

# Parallel Computing & OpenMP Introduction

Luca Tornatore - I.N.A.F.



**“Foundation of HPC” course**



DATA SCIENCE &  
SCIENTIFIC COMPUTING  
2021-2022 @ Università di Trieste

# Outline



Introduction



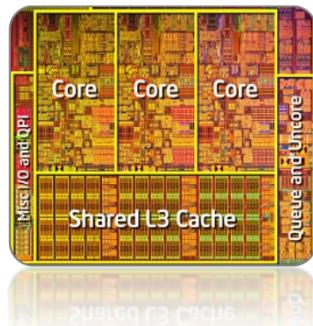
Parallel  
Computing



Intro to  
OpenMP



# ntroduction Outline



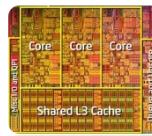
The race  
to multicore



Intro to Parallel  
Computing

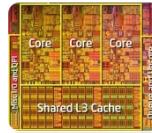


Parallel  
Performance



# Warm-up

A quick recap of what we have  
seen in the Optimization part ...

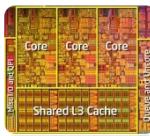


Race to  
Multicore



“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.**”

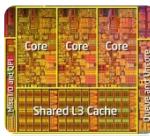
DOE’s Office of Science, 2012



Applications no longer get more performance for free without significant redesign, since 15 years

Since 15 years, the gain in performance is essentially due to fundamentally different factors:

1. Multi-core + Multi-threads
2. Enlarging/improving cache
3. Hyperthreading (smaller contribution)



# Why there is no more “free lunch”?

For instance:

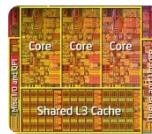
2 Cores at 3GHz are  
basically 1 Core at 6GHz.. ?

False

- ✗ Cores coordination for cache-coherence
- ✗ Threads coordination
- ✗ Memory access
- ✗ Increased algorithmic complexity

Since 15 years, the gain in performance  
is essentially due to  
**fundamentally different factors:**

1. Multi-core + Multi-threads
2. Enlarging/improving **cache**
3. Hyperthreading (*smaller contribution*)



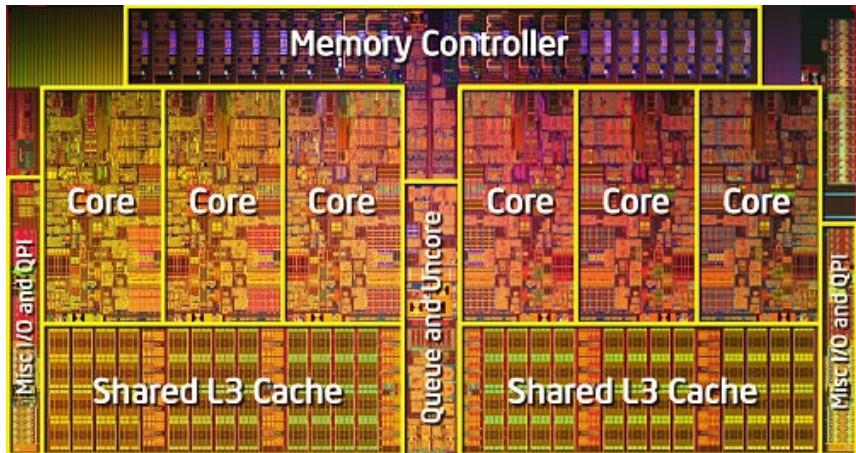
Race to  
Multicore

# Back to the future

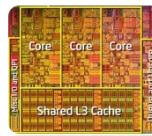


## Message I

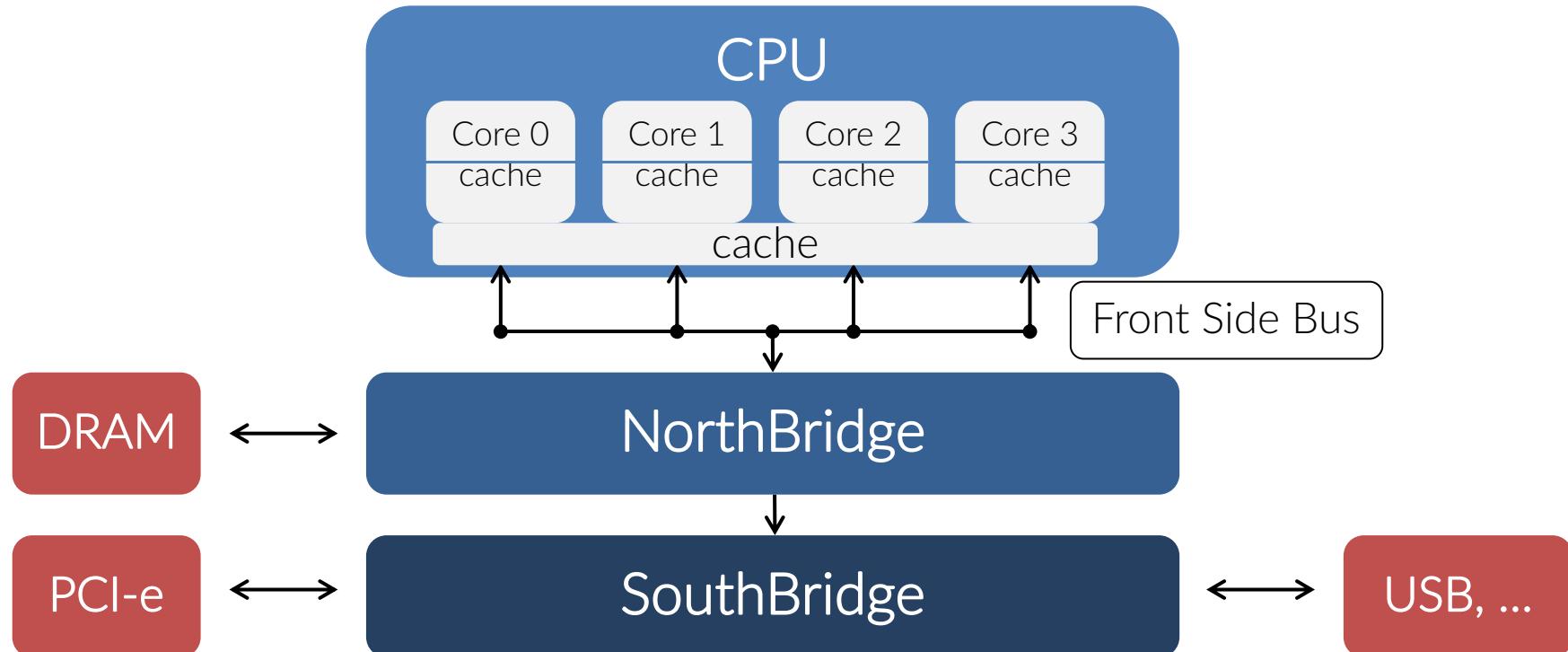
Many-cores CPUs are here to stay

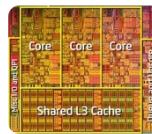


- Concurrency-based model programming (different than both *parallel* and *ILP*): work subdivision in as many independent tasks as possible
- Specialized, heterogeneous cores
- Multiple memory hierarchies



