Parallel Computing & OpenMP Introduction

Luca Tornatore - I.N.A.F.

"**Foundation of HPC" course**

DATA SCIENCE &
SCIENTIFIC COMPUTING

2019-2020 @ Università di Trieste

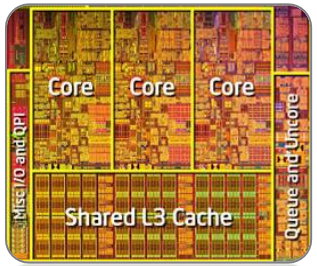
Outline

Introduction

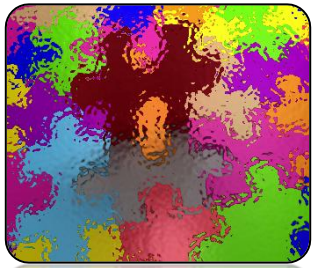
Parallel Computing

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."

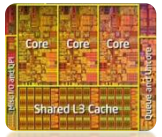
DOE's *Office of Science, 2012*

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

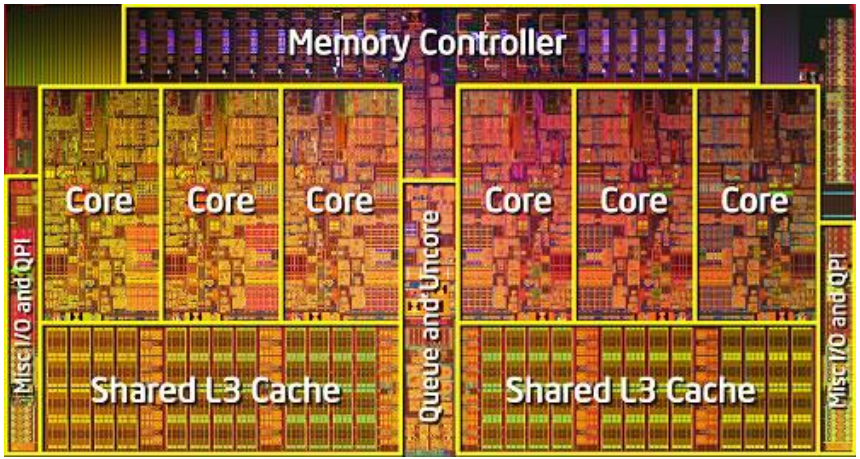
- × Cores coordination for cache-coherence
- × Threads coordination
- × Memory access
- × Increased algorithmic complexity

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)*

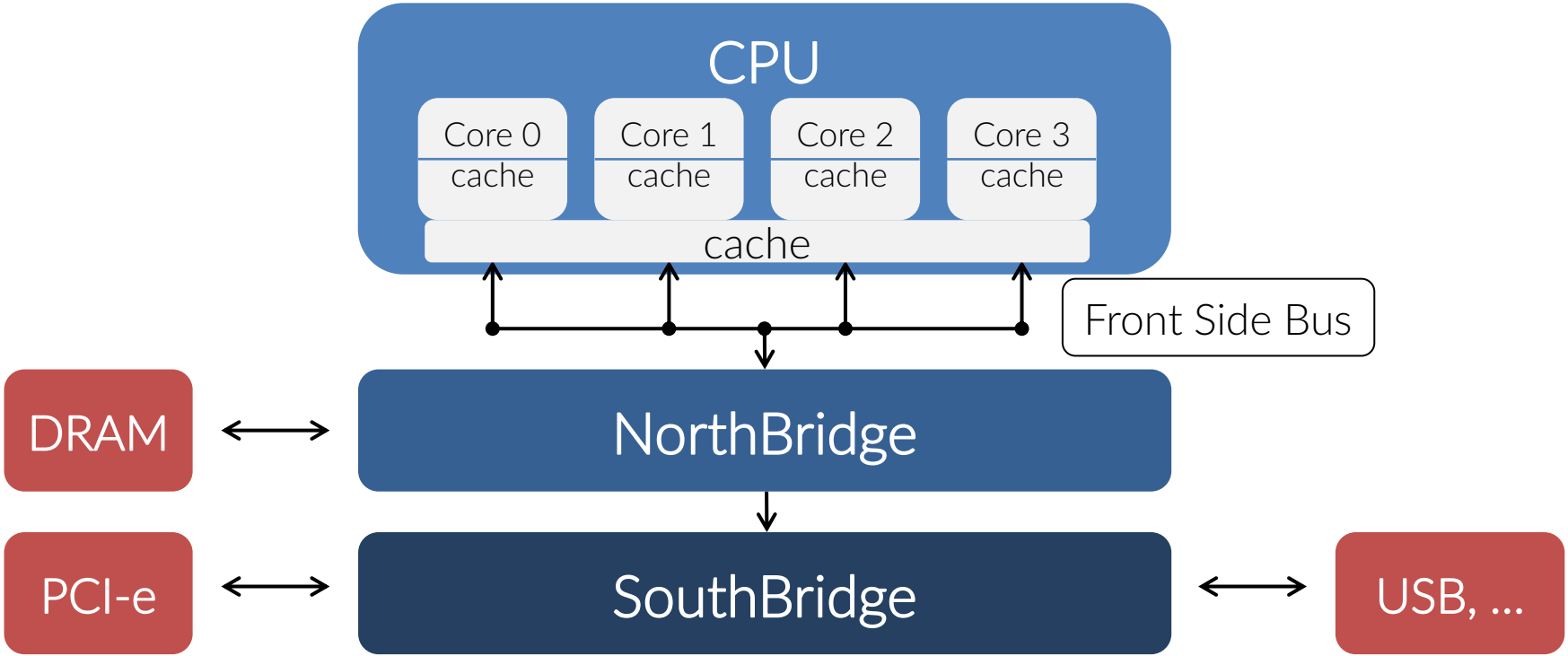
Back to the future

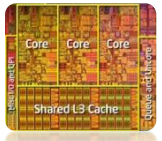
Message I
Many-cores CPUs are here to stay



