

Luca Tornatore - I.N.A.F.

"Foundation of HPC - basic" course



DATA SCIENCE &
SCIENTIFIC COMPUTING
2022-2023 @ Università di Trieste



OpenMP Outline





Parallel Regions



Loops



Parallel & Advanced Parallelism

NUMA **AWARENESS**











```
#pragma omp parallel some clauses here
single-line-here
#pragma omp parallel _some_clauses_here_
{ ... }
#pragma omp parallel for _some_clauses_here_
{ ... }
#pragma omp sections some clauses here
{ ... }
#pragma omp task some clauses here
{ ... }
```

A parallel region can be as short as a single line

There are no limits on the size of the code included within {..}.

The specific construct about for loops

A more general work-sharing construct

This allows task-based parallelism

The region starts at the opening brace and ends at the closing } one.

An implicit synchronization barrier is present at the end of the region (*).

However, for efficiency reasons, it may be that the threads do not die at the end of the region but remain alive until the father process dies

(*) we'll see the details about synchronization later on





When you create a parallel region, through an OpenMP directive, a pool of threads is created.

Each one receives an ID, ranging from 0 to n-1, where n is the number of threads.

Their stack and IP are separated from the others' ones and from the father-process' ones.

Can we check that? What is the fate of the creating process (thread)?

```
int
     i;
register unsigned long long base of stack asm("rbp");
register unsigned long long top of stack asm("rsp");
 printf( "\nmain thread (pid: %d, tid: %d) data:\n"
         "base of stack is: %p\n"
         "top of stack is : %p\n"
                          : %td\n"
         getpid(), syscall(SYS gettid),
         (void*)base of stack,
         (void*)top of stack,
         (void*)base of stack - (void*)&i,
         (void*)&i - (void*)top of stack );
#pragma omp parallel private(i)
    int me = omp get thread num();
    unsigned long long my stackbase;
    asm ("mov %%rbp,%0" : "=mr" (my stackbase));
    printf( "\tthread (tid: %ld) nr %d:\n"
                        "\t\tmy base of stack is %p ( %td from main\'s stack )\n",
                        "\t\tmv i address is %p\n"
                        "\t\t\t\td from my stackbase and %td from main\'s\n",
                        syscall(SYS gettid), me,
                        (void*)my stackbase, (void*)base of stack - (void*)my stackbase,
                        &i, (void*)&i - (void*)my stackbase,
                        (void*)&i - (void*)base of stack);
```







```
main thread (pid: 26291, tid: 26291) data: base of stack is: 0x7ffe6f51efc0
                                                                             threads for the OS.
serial
             top of stack is : 0x7ffe6f51ef60
             &i is
                               : 0x7ffe6f51ef7c
section
                 rbp - &i
                               : 68
                 &i - rsp
                               : 28
              thread (tid: 26291) nr 0:
                      my base of stack is 0x7ffe6f51eee0 ( 224 from main's stack )
                      my i address is 0x7ffe6f51eea0
                               -64 from my stackbase and -288 from main's
             thread (tid: 26293) nr 2:
                      my base of stack is 0x7f7342c82ba0 ( 597747680288 from main's stack )
parallel
                      mv i address is 0x7f7342c82b60
region
                               -64 from my stackbase and -597747680352 from main's
             thread (tid: 26294) nr 3:
                      my base of stack is 0x7f7342880ba0 ( 597751882784 from main's stack )
                      my i address is 0x7f7342880b60
                               -64 from my stackbase and -597751882848 from main's
             thread (tid: 26292) nr 1:
                      my base of stack is 0x7f7343084b20 ( 597743477920 from main's stack )
                      my i address is 0x7f7343084ae0
                               -64 from my stackbase and -597743477984 from main's
parallel regions/
```

pid and tid are the IDs of processes and

Here you see that the master thread stops its activity and join the new pool of threads, while maintaining its own tid.

- Each thread has its own stack; thread 0 inherits its own stack, which has grown to host the data relative to OpenMP parallel region;
- the private is are actually in the threads stacks, and so they are at different memory locations than the shared i.





