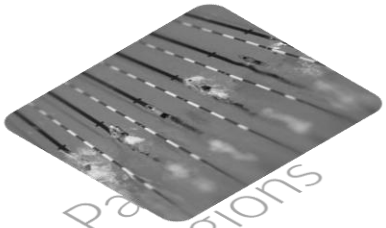
Parallel Computing & OpenMP - Loops

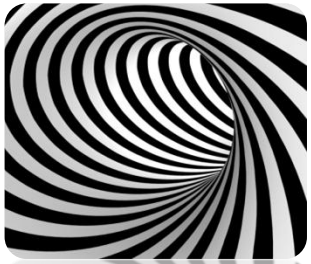
Luca Tornatore - I.N.A.F.

"**Foundation of HPC" course**

DATA SCIENCE &
SCIENTIFIC COMPUTING
2020-2021 @ Università di Trieste

OpenMP Outline





Parallel Loops

Advanced Parallelism

NUMA AWARENESS

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 could 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

TIPS

```
#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

TIPS

```
#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

A *data race* happens when at least two memory accesses
- point the same location
- are performed concurrently by different threads
- are not sync ops

and at least one is a `write`.

A *race condition* is a semantic error in the code. It causes the fact that its behaviour may be non-deterministic and its correctness is affected due to the random ordering of events.

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();
   #pragma omp for
   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();
   #pragma omp for
   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

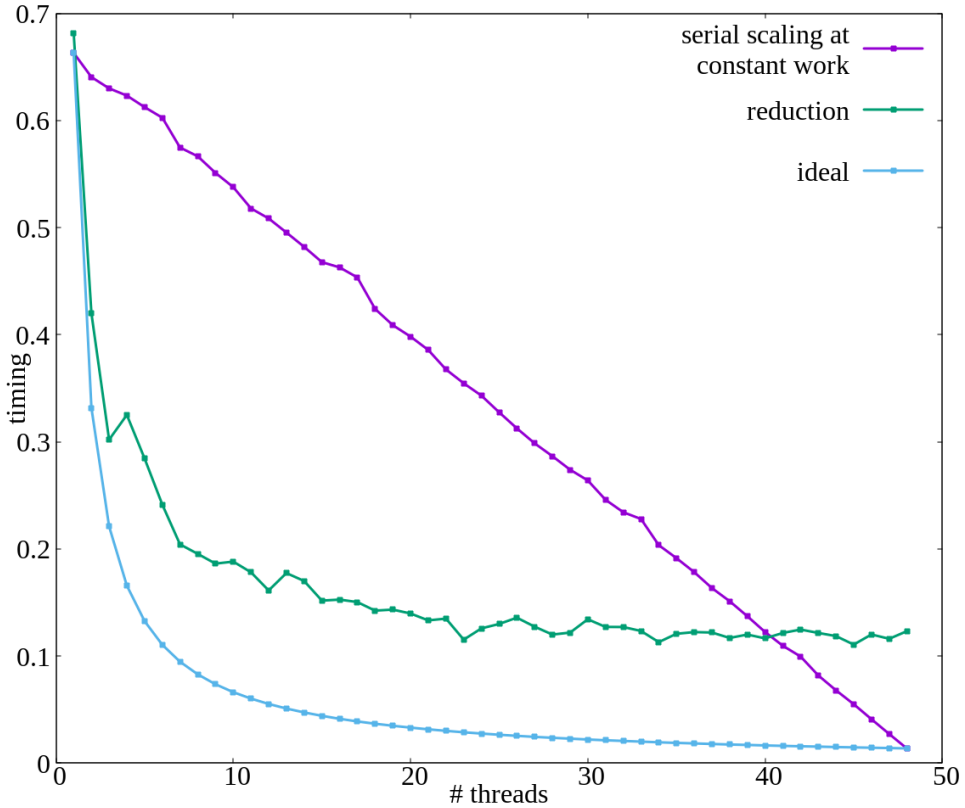
```
#include <omp.h>
double *a;
int     N;
int     nthreads;

#pragma omp master
nthreads = omp_get_num_threads();

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

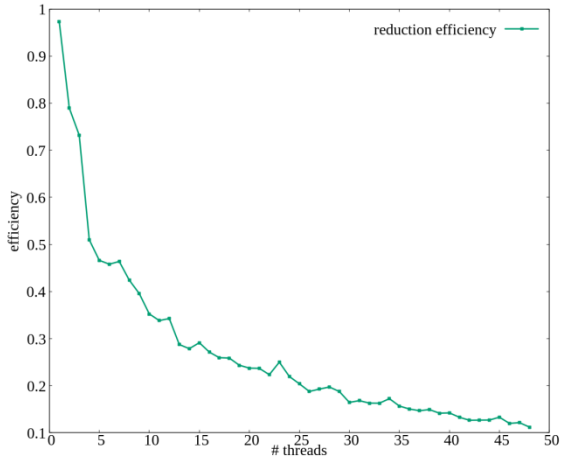
Does this scale ?

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).

Solving the reduction / 6





It seems that, after all, our reduction efficiency is very poor.
One would say that OpenMP is somehow a bad solution. Of course That is not true, we'll learn that this problem is due to a bad thread affinity.

A subtlety to note

These two "`i`"s are two different variables, although both are thread-private.