- 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

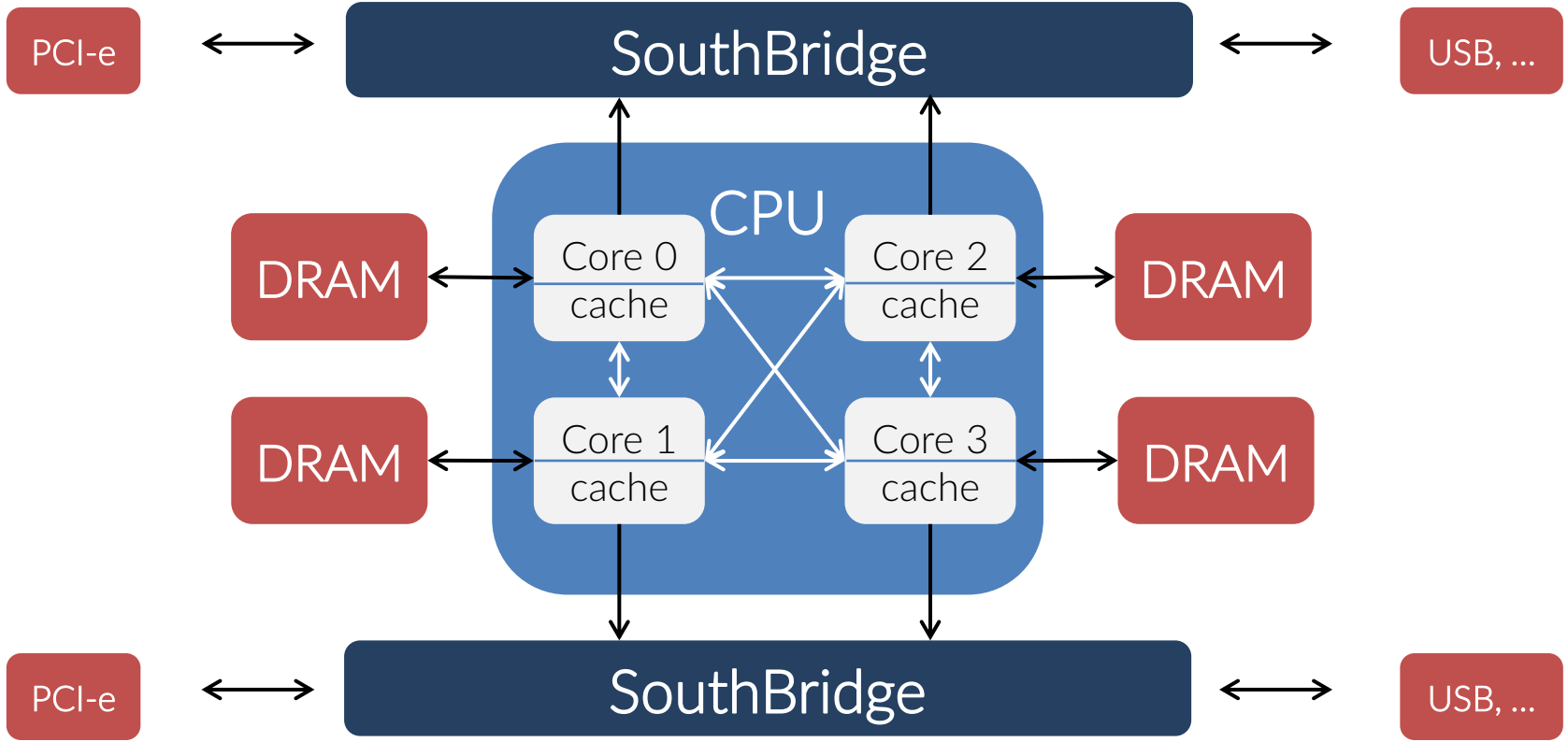




The typical UMA architecture

- 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

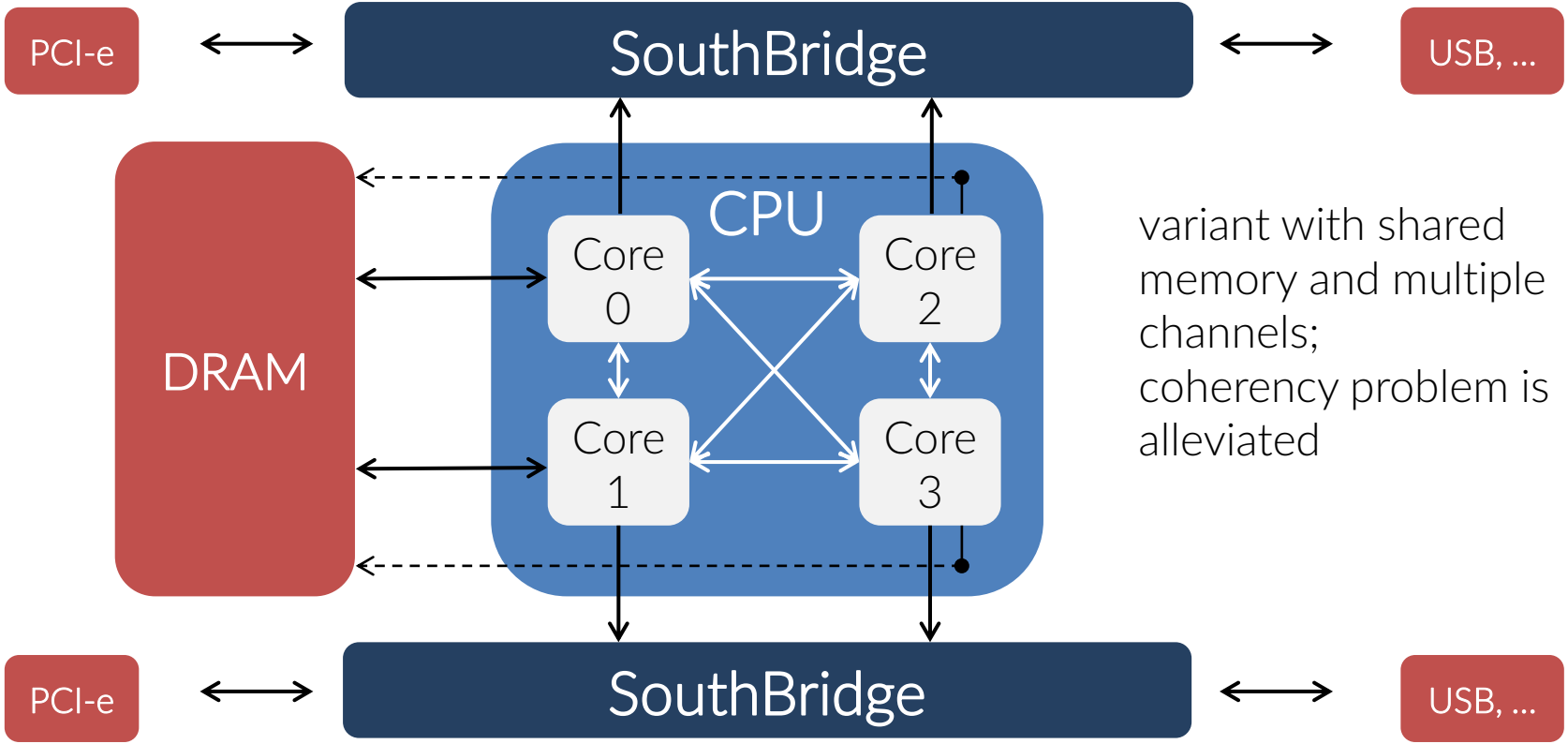
The typical NUMA architecture



The typical NUMA architecture

- The bottleneck of resources (RAM and southbridge) access is eliminated
  - Each core *may* 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

