Parallel Computing & OpenMP - Loops

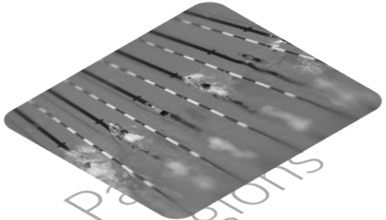
Luca Tornatore - I.N.A.F.

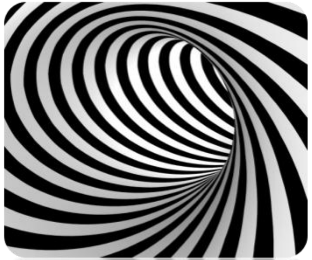
DATA SCIENCE & ARTIFICIAL INTELLIGENCE

SCIENTIFIC & DATA-INTENSIVE COMPUTING

2023-2024 @ Università di Trieste

OpenMP Outline





Parallel
Loops

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 rem     = N % team;
  int mystart = size*mid + ((me < rem)? myid : rem);
  int myend   = size + (rem > 0)*(myid < rem);

  for ( int i = mystart; i < myend; i++ )
    {
      do_something(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

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 region

```
#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];
```

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

TIPS

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

The default policy for memory regions is actually that all the variables defined in serial regions at the moment of entering in the parallel region 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

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

When you declare a `for` construct, it is implicit that the loop counter is private if it was declared outside the parallel region; as such there actually is no need for the clause `private(i)`.

Nevertheless, it is preferrable, for the sake of clarity, to use the C99 standard practice to declare the loop counter inside the for declaration

```
for ( int i = ... ; ... ; ... )
```

OpenMP basic loop

However, variables defined (outside)within the parallel region are automatically (shared)private, and so are the integer indexes used as loop 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 the single threads ?

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 on a first-arrived-first-served basis)

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 still unassigned iterations divided by the number of threads.

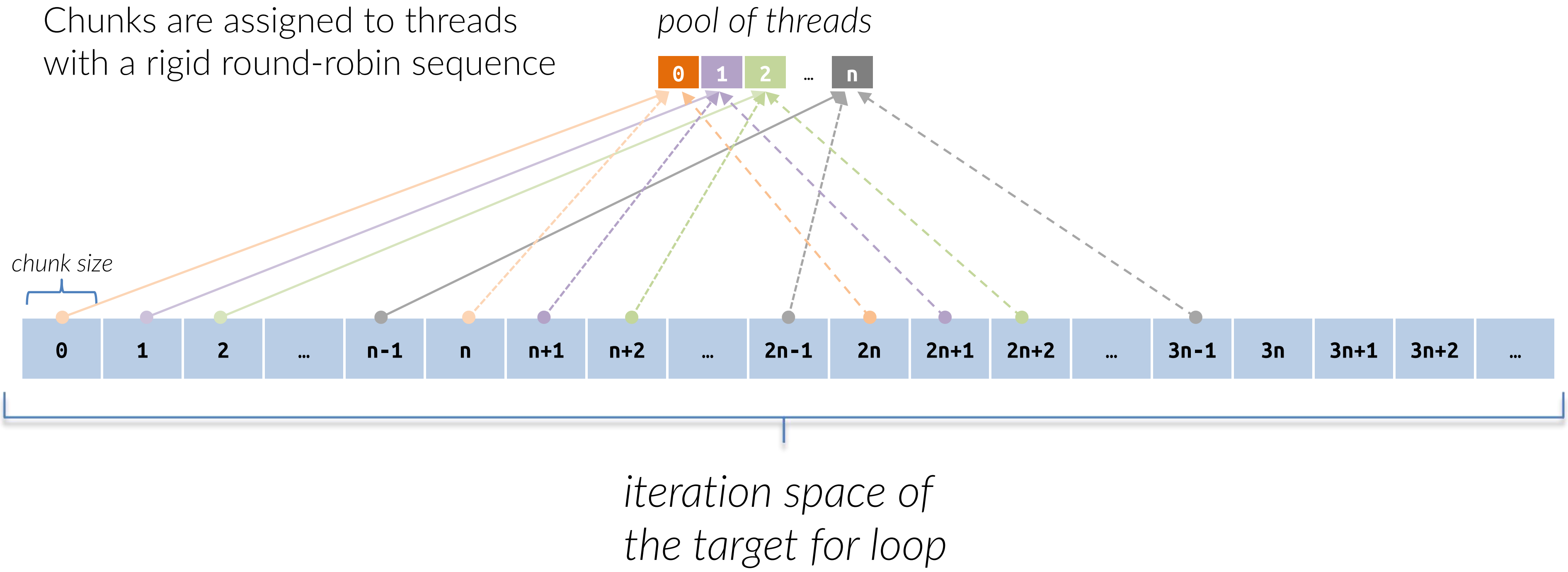
runtime , default

The policy is set at runtime via the env. var. OMP_SCHEDULE or, if that is not defined, to a default implementation-dependent value.