How large is the stack of each thread? The default value is system dependent.

That is the output of the 00 stack and scope code example: it prints the BP and SP pointers of each thread. Then, we know how large is the threads' stack at a given moment (only an integer is defined, plus the system's thread structure), and how much memory is reserved for the stack to grow:

```
thread 0: bp @ 0x7ffc6edea2f0, tp @ 0x7ffc6edea280: 112 B
thread 1: bp @ 0x7fa371d18b20, tp @ 0x7fa371d18ab0: 112 B
                                                                 --> -382202615648 B from top of thread 0
thread 2: bp @ 0x7fa371916ba0, tp @ 0x7fa371916b30: 112 B
                                                                 --> -4202256 B from top of thread 1
thread 3: bp @ 0x7fa371514ba0, tp @ 0x7fa371514b30: 112 B
                                                                 --> -4202384 B from top of thread 2
you did not define OMP STACKSIZE: try to set it and check what happens to the threads' stack pointers
```

In this case, compiled with gcc, it seems that about 4MB are reserved for each thread. The same value holds with pgi, whereas clang seems to reserve 132MB (i.e., the addresses in the virtual space are spaced by 132MB).

However, you can control the size of the threads' stack using the environmental variable omp STACKSIZE: export OMP STACKSIZE=N

Where N is a number, followed by a letter: 'β', 'κ', 'M', 'G' mean bytes, kilobytes, megabytes and gigabytes respectively (if no letter is specified, the number is interpreted as kilobytes).





export OMP STACKSIZE=32K

```
thread 0: bp @ 0x7ffd1a2bbc70, tp @ 0x7ffd1a2bbbd0: 160 B
thread 1: bp @ 0x7f18b4be4af0, tp @ 0x7f18b4be4a50: 160 B
                                                                 --> -980954214624 B from top of thread 0
thread 2: bp @ 0x7f18b4bdab70, tp @ 0x7f18b4bdaad0: 160 B
                                                                 --> -40672 B from top of thread 1
                                                                 --> -40800 B from top of thread 2
thread 3: bp @ 0x7f18b4bd0b70, tp @ 0x7f18b4bd0ad0: 160 B
you defined OMP STACKSIZE as 32K: try to change that value and check what happens to the threads' stack pointers
```

export OMP STACKSIZE=1M

```
thread 0: bp @ 0x7ffc6f8b97f0, tp @ 0x7ffc6f8b9750: 160 B
thread 1: bp @ 0x7fd5bf930af0, tp @ 0x7fd5bf930a50: 160 B
                                                                 --> -166161058912 B from top of thread 0
thread 2: bp @ 0x7fd5bec96b70, tp @ 0x7fd5bec96ad0: 160 B
                                                                 --> -13213408 B from top of thread 1
thread 3: bp @ 0x7fd5beb94b70, tp @ 0x7fd5beb94ad0: 160 B
                                                                 --> -1056608 B from top of thread 2
```

you defined OMP STACKSIZE as 1M: try to change that value and check what happens to the threads' stack pointers





How many threads can be created by a single process?

Also this limit is system-dependent.

On Linux, it depends on the amount of total physical memory: basically, the maximum number of threads is the amount of physical memory divided by the memory amount needed to describe and run a thread, times a factor that accounts for the fact that you don't want all the memory allocated only to make the threads alive.

Within this limit, you can change the behaviour of your OpenMP program by using the env variable OMP THREAD LIMIT.

