a useful re-collection and application to case studies.



# Introduction Outline



Parallel  
Performance

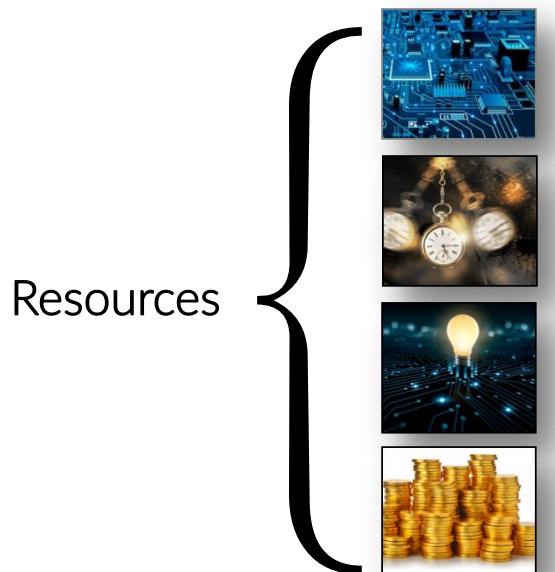


# What is parallel performance



Has we have seen, «performance» is a tag that can stand for many things.

In this frame, with «performance» we mean the relation between the computational requirements and the computational resources needed to meet those requirements.



$$\text{Performance} \approx \frac{1}{\text{resources}}$$

$$\text{Performance ratios} \approx \frac{\text{resources}_1}{\text{resources}_2}$$

Inspired by "parallel performance theory, CIS 410/510 Univ. of Oregon



# What is parallel performance



Performance is a measure of how well the computational requirements are met and, at the same time, of how well the computational resources are exploited.

*“The most constant difficulty in contriving the engine has arisen from the desire to reduce the time in which the calculations were executed to the shortest which is possible.”*

*Charles Babbage, 1791 – 1871*



# Key factors



$n$	Problem size
$T_s(n)$	Serial run-time for a problem of size $n$
$T_p(n)$	Parallel run-time with $p$ processes for a problem of size $n$
$p$	Number of computing units
$f_n$	Intrinsic sequential fraction of the problem of size $n$
 $k(n, t)$	Parallel overhead
Speedup	$Sp(n, t) = \frac{T_s(n)}{T_p(n)}$
Efficiency	$Eff(n, t) = \frac{T_s(n)}{p \times T_p(n)} = \frac{Sp(n, t)}{p}$





# Naïve expectations



- If single processor has  $\sim m$  Mflops, parallel flops performance with  $p$  tasks is  $p \times m$  Mflops.
- If sequential run-time is  $T$ , parallel run-time with  $p$  tasks is  $\propto T/p$ .
- If parallel run-time with  $p$  tasks is  $T$ , and the run-time with  $p_1$  tasks is  $T_1$ , then  $T_1/T_2 \propto p_2/p_1$
- If parallel run-time with  $p$  tasks and problem size  $Z$  is  $T$ , the run-time with size  $Z_1$  is  $T_1 \propto T \times Z_1/Z$ .

Is that correct ?



# Parallel performance



The sequential execution time for a problem of size  $n$  is

$$T_S(n) = T_S(n) \times f_n + T_S(n) \times (1-f_n)$$



Assuming that the parallel fraction of the computation is *perfectly parallel*, meaning that its run time scales as  $1/p$ , then we can express the parallel execution time as

$$T_P(n) = T_S(n, 1) \times f_n + T_S(n) \times (1-f_n) / p$$

And then

$$\text{speedup} : Sp(n, p) = T_S(n) / T_P(n) = \frac{1}{f + \frac{1-f}{p}}$$

$$\lim_{p \gg 1} Sp(n, p) = \frac{1}{f}$$

$$\text{efficiency} : Eff(n, p) = Sp(n, p) / p = \frac{1}{f(p-1)+1}$$

$$\lim_{p \gg 1} Eff(n, p) = 0$$



# Parallel performance



If the parallel fraction ( $1-f_n$ ) is not perfectly parallelizable, we must take into account some *parallel* overhead, and then we can express the parallel execution time as

