

# Advanced OpenMP Sections

Luca Tornatore - I.N.A.F. 

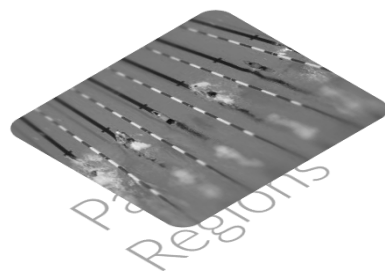
**“Foundation of HPC” course**



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



# OpenMP Outline



Advanced  
Parallelism



# Advanced Parallelism Outline



Advanced  
Parallelism  
in OpenMP



Hybrid codes  
MPI + OpenMP



# Advanced Parallelism Outline



## Advanced Parallelism in OpenMP

Sections

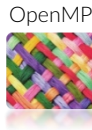
*This lecture*

Tasks





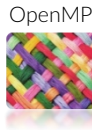
# | SPMD & Work-sharing



- A parallel construct like `for` amounts to create a “**S**ingle **P**rogram **M**ultiple **D**ata” (SPMD) instance: all the threads execute the same code but on different data.
- Other work-sharing constructs are instead about assigning *different execution paths* through the code among the threads (**M**ultiple **P**rogram **M**ultiple **D**ata, MPMD)
  - **section** construct
  - **tasks** construct



# | A general view (recap)



Multi-cores architectures

Around mid of 2000's, it became clear that speedup applications relying on the scaling-up of CPU's frequency was no longer possible.



Heterogeneous computing

Parallel computing

not a direct consequence of the end of "free lunch", but deeply affected by it

New paradigm for programming

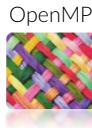
Increasing fine-grain parallelism

The challenge in writing complex scientific and data-intensive applications has increasingly become manifold.

- (1) to identify the parts of the works that can be parallelized, and to expose that parallelism;
- (2) to consider about resource contention, particularly to concurrent data access
- (3) to identify a finer-grained parallelism, decomposing the workflow in smaller well-defined "sequences" of operations that use a subset of data, with well-defined dependencies with other "sequences"



# Different approaches



I

Providing concurrency mechanisms through APIs

The programmer is in charge to find and implement the parallelism, and to manage the concurrency

MPI, POSIX threads  
early OpenMP

II

OOO-features and general parallel patterns

The aim is to alleviate the programmer's burden making the technical details as invisible (transparent) as possible

Intel TBB, HPX,  
FastFlow, ...

III

Inherent parallel constructs

Native parallelism in the language, improve readability and compactness, support sync and concurrency control.  
Mostly based on Partitioned Global Address Space (PGAS), which focuses on data instead of communications/concurrency controls

OpenMP, UPC, Chapel,  
CoArray Fortran,...

IV

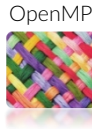
Task-based approach, automatic extraction of parallelism

“tasks” are defined in different ways, and a graph of dependencies is derived (implicitly or explicitly): nodes are the procedures and edges are the relations among them.  
The “assignment” of data regions to the tasks determines the safe concurrent access.

Intel TBB, Cilk,  
Charm++, TensorFlow,  
StarPU, ompSs, late  
OpenMP,...



# | OpenMP sections



The easiest way in OpenMP to get MPMD (Multiple Program Multiple Data), i.e. different threads executing different pieces of codes („sections”) to accomplish different jobs.

It is useful when there is an *established number of independent code units* at compile-time:

```
#pragma omp sections clauses...  
{  
    #pragma omp section  
    { code block }  
  
    #pragma omp section  
    { code block }  
}
```

• **Independent** structured block of codes

Each block is executed by one, and only one thread; the assignment is implementation dependent

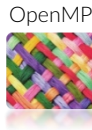
If there are *more sections than threads*, some thread executes more than one section

If there are *more threads than sections*, some threads wait at the **implied barrier at the end** of the sections construct.