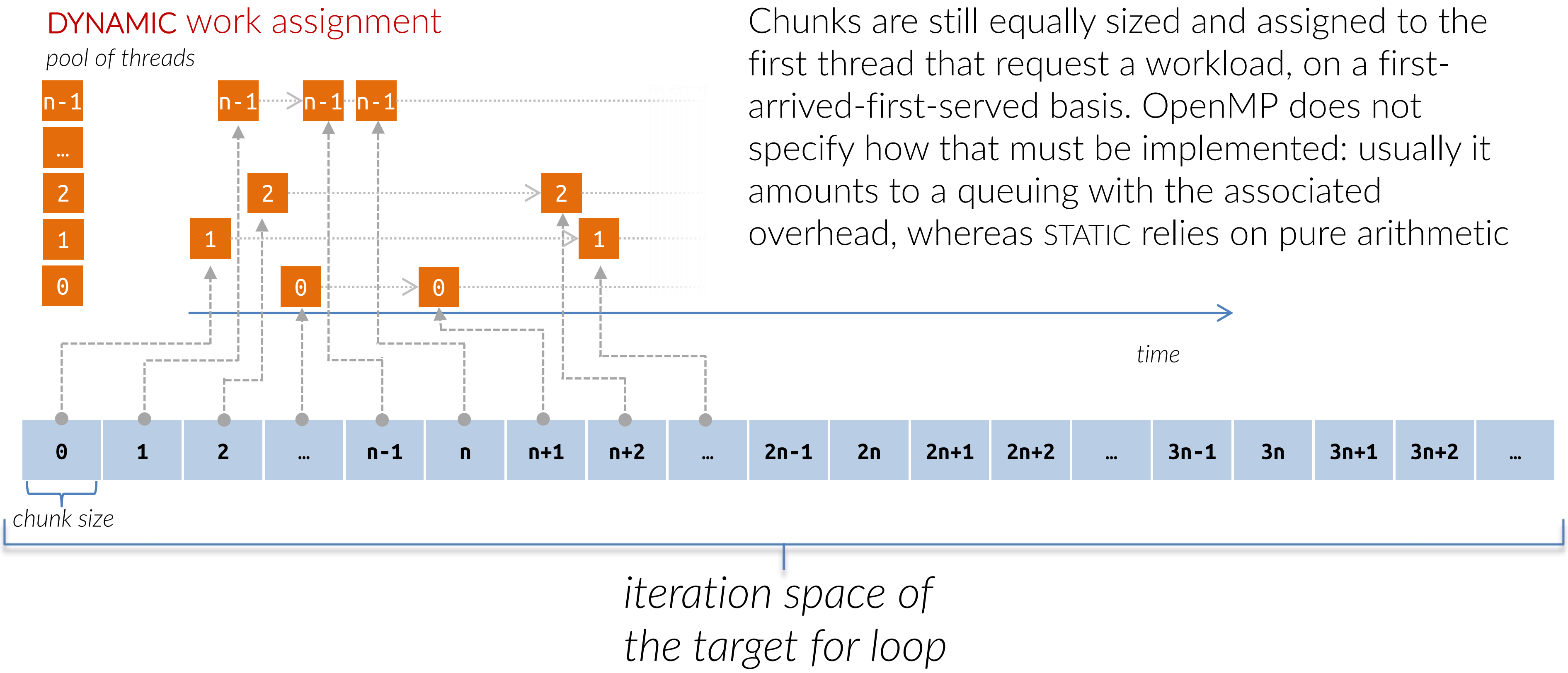
OpenMP work assignment in loops

STATIC work assignment

Chunks are assigned to threads with a rigid round-robin sequence



OpenMP work assignment in loops



Chunks are still equally sized and assigned to the first thread that request a workload, on a first-arrived-first-served basis. OpenMP does not specify how that must be implemented: usually it amounts to a queuing with the associated overhead, whereas STATIC relies on pure arithmetic