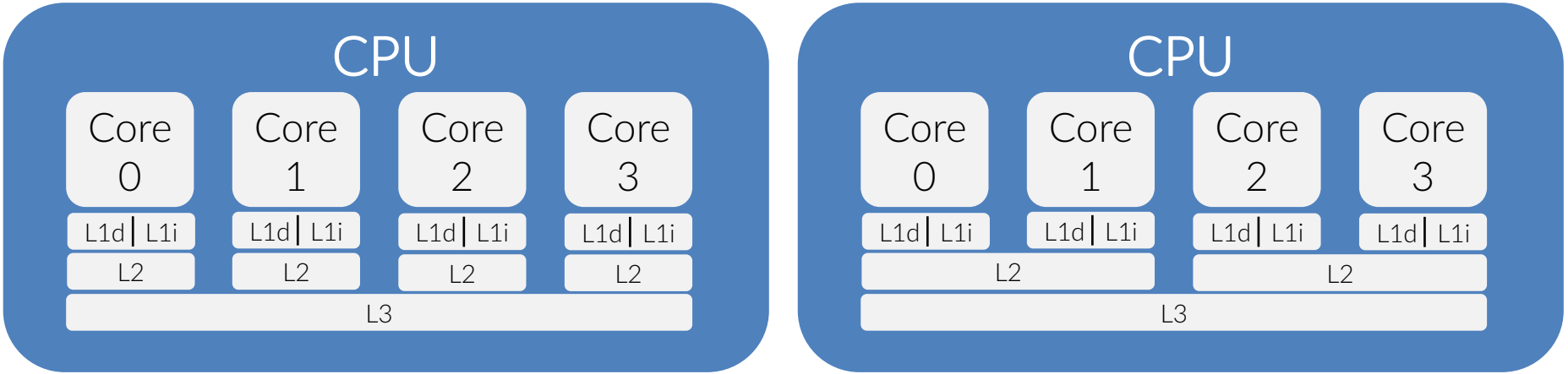
The typical NUMA architecture



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

The typical NUMA architecture

Cache hierarchy can have different topologies



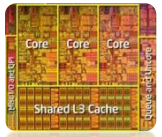
Cache coherence

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: MESI

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

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

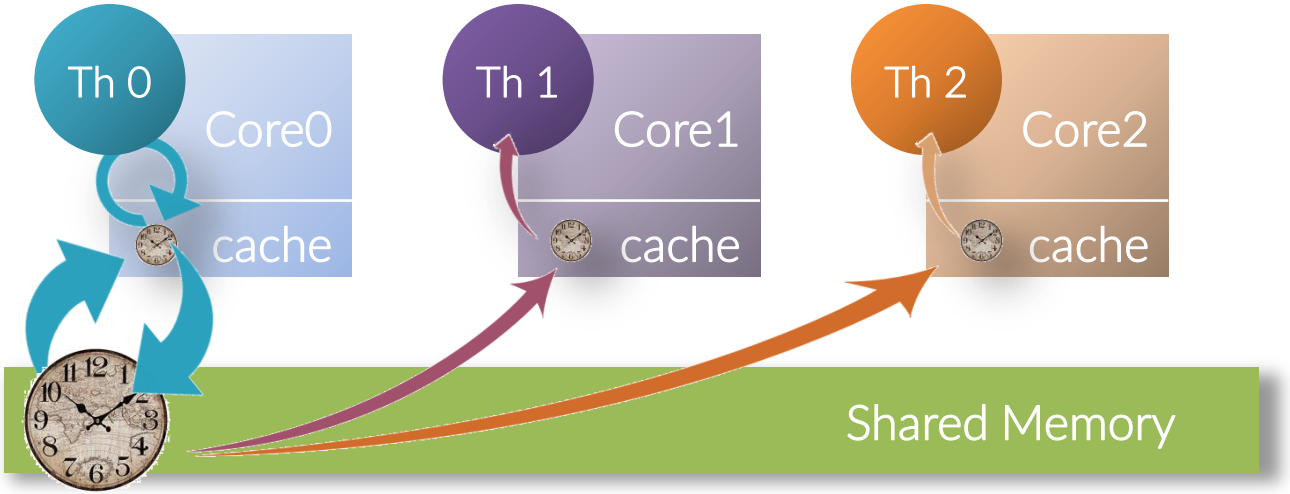
see:
MD64 Architecture Programmer's Manual
Volume 2: System Programming

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

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

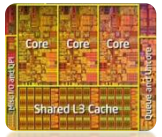
		Time (in secs)	Action	cache status		
M	Modified			Core0	Core1	Core2
E	Exclusive	0	-	I	I	I
S	Shared	1	Th0 reads	E	I	I
I	Invalid	2	Th0 writes[0]	E	I	I
		2.3	Th1 reads[1]	S	S	I
		2.7	Th2 reads	S	S	S
		3	Th0 writes[2]	M	I	I
		4	Th0 writes	M	I	I
		4.4	Th2 reads[1]	S	S	S
		5	Th0 writes	M	I	I
		…	…	…	…	…

[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".
0'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 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

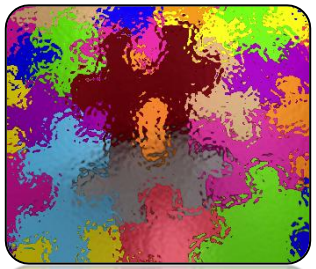
A cache line should contain only one type of data.

Put variables in the order they will be used.

Modified variables should stay together, they are the 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

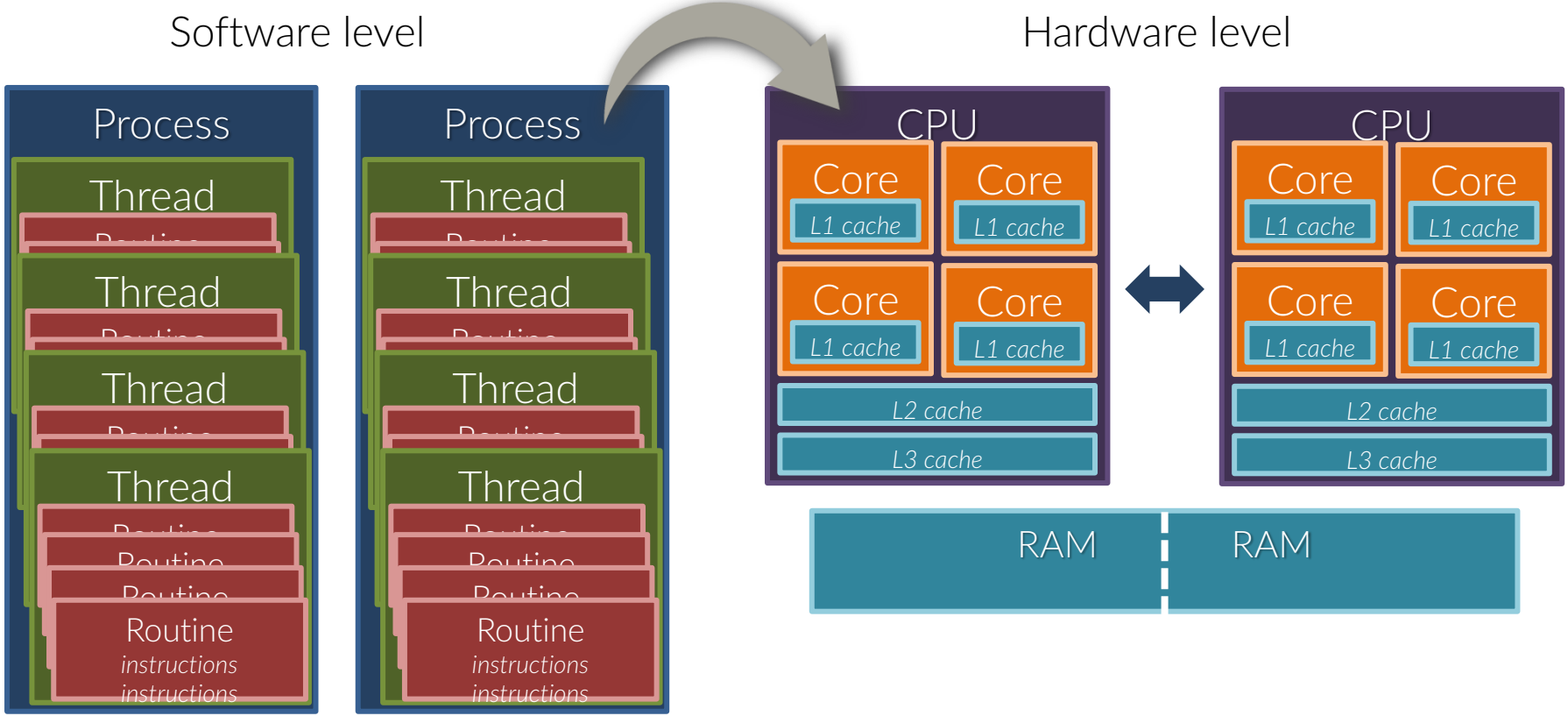
processors, cores, accelerators

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 ?