# | OpenMP sections

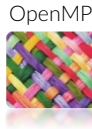


The clauses supported by the `sections` construct are the following:

```
private(list)  
firstprivate(list)  
lastprivate(list)  
reduction(operator:list)  
nowait
```



# | OpenMP sections



```
#pragma omp sections clauses...
```

```
{
```

```
    #pragma omp section
```

```
    function_a(...);
```

```
    #pragma omp section
```

```
    { do_something here
```

```
      function_b();
```

```
    }
```

```
    #pragma omp section
```

```
    { do_something here
```

```
      function_c();
```

```
    }
```

```
}
```

Typical usage of sections is when you have a pre-defined amount of work that can be split in smaller pieces or independent components.

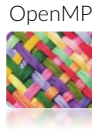
For instance, that may be the case when you have parallel I/O.

Or when you have a computation made up by more independent trunks.

However, this “mechanism” is somehow rigid and may result in a severe work unbalance



# OpenMP sections - ex.0



```
for( int ii = 0; ii < N; ii++ )  
    result += heavy_work_0(array[ii]) +  
             heavy_work_1(array[ii]) +  
             heavy_work_2(array[ii]) ;
```



```
#pragma omp sections reduction(+:result)  
{  
  
    #pragma omp section  
    {  
        double myresult = 0;  
        for( int jj = 0; jj < N; jj++ )  
            myresult += heavy_work_0( array[jj] );  
        result += myresult;  
    }  
  
    #pragma omp section  
    {  
        double myresult = 0;  
        for( int jj = 0; jj < N; jj++ )  
            myresult += heavy_work_1( array[jj] );  
        result += myresult;  
    }  
  
    #pragma omp section  
    {  
        double myresult = 0;  
        for( int jj = 0; jj < N; jj++ )  
            myresult += heavy_work_2( array[jj] );  
        result += myresult;  
    }  
}
```

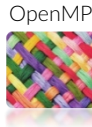
The calls to `heavy_work_?`() are mutually independent. Then, it is possible to separately call the 3 functions for each array entry using the `section` construct.

However, it has no flexibility: the speedup just does not increase while a larger number of threads is used.





# OpenMP sections - ex.0



```
for( int ii = 0; ii < N; ii++ )  
    result += heavy_work_0(array[ii]) +  
             heavy_work_1(array[ii]) +  
             heavy_work_2(array[ii]) ;
```



The calls to `heavy_work_?()` are mutually independent. Then, it is possible to separately call the 3 functions for each array entry using the `section` construct.

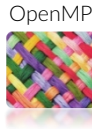
However, it has no flexibility: the speedup just does not increase while a larger number of threads is used.

```
-----  
executing 00_sections  
-----  
  
there are 4 NUMA nodes  
  
    running the serial version  
        running 0/3  
        running 1/3  
        running 2/3  
34.0559 +- 0.0323754  
    running with 3 threads  
        running 0/3  
        running 1/3  
        running 2/3  
15.1382 +- 0.0391657  
    running with 4 threads  
        running 0/3  
        running 1/3  
        running 2/3  
15.0902 +- 0.00148997  
    running with 8 threads  
        running 0/3  
        running 1/3  
        running 2/3  
15.2701 +- 0.257149  
    running with 12 threads  
        running 0/3  
        running 1/3  
        running 2/3  
15.3972 +- 0.206368  
    running with 16 threads  
        running 0/3  
        running 1/3  
        running 2/3
```





# OpenMP sections - ex.1



```
#pragma omp sections reduction(+:result)
{

    #pragma omp section
    {
        double myresult = 0;
        for( int jj = 0; jj < N; jj++ )
            myresult += heavy_work_0( array[jj] );
        result += myresult;
    }

    #pragma omp section
    {
        double myresult = 0;
        for( int jj = 0; jj < N; jj++ )
            myresult += heavy_work_1( array[jj] );
        result += myresult;
    }

    #pragma omp section
    {
        double myresult = 0;
        for( int jj = 0; jj < N; jj++ )
            myresult += heavy_work_2( array[jj] );
        result += myresult;
    }

}
```