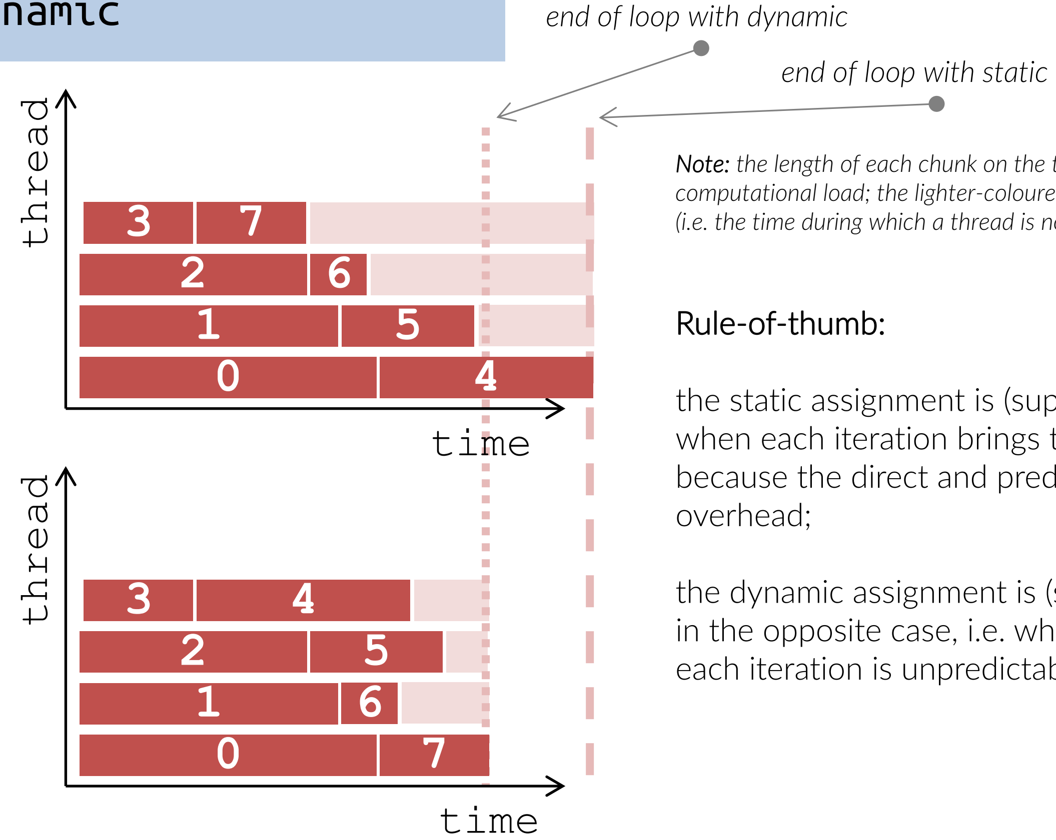
Static vs Dynamic

Static assignment

```
th 0    ← chunk 0
th 1    ← chunk 1
th 2    ← chunk 2
th 3    ← chunk 3
th 4    ← chunk 4
th 5    ← chunk 5
th 6    ← chunk 6
th 7    ← chunk 7
…
```

Dynamic assignment

```
Chunks are assigned
to the first free
thread
```



Note: the length of each chunk on the time-line is "proportional" to its computational load; the lighter-coloured area represent the idle time of each thread (i.e. the time during which a thread is not processing any chunk).

Rule-of-thumb:

the static assignment is (supposed to be) most effective when each iteration brings the same computational work, because the direct and predictable assignment has a smaller overhead;

the dynamic assignment is (supposed to be) most effective in the opposite case, i.e. when the computational load of each iteration is unpredictable.

`parallel_loops/04_scheduling.c`

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)
{
    drink_mycoffees( morning_coffees );
    use_pi( PI );
    password = setup_mypassword();
    int exam_passed = 0;
    while (!exam_passed) { exam_passed = try() }
   #pragma omp for lastprivate( final_mark)
    for( int i; i < nthreads; i++ )
      final_mark = exam_passed;
}
```

parallel_loops/
05_first_and_last_private.c

`PI` and `morning_coffees` are private copies of the homonym variables outside the parallel region;

`password` is just a different variable that has nothing to do with the outer `password`;

`final_mark`, which arrives at the last for cycle as a shared variable, at the end will contain the last `final_mark` assigned

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*)

```c
#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];
```

this won't work -------->

```c
#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

Anatomy of a data race

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
- at least one is a `write`.

A *race condition* is a semantic error in the code. Due to the random ordering of events, it leads the fact that its behaviour is non-deterministic and the result is not correct.

You find here a very nice discussion on these two concepts https://blog.regehr.org/archives/490

Anatomy of a data race


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

Let's get back to our example to examine what may happen in reality.

Let's suppose that we are using 2 threads and that array `a` has the following 10 entries:

```
a[] = {1,2,3,4,5,6,7,8,9,10};
```

If we use schedule static, `thread 0` will process `a[0:4]` and `thread 1` will process `a[5:9]`
Remember that the single-line instruction

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

actually requires in the following simplified (pseudo)intructions:

```
mov sum → regA      fetch sum from memory register A;
add a[i],regA
mov regA → sum
```

Anatomy of a data race

Then, the following is one among many possible erratic combinations of execution.

