Parallel Computing & OpenMP Introduction

Luca Tornatore - I.N.A.F.

"**Foundation of HPC" course**

DATA SCIENCE &
SCIENTIFIC COMPUTING

2019-2020 @ Università di Trieste

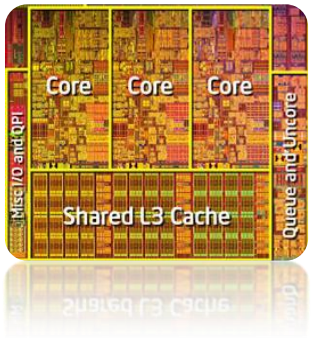
Outline

Introduction

Parallel Computing

Intro to basic OpenMP

Introduction 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 …

"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."

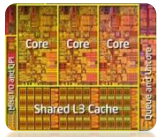
DOE's *Office of Science, 2012*

Why there is no more “free lunch”?

Applications no longer get more performance for free without significant re-design, 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)*



Race to Multicore

Why there is no more “free lunch”?

For instance:

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

False

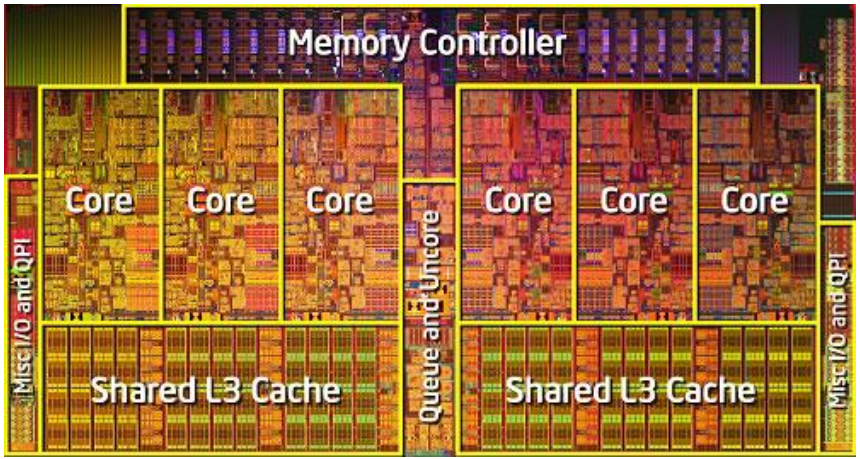
- × Cores coordination for cache-coherence
- × Threads coordination
- × Memory access
- × Increased algorithmic complexity

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)*

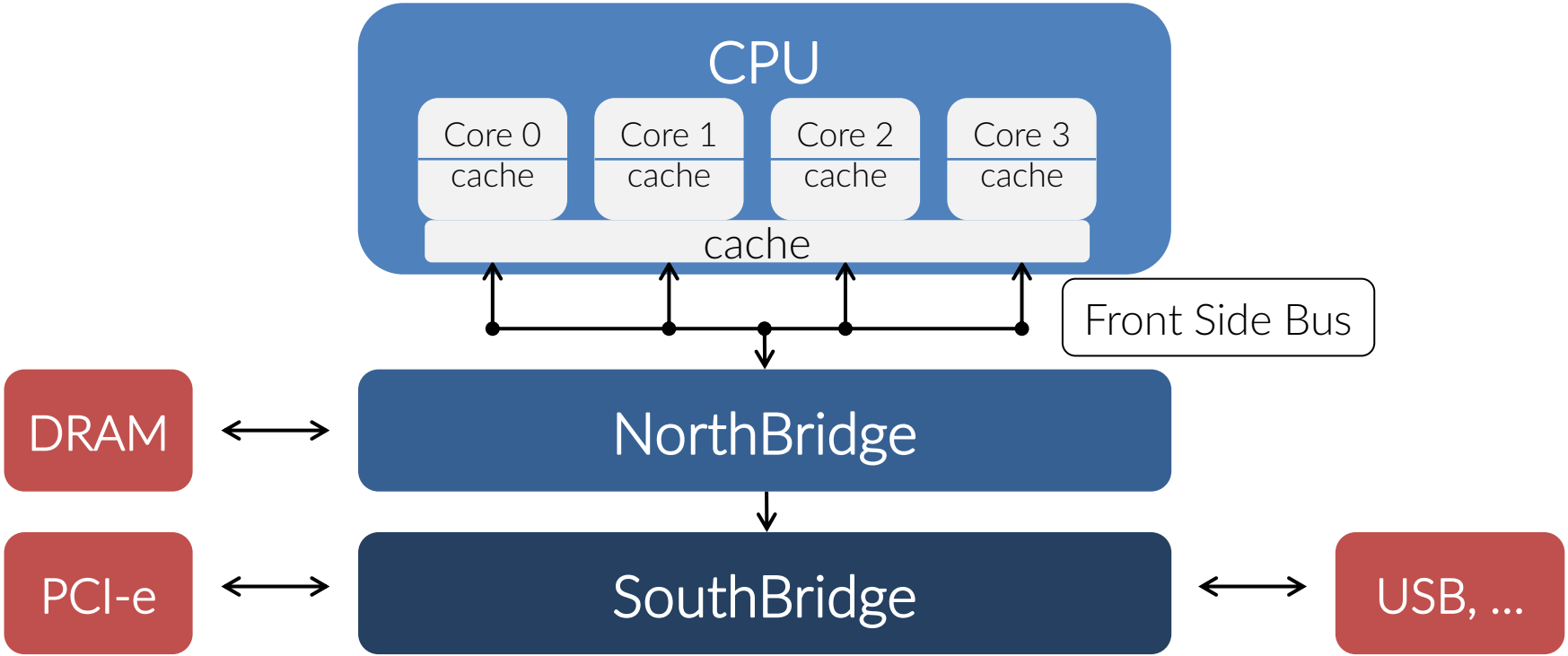
Back to the future

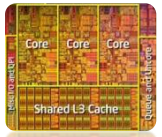
Message I
Many-cores CPUs are here to stay