```
#include <omp.h>
#pragma omp parallel
{
   int i;
   #pragma omp for
   for (i = 0; i < N; i++ )
      {
         …;
      };
}
```

parallel_loops/
01b_array_sum.c

```
luca@GGG:~/work/TEACHING/CODES/OpenMP/parallel_loops$ ./01b_array_sum
omp summation with 4 threads
thread 0 : &i is 0x7ffc27c4b954
        thread 0 : &loopcounter is 0x7ffc27c4b958
thread 1 : &i is 0x7f59f56c3ae4
        thread 1 : &loopcounter is 0x7f59f56c3ae8
thread 2 : &i is 0x7f59f52c1b64
        thread 2 : &loopcounter is 0x7f59f52c1b68
thread 3 : &i is 0x7f59f4ebfb64
        thread 3 : &loopcounter is 0x7f59f4ebfb68
Sum is 4950, process took 0.000830412 of wall-clock time
```

OpenMP work assignment in loops

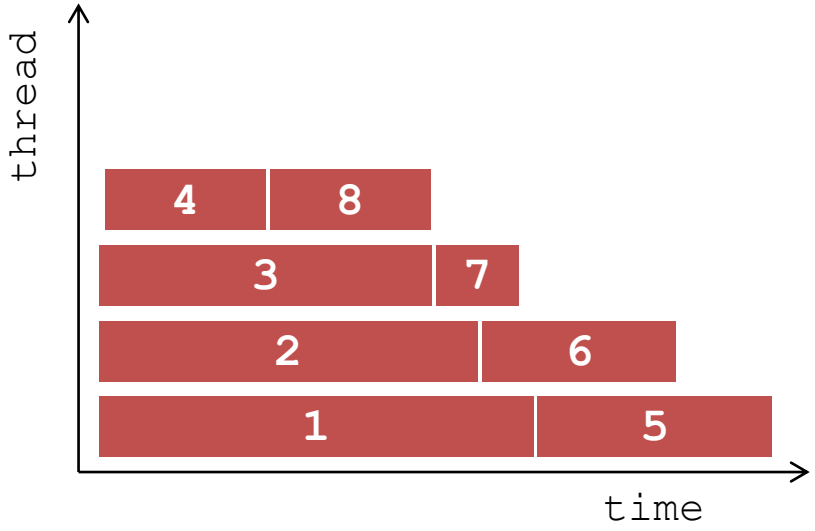
How the work is assigned to the single threads ?

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

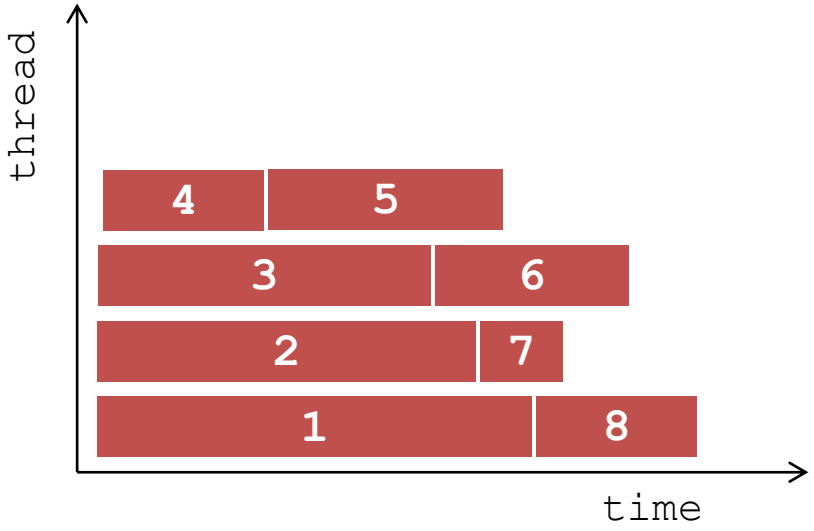
schedule(**static**, *chunk-size*)	The iteration is divided in chunks of size *chunk-size* (or in ~equal size) distributed to threads in circular order
schedule(**dynamic**, *chunk-size*)	The iteration is divided in chunks of size *chunk-size* (or size 1) distributed to threads in no given order (a thread 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.

OpenMP work assignment in loops

Static vs Dynamic



Static assignment



Dynamic assignment

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*

firstprivate & lastprivate

```
double PI              = 3.1415blablabla;
int    morning_coffees = MAX_INTEGER;
char   password[]      = “dont_ask_dont_tell”
int    final_mark;

#pragma omp parallel firstprivate( PI, morning_coffees) private(password) lastprivate( final_mark)
{
    drink_mycoffees( morning_coffees );
    use_pi( PI );

    password = setup_mypassword();
    int exam_passed = 0;
    while (!exam_passed) { exam_passed = try() }

    final_mark = exam_passed;
}
```

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 (must be perfectly nested)

```
#pragma omp for collapse(2)
for ( int ii = 0; ii < Nrows; ii++ )
   for ( int jj = 0; jj < Ncol; jj++ )
      A[i][j] = B[i][j] * C[i][j];
```

```
#pragma omp for collapse(2)
for ( int ii = 0; ii < Nrows; ii++ ) {
   D[i] = function_of_(i);
   for ( int jj = 0; jj < Ncol; jj++ )
      A[i][j] = B[i][j] * C[i][j] + D[i]; }
```

`nowait`

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

that's all, have fun