Not protecting the access to S, leads to an incorrect access.

Cycle	S value in memory	Thread 0 operation	note	Thread 1 operation	note
…	…	…		…	
0	0	-	*let's suppose thread 0 is 1 cycle late*	fetch S → regA	*now regA = 0*
1	0	fetch S → regA	*now regA = 0*	add regA, a[i]	*a[i=5] = 5*
2	0	add regA, a[i]	*a[i=1] = 1*	fetch regA → S	*regA = 5*
3	5	fetch regA → S	*regA = 1*	inc i	*i = 6*
4	1	inc i	*i = 2*	fetch S → regA	*now regA = 1*
5	1	fetch S → regA	*now regA = 1*	add regA, a[i=6]	*a[i=6] = 6*
6	1	add regA, a[i]	*a[i=2] = 2*	fetch regA → S	*regA = 7*
7	7	fetch regA → S	*regA = 3*	inc i	*i = 7*
…	…	…		…	

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 ? Of course no! why?

```
#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

Solving the reduction

Why the code in the previous slide does not scale?

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**

False sharing

The memory sharing is when you explicitly and purposely share some memory region among the threads; that is intentional and at the very ground of “shared-memory paradigm”. You have to face all the issues about memory synchronization and caches and so on, but that is the tool to pay.

False sharing happens when each thread explicitly accesses a memory location that is *different* than any other thread in the parallel pool BUT at least some of those memory locations are mapped on the *same cache line*.
The effect of this is a very poor efficiency when te threads run on different cores.

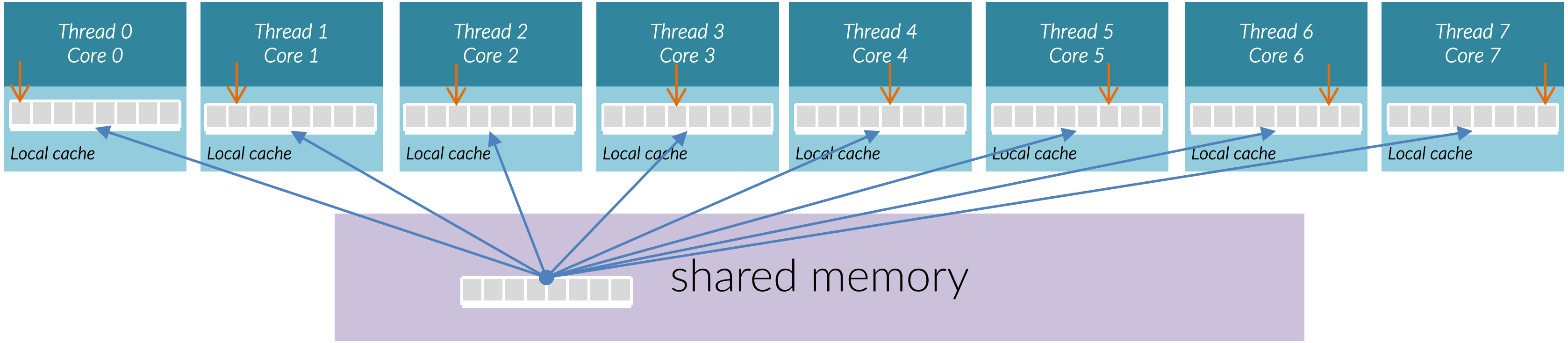
NOTE: the false sharing is an issue when it happens a large amount of times, like in the previous example. Having an array that stores values peculiar for each threads so that they are exposed to all the other threads that access them only once a while, is something very common and not an issue (it actually is what shared memory is about..).

Anatomy of a false sharing