We can add some flexibility by spawning a nested parallel region in each section



examples\_sections/  
01\_sections\_nested.c

```
#pragma omp sections reduction(+:result)
{
    #pragma omp section
    {
        double myresult = 0;
        #pragma omp parallel for reduction(+:myresult)
        for( int jj = 0; jj < N; jj++ )
            myresult += heavy_work_0( array[jj] );
        result += myresult;
    }

    #pragma omp section
    {
        double myresult = 0;
        #pragma omp parallel for reduction(+:myresult)
        for( int jj = 0; jj < N; jj++ )
            myresult += heavy_work_1( array[jj] );
        result += myresult;
    }

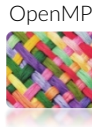
    #pragma omp section
    {
        double myresult = 0;
        #pragma omp parallel for reduction(+:myresult)
        for( int jj = 0; jj < N; jj++ )
            myresult += heavy_work_2( array[jj] );
        result += myresult;
    }

}
```





# OpenMP sections - ex.1



```
#pragma omp sections reduction(+:result)
{

    #pragma omp section
    {
        double myresult = 0;
        for( int jj = 0; jj < N; jj++ )
            myresult += heavy_work_0( array[jj] );
        result += myresult;
    }

    #pragma omp section
    {
        double myresult = 0;
        for( int jj = 0; jj < N; jj++ )
            myresult += heavy_work_1( array[jj] );
        result += myresult;
    }

    #pragma omp section
    {
        double myresult = 0;
        for( int jj = 0; jj < N; jj++ )
            myresult += heavy_work_2( array[jj] );
        result += myresult;
    }

}
```



We can add some flexibility by spawning a nested parallel region in each section

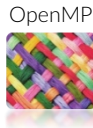
```
# -----
# idx 1
# 01 20k 20k
#   nested parallelism
#   first level is always 3 threads
#
1      31.7      0.01
3      14.21     0.008
6       7.43     0.014
12     3.89     0.14
24     2.03     0.021
30     1.7      0.01
36     1.42     0.017
```



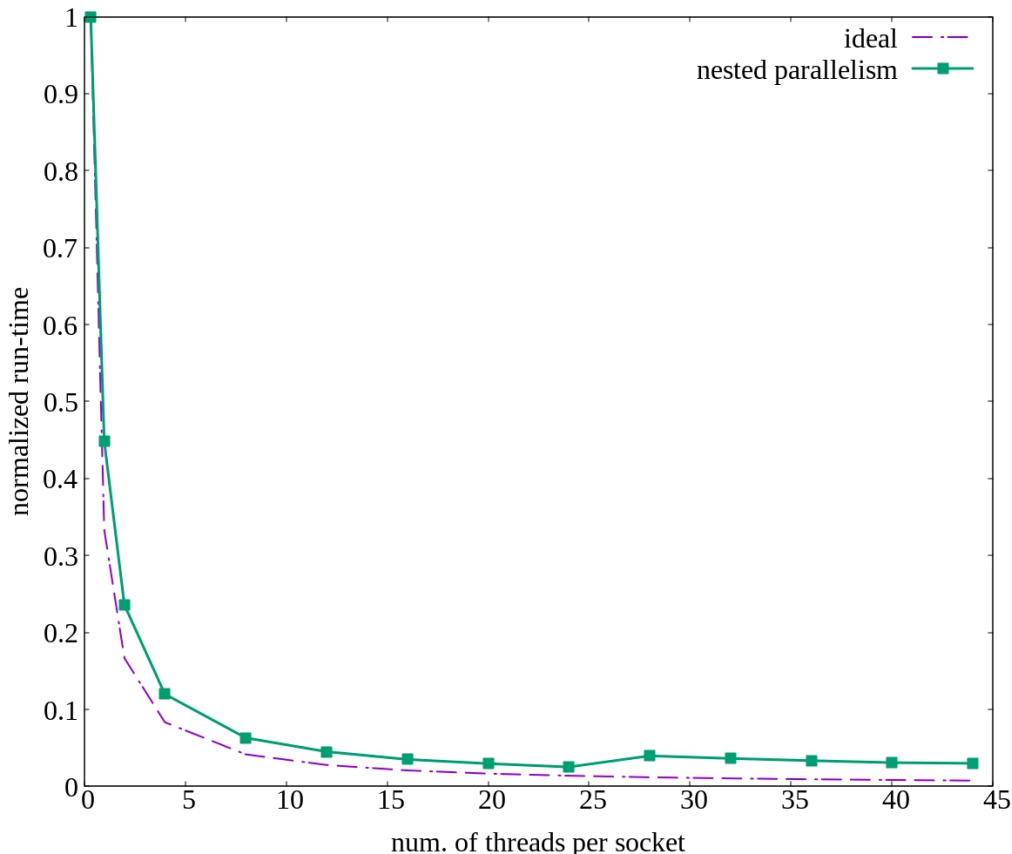
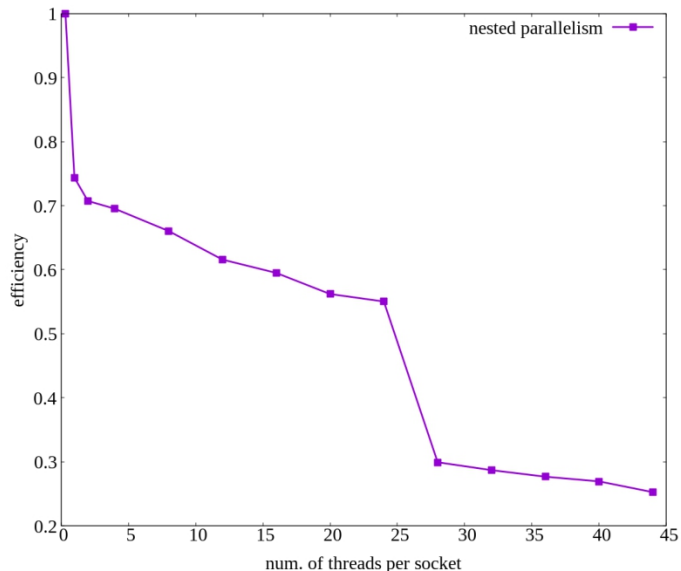
examples\_sections/  
01\_sections\_nested.c