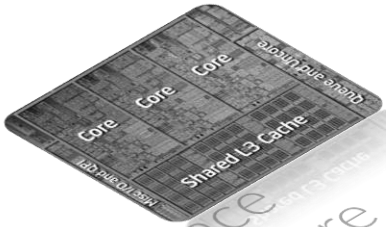
For two main reasons:

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

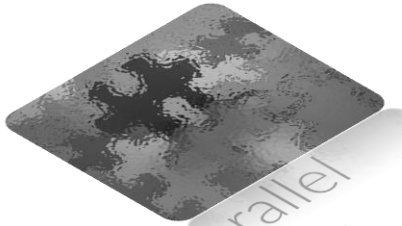
2. Problem size (~ data size)

   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, «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 $= Eff(n,p) \leq Sp(n,p) / p$

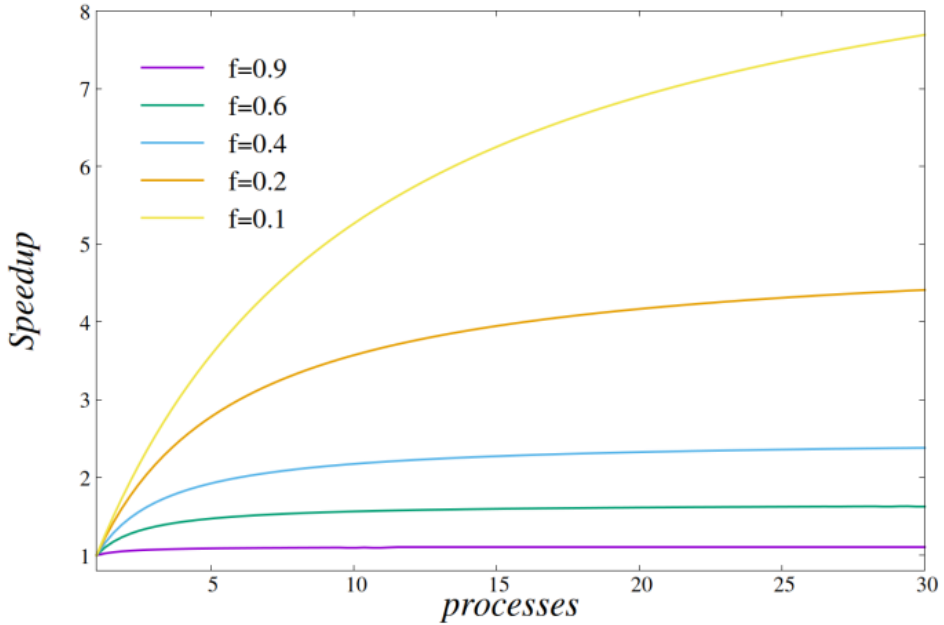
Amdahl's law

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

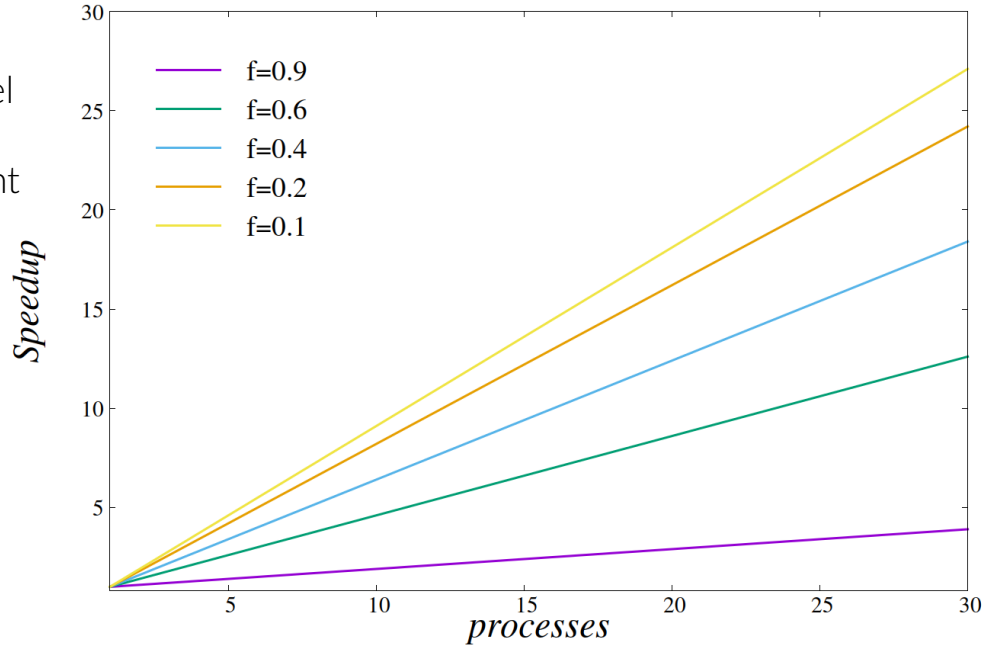
Then, usually the speedup increases with problem size

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: $$e(n,p) = \frac{\frac{1}{Sp(n,t)} - \frac{1}{p}}{1 - \frac{1}{p}}$$

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

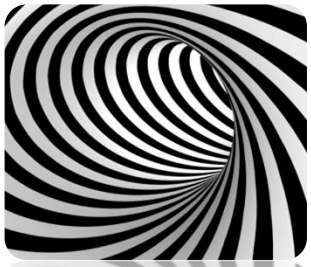
Outline

Intro to basic OpenMP

OpenMP Outline

Introduction & Concept

Parallel Regions



Parallel Loops

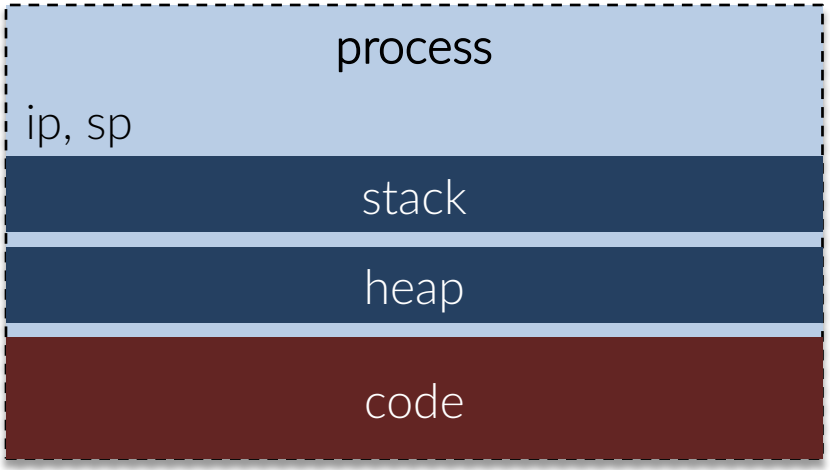
Sections & Task

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

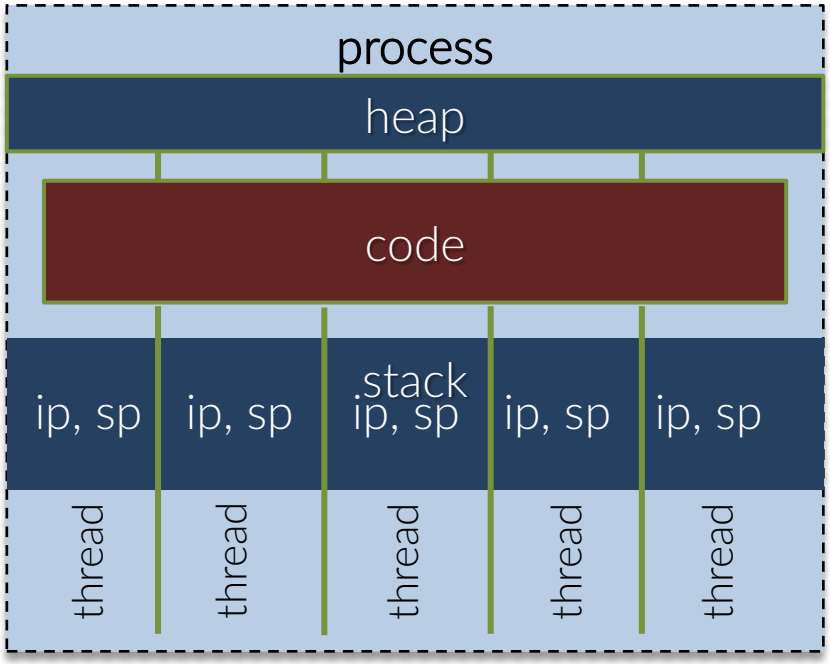


Threads and processes

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*.