```
…
double sum[nthreads];
…
#pragma omp parallel
{ modify sum[me]; }
```

Each thread is modifying a memory location that is not accessed by other threads, but that is contiguous to the memory locations modified by the other threads. When we described the MESI protocol for the cache coherence, we referred to abstract “data”; however, remember that the caches *always* work by *lines* and not by single bytes and, consequently, that some bytes are modified in a cache line the whole line is flagged as modified/invalid.

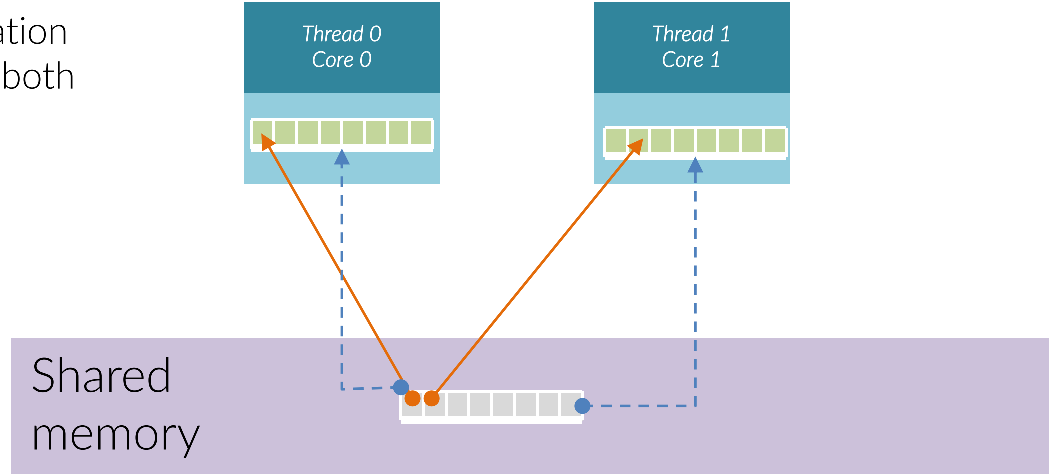
Let’s suppose then that 8 threads are running on 8 different cores and that each of them is writing in an array’s element `array[thread_id]`; we also suppose this array be aligned to a cache line.



Anatomy of a false sharing

Let's concentrate on only 2 threads in order to understand in details how false sharing works.

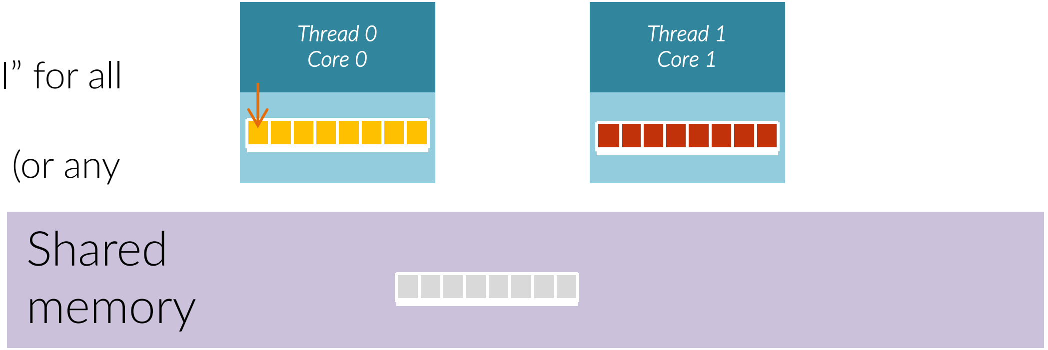
[0] both threads read their target memory location
[1] the entire line is loaded up in the cache of both cores
[2] the line is flagged as "S" (shared)