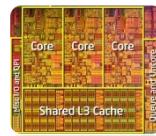
# The typical UMA architecture



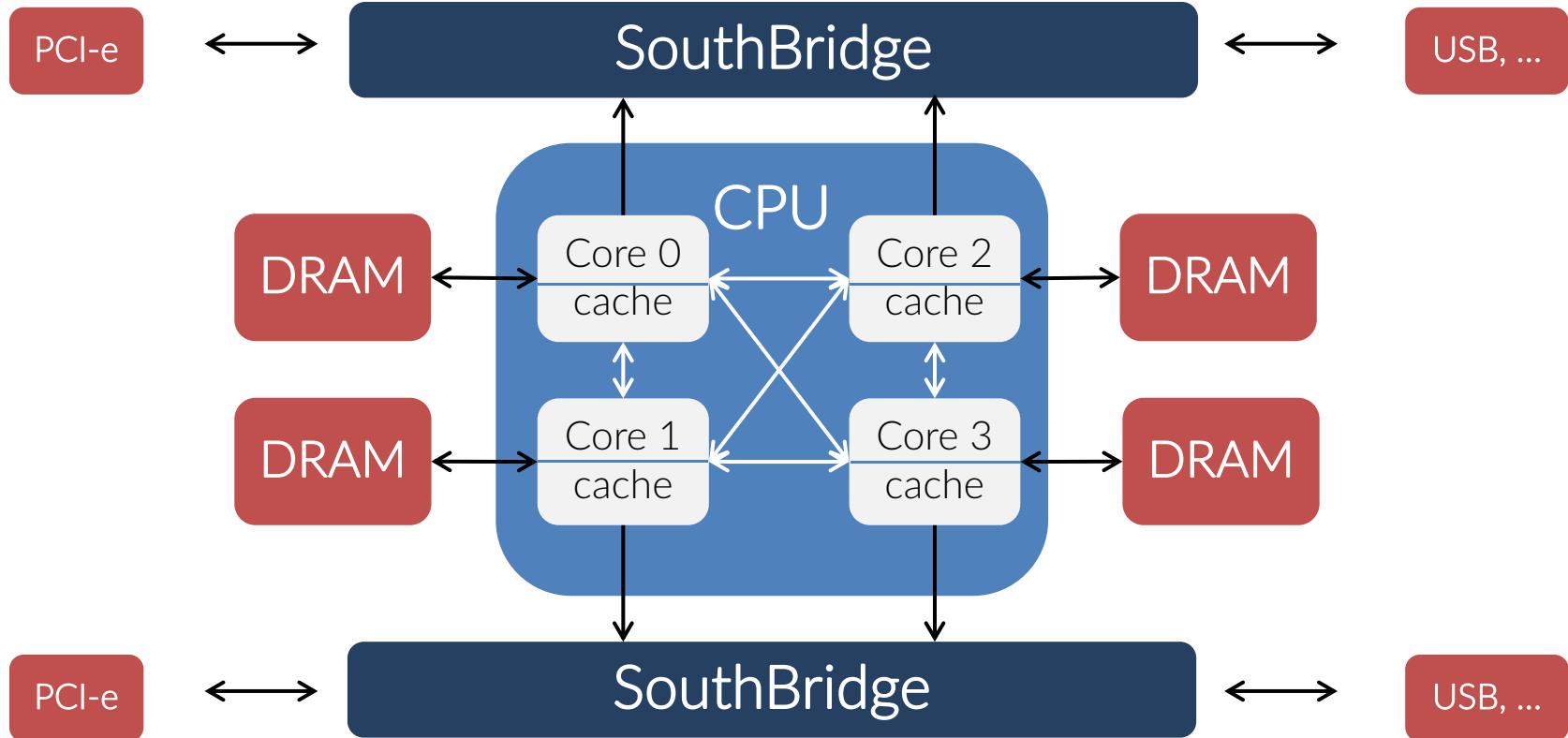


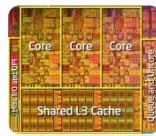
# | The typical UMA architecture

- The RAM can be accessed by one core at a time
  - this lowers the effective bandwidth
  - data coherence is easier
- The faster SRAM was introduced as caches to keep up with the increase of cores' clock
- FSB and RAM access is the main bottleneck

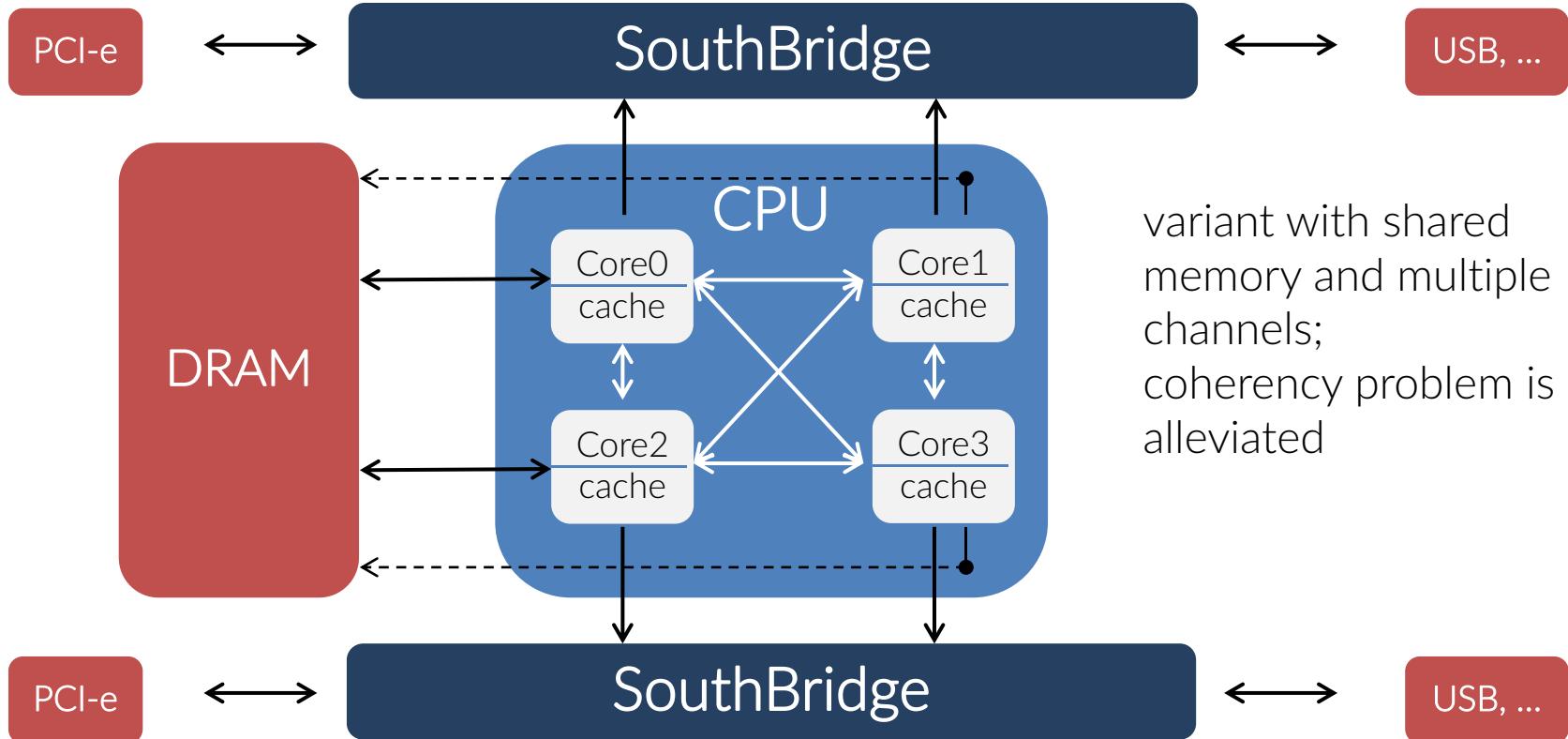


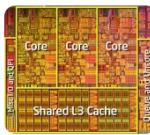
# The typical NUMA architecture



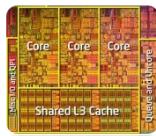


# The typical NUMA architecture



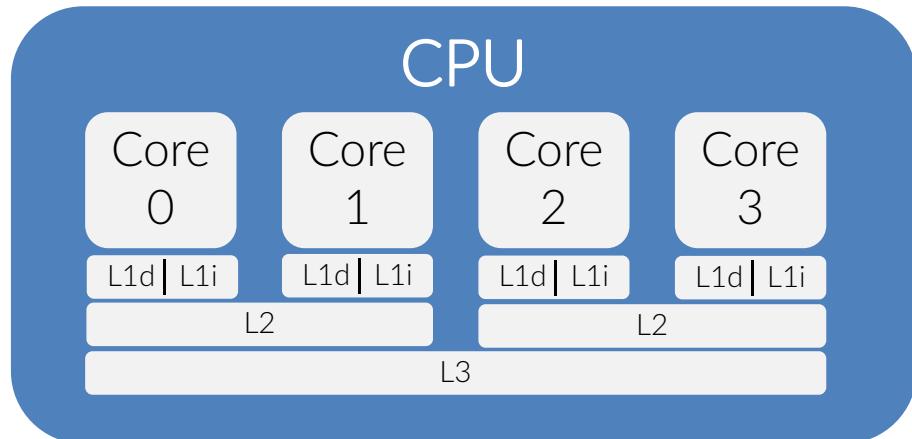
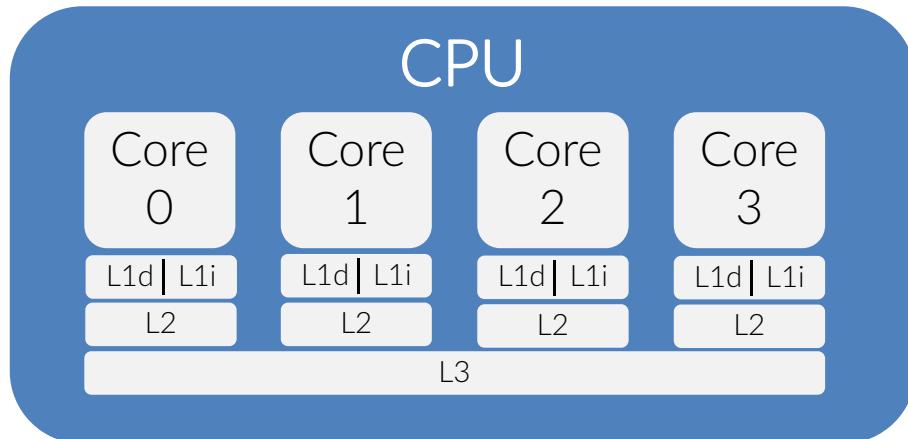


