Introduction to OpenMP

Luca Tornatore - I.N.A.F.

DATA SCIENCE & ARTIFICIAL INTELLIGENCE

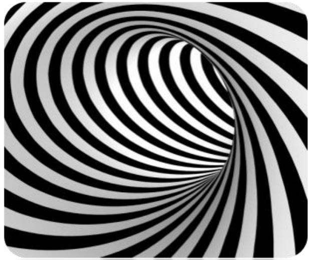
SCIENTIFIC & DATA-INTENSIVE COMPUTING

2024-2025 @ Università di Trieste

OpenMP Outline

Introduction & Concept

Parallel Regions



Parallel Loops

NUMA AWARENESS

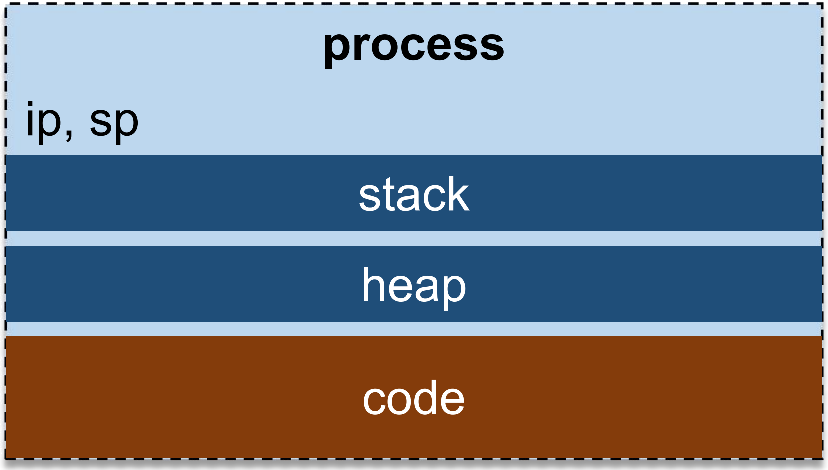
Threads and processes

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

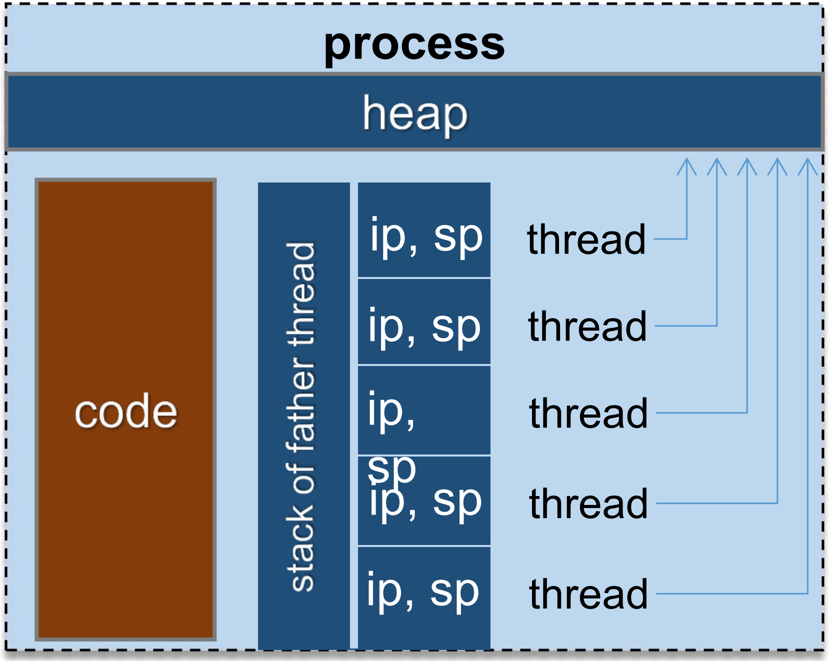


Threads and processes

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.