# OpenMP sections - ex.1

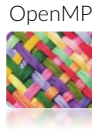


We do obtain some scalability here, although far from perfect





# | OpenMP sections



To add some more flexibility in the `sections` mechanism, some dependences among the sections can be enforced explicitly by using some usual tools as `locks` and `semaphores`:

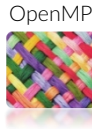
- **locks** : a mechanism that causes a behaviour very similar to critical regions but with more flexibility. A memory region is used as a flag to signal that some task, or some data, is/are under processing and that anybody else can not go on with any task that depends on that locked task/data being free.

*Note: OpenMP has native locks, but we're not covering this topic.*

- **semaphores** : similar to locks, but it is just a shared memory region used as a signal about something.



# | OpenMP sections



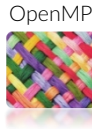
Let's suppose that in our example the data are “arriving” from outer space in irregular bunches with irregular delay.

Then, a section could be devoted to receive the data while the other ones to processing the data *as soon as they are ready*.

Since the sections are independent, it is not required that they process the same bunches of data anytime. In fact, we profit from this to acquire even more flexibility.



# OpenMP sections



```
int semaphores[3] = {0};
#pragma omp sections clauses...
{
    #pragma omp section
    { while( there_is_work0 ) {
        function_0(...);
        set_semaphore(0, ...); } }

    #pragma omp section
    { while( there_is_work1 ) {
        check_semaphore(0, ...);
        function_1();
        set_semaphore(1, ...);} }

    #pragma omp section
    { while( there_is_work2 ) {
        set_semaphore(1, ...);
        function_2(); } }
}
```