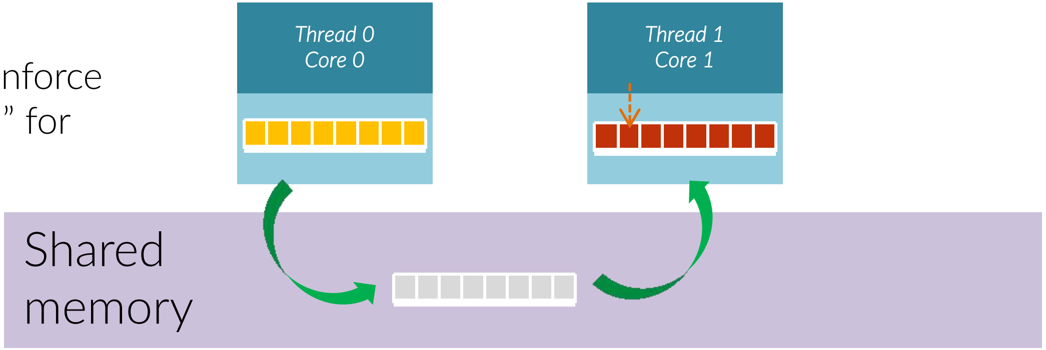
parallel_loops/
03_falsesharing.c
04_falsesharing_fixed.c

Anatomy of a false sharing

[3] th 0 writes its target location
[4] the line is flagged as “M” for thread 0 and “I” for all the others threads
[5] th 0 could write again on its target location (or any other one in the line) w/o modifying the situation

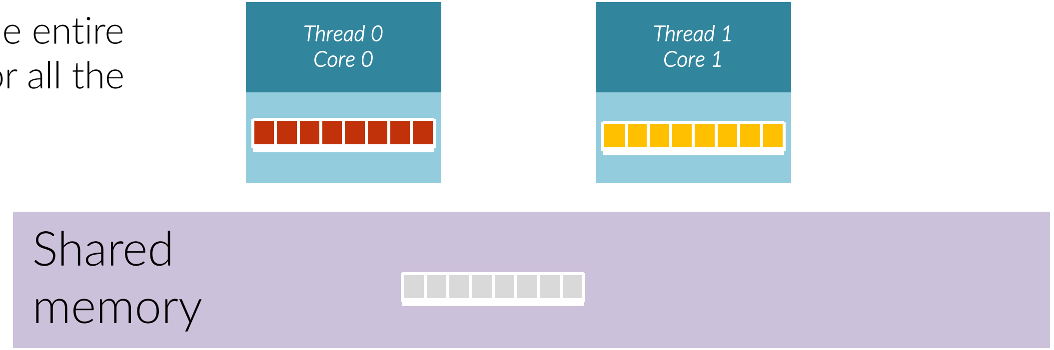


[5] thread 1 wants to write its target location
[6] the entire cash line must be re-flushed to enforce cache-coherence because it is flagged as “I” for thread 1

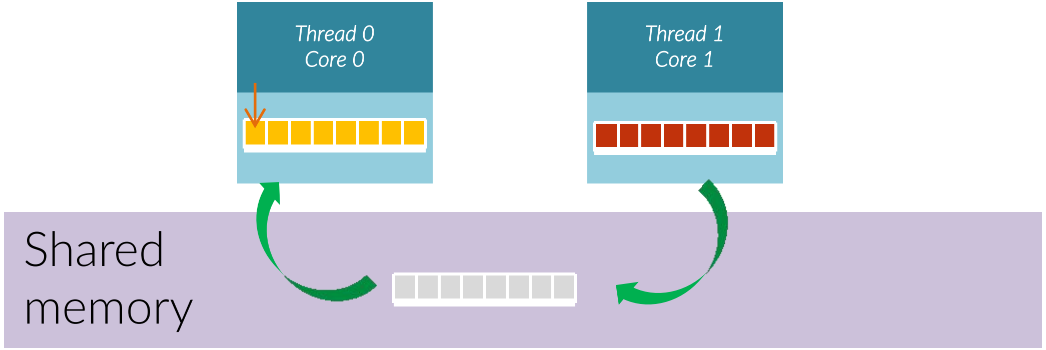


Anatomy of a false sharing

[7] once th 1 has written its target location, the entire cache line is flagged “M” for th 1 and “I” for all the other threads



... and so on...



Solving the reduction /5

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

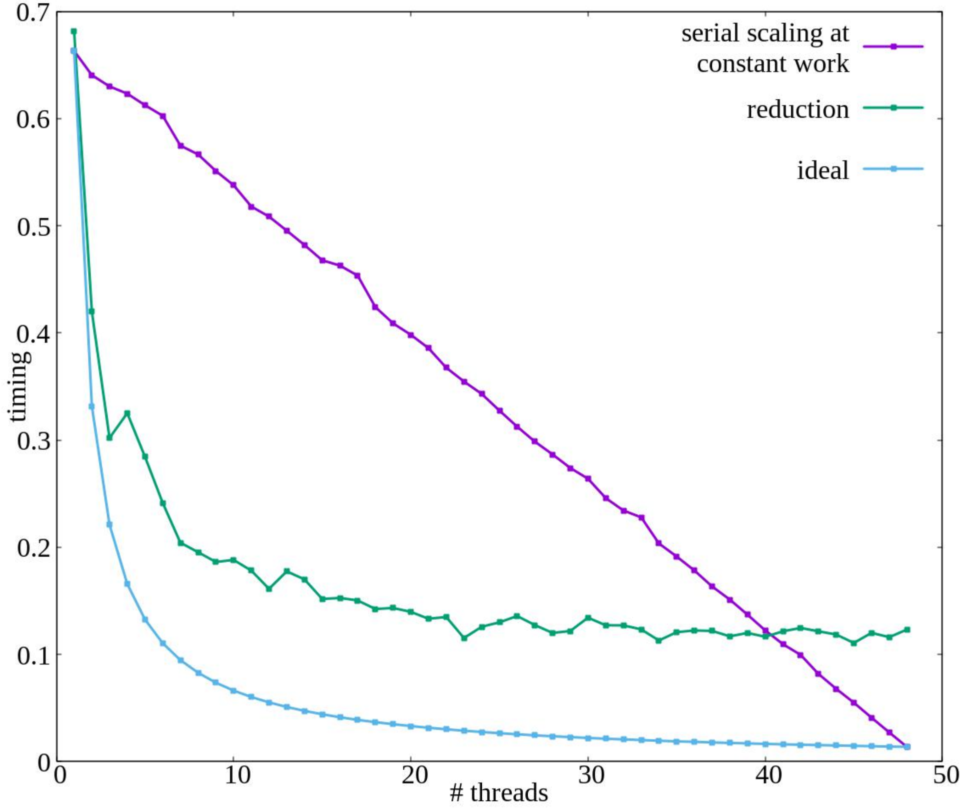
parallel_loops/03_falsesharing_fixed.c

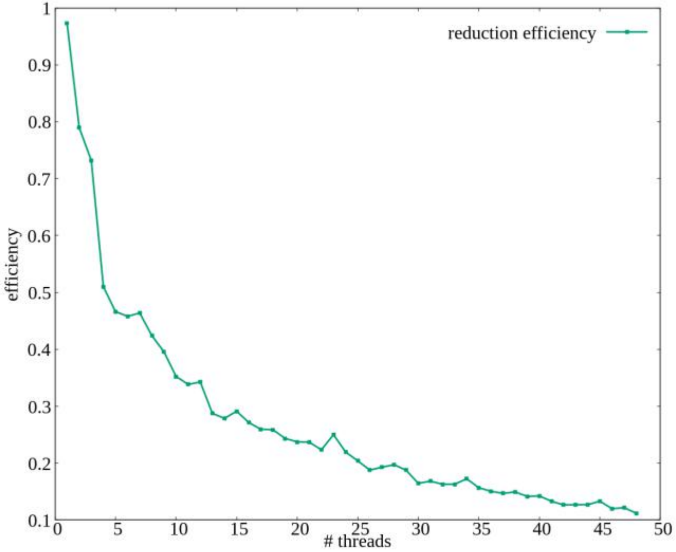
```
#include <omp.h>
double *a;
int      N;
int      nthreads;

