Introduction to
High-Performance Computing

Giorgio Amati
Alessandro Ceci

Corso di dottorato in Ingegneria Aeronautica e Spaziale 2025

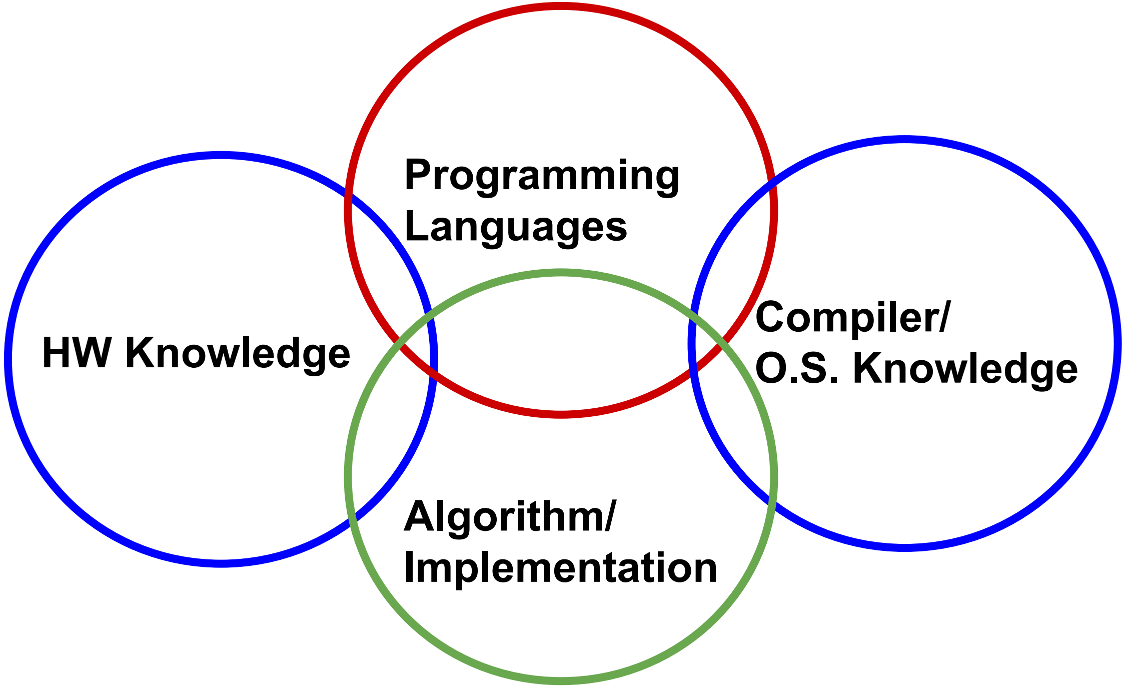
g.amati@cineca.it/g.amaticode@gmail.com
alessandro.ceci@uniroma1.it

Agenda

- ✓ HPC: What is it?
- ✓ Hardware: how it works
- ✓ Algorithm vs. Implementation
- ✓ Compiler + Floating point + I/O
- ✓ **HW & Parallel Paradigm**
  - ■ **Shared Memory parallelization: openmp et al.**
- ✓ Conclusions & Comments

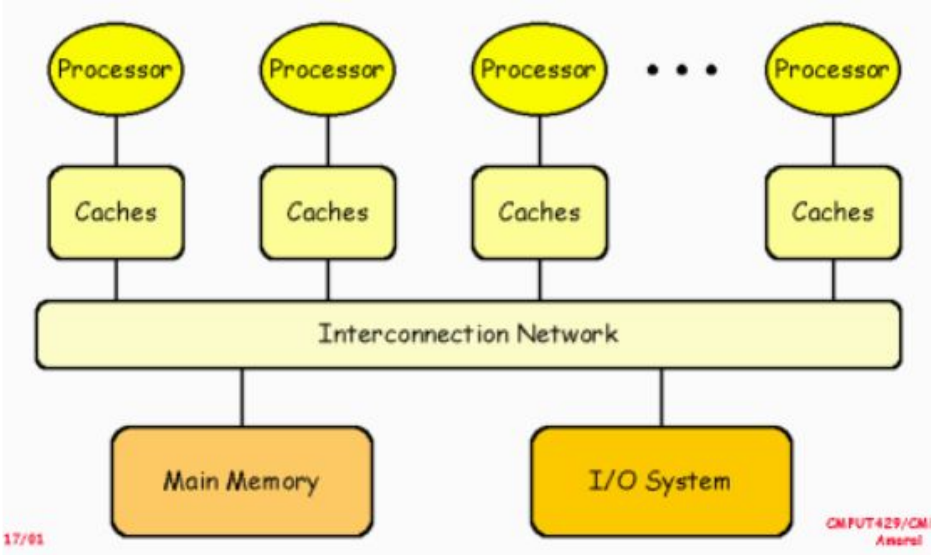
HPC: what it is?

- ✓ These are the main skills for efficient HPC



Shared Memory Machine

- ✓ A shared-memory system is an architecture consisting of a number of processors, all of which have direct (i.e. hardware) access to all the main memory in the system. This permits **any** of the system processors to access data that **any** of the other processors has created or will use
  - ○ **UMA**: Uniform Memory Access
  - ○ **NUMA**: Non Uniform Memory Access



Amdhal’s law (strong scaling)