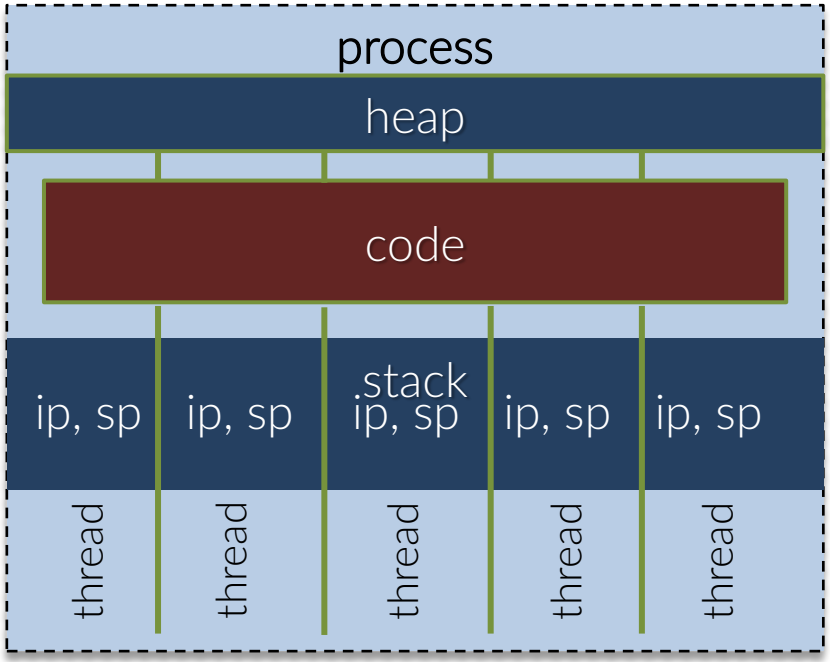
Spawning threads inside a process is much less costly than creating processes.



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 **M**ulti**P**rocessing

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.

What is OpenMP

1997	OpenMP for Fortran, 1.0
1998	OpenMP for C/C++, 1.0
2000	OpenMP for Fortran, 2.0
2002	OpenMP for C/C++, 2.0
2005	OpenMP 2.5
2008	OpenMP 3.0
2013	OpenMP 4.0
2015	OpenMP 4.5
2016	OpenMP 5.0

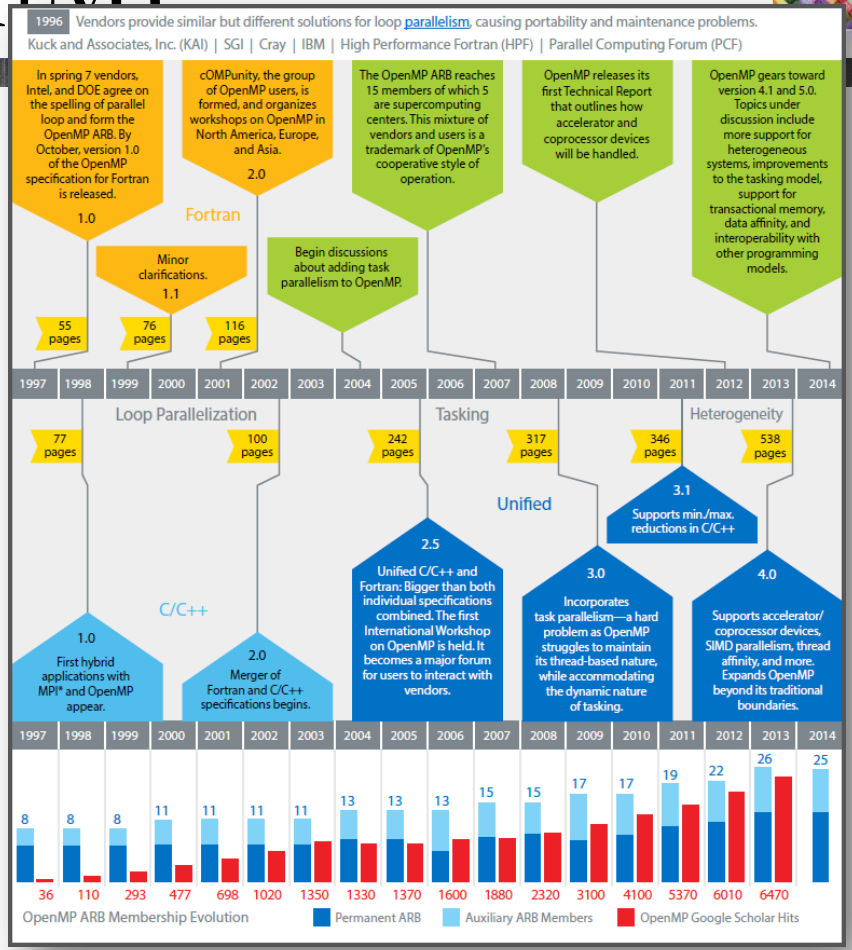
regular loops parallelization

Irregular parallelism task-based parallelism

accelerators / tasks+ / atomics / affinity / SIMD / user reduction

off-loading to accelerators more on tasks and workshares, ...

from http://openmp.org



What is OpenMP

Advantages of a directive-based approach

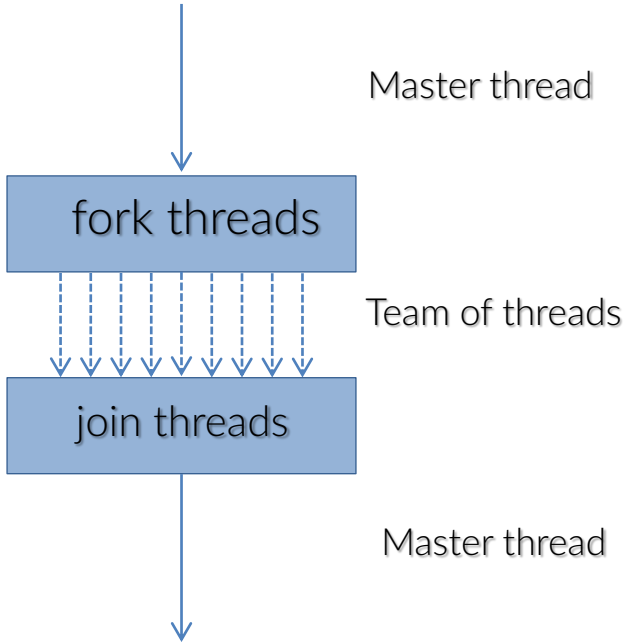
- **Abstraction**
  Subtleties of `pthread` and hardware-specific aspects are hidden. You can focus on data and workflow much more easily.
- **Efficiency.**
  The learning curve to achieve reasonable results is much shallower. The code's design is easier, the result/effort ratio is favourable with respect to `pthread`.
- **Incremental approach**
  No need to re-write your whole code. You start concentrating on some sections only, following a 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