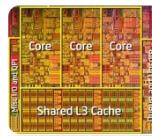
- The bottleneck of resources (RAM and southbridge) access is eliminated
  - Each core *may* (*usually it does*) have its own DRAM module
- NUMA was originally developed to link several sockets, but it also evolved *inside* a single socket
- NEW problems:
  - data coherence (a variable *may* resides in a single DRAM module, unless you replicate data)
  - accessing DRAM has different costs



# The typical NUMA architecture

Cache hierarchy can have different topologies





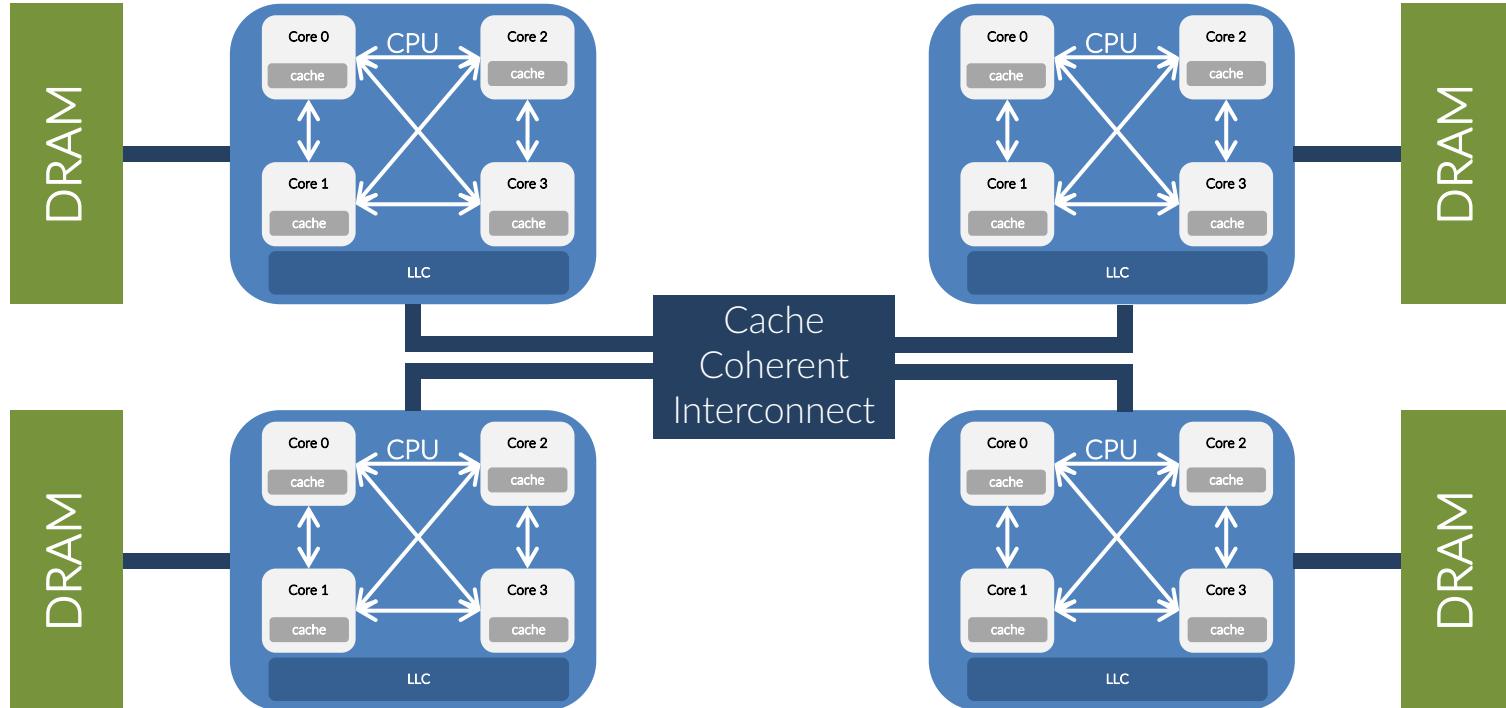
Race to  
Multicore

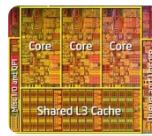
# The typical NUMA architecture

Introduction



A zoom-out to a multi-socket node





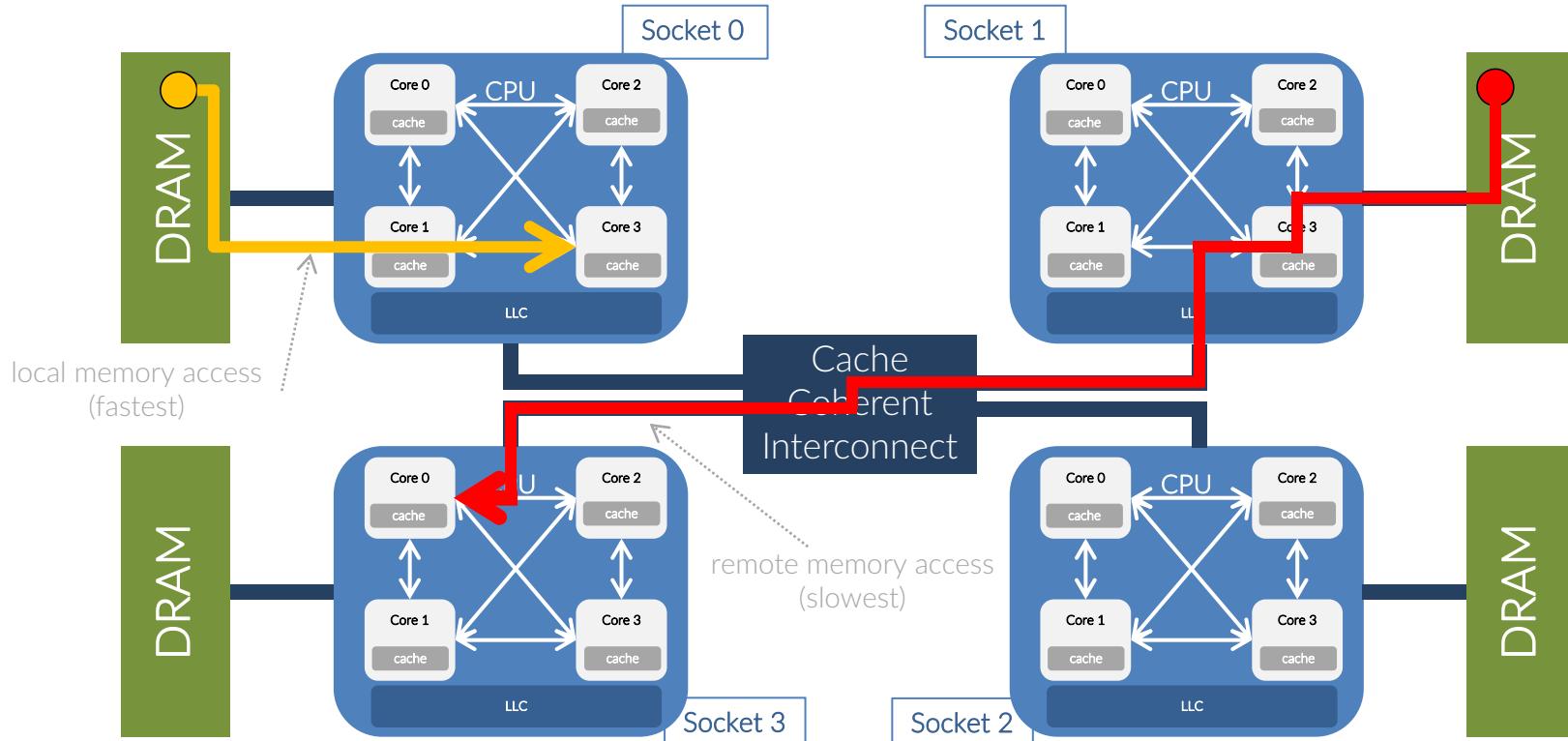
Race to  
Multicore

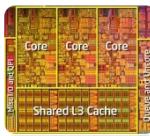
# The typical NUMA architecture

Introduction



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





## The typical NUMA architecture



Two examples, both of nodes with 4 sockets each

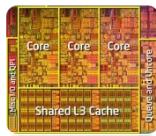
```
[ltornatore@hp10 ~]$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit      hyperthreading off
Byte Order:            Little Endian
CPU(s):                40
On-line CPU(s) list:  0-39
Thread(s) per core:   1
Core(s) per socket:   10
Socket(s):             4
NUMA node(s):          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 }
```