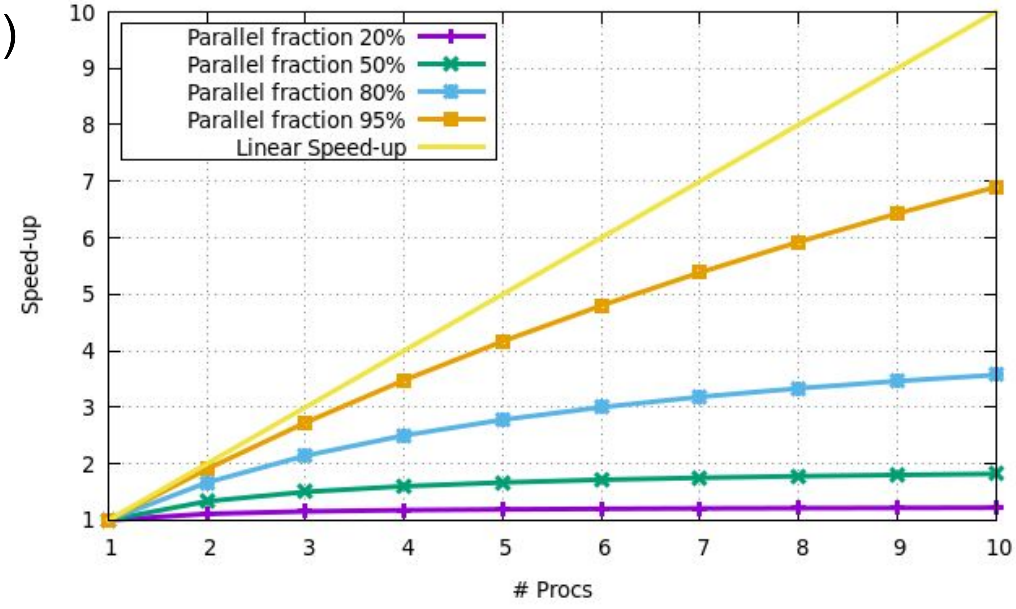
✓ **Fixed problem size**

✓ F = parallel fraction of the code (F < 1,00)

✓ N = #of processors

✓ S = velocity increment (Speed-up)

$$S = \frac{1}{(1-F)+\frac{F}{N}}$$



Amdhal’s law (strong scaling)

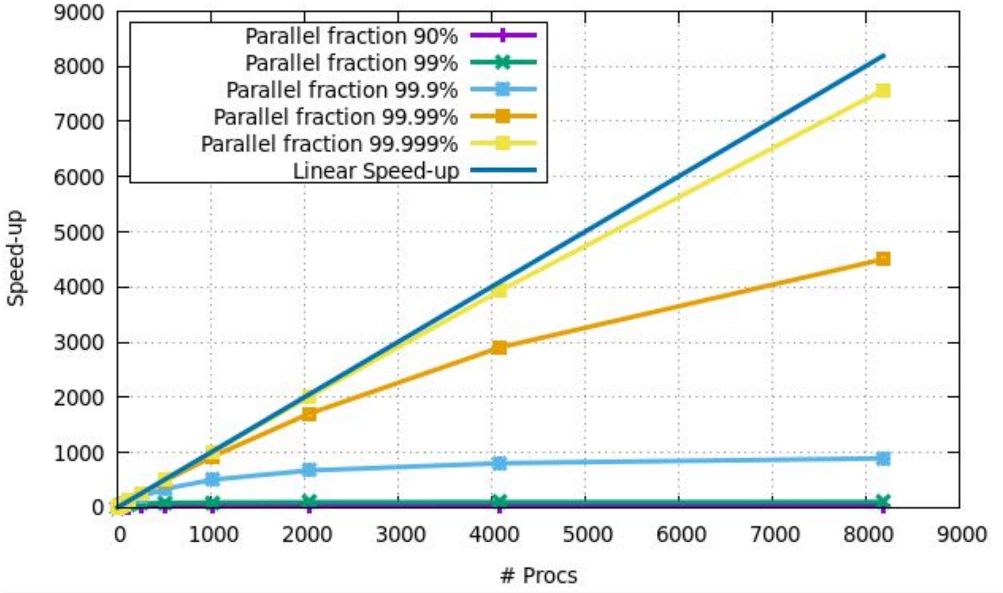
✓ **Fixed problem size**

✓ F = parallel fraction of the code (F < 1,00)

✓ N = #of processors

✓ S = velocity increment (Speed-up)

$$S = \frac{1}{(1-F)+\frac{F}{N}}$$

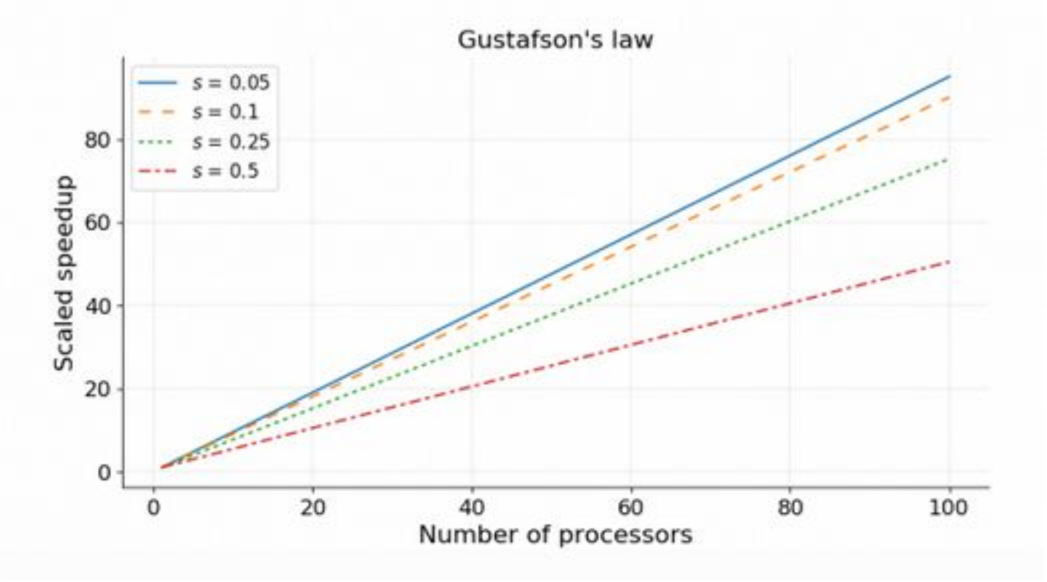


✓ Any hint?