- 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



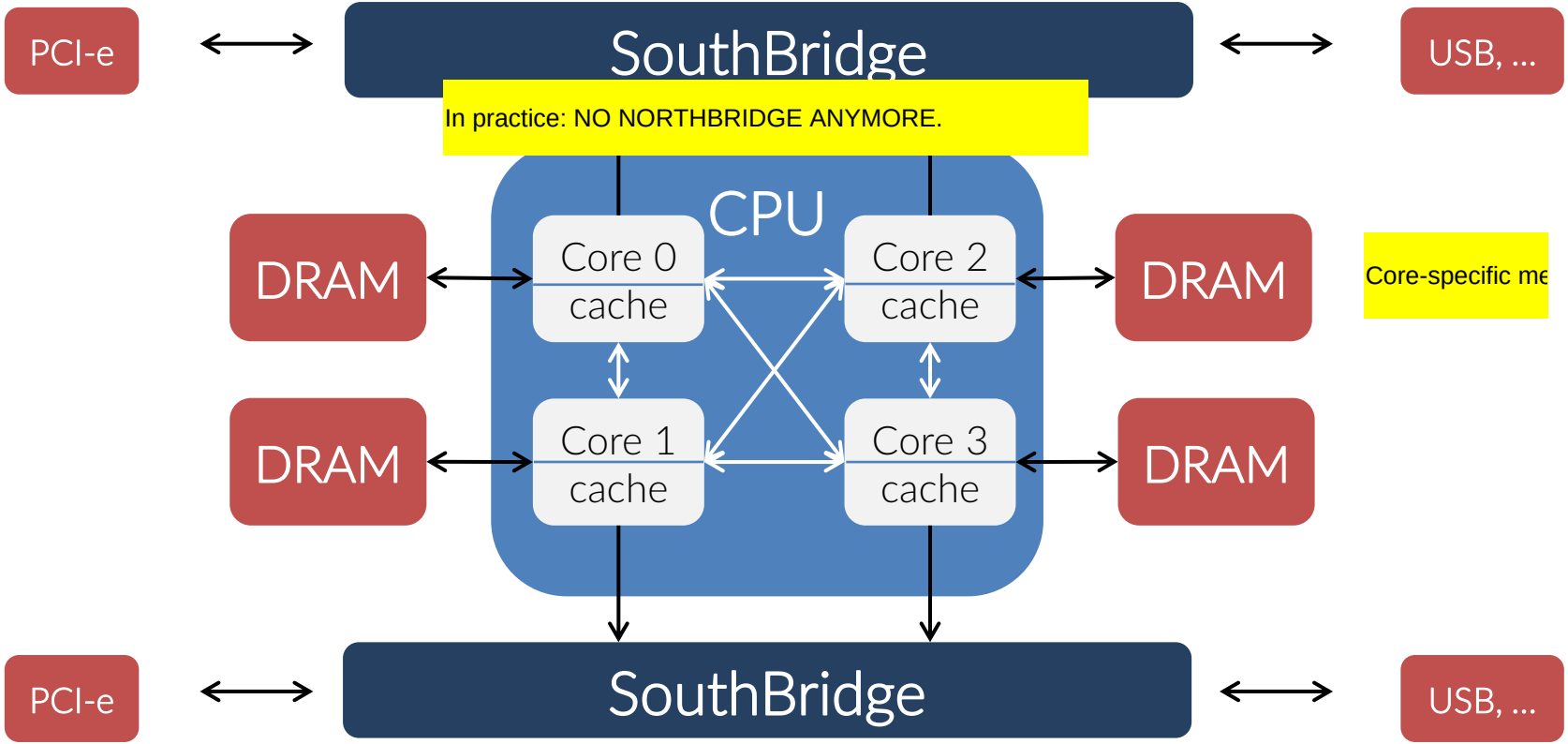


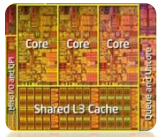
Race to Multicore

The typical UMA architecture

- The RAM can be accessed by one core at a time
  - this lower 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 are the main bottlenecks