#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. Although one could conclude that OpenMP is somehow a bad affair, hidden in these plots there is a very important issue in multi-threading that we inquire in the next lectures.

An important detail

Note

TIPS

When you declare a `omp for` *inside an existing* `parallel` *region*, there is a fundamental difference between the two codes here below. In the snippet A on the left, the `for` is declared without the `parallel`, whereas on the right, in snippet B, it has the `parallel`.

In A the `for` is shared among the threads that form the pool of the outer `parallel` region.

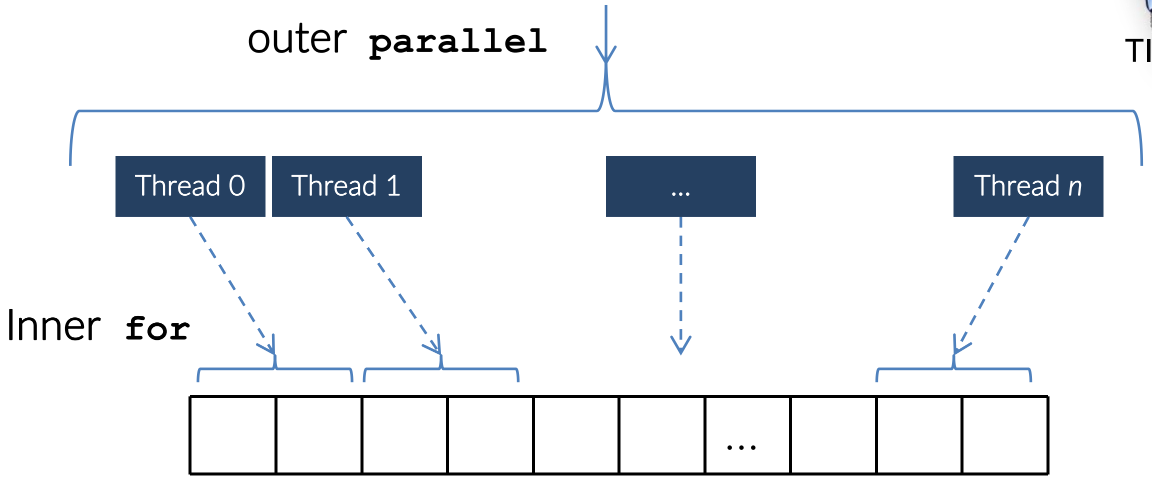
In B *every thread of the pool* is creating a new `parallel` region and inside each of the new `parallel` regions the new pools of threads will execute the `for`. So, in case B, if there are *n* threads in the outer `parallel` you will have *n* `for` cycle executed.