(Gustafsson laws): Weak scaling

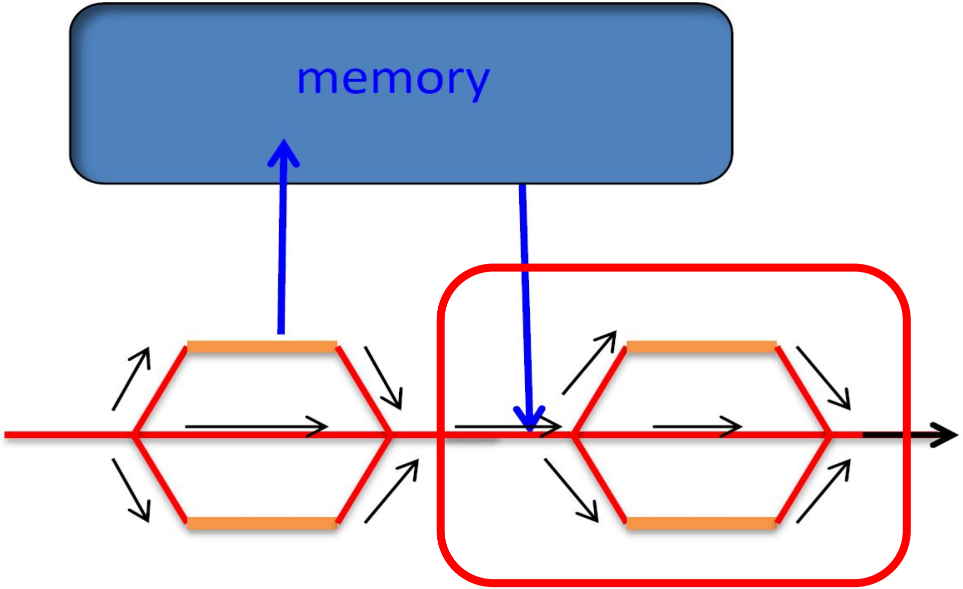
- ✓ **Scaled problem size**
- ✓ S = velocity increment (Speed-up)
- ✓ s = serial part
- ✓ p = parallel part
- ✓ n = #of processors

$$\boxed{S = s + p*n}$$



Shared memory parallelism

- ✓ Work-sharing model
- ✓ One process (master thread) that:
  - ○ “forks” in different threads
  - ○ each threads access to a common memory
  - ○ each threads performs some operation
  - ○ all threads joins the master threads
- ✓ Risk of “race conditions”
- ✓ No risk of “deadlocks”



OpenMP in few slide

- ✓ De-facto standard Application Programming Interface (API) to write shared memory parallel applications in C, C++, and Fortran
- ✓ Consists of compiler directives, runtime routines and environment variables
- ✓ Shared Memory parallelism: work sharing
- ✓ A single program that share work between “workers” (threads)
  - ○ Fork/join
- ✓ **Programmer has to share work between threads and keep data coherence**
- ✓ **Programmer has not to take care of data movement but to say which data is global (i.e. external and accessible to every threads) and which local (i.e. private to one threads)**
- ✓ **Incremental (loop by loop)**
- ✓ **From rel. 4.5 support for offloading to GPU (still to work on)**
- ✓ **Directive based: no code modification (in principle…)**

OpenMP in one slide: Fortran

Matrix matrix multiplication:

```
!$OMP PARALLEL DO &
!$OMP DEFAULT(NONE) &
!$OMP PRIVATE(i,j,k)  &
!$OMP SHARED(a,b,c,n)
    do j = 1, n
       do k = 1, n
          do i = 1, n
             c(i,j) = c(i,j) + a(i,k)*b(k,j)
          enddo
       enddo
    enddo
!$OMP END PARALLEL DO
```

1 Thread (serial) Mflops = 5682
8 threads Mflops = 37590

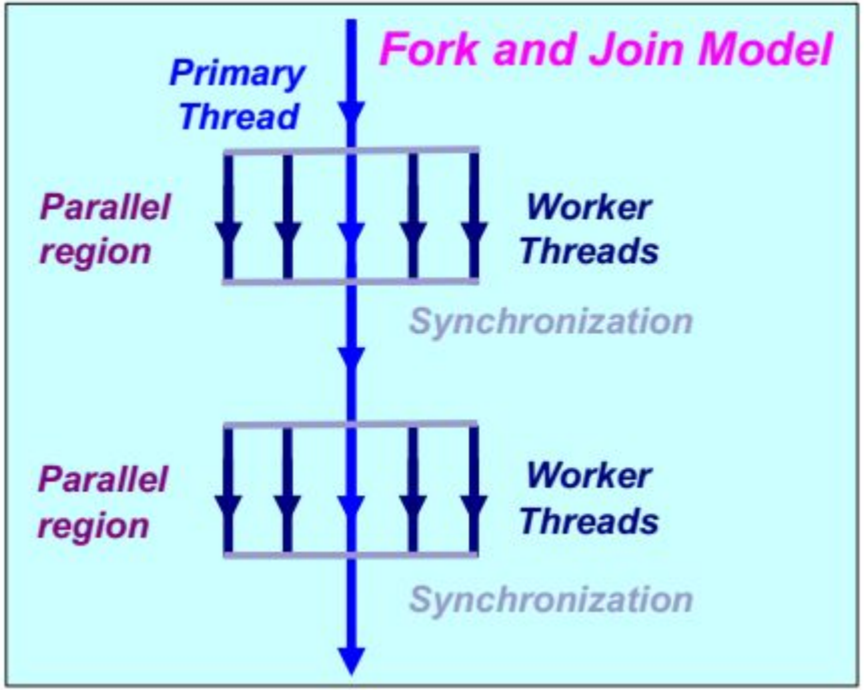
OpenMP in one slide: C

Matrix matrix multiplication:

```
#pragma omp parallel shared(a,b,c),private(i,j,k)
#pragma omp for
 for (i = 0; i < nn; i++) {
  for (k = 0; k < nn; k++) {
    for (j = 0; j < nn; j++) {
        c[i][j] = c[i][j] + a[i][k]*b[k][j];
    }
  }
}
```