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

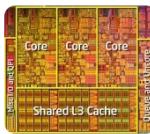
```
[ltornatore@gen10-01 ~]$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit      hyperthreading on
Byte Order:            Little Endian
CPU(s):                96
On-line CPU(s) list:  0-95
Thread(s) per core:   2
Core(s) per socket:   12
Socket(s):             4
NUMA node(s):          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 }
```

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



How to discover the topology of your node:

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



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

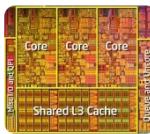
- **numactl** tool

**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
```



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

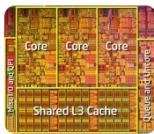
- **numactl** tool

- **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)
```



Race to  
Multicore

Introduction

# The typical NUMA architecture

How to discover the topology of

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

Machine (256GB total)

NUMANode L#0 (#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 (#1 64GB) + Package L#1 + L3 L#1 (25MB)

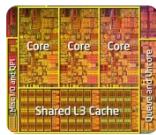
L2 L#10 (256KB) + L1d L#10 (32KB) + L1i L#10 (32KB) + Core L#10 + PU L#10 (P#5)  
L2 L#11 (256KB) + L1d L#11 (32KB) + L1i L#11 (32KB) + Core L#11 + PU L#11 (P#6)  
L2 L#12 (256KB) + L1d L#12 (32KB) + L1i L#12 (32KB) + Core L#12 + PU L#12 (P#7)  
L2 L#13 (256KB) + L1d L#13 (32KB) + L1i L#13 (32KB) + Core 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 (#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)

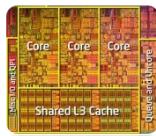
NUMANode L#3 (#3 64GB) + Package L#3 + L3 L#3 (25MB)

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)



How to discover the topology of your node:

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

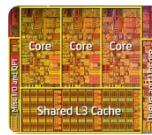


# The typical NUMA architecture

How to discover the topology of your node:

- `numactl` to
- `cpuinfo` to
- `hwloc` (by C
- directly exp

Jump to [Numa Awareness](#) to explore how to orchestrate data and OpenMP threads on NUMA architectures



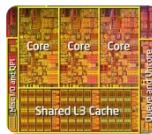
# Cache coherence

**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**

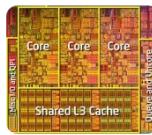
X is used by multiple cores; changes need to be signalled

**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

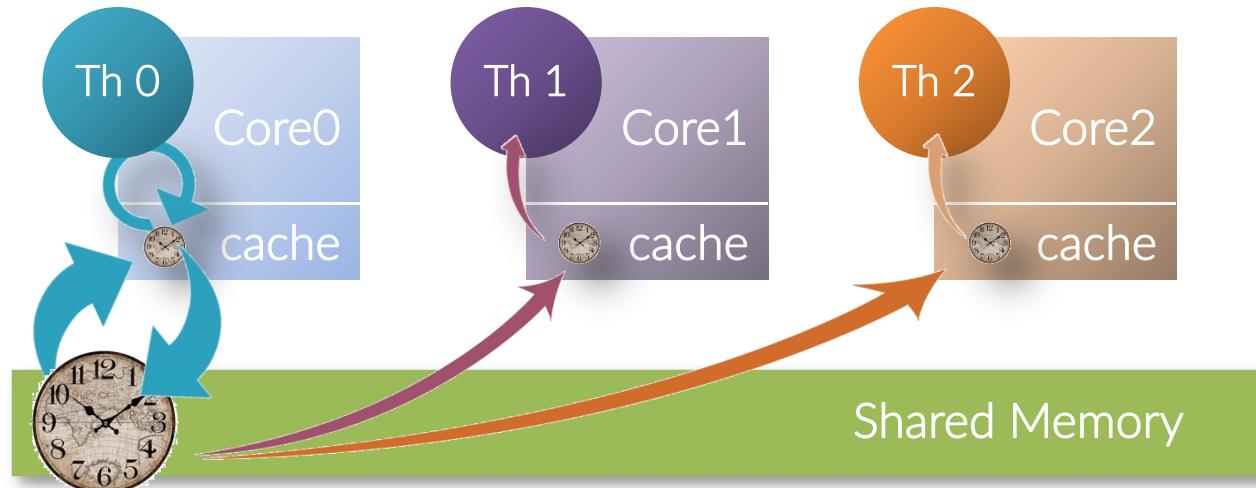
In the above, "X" stands for any given memory location

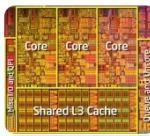


# Cache coherence: MESI - an example

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.





# Cache coherence: MESI – an example



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

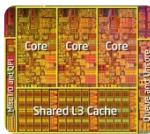
0 Core0 is the only one using the value, that is then “Exclusive”. No signal needs to be sent around.

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.



# Introduction Outline



Intro to Parallel  
Computing



Parallel  
Performance

# | 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..

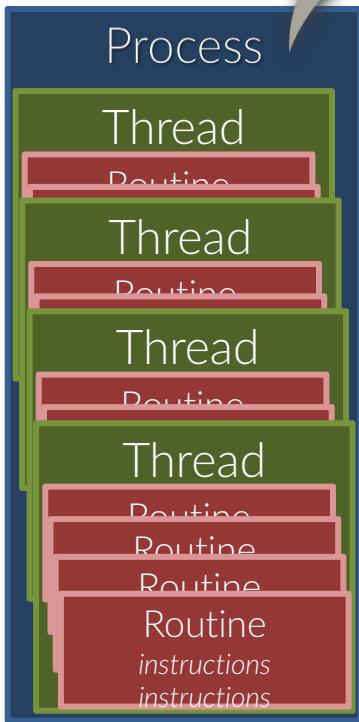
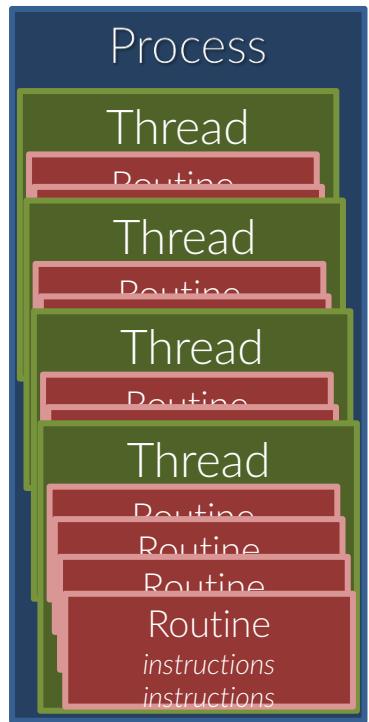
The software entities run on underlying **computational hardware entities** as processors, cores, accelerators

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

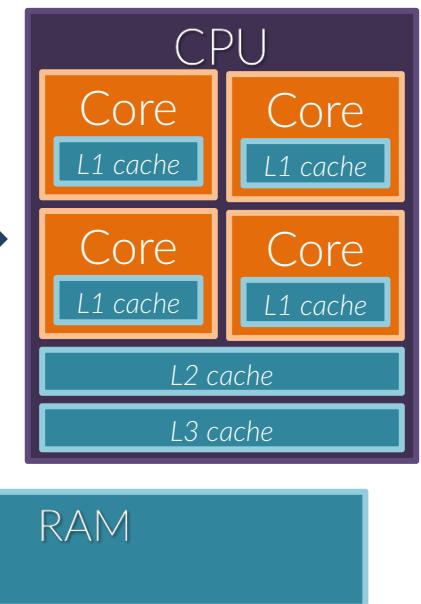
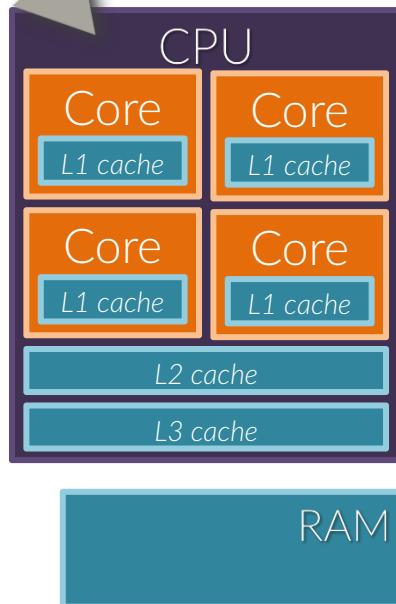
The *exploitation/access* of hardware resources (computational and storage) is **concurrent** among software entities

# | What is parallel computing ?

Software level



Hardware level





# I Why parallelism ?

For two main reasons:

## 1. Time-to-solution

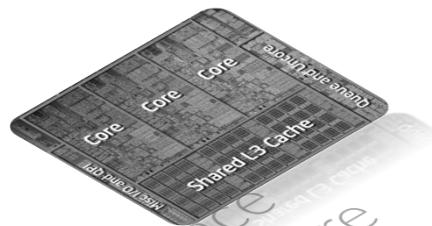
- To solve the same problem in a smaller time
- To solve a larger problem in the same time

## 2. Problem size ( $\sim$ data size)

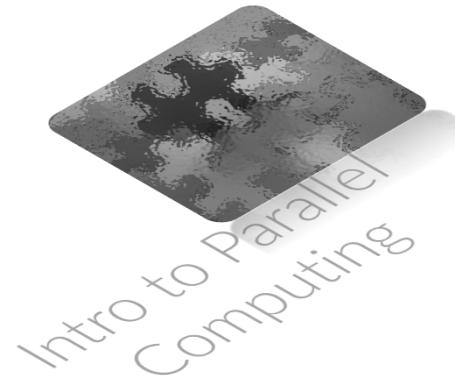
To solve a problem that could *not* fit on the memory addressable by a single computational units (or that could fit in the space around a single computational units without serious performance loss)



# Introduction Outline



The race  
to multicore



Intro to Parallel  
Computing



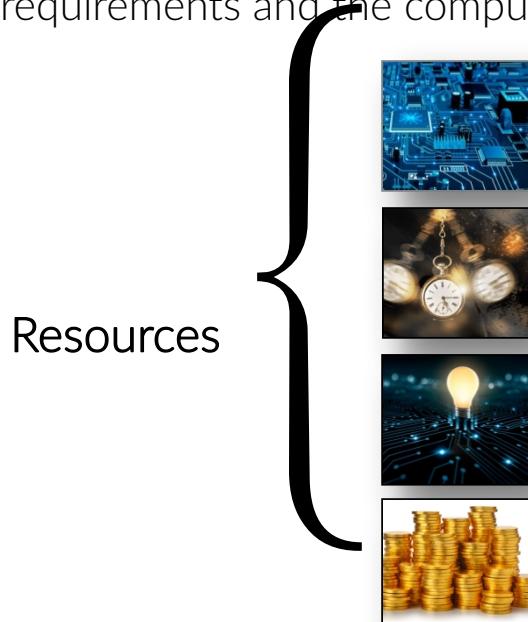
Parallel  
Performance



# What is parallel performance

As 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.



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

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



# 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

$T_p(n)$

Parallel run-time

$p$

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



Sequential execution time is

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

Assuming that the parallel fraction of the computation is *perfectly parallel*, parallel execution time is

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

And then

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

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

unless  $k(n,t) = 0$ .



# Amdahl's law

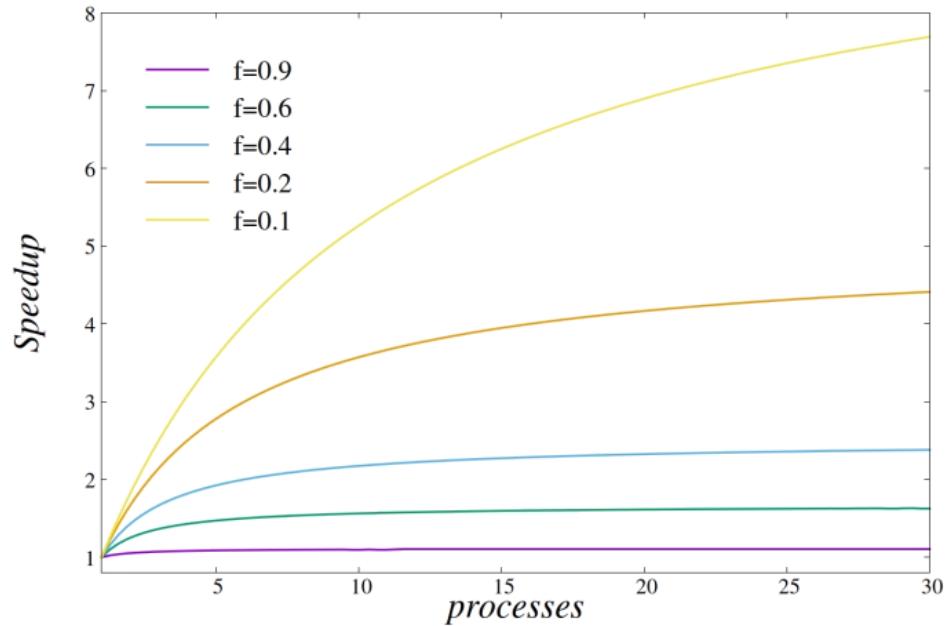


If  $f$  is the fraction of the code which is intrinsically sequential,

the speedup is then

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

Note that we wrote  $f$  instead of  $f_n$  and we are neglecting the parallel overhead





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

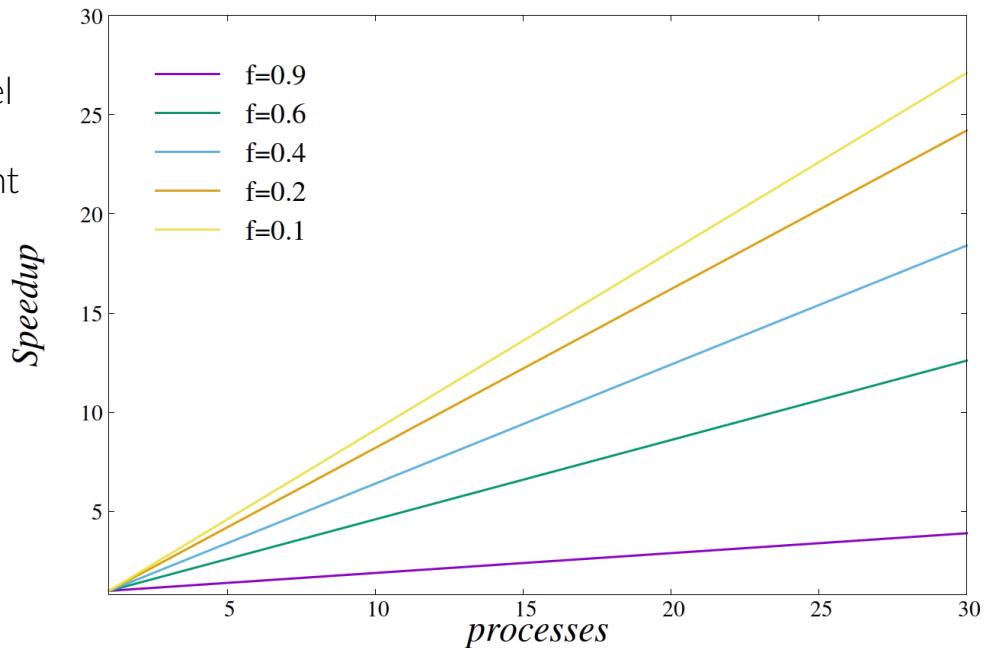
$$T_p \propto a+b \text{ using } p \text{ processes}$$

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:

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





# 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.

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

# Outline



Introduction



Parallel  
Computing



Intro to  
OpenMP



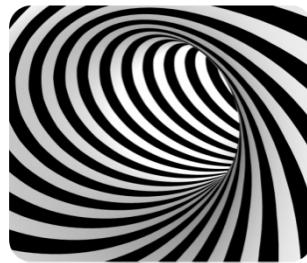
# OpenMP Outline



Introduction  
&  
Concept



Parallel  
Regions



Parallel  
Loops



Sections  
& Task



NUMA  
AWARENESS



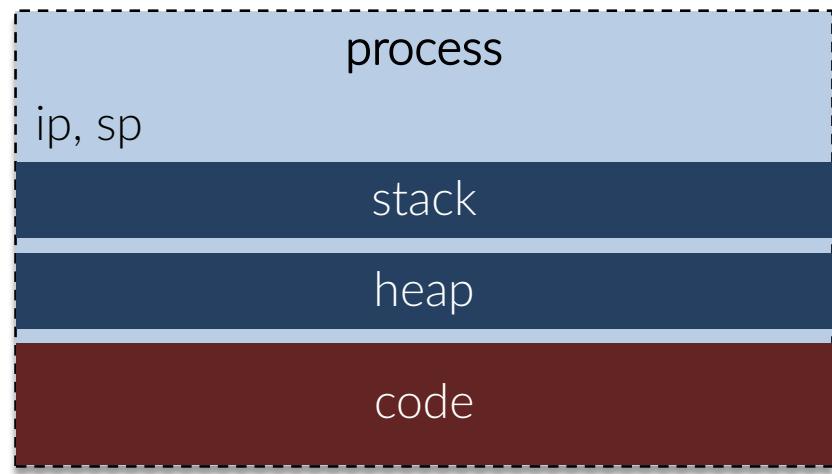
# Threads and processes

A **process** is an independent sequence of instructions *and* the ensemble of resources needed for their execution.