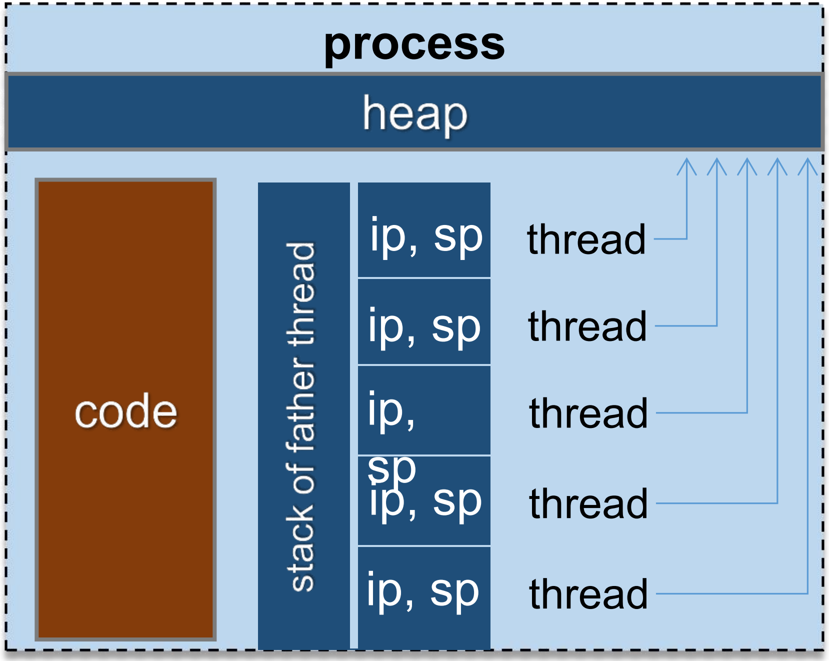
In geneal spawning threads inside a process is much less costly than creating processes.



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.

i.e

Shared-Memory

vs

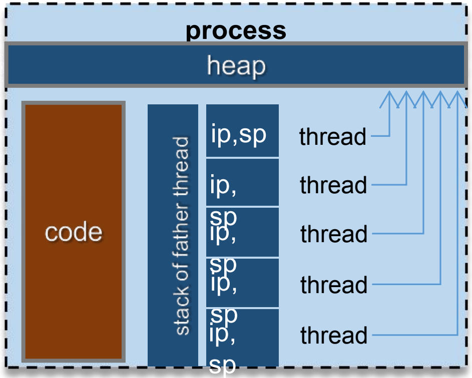
Distributed-Memory

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



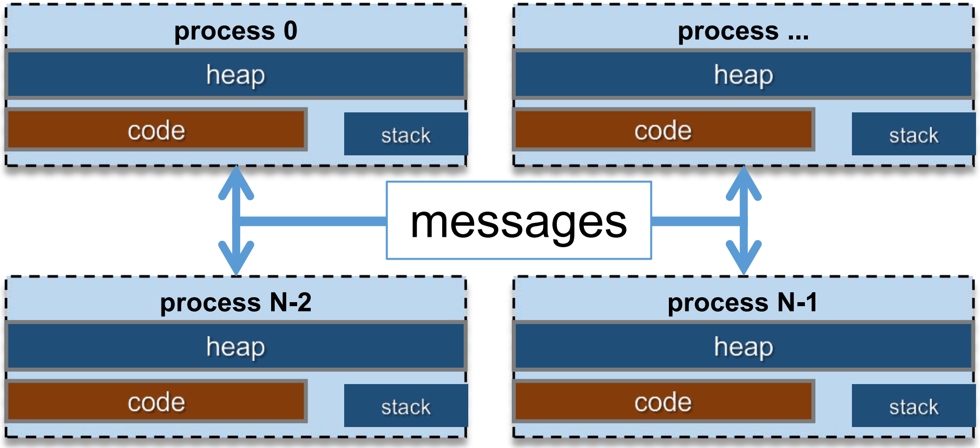
Distributed-Memory

(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-Me

(e.g. OpenMI

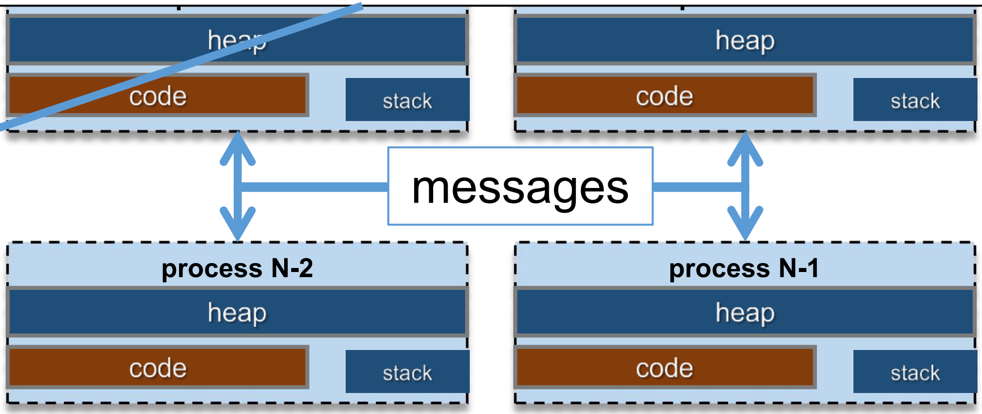
A unique process
number of threac
memory space th
the threads