You can check the OS' limit with

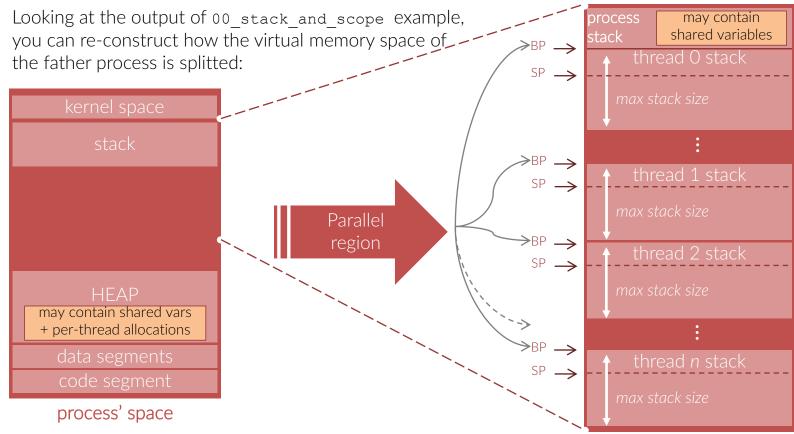
cat /proc/sys/kernel/threads-max

or by calling the omp library function

int omp get max threads()

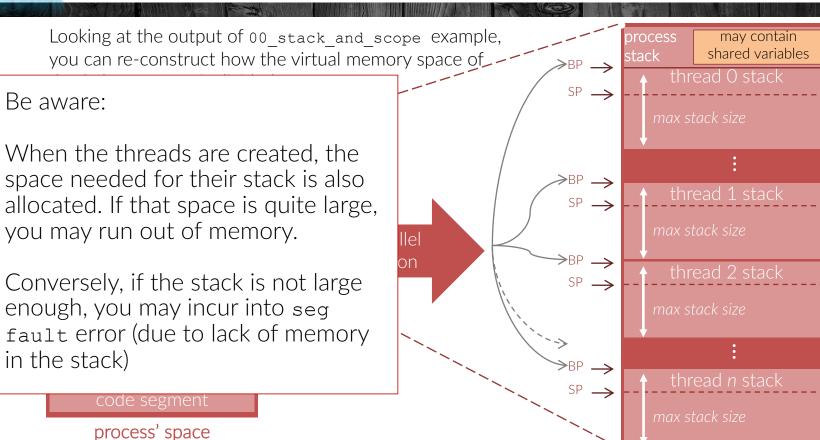






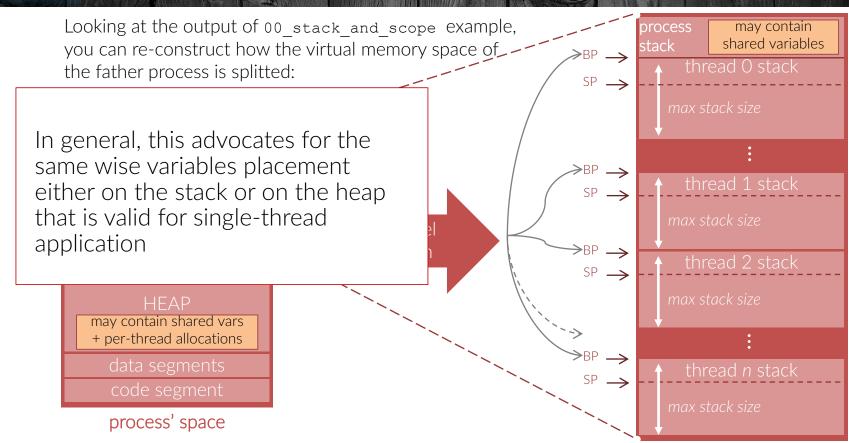
















```
returns the ID of the calling thread
int nthreads
int my thread id = 0;
#ifdef _OPENMP
#pragma omp parallel
    my_thread_id = omp_get_thread_num(); 
    #pragma omp master
    nthreads
                   = omp_get_num_threads();__
#endif
          returns the number of threads active
          in that parallel region
```

By default rules, variables declared outside a parallel region are shared, whereas those declared inside a region are private.

Then both nthreads and my thread id are shared.

As a consequence, in this example:

- all threads are writing in my thread id, in undefined order
- only the master thread is writing in nthreads

The value of my thread id is unpredictable, because it depends on the run-time order by which the threads access it and by each a thread accesses it again to write it.

parallel regions/

01 simple pr wrong.c





```
int i:
#pragma omp parallel private(i)
   ... using i
```

This code, is perfectly following the OpenMP standards and of which you still find several examples - i.e. shared variables defined as private at the entrance of the parallel regions.