OpenMP execution Model

- ✓ Work-sharing model
- ✓ One process (master thread) that:
  - ○ “forks” in different threads
  - ○ each threads access to a common memory
  - ○ each threads performs some operation
  - ○ all threads joins the master threads



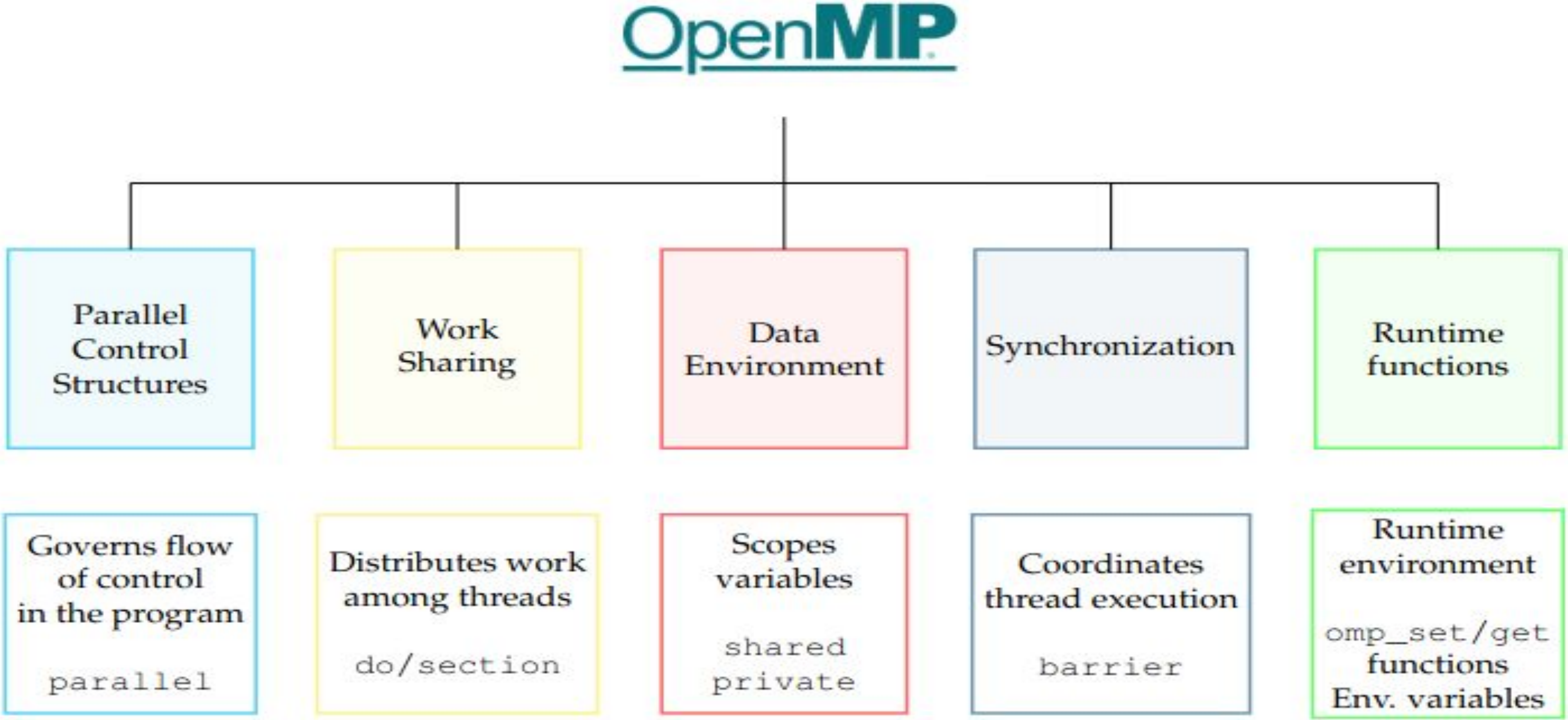
OpenMP API

It can handle many “features”:

- ✓ synchronization
  - ○ wait/nowait
- ✓ reductions
- ✓ cancellation
- ✓ data allocation….
- ✓ data movement
  - ○ shared/private data
- ✓ thread binding
- ✓ memory affinity (rel. 5.x)
- ✓ simd, vectorization, ….
- ✓ Tasking

ONLY BASIC FEATURES WILL BE DESCRIBED

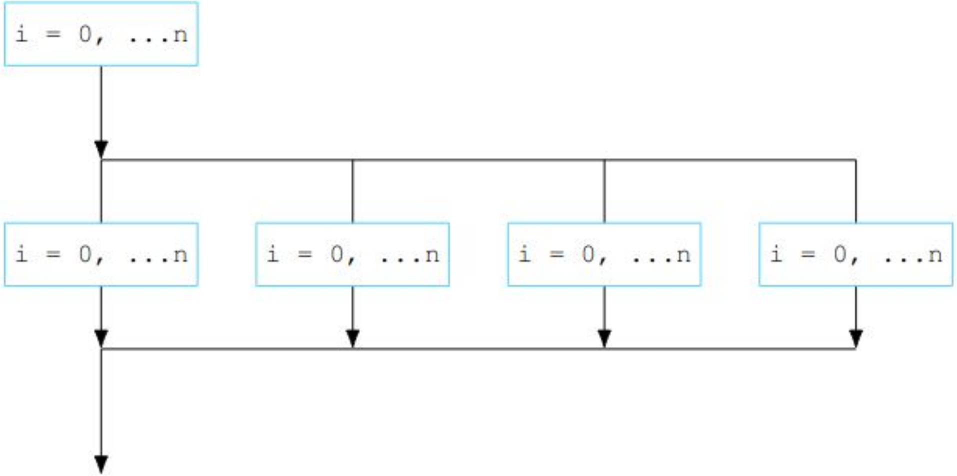
OpenMP



OpenMP
Parallel Control Structures
Work Sharing
Data Environment
Synchronization
Runtime functions
Governs flow of control in the program
parallel
Distributes work among threads
do/section
Scopes variables
shared
private
Coordinates thread execution
barrier
Runtime environment
omp_set/get functions
Env. variables