A program needs much more than just its binary code (i.e. the list of ops to be executed): it needs to access to a protected memory space and to access system resources (e.g. files and network).

A “process” is then a program that has been allocated with the necessary resources by the operating system.

There may be different **instances** of the same program as different, independent processes





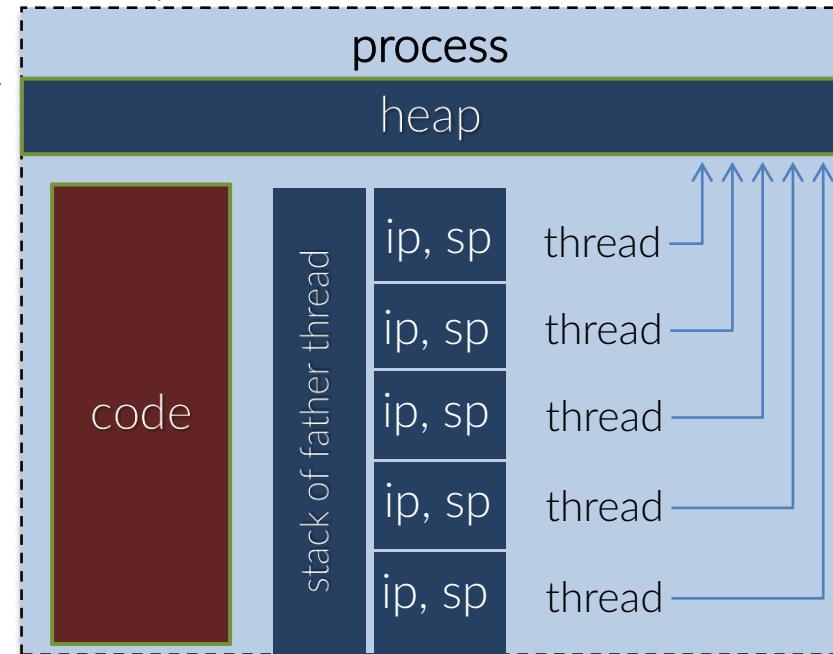
# Threads and processes

A **thread** is an independent instance of code execution *within* a process. There may be from one to many threads within the same process.

Each thread shares the same code, memory address space and resources than its father process.

While each thread has its own stack, ip and sp, the heap will be shared among threads, which then operate in *shared-memory*. threads also share the stack of the father thread.

In general spawning threads inside a process is much less costly than creating processes.

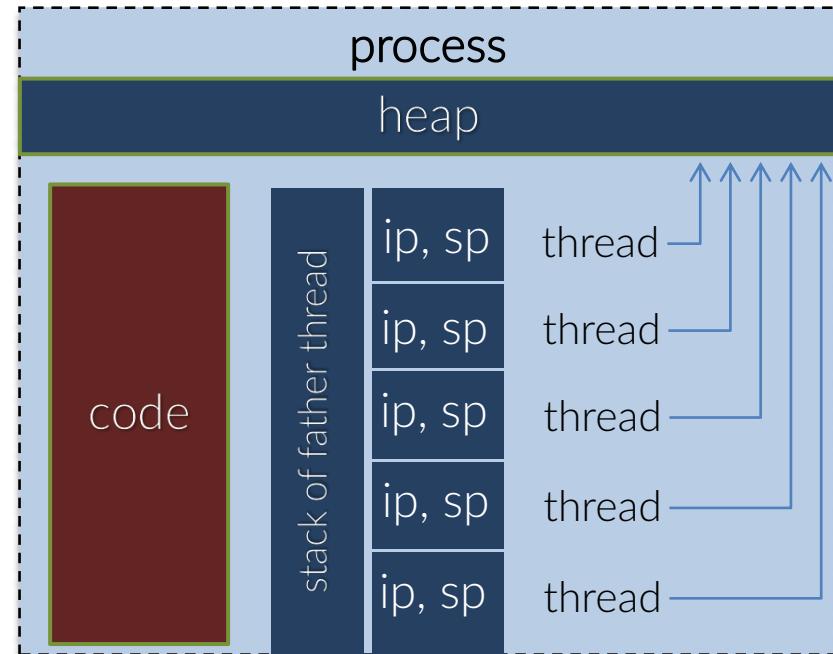




# Threads and processes

A thread can run either on the same computational units of its father process or on a different one.

A computational unit nowadays amounts to a **core**, either inside the same CPU (socket) on which the father process runs, or inside a sibling socket in the same NUMA region.





# | What is OpenMP

OpenMP is a standard API to enable shared-memory parallel programming:  
**Open specifications for MultiProcessing**

It allows to write multi-threaded programs with a standard behaviour through the usage of a set of compiler directives to be inserted in the source code:

- Pragmas '#' in C/C++
- Specially formatted comments in Fortran

Both fine- and coarse-grain parallelism are possible, from loop-level to explicit assignment to threads.



# | OpenMP vs MPI

i.e

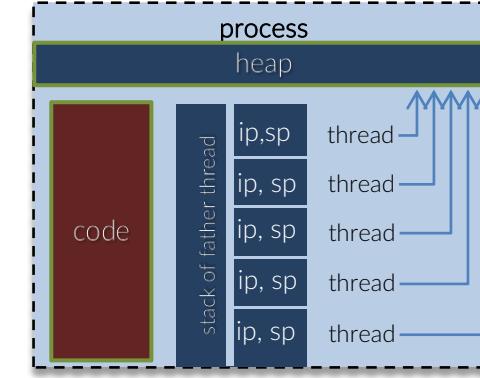
Shared-Memory  
vs  
Distributed-Memory



# OpenMP vs MPI

## Shared-Memory (e.g. OpenMP)

A unique process that spawns a number of threads. There is a unique memory space that is accessible by all the threads

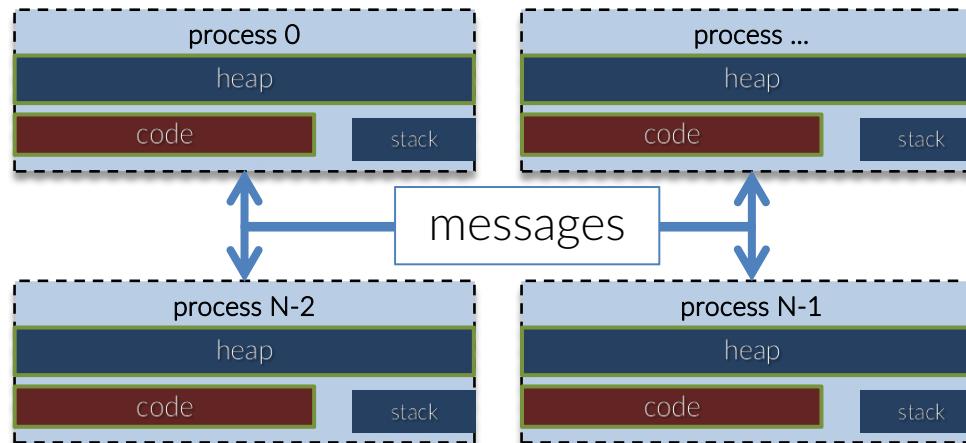


## Distributed-Memory (e.g. MPI)

$N$  processes are created, each with its own copy of the code and its own memory space.

A process *can not* access the memory space of another process.

The processes communicate through messages.

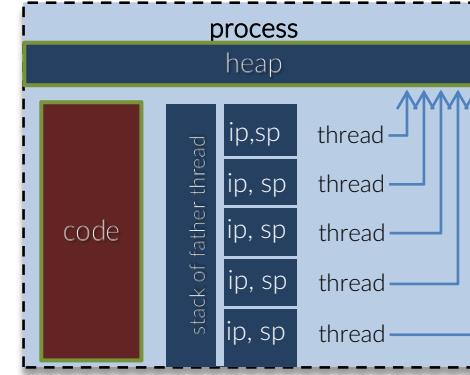




# OpenMP vs MPI

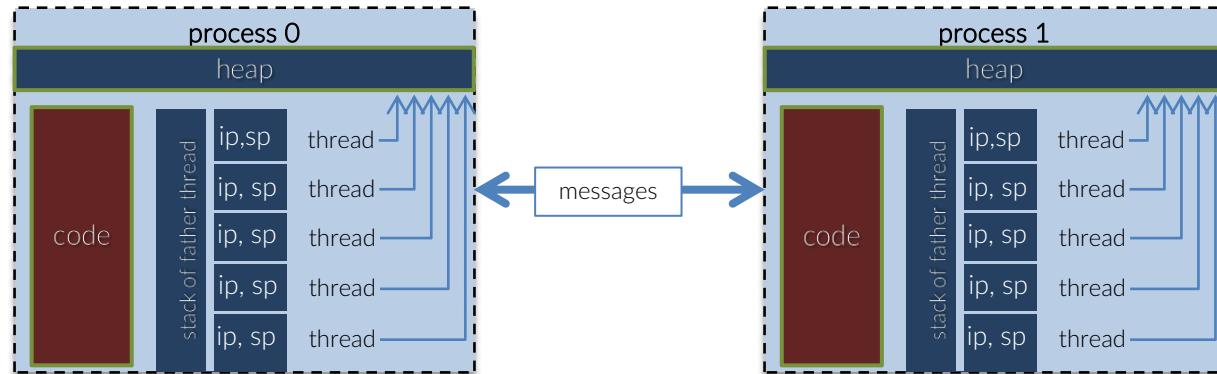
## Shared-Memory (e.g. OpenMP)