However, in my opinion, it's just a relic from old times and a wrong way of thinking and programming.

In fact, "i" as used in the parallel region refers to a memory region that is different than the "i" in the serial region: as such, this coding style looks only a source of confusion and lacks of cleariness.









```
int i:
..using i
#pragma omp parallel
  int j;
  ..using j
..using i
```



The effect of the previous code is the same than this one: you have an integer in the stack of each thread, without any need of referring it with the same name

In general, then, my advice is: if you need a private variable in the parallel region, just define it clearly inside the parallel region.

If a variable is defined in the serial region, it must have a scope and a usage in the serial region.

Some possibile handy exception will be discussed later, like the usage Of firstprivate Or lastprivate



Controlling the number of threads



By default the number of threads spawned in a parallel regions is the number of cores available. However, you can vary that number in several way

- OMP NUM THREADS environmental variable
- #pragma omp parallel num threads(n)
- omp_set_num_threads(n); #pragma omp parallel





Controlling the number of threads



By default the number of threads spawned in a parallel regions is the number of cores available. However, you can vary that number in several way

- OMP_NUM_THREADS environmental variable
- #pragma omp parallel num threads(n)
- omp_set_num_threads(n); #pragma omp parallel

That works if omp dynamic variable is set to TRUE, otherwise the number of threads spawned is strictly equal to OMP NUM THREADS. This setting can be changed through

omp set dynamic(true)



Foundations of HPC - basic

Data Science & Scientific Computing - 2022/2023





```
#pragma omp parallel
    int my thread id = omp get thread num();
    printf( "greetings from thread num %d\n",
            my thread id );
```

The order by which the greetings appear in the output is not deterministic.

In general, unless either you explicitly requires so – and that is possible only for some constructs, or you directly code it, there is no given order, since the threads are independent entities.

How would you build-up an enforcement of the order in the parallel region shown here on the left?





```
int order = 0;
#pragma omp parallel
    int myid = omp_get_thread_num();
    #pragma omp critical
    if ( order == myid ) {
       printf( "greetings from thread num %d\n",
               my thread id );
        order++; }
```

parallel regions/ 04 order of threads wrong.c

The CRITICAL directive defines a region which is executed by all threads but by a single thread at a time. Which starts to sound like an "ordering".

However, although the threads queue up at the begin of the region, no particular order is imposed to that queue. As a consequence, the result is unpredictable, aside the fact that thread o will print the message and increase order.

For instance, if thread 2 is gueued immediately after, it will execute the if evaluation which will fail (order would be equal to 1) and the thread 2 will not print the message at all.





```
int order = 0;
#pragma omp parallel
    int myid = omp_get_thread_num();
    int done = 0
    while (!done) {
    #pragma omp critical
    if ( order == myid ) {
       printf( "greetings from thread num %d\n",
               my thread id );
        order++; done=1; } }
```

In this second implementation, the while cycle enforces the threads that fails the if condition to keep trying until the correct order is ensured.

Once a thread has executed the critical region, it then exits the while and join the implicit synchronization barrier at the end of the parallel region.

A note: without the critical directive, this code could work as well on most platform. However, it is still unreliable since it is not guaranteed that each thread complete the region before the successive threads enter in it. For instance, the compiler could have been reshuffled the order++ instruction placing it before the print and so the thread *n+1* could enter the region before of the print of the thread n and its message could appear as first.





```
int nthreads = 1;
#pragma omp parallel
    int myid = omp get thread num();
   #pragma omp master
    int nthreads = omp get num threads();
   #pragma omp barrier
   #pragma omp for ordered
    for ( int i = 0; i < nthreads; i++ )</pre>
      #pragma omp ordered
       printf( "greetings from thread num %d\n",
               my thread id );
```

In this third implementation, we use the clause ordered which force a work-sharing loop to be executed in the order of the loop iteration. ordered may also be a stand-alone directive that specifies cross-iteration dependences.