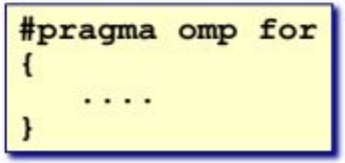
Parallel control structure

- ✓ **`parallel/end parallel`**
- ✓ It opens the parallel region where
  - ○ all threads perform the same iterations
  - ○ not yet worksharing
  - ○ each thread executes the same code (redundancy)

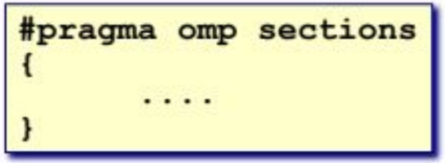


Worksharing Contructs

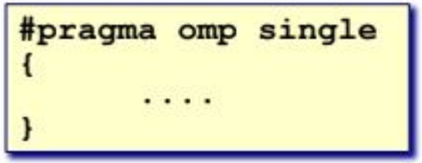
✓ The work is distributed over the threads
✓ Must be enclosed in a parallel region
✓ Must be encountered by all threads in the team, or none at all
✓ No implied barrier on entry
✓ Implied barrier on exit (unless the nowait
✓ clause is specified)
✓ A work-sharing construct does not launch any new threads
✓ Different construct
  ○ for/loop
  ○ sections
  ○ single
  ○ ….



```
#pragma omp for
{
    ....
}
```



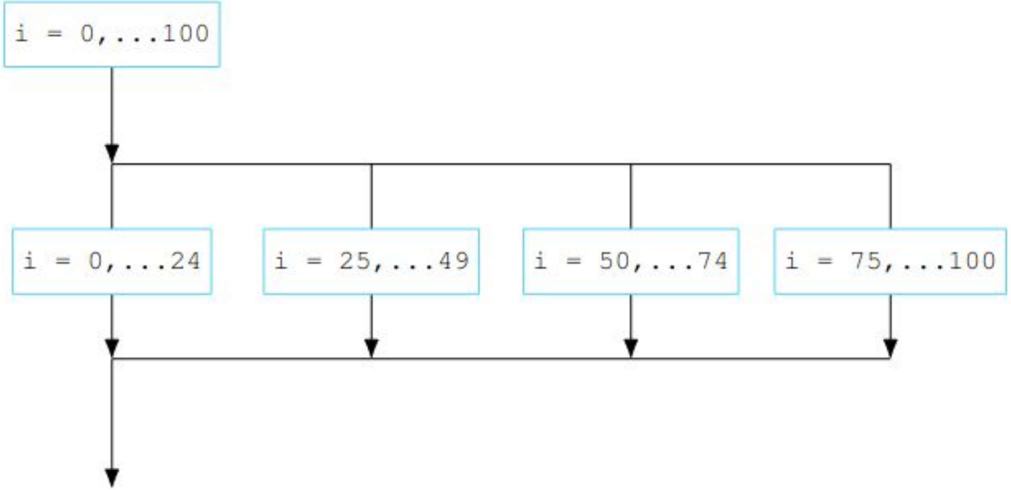
```
#pragma omp sections
{
        ....
}
```



```
#pragma omp single
{
        ....
}
```

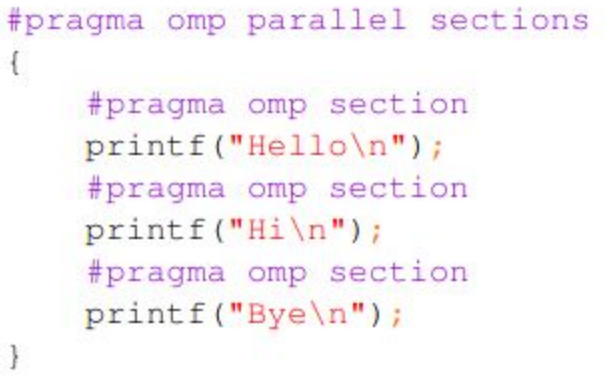
Worksharing: `loop/do`

- ✓ `loop/do`
- ✓ **distribute work between threads**
  - ○ default: all threads perform the same number of iterations
  - ○ It scales with the number of available threads (if the are enough iterations)

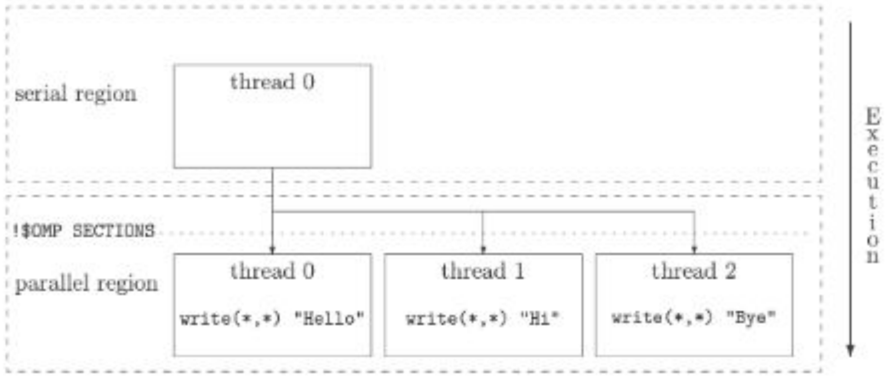


Worksharing: `section`

- ✓ **`sections`**
  - ○ Each section is execute by a different threads
  - ○ Each section could be a complete different piece of code
  - ○ could be hard to scale with the number of threads



```
#pragma omp parallel sections
{
    #pragma omp section
    printf("Hello\n");
    #pragma omp section
    printf("Hi\n");
    #pragma omp section
    printf("Bye\n");
}
```