The typical NUMA architecture



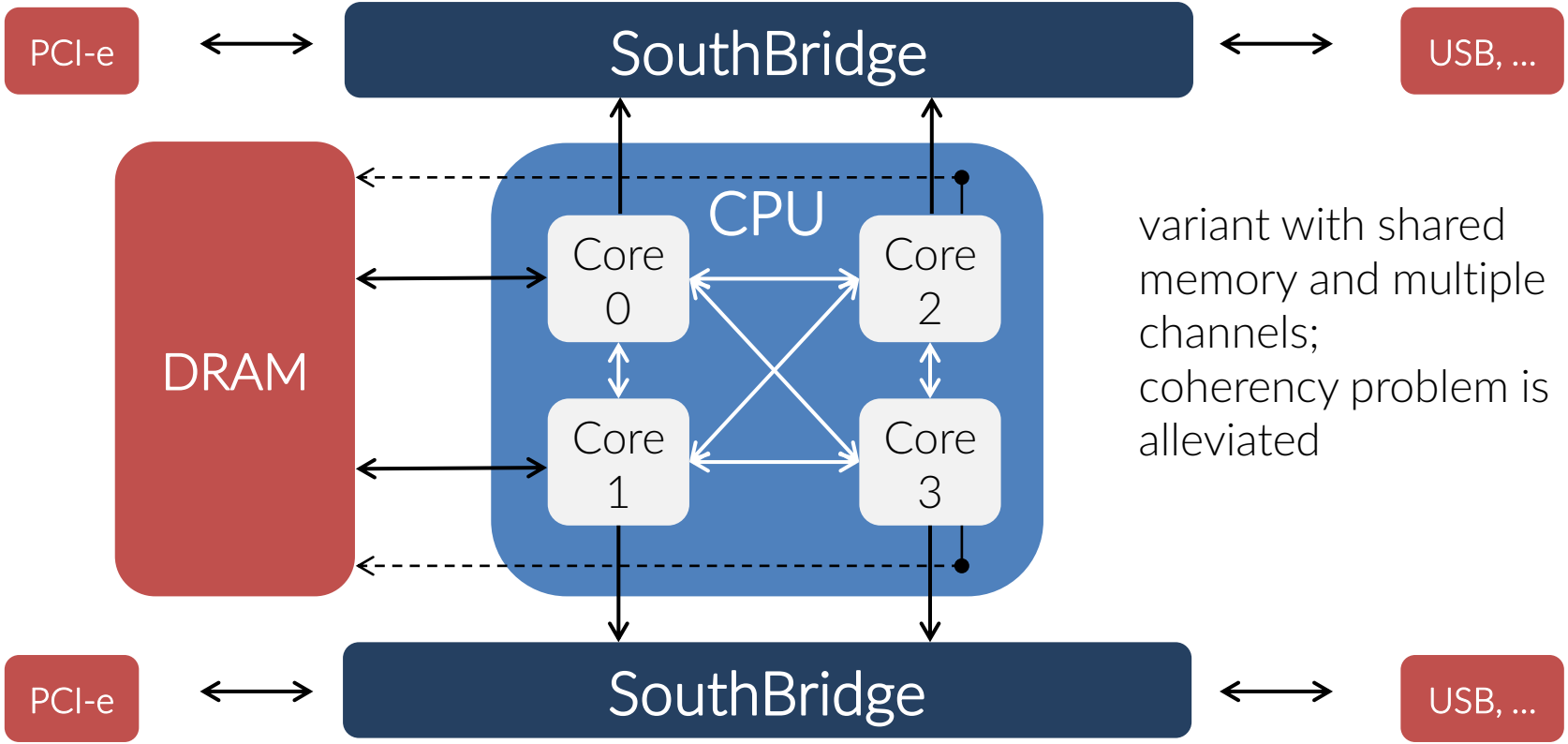


Race to Multicore

The typical NUMA architecture

- The bottleneck of resources (RAM and southbridge) access is eliminated
  - Each core *may* 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)
  - accessing DRAM has different costs

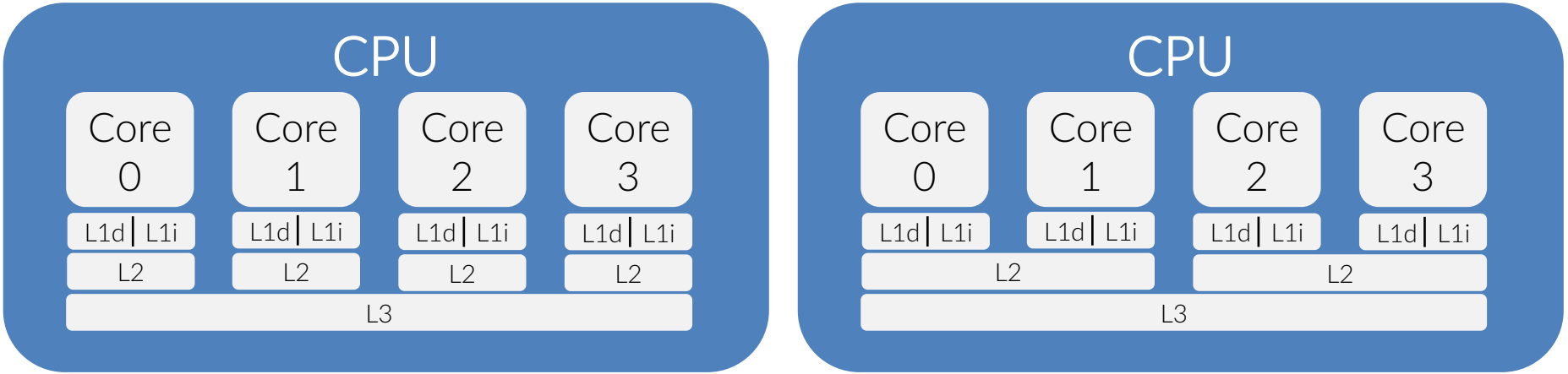
The typical NUMA architecture



variant with shared memory and multiple channels;
coherency problem is alleviated

The typical NUMA architecture

Cache hierarchy can have different topologies



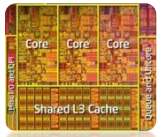
Cache coherence

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, the change must be propagate.

- 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

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

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

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

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)

"X" stands for a given memory location

Great powers, great responsability

A
Variables used by a single core
They should resides 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

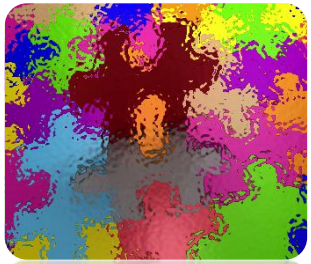
A cache line should contain only one type of data.

Put variables in the order they will be used.

Modified variables should stay together, they are the slowest ones.

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 offer *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