A note: without the barrier directive, it may happen that some threads (potentially all of them) arrive at the worksharing contruct for with a wrong value of nthreads







```
#pragma omp parallel
    #pragma omp critical
    { code block
    #pragma omp atomic
    assignment instruction
    #pragma omp single
    { code block
    #pragma omp master
    { code block
```

The critical directive ensures that only a thread at a time executes the block. All the threads will execute it, although in unspecified order.

Like critical but limited to the following single line. The instruction can only be an assignment in the form x binop = expr

Only one among the threads will enter the code block

Only the master thread will enter the code block





```
#pragma omp parallel
    #pragma omp critical
    { code block
    #pragma omp atomic
    assignment instruction
    #pragma omp single
    { code block
    #pragma omp master
    { code block
```

A barrier (in the form of queuing) is present at the entry point; no one at the end point, i.e. the threads exiting the region will continue the run.

Synchronization as in critical region.

Implicit synchronization at the end. Only one thread is inside the region, the others are waiting at the end of the region.

No explicit synchronization at all. Only the thread 0 is inside the region, the other ones can be everywhere else.







Rationale of

#pragma omp atomic assignment instruction

(*) std C and C++ also expose to the programmer a large bunch of atomic instructions

When you deal with shared variables, ensuring that the workflow maintains the semantic correctness of your code is fundamental. For instance, the assignment

$$a += b$$

(where either a, b or both are shared) is meaningful only if the value of a (or b, or both) does not change in the middle of the instruction itself.

In fact, while a single assembly instruction is guaranteed not to be ever interrupted on the fly, we have to take into account that even a single C line translates in several asm instructions. So, the value of a shared variable could have been changed in the middle of that groups of asm instructions, breaking the correctness of the code (*).

To avoid that, it is necessary to "protect" the sensible regions.

An atomic assignment has, obviously, a much lower overhead than a critical region and must be preferred in the appropriate cases.







Special clauses for **atomic**

read var = mem

write mem = var

Rationale of

#pragma omp atomic assignment instruction

Clauses: read, write, update. capture

update

mem binop= expr

val = mem++,

val = ++memval = mem - -, $val = --mem_{\bullet}$

capture division val = mem binop expr

expr} {mem binop= expr; val = mem

{val = mem; mem binop=

causes an atomic read of the location designated by mem regardless of the native machine word size

causes an atomic write of the location designated by mem regardless of the native machine word size

Causes an atomic update of the location designated by mem using the designated operator or intrinsic

Causes an atomic update of the location designated by mem using the designated operator or intrinsic while also capturing the original or final value of the location designated by x with respect to the atomic update.

The original or final value of the location designated by mem is written in the location designated by val, depending on the form of the atomic construct. structured block, or statements, following the usual language semantics







```
#pragma omp critical
   "A" code block
#pragma omp critical
   "B" code block
#pragma omp critical(my loop)
   code block
```

Named critical regions

In OpenMP it exists a unique global critical section.

Hence, when you define a critical section, it is logically considered to be part of the global one.

As a consequence only one thread can be inside any of the unnamed critical sections, which of course limits the perfomance whn more than one region is present.

However, that can be cured by the *named regions*, defined as the last one in the code snippet here on the left.

In that example, only one thread can be either in region "A" or in region "B": so, if one thread is executing "A", another one is forced to delay entering "B" even it would have been reading for it.

Instead, the named critical regions can be executed at the same time (by different threads, of course, and only by one thread at a time).







```
#pragma omp critical
  some_assignment_to x;
#pragma omp atomic
x op= value
```

critical and atomic

Consider that by design atomic protects memory regions, while critical protects code regions.

Hence they are not mutually exclusive: a thread may be executing the critical region while another may be executing the atomic.







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

```
#pragma omp master
{ ...code block... }
```

Is there any difference between using single or master?

There obviously is if you're assigning some special workflow to thread 0.

If not, the entity of the difference depends on the implementation details.

In general, using master requires only a simple test on the thread id, while using single requires more synchronization (the openmp infrastructure has to keep track about what thread is in the region and so on).