$$T_P(n) = T(n,p) = T_S \times f_n + T_S \times (1-f_n)/p + k(n,t)$$



And then

$$\text{speedup} = Sp(n,p) \leq T_S / T_P$$

$$\text{efficiency} = Eff(n,p) \leq Sp(n,p) / p$$



# Amdahl's law



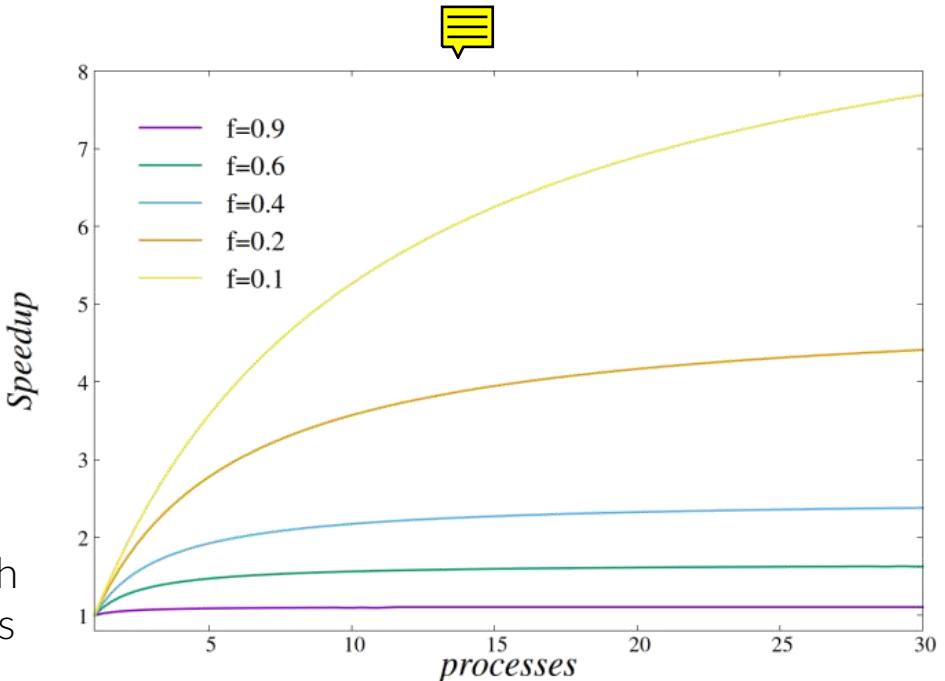
If  $f$  is the fraction of the code which is intrinsically sequential, as we have seen the speedup is

$$Sp(n, t) \leq \frac{1}{f + \frac{1-f}{p}} \leq \frac{1}{f}$$

i.e. the serial fraction is severely limiting the achievable speedup.

Is that a problem ?

Actually tens of years ago there was a huge debate about whether or not a huge effort towards developing massive parallelism (which means developing hw and ws technology) was in order or not.





There are some significant issues in the Amdhal's law shown in the previous slide:

- No matter of how many processes  $p$  are used, the maximum achievable speedup is determined by  $f$  (and is quite low for ordinary problems).
- The problem size  $n$  is kept fixed when estimating the possible speedup while the number of processes increases (*strong scaling*).  
However, most often the problem size increases as well.
- The parallel overhead  $k(n, t)$  is ignored, which leads to an optimistic estimate of the speedup, and usually,  $p(n)/t > k(n,t)$
- The fraction of sequential part may decrease when the problem size increases

Then, usually the speedup increases with problem size



# Gustafson's law

However, normally when you increase the problem's size, the parallelizable part increases way more than the sequential part.

If we consider the workload as the sum

$$w = a+b$$

where  $a$  and  $b$  being the serial and the parallel work, and we assign the same amount of workload to every process, that would amount to a serial run-time