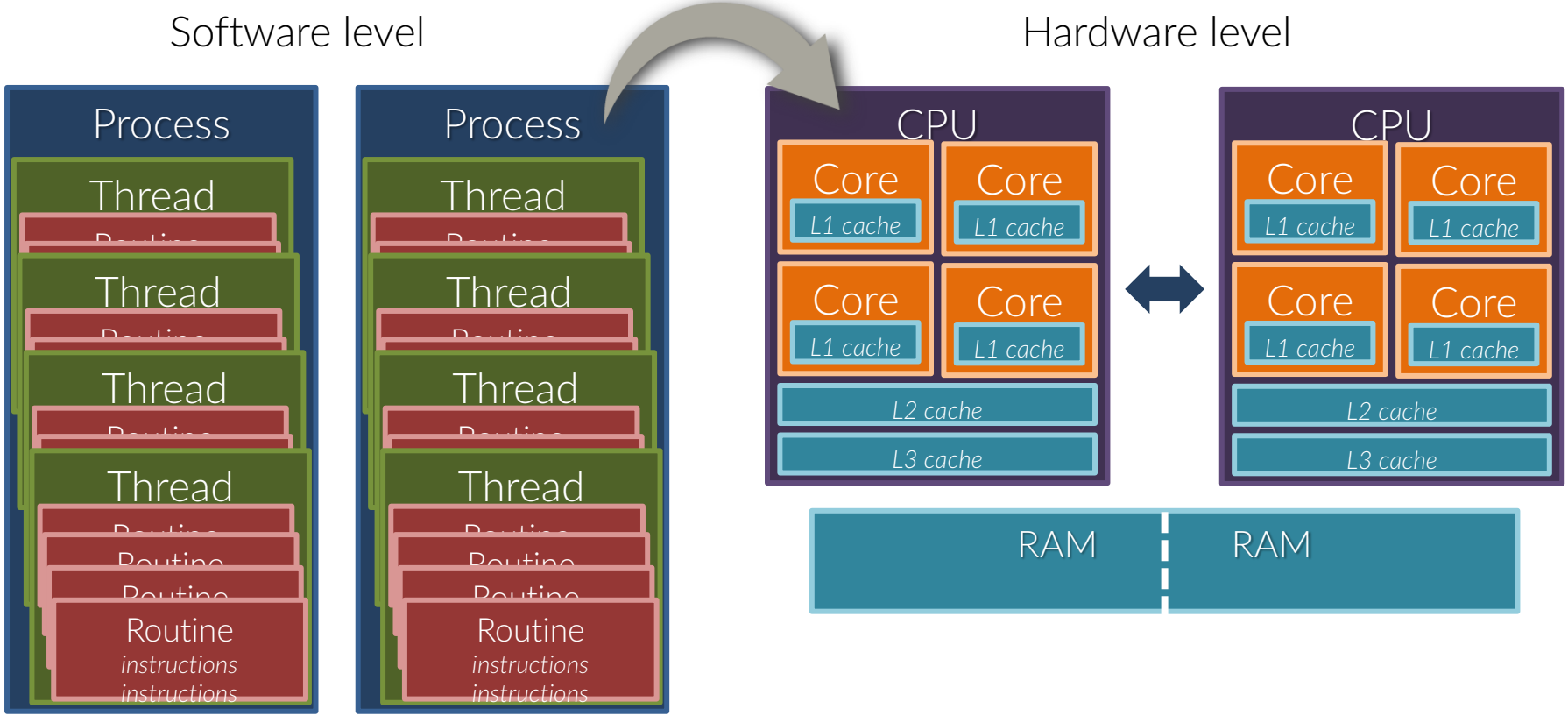
processors, cores, accelerators

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 ?



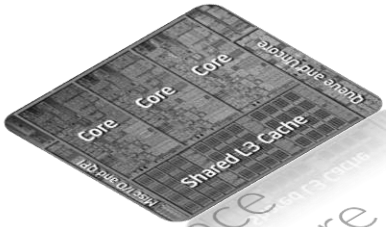
For two main reasons:

1. Time-to-solution
   - To solve the same problem in a smaller time
   - To solve a larger problem in the same time

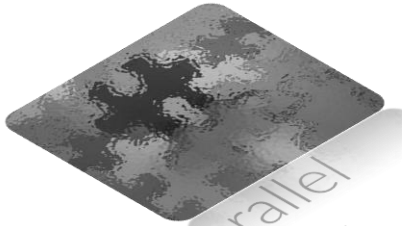
2. Problem size (~ data size)

   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)

Introduction Outline



The race to multicore



Intro to Parallel Computing

Parallel Performance

What is parallel performance

Has we have seen yesterday, «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.

Resources

Hardware

Time

Energy

Money

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

$$\text{Performance ratios} \approx \frac{resources_1}{resources_2}$$

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
$T_p(n)$	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 ~ *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

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

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

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

And then

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

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

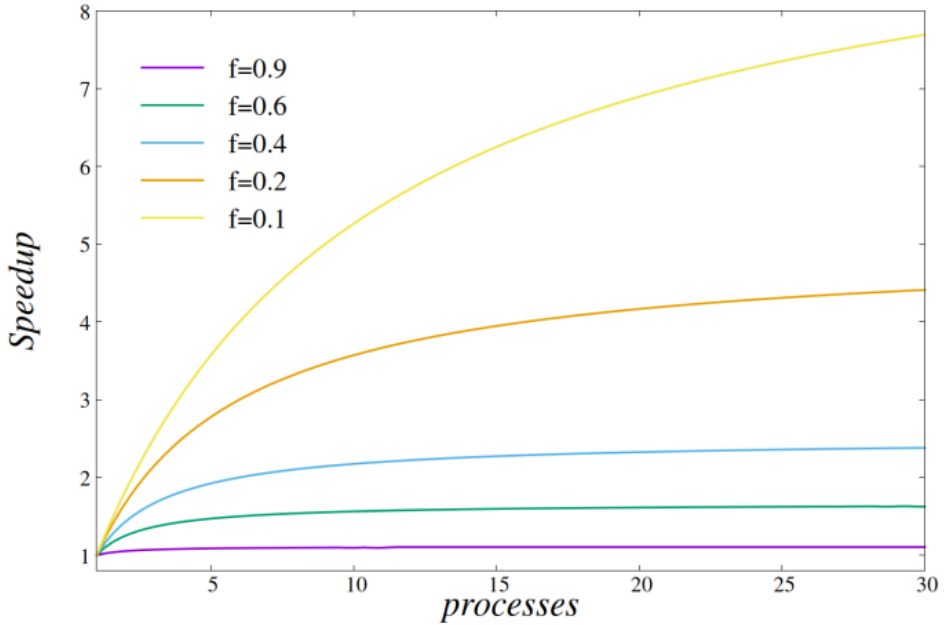
Amdahl's law

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



Amdahl's law

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 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

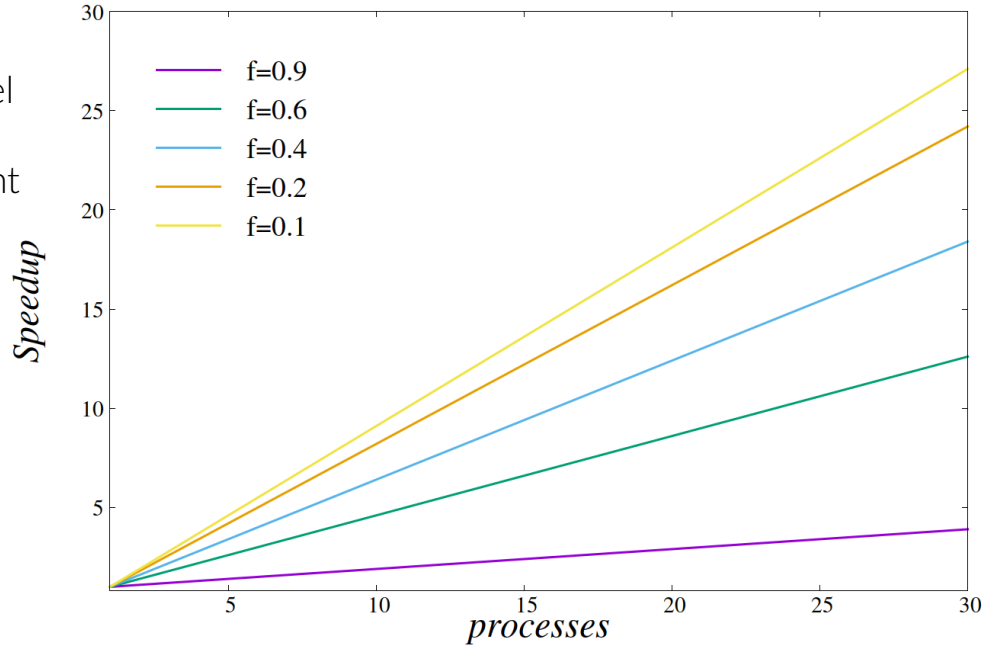
Then, usually the speedup increases with problem size