## What semaphores may look like

A simple semaphore implementation (conceptual, many details have been discarded)

```
void set_semaphore( int i, int val ) {
    // in principle you don't need atomic here
    // because each entry should be modified by
    // only one thread
    semaphore[i] = val; }

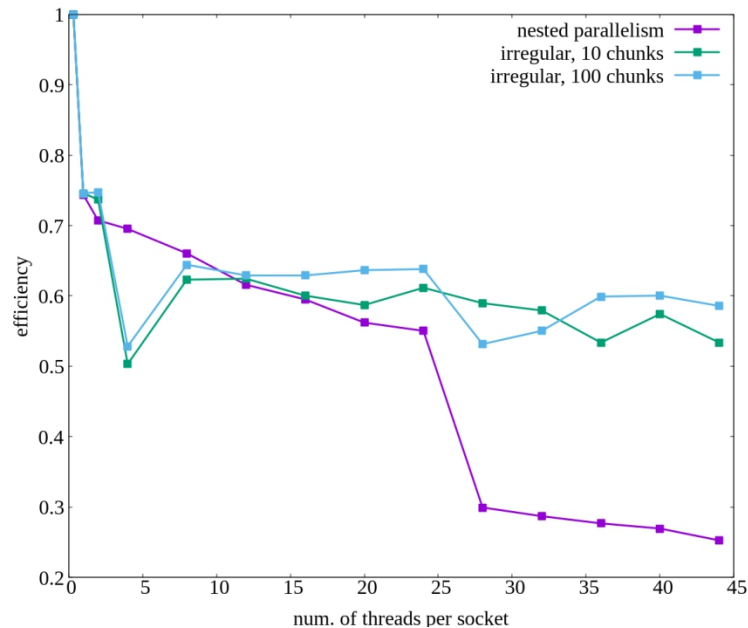
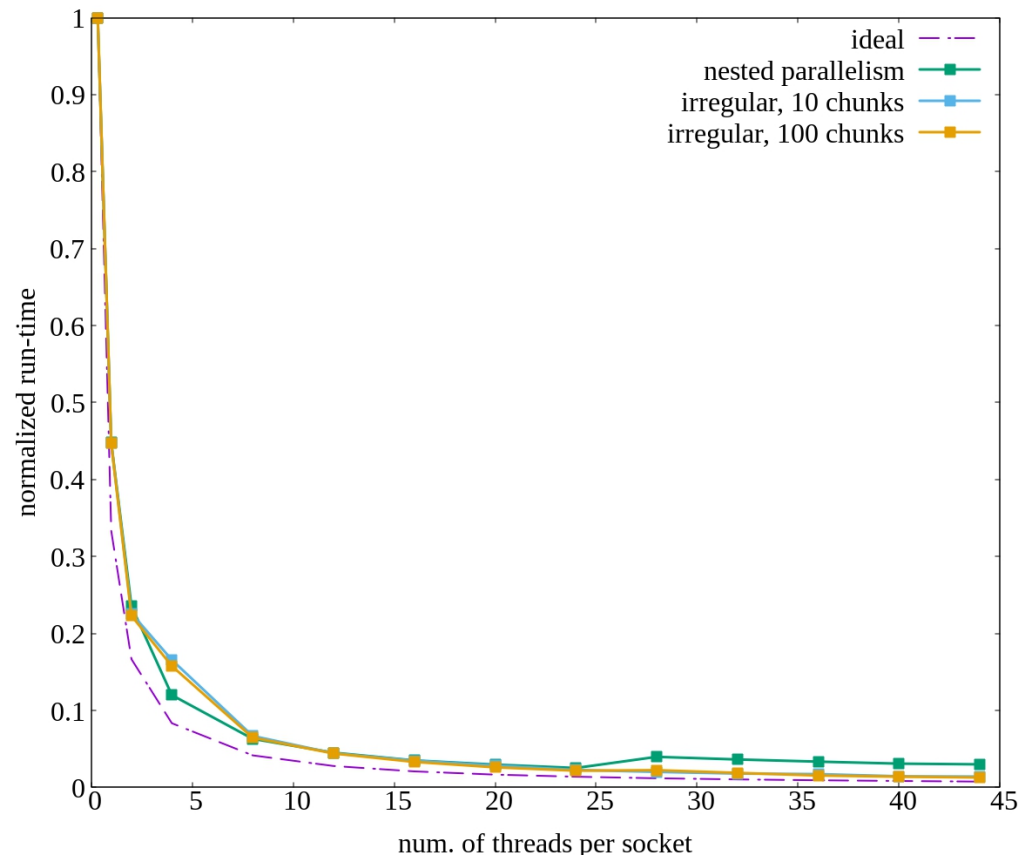
void check_semaphore( int i, int val ) {
    while( semaphore[i] != val) {
        sleep_or_spin_a_while(...); } }

void sleep_or_spin_a_while( int a_while ) {
    // you may use a call to nanosleep() instead
    volatile int sum = 0;
    for( int ii = 0; ii < N; ii++ ) sum += ii; }
```





# OpenMP sections



parallel\_tasks/  
02\_sections\_nested\_irregular.c

that's all, have fun

"So long  
and thanks  
for all the fish"