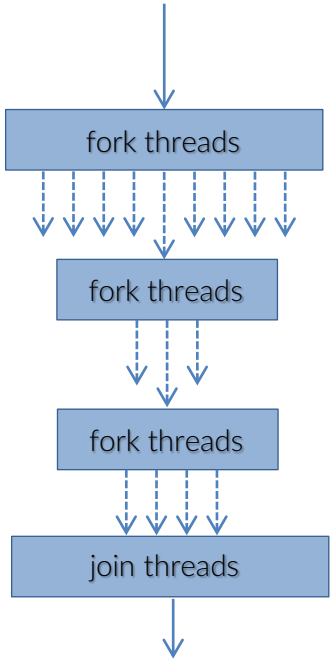
use a single master thread for serial operations



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

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

- directives for parallel regions
  - #pragma omp parallel directive
- work-sharing constructs
  - #pragma omp for
  - #pragma omp section
  - #pragma omp task
- data attributes
  - shared, private, firstprivate, lastprivate, copyin, copyprivate, threadprivate clauses
- Synchronization
  - critical, atomic, barrier, ordered, flush, nowait
- run-time
  - Lot of run-time functions and utils
    omp_set_num_threads(), omp_get_thread_num(), omp_get_num_threads(), …

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:

```
   …
#ifdef _OPENMP
   …
   …
#endif
   …
```

omp_versions.c

Conditional compilation

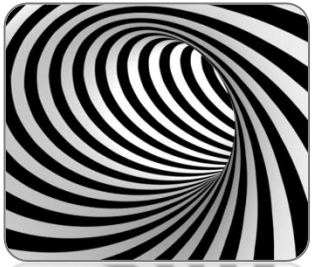
What is this useful for?

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

Sections & Task

Parallel Regions

```
#pragma omp parallel _some_clauses_here_
single-line-here
```

A parallel region can be as short as a single line

```
#pragma omp parallel _some_clauses_here_
{ … }
```

There are no limits on the size of the code included within {..}.

```
#pragma omp parallel for _some_clauses_here_
{ … }
```

The specific construct about `for` loops

```
#pragma omp sections _some_clauses_here_
{ … }
```

A more general work-sharing construct

```
#pragma omp task _some_clauses_here_
{ … }
```

This allows task-based parallelism

The region starts at the opening { brace and ends at the closing } one.
An implicit synchronization barrier is present at the end of the region (*).

However, for efficiency reasons, it may be that the threads do not die at the end of the region but remain alive until the father process dies

(*) we'll see the details about synchronization later on

The threads factory

When you create a parallel region, through an OpenMP directive, a pool of threads is created.
Each one receives an ID, ranging from 0 to *n-1*, where *n* is the number of threads.

Their stack and IP are separated from the others' ones and from the father-process' ones.

Can we check that?
What is the fate of the creating process (thread) ?

```
int    i;

register unsigned long long base_of_stack asm("rbp");
register unsigned long long top_of_stack asm("rsp");

 printf( "\nmain thread (pid: %d, tid: %d) data:\n"
         "base of stack is: %p\n"
         "top of stack is : %p\n"
         "&i is           : %p\n"
         "   rbp - &i     : %td\n"
         "   &i - rsp     : %td\n"
         "\n\n",
         getpid(), syscall(SYS_gettid),
         (void*)base_of_stack,
         (void*)top_of_stack,
         &i,
         (void*)base_of_stack - (void*)&i,
         (void*)&i - (void*)top_of_stack );

#pragma omp parallel private(i)
  {
    int me = omp_get_thread_num();
    unsigned long long my_stackbase;
    __asm__("mov %%rbp,%0" : "=mr" (my_stackbase));

    printf( "\tthread (tid: %ld) nr %d:\n"
                      "\t\tmy base of stack is %p ( %td from main\'s stack )\n",
                      "\t\tmy i address is %p\n"
                      "\t\t\t%td from my stackbase and %td from main\'s\n",
                      syscall(SYS_gettid), me,
                      (void*)my_stackbase, (void*)base_of_stack - (void*)my_stackbase,
                      &i, (void*)&i - (void*)my_stackbase,
                      (void*)&i - (void*)base_of_stack);
}
```

parallel_regions/
00_stack_and_scope.c

The threads factory

serial section

parallel region

```
main thread (pid: 26291, tid: 26291) data:
base of stack is: 0x7ffe6f51efc0
top of stack is : 0x7ffe6f51ef60
&i is           : 0x7ffe6f51ef7c
   rbp - &i     : 68
   &i - rsp     : 28


thread (tid: 26291) nr 0:
        my base of stack is 0x7ffe6f51eee0 ( 224 from main's stack )
        my i address is 0x7ffe6f51eea0
                -64 from my stackbase and -288 from main's
thread (tid: 26293) nr 2:
        my base of stack is 0x7f7342c82ba0 ( 597747680288 from main's stack )
        my i address is 0x7f7342c82b60
                -64 from my stackbase and -597747680352 from main's
thread (tid: 26294) nr 3:
        my base of stack is 0x7f7342880ba0 ( 597751882784 from main's stack )
        my i address is 0x7f7342880b60
                -64 from my stackbase and -597751882848 from main's
thread (tid: 26292) nr 1:
        my base of stack is 0x7f7343084b20 ( 597743477920 from main's stack )
        my i address is 0x7f7343084ae0
                -64 from my stackbase and -597743477984 from main's
```

The main thread has always the same `tid` than the `pid` of the father process.
The `thread 0` of the parallel region is still the same thread.

- Each thread has its own stack; `thread 0` inherits its own stack, which has grown to host the data relative to OpenMP parallel region;
- the private `i`s are actually in the threads stacks, and so different memory regions than the shared `i`.

`parallel_regions/00_stack_and_scope.c`

Parallel Regions


```
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
```

By default rules, variables declared outside a parallel region are *shared*, whereas those declared inside a region are *private*.
Then both `nthreads` and `my_thread_id` are shared.

As a consequence, in this example:
- all threads are writing in `my_thread_id`, in undefined order
- only the master thread is writing in `nthreads`

The value of `my_thread_id` is unpredictable, because it depends on the run-time order by which the threads access it and by each a thread accesses it again to write it.

parallel_regions/
01_simple_pr_wrong.c
02_simple_pr.c

Controlling the number of threads

By default the number of threads spawned in a parallel regions is the number of cores available.
However, you can vary that number in several way

- `OMP_NUM_THREADS environmental variable`
- `#pragma omp parallel num_threads(n)`
- ```
  omp_set_num_threads(n);
  #pragma omp parallel
  ```

`parallel_regions/`
`03_num_of_threads.c`
  ```

Controlling the number of threads

By default the number of threads spawned in a parallel regions is the number of cores available.
However, you can vary that number in several way

- `OMP_NUM_THREADS` environmental variable
- `#pragma omp parallel num_threads(n)`
- `omp_set_num_threads(n);`
  `#pragma omp parallel`

That works if `OMP_DYNAMIC` variable is set to `TRUE`, otherwise the number of threads spawned is strictly equal to `OMP_NUM_THREADS`.
This setting can be changed through
`omp_set_dynamcs( true )`

parallel_regions/
03_num_of_threads.c

The ordering of the threads


```
#pragma omp parallel
{
    int my_thread_id = omp_get_thread_num();
    printf( "greetings from thread num %d\n",
            my_thread_id );
}
```

The order by which the greetings appear in the output is not deterministic.

In general, unless either you explicitly requires so – and that is possible only for some constructs, or you directly code it, there is no given order since the threads are independent entities.

How would you build-up an enforcement of the order in the parallel region here on the left ?

The ordering of the threads