A unique process that spawns a number of threads. There is a unique memory space that is accessible by all the threads



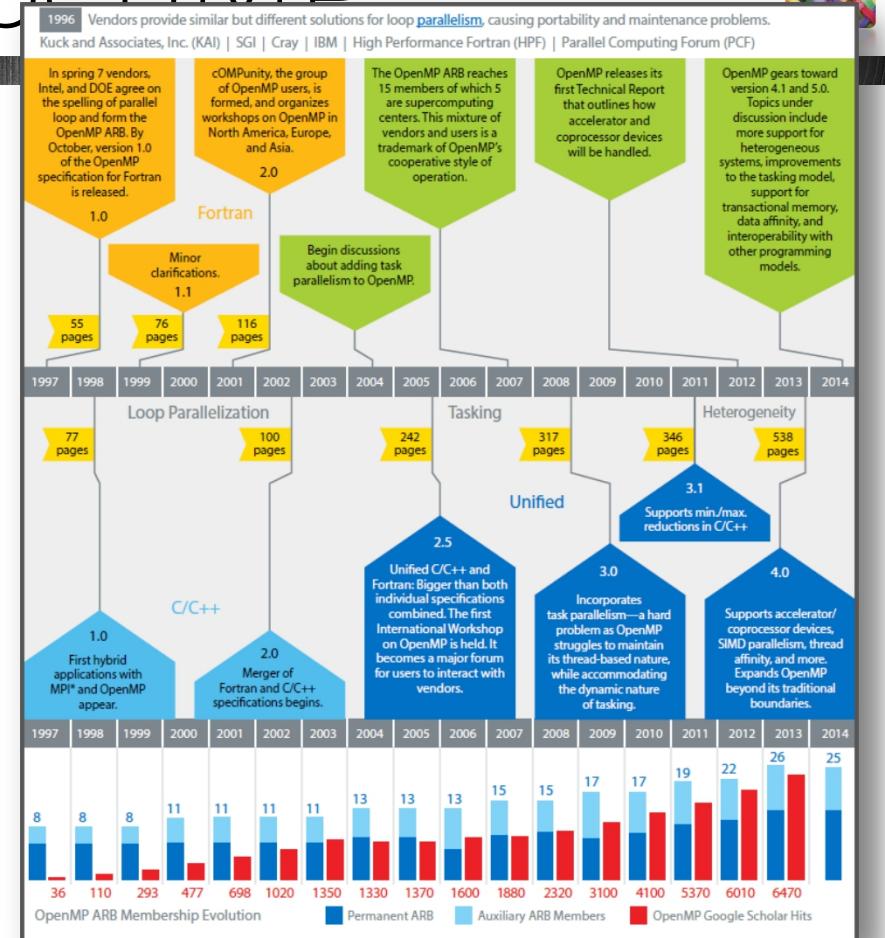
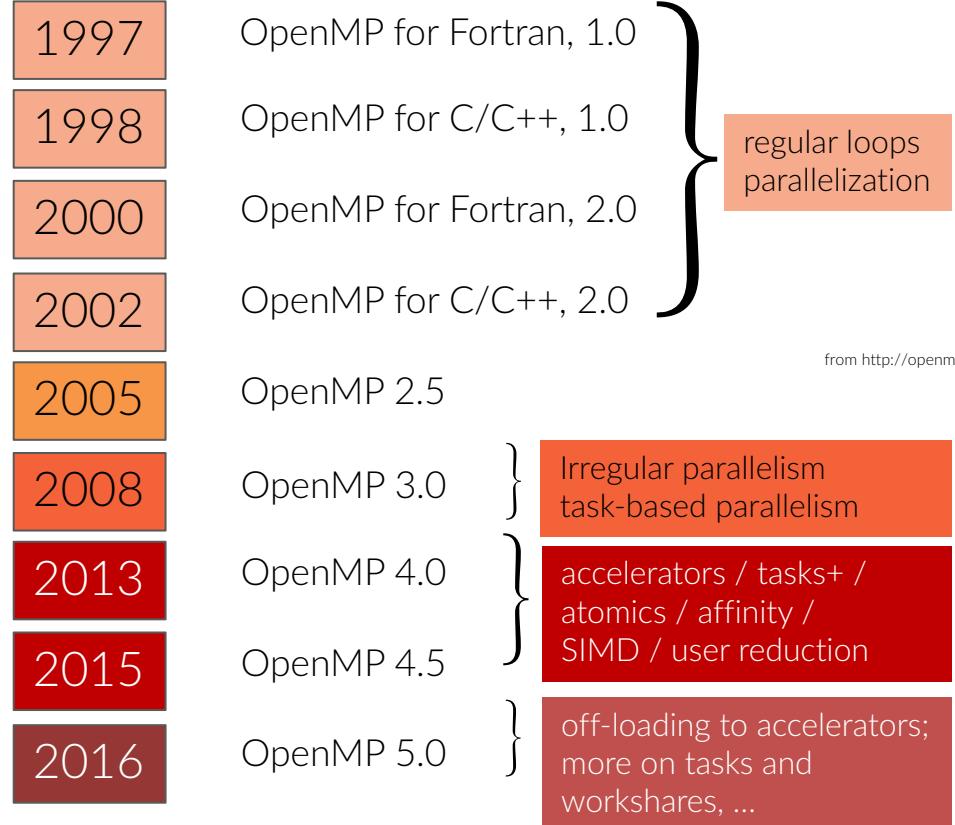
## Distributed-Memory (e.g. MPI) + Shared-Memory

N processes are created, each with its own copy of the code and its own memory space. Each process may spawn a number of threads as in shared-memory. A process *can not* access the memory space of another process (nor any of its threads can). The processes communicate through messages.





# What is OpenMP





# | What is OpenMP

Advantages of a directive-based approach

- **Abstraction**  
Subtleties of `pthread` and hardware-specific aspects are hidden. You can focus on data and workflow much more easily.
- **Efficiency.**  
The learning curve to achieve reasonable results is much shallower. The code's design is easier, the result/effort ratio is favourable with respect to `pthread`.
- **Incremental approach**  
No need to re-write your whole code. You start concentrating on some sections only, following the suggestions from profiling.
- **Portability.**  
The compiler will take care of this for you. You still have to develop a design able to adapt to different topologies.
- **One source**  
Through conditional compilation, serial and parallel versions can easily coexist.



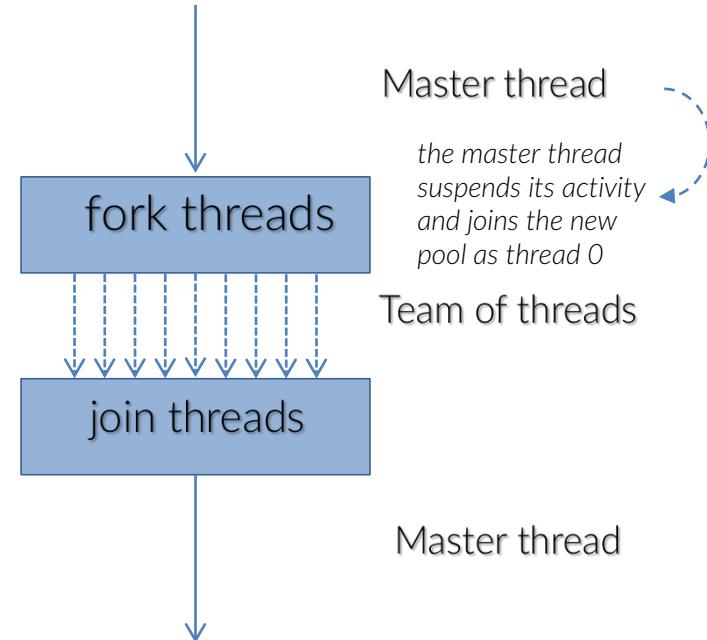
# OpenMP programming model



use a single master thread  
for serial operations

spawn a team of threads to  
perform parallel work

use a single master thread  
for serial operations

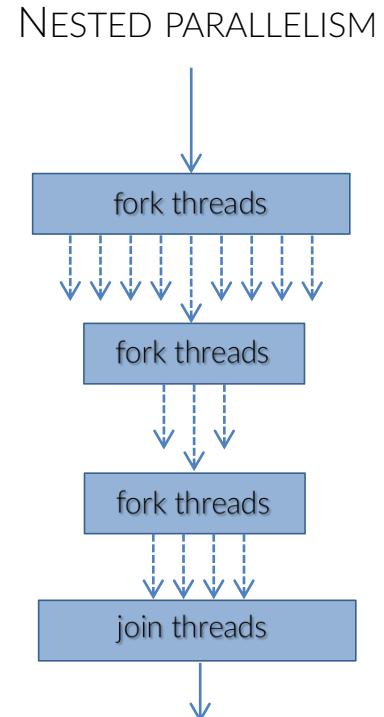


This is called “fork-join” model: a thread meets, at some point in its existence, a *directive* that activates the creation of a pool of children threads.