In general, if you have some insight about the fact that the workload before that point may be systematically unbalanced, so that some thread will likely arrive first, using single is ok. Otherwise, it may be better to use master.

Work assignment



Inside a parallel region, the work can be assigned to each threads (actually, that's why PR are created..):

```
#pragma omp parallel
                                                                                      dynamic extent
                                               if( myid % 3 == 0)
                                                   result = heavy_work_0();
                                               else if ( myid % 3 == 1 )
                                                                                       parallel region/
  results[0] = heavy work 0();
                                                                                       06_assign_work.c
                                                   result = heavy work 1( );
  results[1] = heavy work 1();
                                               else if ( myid % 3 == 2 )
                                                                                       run with a second
  results[2] = heavy work 2();
                                                                                       argument >0 to check
                                                   result = heavy work 2( ); V
                                                                                       about it
                                               if ( myid < 3 )
                                                   results[mvid] = result;
parallel region/
```

06a assign work.c



Conditional creation of PR



It is possible to spawn a parallel region only if some conditions are met:

#pragma omp parallel if(any valid C expression)

```
#pragma omp parallel if( amount of work > high )
    if( myid % 3 == 0)
        result = heavy work 0();
    else if ( myid % 3 == 1 )
        result = heavy_work_1( );
    else if ( myid % 3 == 2 )
        result = heavy work 2();
    if ( myid < 3 )
        results[myid] = result;
```

If the condition is not fulfilled. the threads are not created and only the master thread will execute the code block.

```
parallel region/
06a assign work.c
if the first argument, that determines
the amount of work, is \leq 10, the
parallel region is not created
```



Conditional creation of PR





What is this useful for?

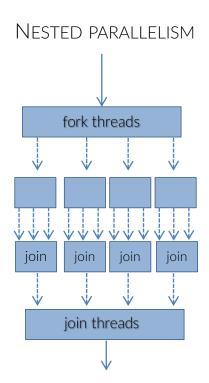
The overhead of the creation of a parallel region is of the order of $\sim 10 \, \mu seconds$ (that figure is dependent on the system, the compiler, the number of threads,...).

Then, you may want to create a parallel region (i.e. to spawn more threads) only if the serial execution of that code section is at least several times this overhead.



Nested Parallel Regions





Nested parallelism is explicitly_permitted in OpenMP. Whether it is active or not, depends on the value of some environmental variables:

```
(*) default value is FALSE
OMP NESTED=<TRUE | FALSE>
OMP NUM THREADS=N_0 < N_1, ... >
OMP MAX ACTIVE LEVELS=n
```

Which you can change by calling the appropriate omp functions

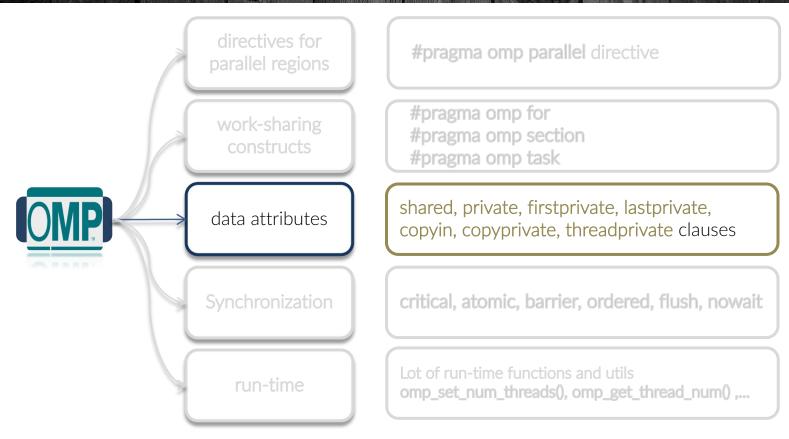
```
omp_get_nested(), omp_set_nested( cond )
```