```
int order = 0;

#pragma omp parallel
{
    int myid = omp_get_thread_num();

    #pragma omp critical
    if ( order == myid ) {
       printf( "greetings from thread num %d\n",
               my_thread_id );
        order++; }
}
```

The CRITICAL directive defines a region which is executed by all threads but by a single thread at a time. Which starts to sound like an “ordering”.

However, although the threads queue at the begin of the region, no particular order is imposed to that queue. As a consequence, the result is unpredictable, aside the fact that `thread 0` will print the message and increase `order`.
For instance, if `thread 2` is queued immediately after, it will execute the `if` evaluation which will fail (`order` would be equal to 1) and the `thread 2` will not print the message.

parallel_regions/
04_order_of_threads_wrong.c

The ordering of the threads


```
int order = 0;

#pragma omp parallel
{
    int myid = omp_get_thread_num();
    int done = 0

    while (!done) {
    #pragma omp critical
    if ( order == myid ) {
       printf( "greetings from thread num %d\n",
               my_thread_id );
        order++; done=1; } }
}
```

parallel_regions/
05_order_of_threads.c

In this second implementation, the `while` cycle enforces the threads that fails the if condition to keep trying until the correct order is ensured.

Once a thread has executed the `critical` region, it then exits the `while` and join the implicit synchronization barrier at the end of the parallel region.

A note: without the `critical` directive, this code could work as well on most platform. However, it is still unreliable since it is not guaranteed that each thread complete the region *before* the successive threads enter in it. For instance, the compiler could have been reshuffled the `order++` to be before the print and so the `thread` *n+1* could enter the region before of the print of the `thread n` and its message could appear as first.

Specializing execution in PR


```
#pragma omp parallel
{
    #pragma omp critical
    { code block
    }

    #pragma omp atomic
    assignment instruction

    #pragma omp single
    { code block
    }

    #pragma omp master
    { code block
    }
}
```

Guards A CODE REGION

The **`critical`** directive ensures that only a thread at a time executes the block. All the threads will execute it, although in unspecified order.

Like **`critical`** but limited to the following single line.
The instruction can only be an assignment in the form

x binop = expr

GUARDS MEMORY!

Only one among the threads will enter the code block

Only the master thread will enter the code block

Specializing execution in PR


```
#pragma omp parallel
{
    #pragma omp critical
    { code block
    }

    #pragma omp atomic
    assignment instruction

    #pragma omp single
    { code block
    }

    #pragma omp master
    { code block
    }
}
```

A barrier (in the form of queuing) is present at the entry point; no one at the end point, i.e. the threads exiting the region will continue the run.

Synchronization as in critical region.

No explicit synchronization at all (but see relative tips in the following slides).
Only the executing thread is inside the region, the other ones can be everywhere else.

Specializing execution in PR

Rationale of

```
#pragma omp atomic
assignment instruction
```

When you deal with shared variables, ensuring that the workflow maintains the semantic correctness of your code is fundamental.
For instance, the assignment

```
a += b
```

(where either `a`, `b` or both are shared) is meaningful only if the value of `a` (or `b`, or both) does not change in the middle of the instruction itself.

In fact, while a single assembly instruction is guaranteed not to be ever interrupted on the fly, we have to take into account that even a single C line translates in several asm instructions. So, the value of a shared variable could have been changed in the middle of that groups of asm instructions, breaking the correctness of the code.

To avoid that, it is necessary to "protect" the sensible regions.

An **`atomic`** assignment has, obviously, a much lower overhead than a **`critical`** region and must be preferred in the appropriate cases.

Specializing execution in PR


```
#pragma omp critical
{
   "A" code block
}
#pragma omp critical
{
   "B" code block
}
#pragma omp critical(my_loop)
{
   code block
}
```

Named critical regions

In OpenMP it exists a unique global `critical` section.
Hence, when you define a `critical` section, it is logically considered to be part of the global one.
As a consequence *only one thread can be inside any of the unnamed `critical` sections*, which of course limits the perfomance whn more than one region is present.

However, that can be cured by the *named regions*, defined as the last one in the code snippet here on the left.
In that example, only one thread can be either in region "A" or in region "B": so, if one thread is executing "A", another one is forced to delay entering "B" even it would have been reading for it.

Instead, the named `critical` regions can be executed at the same time (by different threads, of course, and only by one thread at a time).

Specializing execution in PR


```
#pragma omp critical
{ …
  some_assignment_to x;
  …
}

#pragma omp atomic
x op= value
```

`critical` *and* `atomic`

Consider that by design `atomic` protects *memory regions*, while `critical` protects code regions.

Hence they are not mutually exclusive: a thread may be executing the critical region while another may be executing the atomic.

Specializing execution in PR


```
#pragma omp single
{ …code block… }
```

```
#pragma omp master
{ …code block… }
```

Is there any difference between using `single` or `master` ?

There obviously is if you're assigning some special workflow to `thread 0`.
If not, the entity of the difference depends on the implementation details.

In general, using `master` requires only a simple test on the thread id, while using `single` requires more synchronization (the `openmp` infrastructure has to keep track about what thread is in the region and so on).

In general, if you have some insight about the fact that the workload before that point may be systematically unbalanced, so that some thread will likely arrive first, using `single` is ok.
Otherwise, it may be better to use `master`.

Work assignment

Inside a parallel region, the work can be assigned to each threads (actually, that's why PR are created..):

```
…
results[0] = heavy_work_0();
results[1] = heavy_work_1();
results[2] = heavy_work_2();
…
```

parallel_region/
06_assign_work.c

```
#pragma omp parallel
{
    if( myid % 3 == 0)
        result = heavy_work_0( );
    else if ( myid % 3 == 1 )
        result = heavy_work_1( );
    else if ( myid % 3 == 2 )
        result = heavy_work_2( );

    if ( myid < 3 )
        results[myid] = result;
}
```

dynamic extent

parallel_region/
06_assign_work.c

run with a second argument >0 to check about it

Conditional creation of PR

It is possible to spawn a parallel region only if some conditions are met:

- `#pragma omp parallel if( any valid C expression )`

```
#pragma omp parallel if( amount_of_work > high )
{
    if( myid % 3 == 0)
        result = heavy_work_0( );
    else if ( myid % 3 == 1 )
        result = heavy_work_1( );
    else if ( myid % 3 == 2 )
        result = heavy_work_2( );