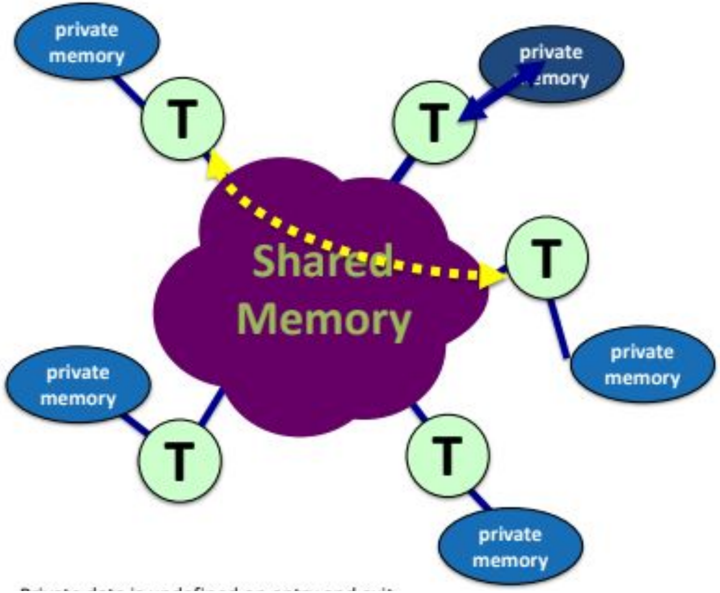


Worksharing: `single/master/critical`

- ✓ **`single/master/critical`**
  - ○ **`single`**: when only one (first in) execute the work sharing section
  - ○ **`master`**: only the master thread execute the work sharing section
  - ○ **`critical`**: this work sharing section is executed by all the threads but non in parallel, only one threads at time.

OpenMP memory model

- ✓ All threads have access to the same, globally shared memory
- ✓ Data in private memory is only accessible by the thread owning this memory
- ✓ No other thread sees the change(s) in privatememory
- ✓ Data transfer is through shared memory and is 100% transparent to the application



Main Data Sharing Clauses

- ✓ **`private(list)`**
  - ○ All variables defined are uninitialized, each threads has is own private copy
- ✓ **`firstprivate(list)`**
  - ○ All variables defined are initialized, each threads has is own private copy with the same value, at the begin
- ✓ **`shared(list)`**
  - ○ All variables defined are initialized and accessible for each thread
- ✓ **`reduction(+:list)`**
  - ○ Perform a reduction (e.g. sum) all over the variables in the list

`default` clause

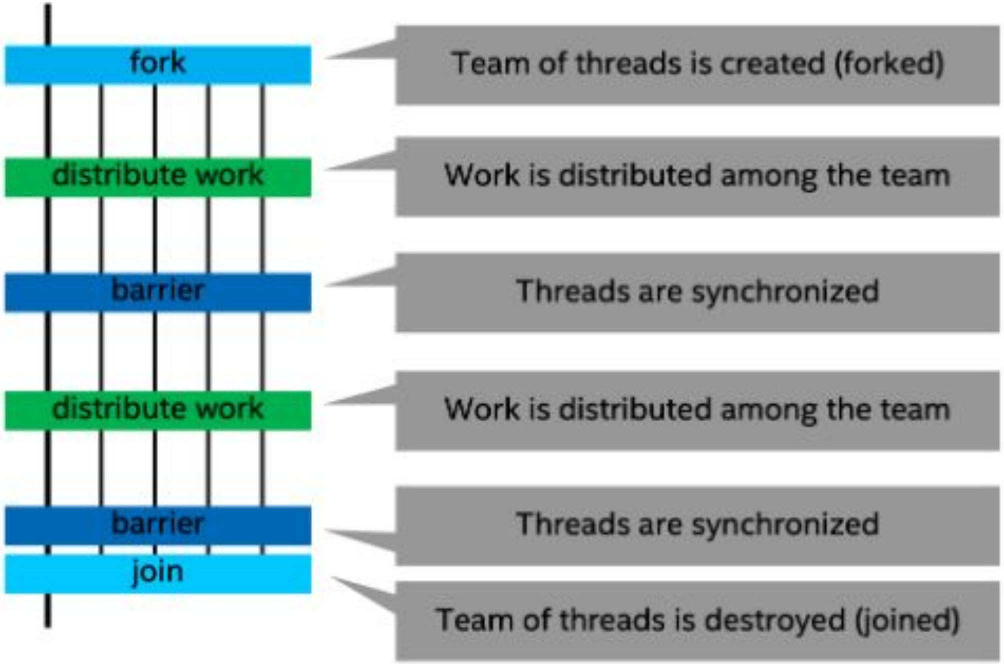
- ✓ **`default(none)`**
  - ○ All variables are undefined: the must be defined shared or private
- ✓ **`default(shared)`**
  - ○ All variables are defined shared; you must define the private one
- ✓ **`default(private)`**
  - ○ All variables are defined shared; you must define the shared one
- ✓
- ✓ Best practice
  - ○ Use **`default(none)`**; compile rise an error if a variable is not defined….