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

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

An important detail

TIPS



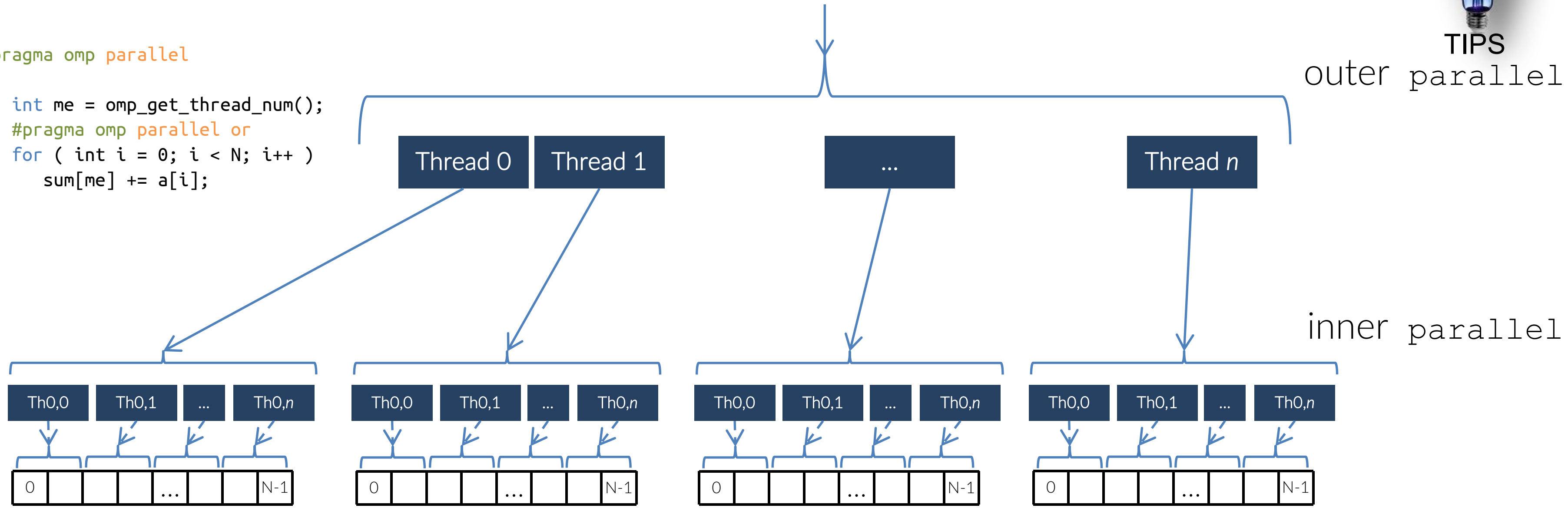

```
A
#pragma omp parallel
{
   int me = omp_get_thread_num();
   #pragma omp for
   for ( int i = 0; i < N; i++ )
      sum[me] += a[i];
}
```

A unique **for** is subdivided among the threads that are in the pool

An important detail

TIPS

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

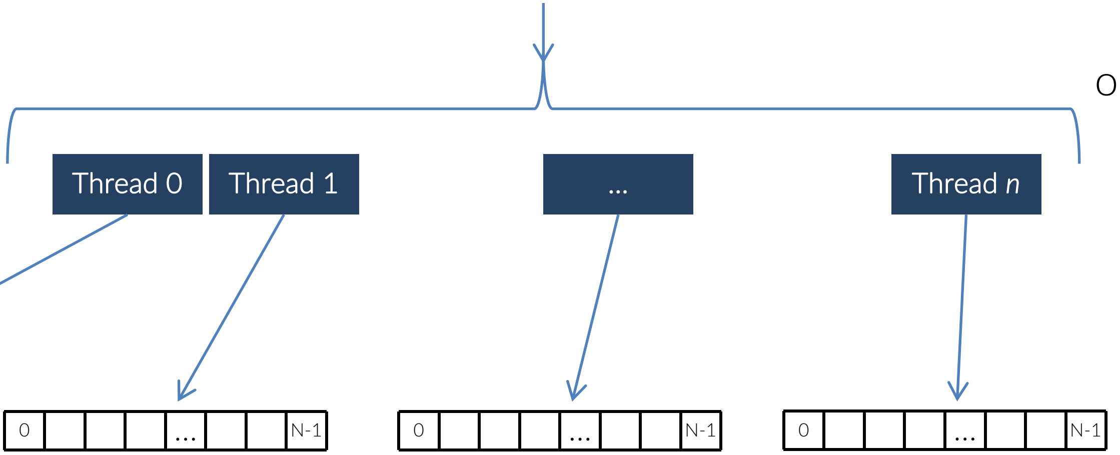


Here every thread hat participates in the outer parallel region spawns a new parallel region whose participating threads will share the for cycle. Then, we have *n* nested parallel regions each of which performs the entire for loop.

An important detail

TIPS
outer `parallel`

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



If the parallel nesting was not active at this level, each thread will enter alone in the inner parallel region executing it sequentially.
Then, each thread will execute the entire for loop.

A subtlety to note

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

```c
#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
```

that’s all, have fun