Actually MPI 3.0 introduced special tools to
(1) allow shared-like accesses among tasks that run on cores that share the memory;
(2) allow direct memory access to the memory of other MPI tasks in general, which is called Remote Memory Access

Distributed-Memory

(e.g. MPI)

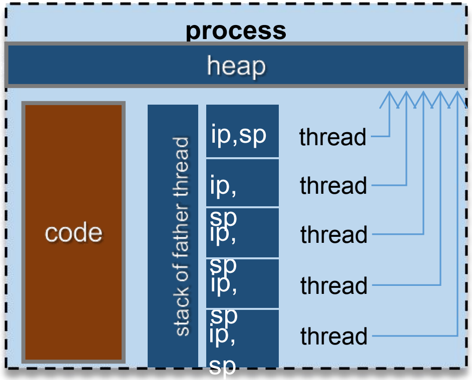
N processes are created, each with its own copy of the code and its own memory space.
A process ***can not*** access the memory space of another process.
The processes communicate through *messages*.



OpenMP vs MPI

Shared-Memory
(e.g. OpenMP)

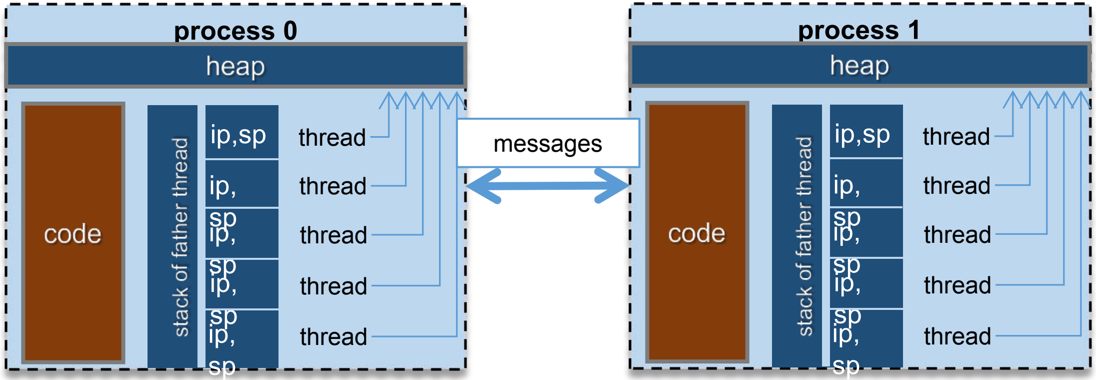
A unique process that spawns a number of threads. There is a unique memory space that is accessible by all the threads



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*.