Scheduling clause

OpenMP allows to define the chunk to dispatch to the tasks via scheduling clause

- ✓ **`static`**
  - ○ Minimal overhead
  - ○ same chunk for all the threads in a fixed round-robin way
  - ○ useful when first n/2 iteration ask for different time with respect next n/2 iterations
  - ○ chunk size can be defined by user: **`schedule(static,10)`**
- ✓ **`dynamic`**
  - ○ each chunk is allocated at a thread in a first-in first-our way
  - ○ useful when the iterations presents different execution time
  - ○ chunk size can be defined by user: **`schedule(dynamic,2)`**
- ✓ **`guided`**
  - ○ like dynamic but with reduction of the chunk size
  - ○ useful for unbalanced workflow
  - ○ chunk size can be defined by user: **`schedule(guided,100)`**

OpenMP workflow

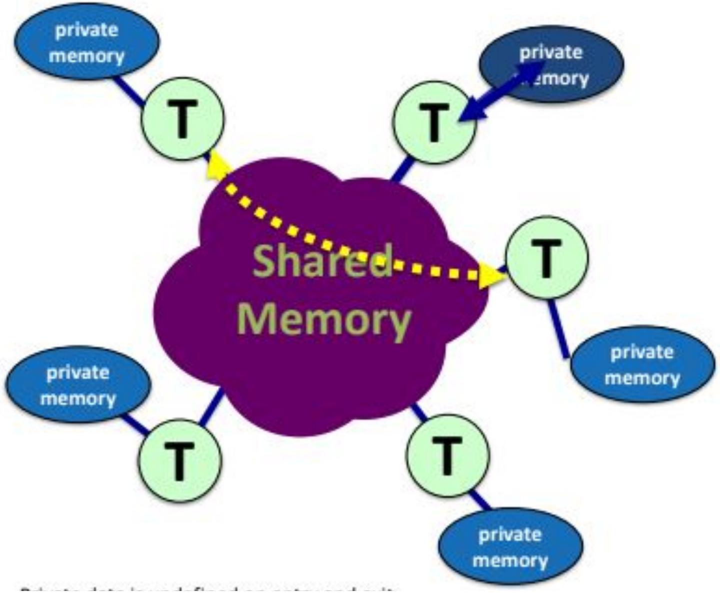

fork
Team of threads is created (forked)
distribute work
Work is distributed among the team
barrier
Threads are synchronized
distribute work
Work is distributed among the team
barrier
Threads are synchronized
join
Team of threads is destroyed (joined)

Developer must take care of

✓ workshare region
✓ data scope
✓ scheduling
✓ thread synchronization

Data Race

- ✓ when threads access to the same memory location
- ✓ It is up to the developer to avoid data races with correct synchronization model
  - ○ `barrier`
  - ○ `single`
  - ○ `ordered`
  - ○ `master`



How compile?

✓ Code directives look like a comment and are ignored by the compiler

✓ You can activate directives using an OpenMP compliant compiler by the appropriate flags

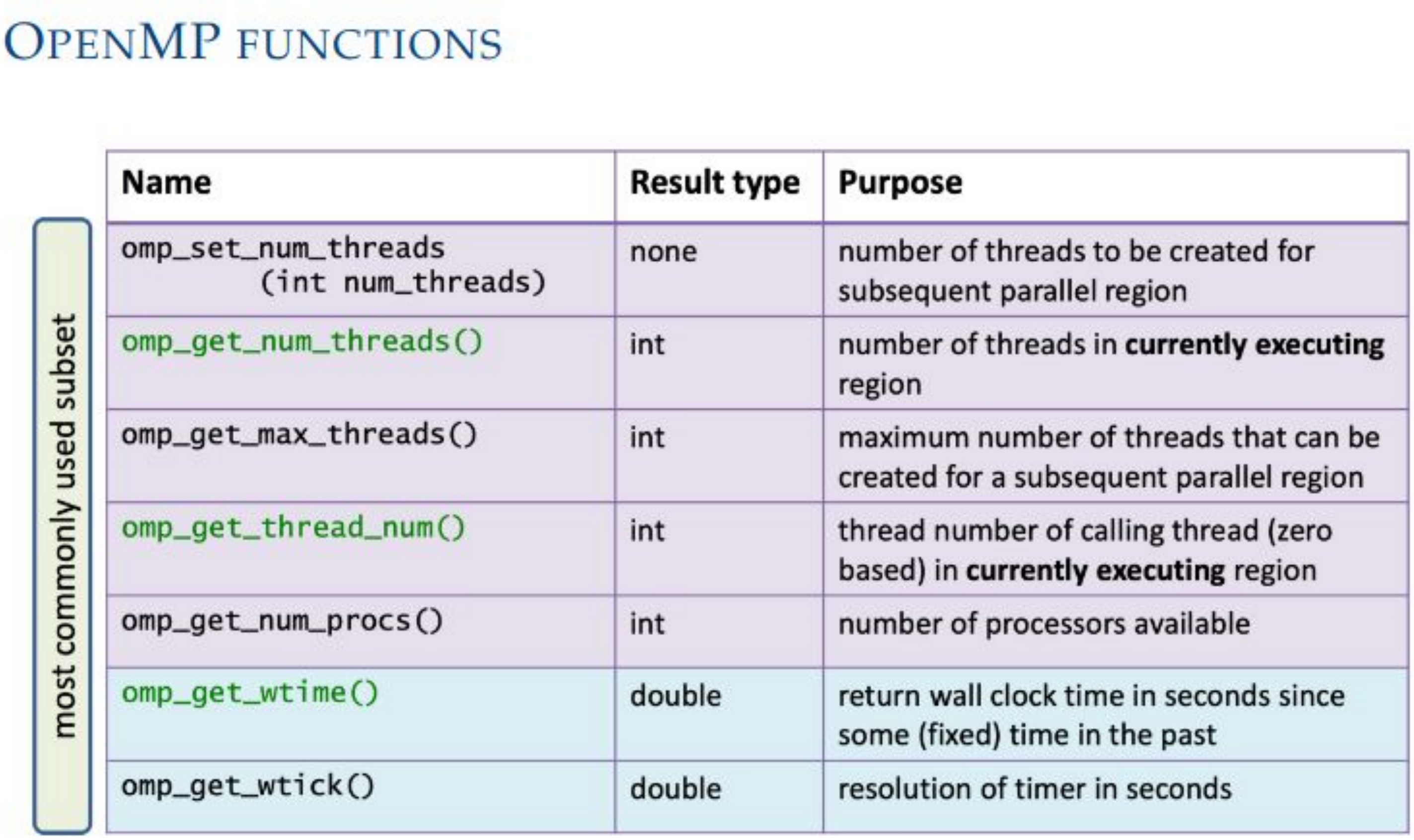
Compiler	Fortran	C/C++
GNU	`gfortran -fopenmp`	`gcc/g++ -fopenmp`
Intel	`ifort/ifx -qopenmp`	`icc/icpc -qopenp`
PGI/Nvidia	`pgfort/nvfortran -mp`	`pgcc/nvcc -mp`