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
$T_P \propto a+b$ using p 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 they 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 waist 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: $$e(n,p) = \frac{\frac{1}{Sp(n,t)} - \frac{1}{p}}{1 - \frac{1}{p}}$$

Experimentally-measured fraction of time spent doing SERIAL

Let's check a couple of examples:

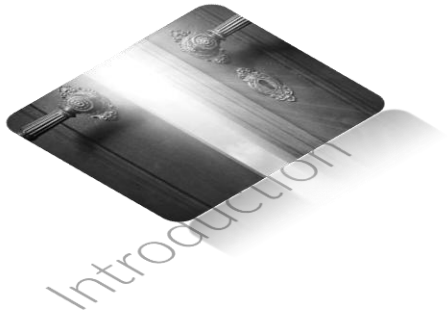
p	2	4	8	10	20
Sp(p)	1.69	2.6	3.52	3.79	4.49
E(p)	0.18	0.18	0.18	0.18	0.18

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

Outline



Intro to basic OpenMP

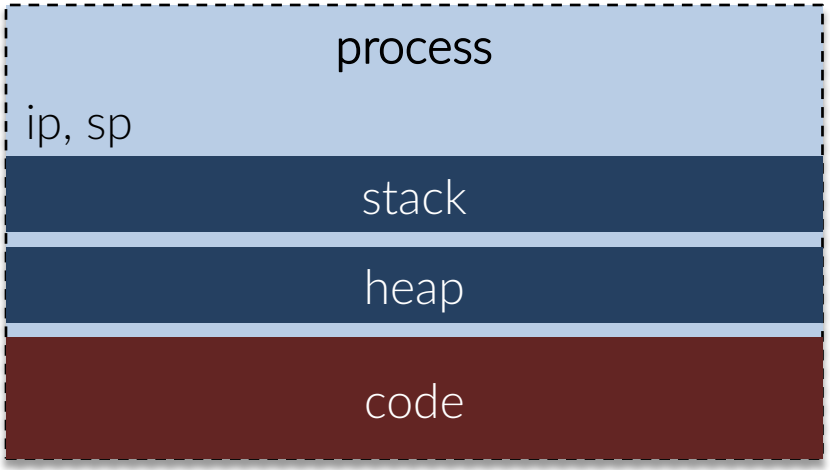
The difference between **threads** and **processes**

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).

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

SIMPLY PUT: Code is code; A PROCESS IS CO

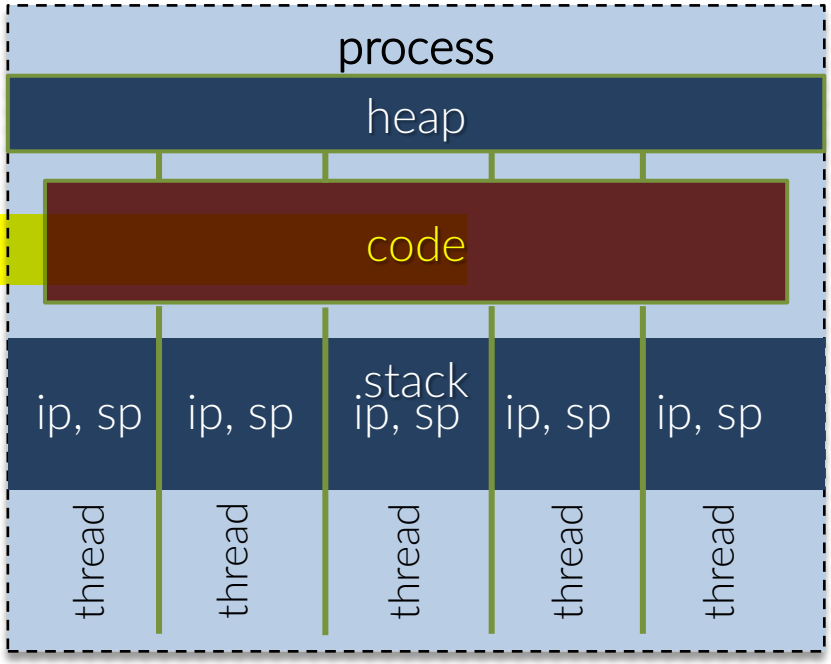


The difference between **threads** and **processes**

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 share 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*.

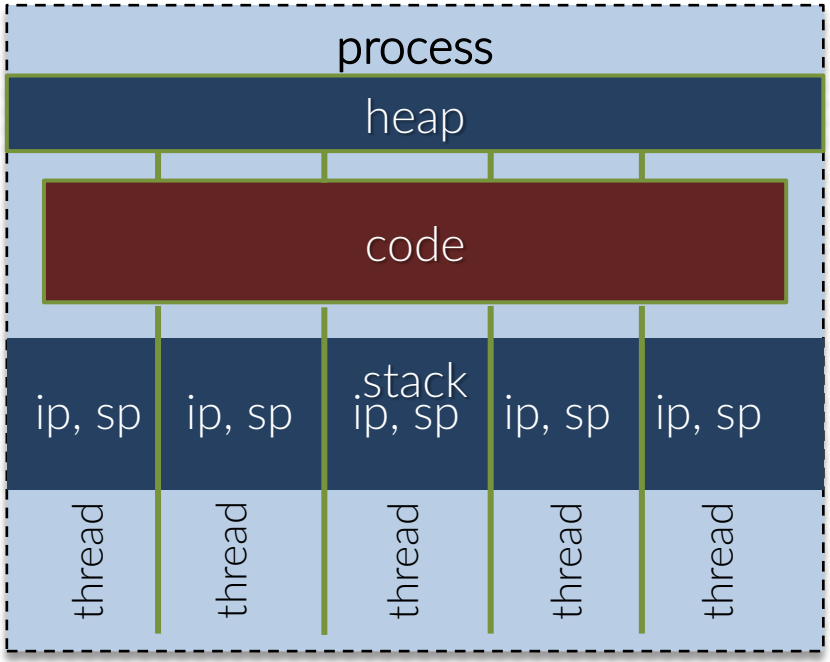
Spawning threads inside a process is much less costy than creating processes.