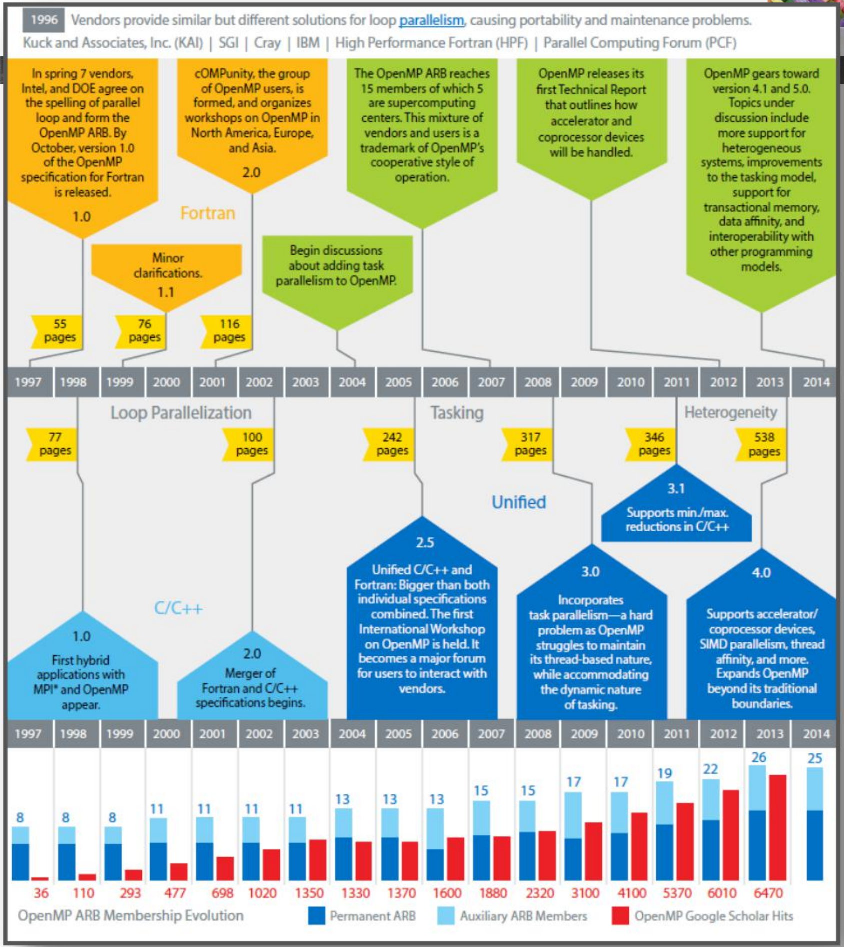


Intro & Concept

What is OpenMP

1997	OpenMP for Fortran, 1.0	regular loops parallelization
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	Irregular parallelism task-based parallelism
2013	OpenMP 4.0	accelerators / tasks+ / atomics / affinity / SIMD / user reduction
2015	OpenMP 4.5	
2016	OpenMP 5.0	off-loading to accelerators; more on tasks and workshares, …

from http://openmp.org



What is OpenMP

Advantages of a directive-based approach

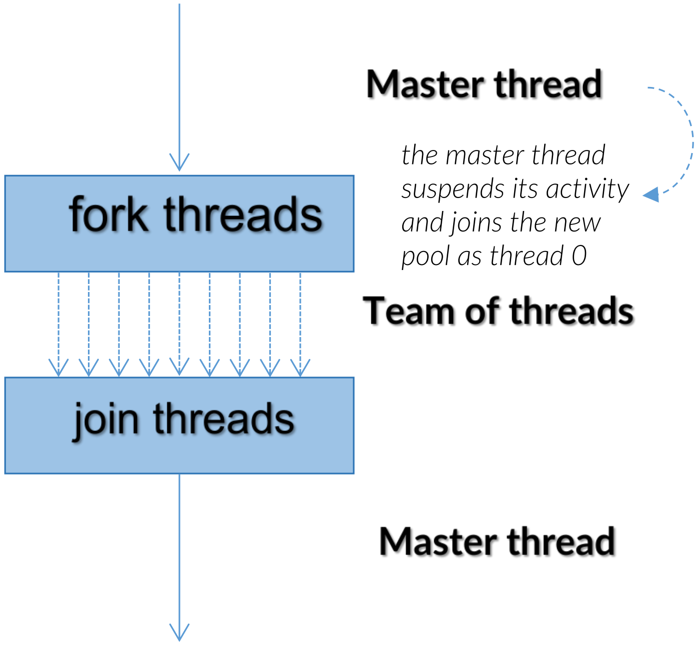
- **Abstraction**
  Subtleties of `pthread` and hardware-specific aspects are hidden. You can focus on data and workflow much more easily.
- **Efficiency.**
  The learning curve to achieve reasonable results is much shallower. The code's design is easier, the result/effort ratio is favourable with respect 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

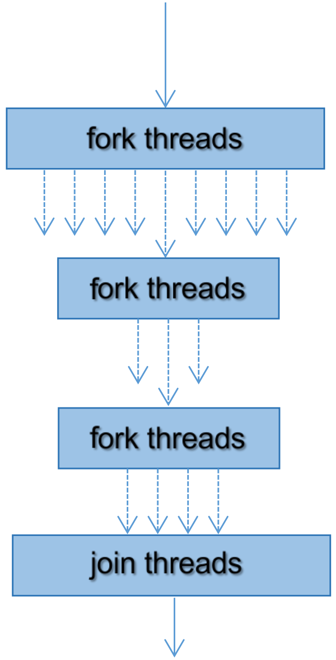


This is called "fork-join" model: a thread meets, at some point in its existence, a *directive* that activates the creation of a pool of children threads.

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

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.

What happens if a function is called from a parallel region?

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

static extent

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.

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

static extent

```
double foo( double *A, int N )
{
    double mysum = 0;

    for ( int ii = 0; ii < N; ii++ )
       mysum += A[ii];
    return sum;
 }
```

dynamic extent

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);
   …
}
```

static extent

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

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 mysum = 0;
  #pragma parallel for reduction(+:sum)
   for ( int ii = 0; ii < N; ii++ )
      mysum += A[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

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
my_thread_id = omp_some_function();
#endif
  ...
```

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

Conditional compilation

TIPS

What the `_OPENMP` is 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).

that’s all, have fun