# OpenMP programming model

- Threads access and modify shared memory regions
  - explicit or implicit synchronization protect against race conditions
  - there is no concept like explicit “message-passing”
  - loop-carried dependencies hamper any parallel speedup
  - shared-variable attributes are vital to reduce or avoid race conditions or the need for synchronization
- Each thread performs its part of parallel work in a separate space and stack that are not visible to other threads or outside the parallel region
- Nested parallelism is explicitly permitted
- The number of threads can be dynamically changed before a parallel region





# | OpenMP directives

An OpenMP directive is a specially-formatted pragma for C/C++ and comment for FORTRAN codes.

Most of the directives apply to *structured code block*, i.e. a block with a single input and a single output points and no branch within it.

The directives allows to

- create team of threads for parallel execution
- manage the sharing of workload among threads
- specify which memory regions (i.e. variables) are shared and which are private to each threads
- drive the update of shared memory regions
- synchronize threads and determine atomic/exclusive operations

DECLARE PARALLEL REGION

**!\$OMP PARALLEL**

...

**!\$OMP END PARALLEL**

**#pragma omp parallel**

{

...

}



# Dynamic extent

As we have seen in the previous slide, the lexical scope of structured blocks defines the *static extent* of an OpenMP parallel region.

Every function call from within a parallel region determines the creation of a *dynamic extent* to which the same directives apply.

The dynamic extent includes the original static extent and all the instructions and further calls along the call tree.

The functions called in the dynamic extent can contain additional OpenMP directives.

```
#pragma omp parallel
{
    double *array;
    int N;
    ...
    sum = foo(array, N);
    ...
}

double foo( double *A, int N )
{
    double sum = 0;
    #pragma parallel for reduction(+:sum)
    for ( int ii = 0; ii < N; ii++ )
        sum += array[ii];
    return sum;
}
```

static extent

dynamic extent

“orphan” directive



# Dynamic extent

As we have seen in the previous slide, the lexical scope of structured blocks defines the *static extent* of an OpenMP parallel region.

Every function call from within a parallel region determines the creation of a *dynamic extent* to which the same directives apply.

The dynamic extent includes the original static extent and all the instructions and further calls along the call tree.

The functions called in the dynamic extent can contain additional OpenMP directives.

```
#pragma omp parallel
{
    double *array; ←
    int N; ←
    ...
    sum = foo(array, N);
    ...
}

double foo( double *A, ←int N )
{
    double sum = 0;
    #pragma parallel for reduction(+:sum)
    for ( int ii = 0; ii < N; ii++ )
        sum += array[ii];
    return sum;
}
```

static extent

These will be thread-specific

dynamic extent



# | OpenMP toolbox



OpenMP is made of 3 components:

- 1. Compiler directives**

give indication to the compiler about how to manage threads internals

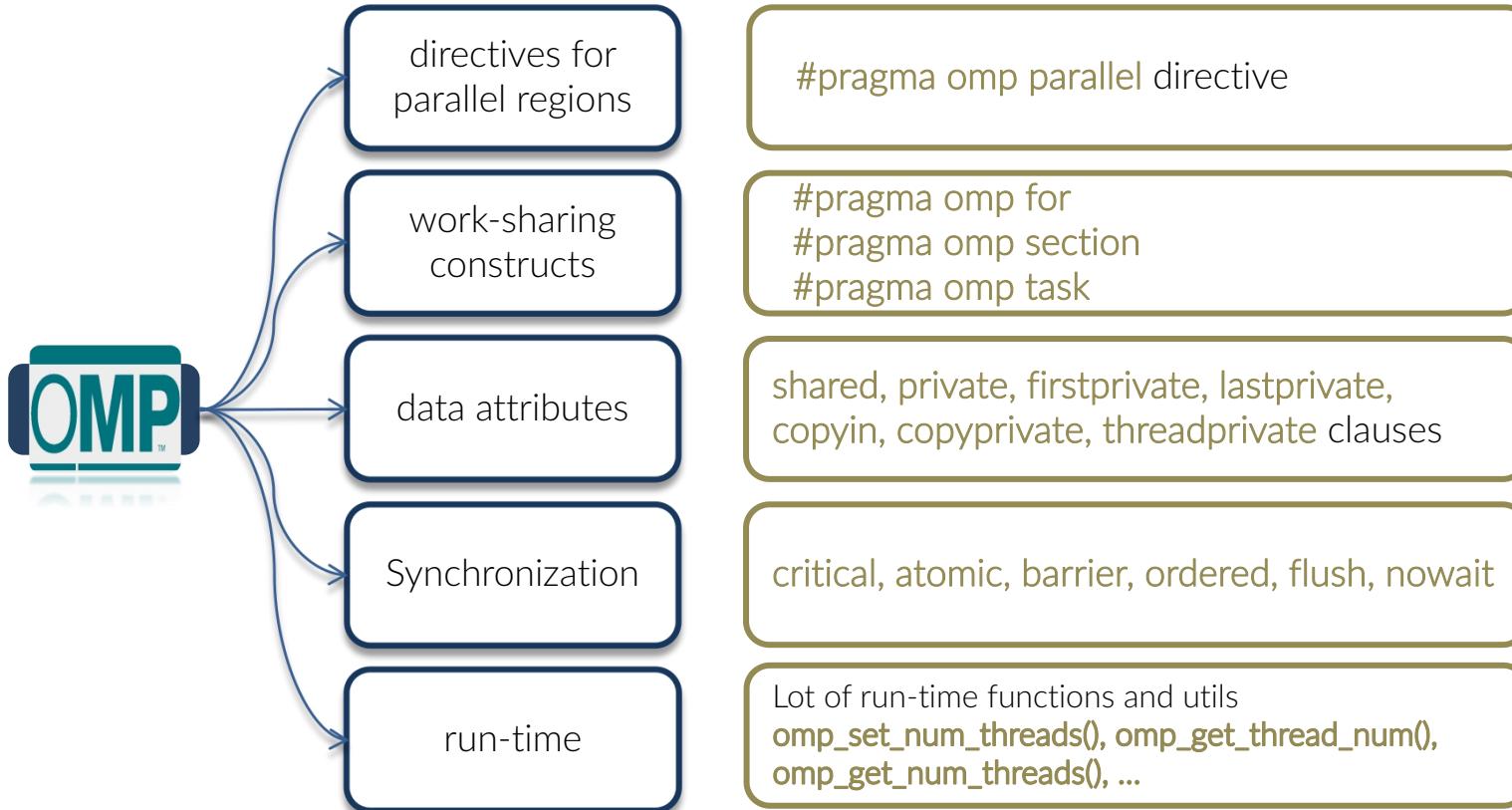
- 2. Run-time libraries** linked by the compiler

- 3. Environment variables**

set by the user, determine the behaviour of the omp library; for instance, the number of threads to be spawned or the requirements about the thread-cores-memory affinity



# | OpenMP toolbox





# | Conditional compilation

By default, when the compiler is instructed to activate the processing of OpenMP directives,

```
gcc -fopenmp ...
icc -fopenmp ...
pgcc -mp ...
```

it defines a macro that let you to conditionally compile sections of the code:

```
...
#define _OPENMP
...
...
#endif
...
...
```



omp\_versions.c : with this example, you see how to determine what version of OpenMP standard is supported by your compiler



# | Conditional compilation



What is this useful for?

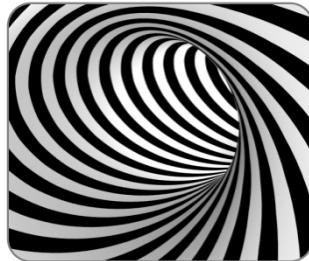
TIPS

To write code that works as well also without OpenMP.

That helps you in assessing the correctness and portability of your code (mostly if you are writing an hybrid code, for instance MPI+OpenMP).



# OpenMP Outline



Parallel  
Regions

Parallel  
Loops

Advanced  
Parallelism



that's all, have fun

"So long  
and thanks  
forall the fish"