The difference between **threads** and **processes**

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

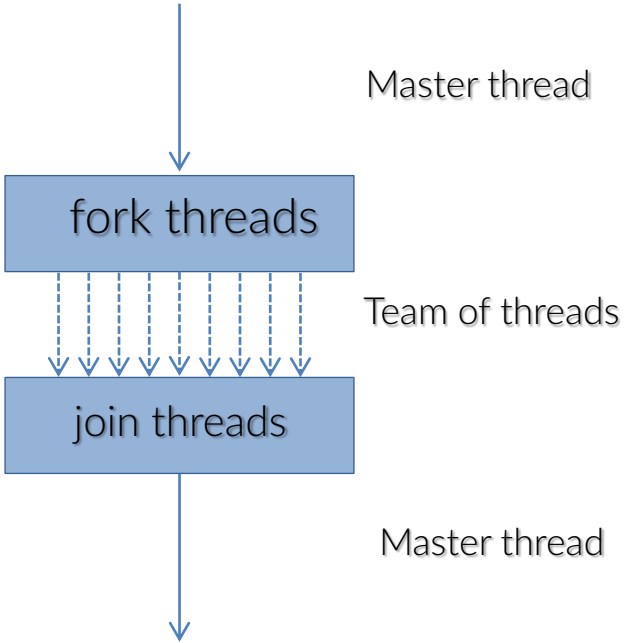
A computational unit nowadays amount to a **core**, either inside the same CPU (socket) on which the father process runs, or inside a sibling socket in the same NUMA region.



What is OpenMP

OpenMP is a standard API to enable shared-memory parallel programming; it allows to write multi-threaded programs, which

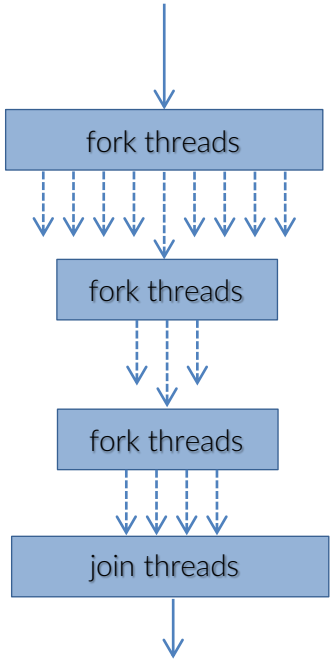
- use a single master thread for serial operations
- spawn a team of threads to perform parallel work
- use a single master thread for serial operations



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 perform its part of parallel work in a separate space and stack that are not visible to other threads and outside the parallel region
- Nested parallelism is explicitly permitted
- The number of threads can be dynamically changed before a parallel region

NESTED PARALLELISM



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

OpenMP toolbox

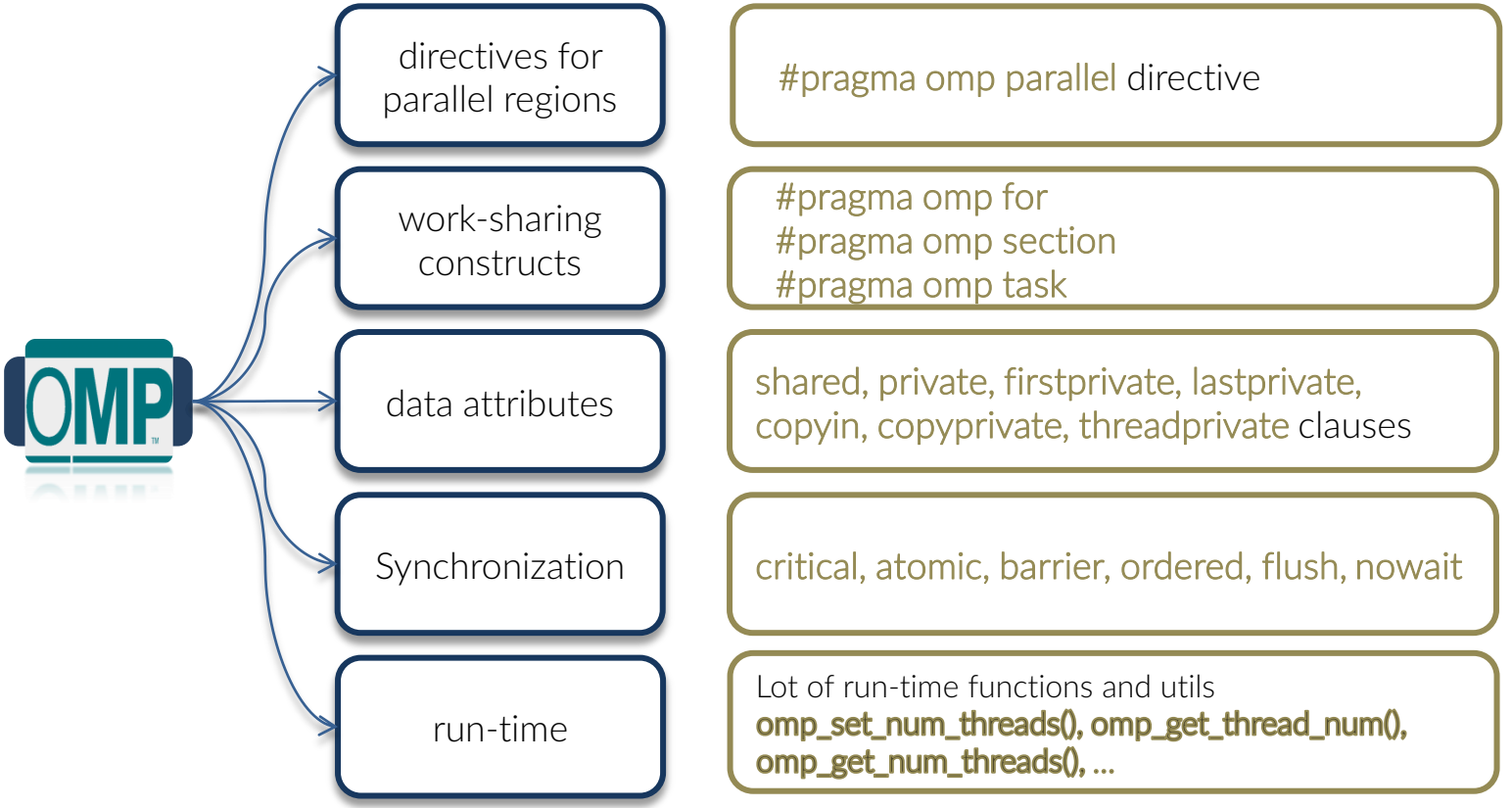
OpenMP is made of 3 components:

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

```
gcc -fopenmp …
icc -qopenmp …

int nthreads     = 1;
int my_thread_id = 0;

#ifdef _OPENMP
#pragma omp parallel
{
    my_thread_id = omp_get_thread_num();
#pragma master
    nthreads     = omp_get_num_threads();
}
#endif
```

Right thing to do!