$$T_s \propto a + p \times b$$

while it still takes

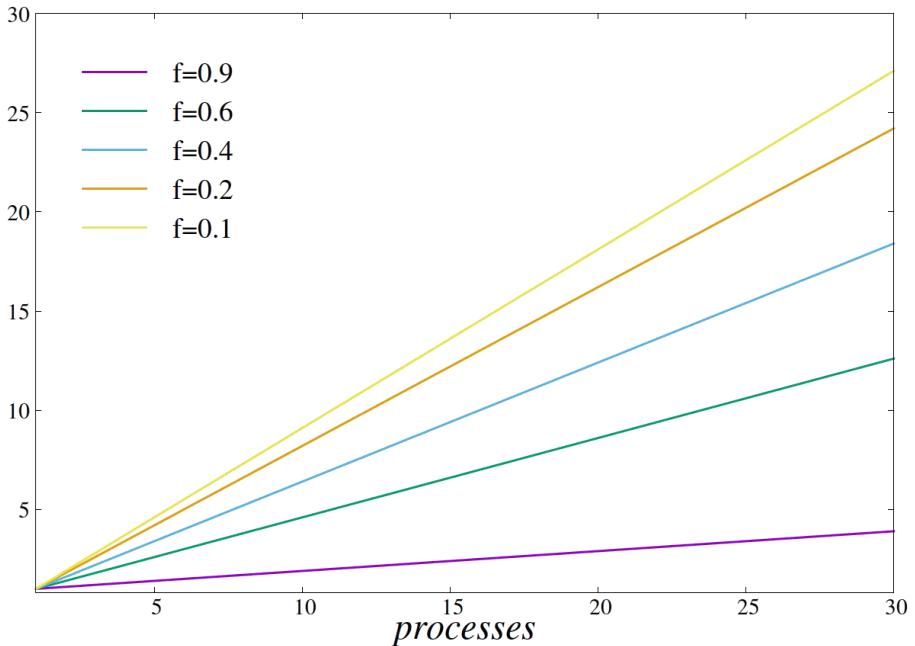
$$T_p \propto a+b \text{ using } p \text{ processes}$$

Hence the speedup is  $[a + p \times b] / [a+b]$ , which if  $f_n = a/(a+b)$  we can rewrite as the Gustafson's law for the speedup:

$$Sp_G(n, t) = p - (p - 1)f_n \leq p$$



Speedup





# Scalability



The two lines of reasoning, the former by Amdhal and the latter by Gustafson, lead us to two different concepts for the *scalability*, which is the ability of a parallel system to increase its efficiency when the number of processes and/or the size of the problem get larger.

1. **STRONG SCALABILITY:** the problem size is fixed,  $p$  increases 
2. **WEAK SCALABILITY:** the workload is fixed, the problem size *and*  $p$  increase 



# Parallel overhead



In parallel computing there may be several sources of overhead due to the parallelization itself:

- Communication overhead
- Algorithmic overhead
- Synchronization
  - Critical paths - Dependencies across different processes
  - Bottlenecks (some processes are stuck and make all the others to waste time)
  - Work-load imbalance
- Thread/processes creation

Hence, if  $k(n,t)$  is the overhead of some kind,  $t_S$  and  $t_P$  the run-time for the serial and the parallel part, the parallel run-time can be written as

$$T_P(n, p) = t_S + \frac{t_P}{p} + k(n, t)$$

Let's define an **experimentally measured** serial fraction of time:

$$e(n, p) = \frac{t_S + k(n, p)}{t_S + t_P}$$



# Parallel overhead



With a little bit of math:  $e(n, p) = \frac{\frac{1}{Sp(n,t)} - \frac{1}{p}}{1 - \frac{1}{p}}$

Let's check a couple of examples:

$p$	2	4	8	10	20
$Sp(p)$	1.69	2.6	3.52	3.79	4.49
$E(p)$	0.18	0.18	0.18	0.18	0.18

The measured serial fraction is constant, the lack of scaling is due to the 0.18 fraction of serial workload.

$p$	2	4	8	10	20
$Sp(p)$	1.67	2.47	3.11	3.22	3.15
$E(p)$	0.2	0.21	0.22	0.23	0.28

The measured serial fraction keep increasing: the lack of scaling is also due parallelization overhead



# Parallel overhead



A simpler, quick-and-dirty way to measure your parallel overhead is to consider the “distance” between your code’s scaling and the perfect scaling.



$$\begin{aligned} T_s(n) - T_p(n, p) &= \\ t_s + t_p - t_s - \frac{t_p}{p} - k(n, p) &= \\ t_p \times \frac{p - 1}{p} - k(n, p) &= \end{aligned}$$

which becomes for ”large”  $p$   
 $\approx t_p - k(n, p)$

- CPU = Socket = processor
- A CPU contains multi cores; “many” cores are multi-cores with large number of cores.
- sometimes (for instance on some software tools that explore the machine topology), the hardware cores are referred as “cpus”
- NUMA nodes (in tools like numactl, ...) = sockets
- MPI processes are also called “tasks”, whereas OpenMP threads are always threads. However, a “task” is also a concept in functional decomposition, as discusses in the OpenMP part that introduce the, in fact, tasks.

that's all, have fun

"So long  
and thanks  
for all the fish"