How manage threads?

- ✓ At run time
  - `export OMP_NUM_THREADS=8`
  - `OMP_NUM_THREADS=3 ./exe`

- ✓ in the code
  - in directive
    - `#pragma openmp parallel num_threads(8)`
  - with a function
    - `void omp_set_number_treads(int num_tr)`

OpenMP API



OpenMP functions

most commonly used subset

Name	Result type	Purpose
omp_set_num_threads (int num_threads)	none	number of threads to be created for subsequent parallel region
omp_get_num_threads()	int	number of threads in **currently executing** region
omp_get_max_threads()	int	maximum number of threads that can be created for a subsequent parallel region
omp_get_thread_num()	int	thread number of calling thread (zero based) in **currently executing** region
omp_get_num_procs()	int	number of processors available
omp_get_wtime()	double	return wall clock time in seconds since some (fixed) time in the past
omp_get_wtick()	double	resolution of timer in seconds

Fortran

Matrix matrix multiplication:

```
!$OMP PARALLEL DO &
!$OMP DEFAULT(NONE) &
!$OMP PRIVATE(i,j,k)  &
!$OMP SHARED(a,b,c,n)
   do j = 1, n
      do k = 1, n
         do i = 1, n
            c(i,j) = c(i,j) + a(i,k)*b(k,j)
         enddo
      enddo
   enddo
!$OMP END PARALLEL DO
```

Different Level of optimization

- ✓ **Single core optimization**
  - ○ Vectorization
  - ○ Data access
  - ○ Serial Optimized libraries
  - ○ Compiler optimization
- ✓ **Single-node optimizations**
  - ○ Intra-node optimization
  - ○ Data access
  - ○ Shared memory Parallelization/Distributed memory Parallelization
  - ○ Offloading
  - ○ Shared Memory Optimized Libraries
- ✓ **Multi-node optimizations**
  - ○ Distributed memory optimization (i.e. load balancing)
  - ○ Parallel Optimized libraries
- ✓ **I/O issues**

You cannot skip one single level of optimization!!

Optimized Libraries

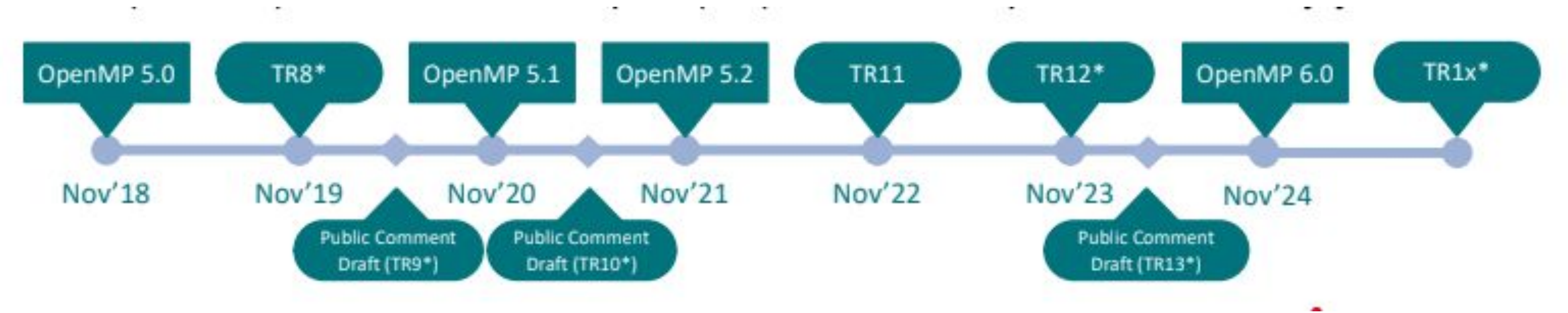
- ✓ There are many optimized libraries
  - ○ Serial: BLAS, LAPACK,...
  - ○ Parallel: PETSc, Trillinos, SCALAPACK, FFTW…
  - ○ Proprietary: ESSL, MKL, ….
- ✓ Usually could be robust or flexible or fast (but few times all together)
- ✓ Sometimes it can help to perform a part of you problem more hard to solve all your problem
- ✓ library maintenance could be a problem, for the open source one

- ✓ **HINT**: do not reinvent the wheel! Look around if you find something that’s fine for you (no one needs to write a parallel FFT from scratch)

OpenMP roadmap

✓ Very long history

✓ There is much more under the hood (rel.5.2 OpenMP api document has 669 pages….)



Reference

- ✓ Introduzione al calcolo parallelo conOpenMP (in italian)
- ✓ A “Hands-on” Introduction toOpenMP
- ✓ ISC24 Tutorial on OpenMP
  - ○ (some slides & images were “borrowed” from here)