OpenMP basic loop

Let's start with a classical and very common problem.

```
double *a;
double  sum = 0
int     N;
…
for ( int i = 0; i < N; i++ )
      sum += a[i];
```

OpenMP basic loop

```
#include <omp.h>
double *a, sum = 0;
int     i, N;

#pragma omp parallel for implicit(none) shared(a,b,c,N) private(i)
for ( i = 0; i < N; i++ )
      sum += a[i];
```

declares what variables are private: despite their name is the same within the parallel region, they have different memory locations and die with the parallel reg.

Ex. 00

This is a **work-sharing** construct; workload is subdivided among threads (the default choice is implementation-dependent)

no implicit assumptions about variables scope

declares what variables are shared; all threads can access and modify those memory locations

OpenMP basic loop

However, variables defined within the parallel region are automatically private, and so are the integer indexes used as cycles counter:

```
#include <omp.h>
double *a, sum = 0;
int     N;

#pragma omp parallel for implicit(none) shared(a,sum,N)
for ( int i = 0; i < N; i++ )
      sum += a[i];
```

How the work is assigned to single threads ?

Ex. 01

OpenMP work assignment in loops

How the work is assigned to single threads ?

```
#pragma omp parallel for schedule(scheduling-type)
for ( int i = 0; i < N, i++ )
```

schedule(**static**, *chunk-size*)	The iteration is divided in chunks of size *chunk-size* (or in ~equal size) distributed to threads in circular order
schedule(**dynamic**, *chunk-size*)	The iteration is divided in chunks of size *chunk-size* (or size 1) distributed to threads in no given order (a thread requests the first available chunks)
schedule(**guided**, *chunk-size*)	The iteration is divided in chunks of minimum size *chunk-size* (or size 1) distributed to threads in no given order like *dynamic*. The chunk size is proportional to the number of unassigned iterations divided by the number of threads.
runtime , default	The policy is set at runtime via env. OMP_SCHEDULE or to intern. var. def-sched-var.

OpenMP control parallel regions

It is possible to modify the behaviour of the parallel regions by run-time omp routines:

- `#pragma omp parallel if( any valid C expression )`
- `#pragma omp parallel num_threads(n)`
-
```
omp_set_num_threads(n);
#pragma omp parallel
```

Clauses in *parallel for*

```
#pragma omp for
        schedule( policy [,chunk])
        ordered
        private ( var list )
        firstprivate ( var list )
        lastprivate (var list )
        shared ( var list )
        reduction ( op: var list )
        collapse (n)
        nowait
```

Clauses in *parallel for*

`private ( var list )`

vars in the list will be private to each thread; despite their name is the same out of the parallel region, they have different memory locations and die with the parallel region.

`firstprivate ( var list )`

the variables in the list are private (in the same sense than in *private*) and are initialized at the value that shared variables have at the begin of the parallel region.

`lastprivate ( var list )`

the shared variables will have the value of the private var in the last thread that ends the work in the parallel region.

Clauses in *parallel for*

`reduction ( op: var list )`

Possible operators are: +, ×, -, max, min, &, &&, |, ||
The initial value of vars is taken into account *at the end* of the parallel for; at the begin of the for, initialization values are what you logically expect: 0 for add, 1 for mul, min and max of the result type for max and min.

`collapse ( n )`

Enable the parallelization of multiple loops level

```
!$omp parallel do collapse(2) schedule (guided)
 do j = 1, n, p
   do i = m1, m2, q
      call dosomething (a, i, j)
   end do
 enddo
```

`nowait`

Ignore the implicit barrier at the end of parallel region or work-sharing construct

SPMD & Work-sharing

- A parallel construct amounts to create a “Single Program Multiple Data” instance: all the threads execute the same code but on different data.
- The work-sharing constructs is instead about assigning different execution paths through the code among the threads.
  - **`section`** construct
  - **`single`** construct
  - **`tasks`**

Back to our OpenMP basic loop

The instruction

```
sum += a[i];
```

- Determine a **data race**: between two synchronization points at least one thread writes to a data location from which another threads reads

```
#include <omp.h>
double *a, sum = 0;
int     N;

#pragma omp parallel for implicit(none) shared(a,sum,N)
for ( int i = 0; i < N; i++ )
      sum += a[i];
```

Back to our OpenMP basic loop

The instruction

```
sum += a[i];
```

- Determine a **data race**: between two synchronization points at least one thread writes to a data location from which another threads reads

```
#include <omp.h>
double *a, sum = 0;
int     N;

#pragma omp parallel for implicit(none) shared(a,sum,N)
for ( int i = 0; i < N; i++ )
      sum += a[i];
```

Solving data race

Let's solve the data race:

```
#include <omp.h>
double *a, sum = 0;
int     N;

#pragma omp parallel for implicit(none) shared(a,sum,N)
for ( int i = 0; i < N; i++ )
#pragma omp critical local_sum
      sum += a[i];
```

#pragma omp critical *optional_name*

builds a region in which only 1 thread at a time can execute code.

#pragma omp atomic

does the same for a single line

Solving data race

Does this scale ?

Ex. 02a

```
#include <omp.h>
double *a, sum = 0;
int     N;

#pragma omp parallel for implicit(none) shared(a,sum,N)
for ( int i = 0; i < N; i++ )
#pragma omp critical local_sum
      sum += a[i];
```

Solving data race

Does this scale ?

Ex. 02a

```
#include <omp.h>
double *a, sum = 0;
int     N;

#pragma omp parallel for implicit(none) shared(a,sum,N)
for ( int i = 0; i < N; i++ )
#pragma omp critical local_sum
      sum += a[i];
```

Of course no! why?

Of course no! why?

Because this solution makes the threads to wait for each other too frequently.