    if ( myid < 3 )
        results[myid] = result;
}
```

If the condition is not fulfilled, the threads are not created and only the master thread will execute the code block.

parallel_region/
06_assign_work.c

if the first argument, that determines the amount of work, is ≤10, the parallel region is **not** created

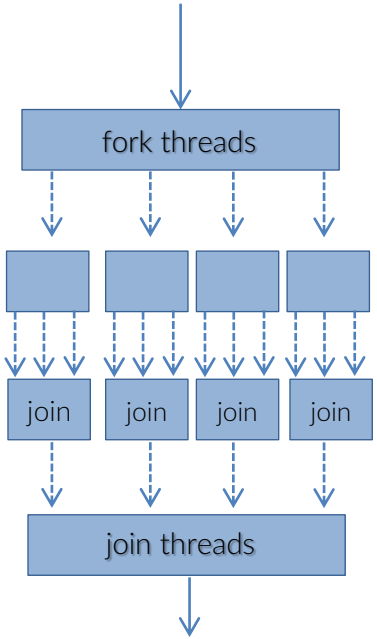
What is this useful for?

The overhead of the creation of a parallel region is of the order of ~10 μseconds (that figure is dependent on the system, the compiler, the number of threads,...).

Then, you may want to create a parallel region (i.e. to spawn more threads) only if the serial execution of that code section is at least several times this overhead.

Nested Parallel Regions

NESTED PARALLELISM



Nested parallelism is explicitly_permitted in OpenMP. Whether it is *active* or not, depends on the value of some environmental variables:

`OMP_NESTED=<TRUE | FALSE>` (*) default value is `FALSE`
`OMP_NUM_THREADS=`N_0`<,`N_1`, … >`
`OMP_MAX_ACTIVE_LEVELS=`*n*

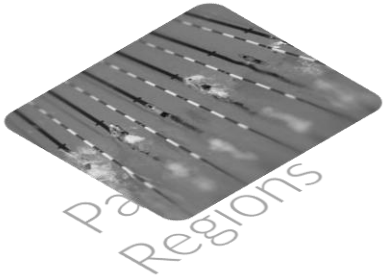
Which you can change by calling the appropriate `OMP_` functions

`omp_get_nested(), omp_set_nested(` ***cond*** **`)`**

(*)OMP_NESTED is deprecated in new OpenMP 5.0. You should use only OMP_MAX_ACTIVE_LEVELS and OMP_NUM_THREADS. OMP_NESTED is in fact redundant. However, at the moment of writing the compilers still support OMP_NESTED.

07_nested_regions.c

OpenMP Outline



Parallel Loops

Sections & Task

OpenMP parallel loops

Loops are one of the most common work structure in HPC, and it is quite common that a vast amount of compute-intensive code resides in loops.

In fact, OpenMP, up to version 2.x, was essentially about quickly and effectively parallelizing loops without much effort.
Hence, OpenMP standard presents a broad amount of features dedicated to parallel `for` loops.

Building up a parallel loop


```
int N = some_workload;
#pragma omp parallel
{
  int myid = omp_get_thread_num();
  int team = omp_get_num_threads();

  int size     = N / team;
  int reminder = N % team;
  int mystart  = size*myid + (myid<reminder)*myid;
  int myend    = size + (myid < reminder);

  printf("task %d is running from %d to %d\n",
         myid, mystart, myend);

  for ( int i = mystart, i < myend; i++ )
    {
      do_smething(i);
    }
}
```

Splitting the work of a for loop among the threads can easily be achieved by directly assigning the boundaries of the loop to each thread.

In this example, we statically assign an equal share `N/nthreads` of iterations per thread, while distributing the remaining `N%nthreads` iteration to the first `N%nthreads` threads.

However, OpenMP has dedicated constructs that offer easier and more flexible mechanisms to share the work within a for loop.

OpenMP basic loop

Let's start with a classical and very common problem in order to understand the appropriate OpenMP work-sharing construct relative to loops.

```
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,sum,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.

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 (outside)within the parallel region are automatically (shared) private, and so are the integer indexes used as cycles counter.

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

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

parallel_loops/
00_array_sum_with_race.c

What happens if you drop the `atomic` directive?
You obtain a result that is smaller than the correct one: why?

How is the work assigned to single threads ?

OpenMP basic loop


```
#pragma omp parallel for implicit(none) shared(a,sum,N) private(i)
```

The default policy for memory regions is actually that all are shared. However, that is a very common source of error – when you have lots of variables, you forgot what is what in your code.

It may be considered a good practice to add `implicit(none)` to all your construct so that to spot any error alike.

OpenMP basic loop


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

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

Without the `atomic` directive, the assignment

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

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

Anatomy of a data race

After having solved the data race

Let's say that we solve the data race introducing the critical region `local_sum`, or an `atomic` directive. Does it scale ?

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

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

parallel_loops/
00_array_sum_with_race.c

Try to run it with a fixed, large enough, N on an increasing number of cores, and take note about the speedup. Then, measure the Parallel overhead

Of course no! why?

Solving the reduction

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 the reduction / 2

However, that is so important that the OpenMP standard offers a simple solution:

parallel_loops/
01_array_sum.c

```
#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.. in this simple case.

Solving the reduction / 3

There is another way in which we can solve the conflicts on the `sum`

```
#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 ?

Solving the reduction /4

There is another way in which we can solve the conflicts on the `sum`

parallel_loops/
02_falsesharing.c

```
#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 the values of **sum[nthreads]** reside in the same cache line(s); hence, when a thread access and modify its location, to maintain the coherence the cache must write-back and reflush. Every time.

That is called **false sharing**

Solving the reduction /5

There is another way in which we can solve the conflicts on the `sum`

parallel_loops/
03_falsesharing_
fixed.c

```
#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 much more memory than needed.
And, above all, we hard-coded a magic number (which is not a good move, in general, since it is not portable).

OpenMP work assignment in loops

How the work is assigned to the 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.

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`

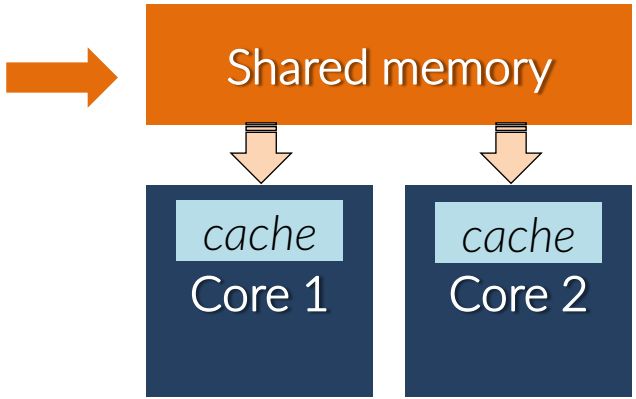
“touchfirst” policy

The matter is: who “owns” the data?

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

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

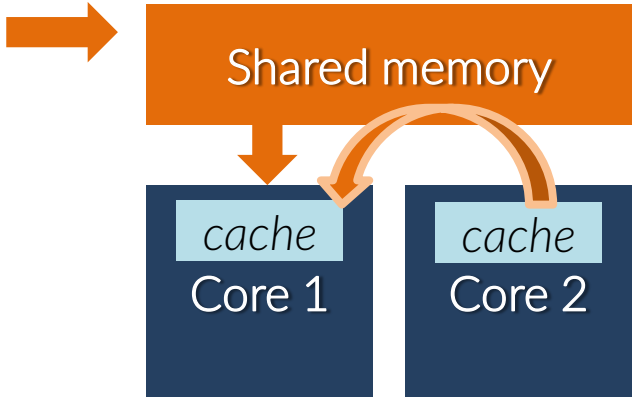
#pragma omp parallel for reduction(+: sum)
for ( int i = 0; i < N; i++ )
      sum += a[i];
```



“touchfirst” policy

The matter is: who “owns” 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

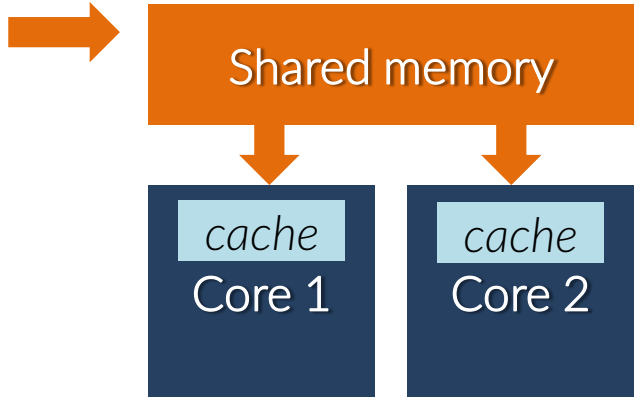
The matter is: who “owns” the data?

```
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. 08



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` preserves 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