- A critical region has **synchronization points** at the start and the end of critical regions, meaning that threads have to communicate with each other and decide who's waiting and who's not.

Other **sync points** are implicit and explicit barriers, locks and flush directives.

Solving data race

However, that is so damn important that of course there is a simple solution:

Ex. 02

```
#include <omp.h>
double *a, sum = 0;
int     N;

#pragma omp parallel for reduction(+: sum)
for ( int i = 0; i < N; i++ )
   sum += a[i];
```

Note that *shared* clause has disappeared, implicit assumptions are ok for us

Solving data race /2

There is another way in which we can solve the data race

Ex. 03

```
#include <omp.h>
double *a;
int N;

int nthreads;
#pragma omp master
Nthreads = omp_get_num_threads();

double sum[nthreads];

#pragma omp parallel
{
   int me = omp_get_thread_num()
   for ( int i = 0; i < N; i++ )
      sum[me] += a[i];
}
```

Does this scale ?

Note the directive
#pragma omp master
#pragma omp single
would have worked as well here

Solving data race /2

There is another way in which we can solve the data race

Ex. 03

```
#include <omp.h>
double *a;
int N;

int nthreads;
#pragma omp master
Nthreads = omp_get_num_threads();

double sum[nthreads];

#pragma omp parallel
{
   int me = omp_get_thread_num()
   for ( int i = 0; i < N; i++ )
      sum[me] += a[i];
}
```

Does this scale ?

Hardly

Because **sum[nthreads]** resides in the same cache line; hence, when a threads access and modify its location, to maintain the coherence the cache must write-back and reflush.
Every time.

That is called **false sharing**

Solving false sharing

There is another way in which we can solve the data race

Ex. 03b

```
#include <omp.h>
double *a;
int N;

int nthreads;
#pragma omp master
Nthreads = omp_get_num_threads();

double sum[nthreads*8];

#pragma omp parallel
{
   int me = omp_get_thread_num()
   for ( int i = 0; i < N; i++ )
      sum[me*8] += a[i];
}
```

Does this scale ?
Better.

However, we are using “a lot” of memory and, above all, we hard-coded a magic number. That is not a good solution.

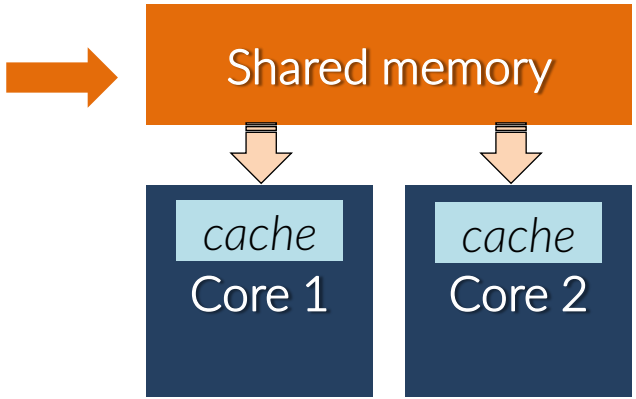
“touchfirst” policy

The matter is: who “belongs” the data?

```
double *a = (double*)calloc( N, sizeof(double);

for ( int i = 0; ii < N; ii++ ) {
   a[i] = initialize(i);

#pragma omp parallel for reduction(+: sum)
for ( int i = 0; i < N; i++ )
      sum += a[i];
```



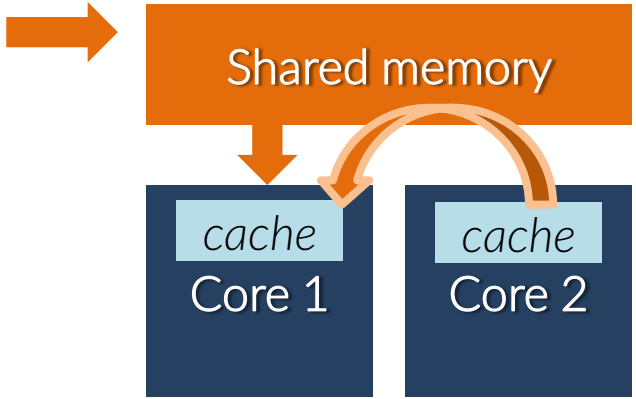
"touchfirst" policy

The matter is: who "belongs" the data?

```
double *a = (double*)calloc( N, sizeof(double);


for ( int i = 0; ii < N; ii++ ) {
   a[i] = initialize(i);

#pragma omp parallel for reduction(+: sum)
for ( int i = 0; i < N; i++ )
      sum += a[i];
```



In this way, the cache of the thread that initialize (first touch) the data is warmed-up

"touch by all" policy

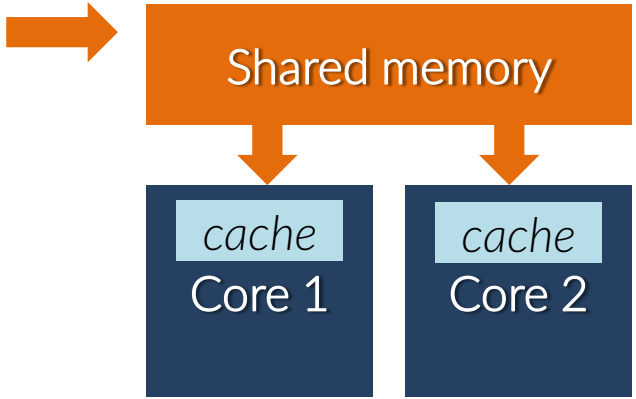
The matter is: who "belongs" the data?

```
double *a = (double*)calloc( N, sizeof(double);

#pragma omp parallel for
for ( int i = 0; ii < N; ii++ ) {
   a[i] = initialize(i);

#pragma omp parallel for reduction(+: sum)
for ( int i = 0; i < N; i++ )
      sum += a[i];
```

Ex. 04



In this way, the cache of each thread is warmed-up with the data it will use afterwards (the scheduling must be the same!)

Global private

```
double *global_pointer;
double  global_damnimportant_variable

#pragma omp threadprivate(global_pointer, global_damnimportant_variable)
```

Threadprivate preserve the global scope of the variable, but make it private for every thread.
Basically, every thread has its own copy if it everywhere you create a thread team.

End of basic

There is way more in OpenMP than this very brief introduction could unveil.

However, you now have a basic toolbox that however allows you to start using OpenMP on more demanding loops in your codes.

that's all, have fun