(*)OMP NESTED is deprecated in new OpenMP 5.0. You should use only OMP MAX ACTIVE LEVELS and OMP NUM THREADS. OMP NESTED is in fact redundant. However, at the moment of writing the compilers still support OMP NESTED.













data attributes

shared, private, firstprivate, lastprivate, copyin, copyprivate, threadprivate clauses

These clauses, that you can use at the opening of parallel regions and constructs allow a fine tuning about how to manage variables that are shared, i.e. that are formally declared in a serial region of the code.



```
int i, j, k;
double *array;
#pragma omp parallel
   // i,j,k and array
   // are shared here
```

The basic rule is that everything that is defined in a serial region is inherited as shared in a parallel region that originates from the former serial region. Global variables are always shared (we'll see and exception).

In this case, i, j, k and array are all shared within the parallel region, meaning that all the threads inside that region can concurrently access them.

To be absolutely clear:

&i, &j, &k and &array are exactly the same for all the threads.



private

```
int
     i, j, k;
double *array;
#pragma omp parallel private(i,k)
   // j and array
   // are shared here
      i and k are unique to each
     thread, they live in each
      thread's stack.
     They are different than
     the original i and k
```

You can specify that a list of variables (*) that exists in the local stack at the moment of the creation of the parallel region are private to each thread inside the parallel region.

That means that the needed space will be reserved in the threads' stack to host variables of the same types. As such, those memory regions are different than the original ones, although within the parallel region those variables are referred in the source code with the same names.

(*) specified as a comma-sperated list of names



private

```
int
      i, j, k;
double *array;
array = (double*)malloc(...);
#pragma omp parallel private(array)
     now array is unique to
      each thread and it does NOT
      point to the region pointed
      by the original array pointer
```

In this case, array is declared as private. As before, it means that in the stack of every thread there will be 8 bytes dedicated to host a variable referred as array in the source code that however is different than the original array variable in the serial region.

Then if the code tries to use it to address. the memory allocated in the serial region, there will be weird things going on. (see the example OpenMP/parallel regions/00_stack_and_sco pe.c)





firstprivate

```
int
       i, j, k;
double *array;
array = (double*)malloc(...);
#pragma omp parallel
firstprivate(array)
   // now array is unique to each thread,
     BUT each copy is initialized to the
     value that the original array has
      at the entry of the parallel region.
     As such, now array can be used to
     access the previously allocated
     memory.
```

If the clause firstprivate() is used instead, every variable listed is private in the same sense than before, but its value is not randomic but it is **initialized** at the value that the corresponding variable in the serial region has at the moment of entering in the parallel region.





lastprivate

```
double last result;
#pragma omp parallel
lastprivate(last result)
  #pragma omp for
  for( int j = 0; j < some_value; j++ )
    last result = calculation( j, ...);
other calculations( last_result, ... );
// at this point, last result has the last
  value from the last iteration in the
  for loop in the parallel region
```

The clause lastprivate() pertains only to the **for** construct.

When used, every variable listed is private in the same sense than before, and its value is not initialized

What is affected is the value that the original variable, the one that is declared in the master thread's stack, has after the parallel region. It will have the value that the private copy of it has in the last iteration of the for loop.

we'll see in detail how the for loop works and how the work is distributed within it.

parallel regions/ 09 clauses lastprivate.c





threadprivate

```
int
        mvN:
double *array;
#pragma omp threadprivate(myN, array)
#pragma omp parallel
   // array does exist and it's private
   // to each thread
   array = ... allocate memory...;
   .. something serial here..
#pragma omp parallel
   // array does exist here and
   // the allocated memory is available
parallel regions/
```

The clause threadprivate applies to global variables and has a global scope. threadprivate variables are private variables that do exist all along the lifetime of the process.

I.e. they are private variables that do not die in between of two different parallel regions.

note: when using threadprivate, the dynamic thread creation is not allowed, i.e. the number of threads in each parallel region is constant.

09 clauses threadprivate.c



copyin

```
int golden values[3];
#pragma omp threadprivate(golden values)
for( int i = 0; i < N; i++ )
    get golden values();
   #pragma omp parallel copyin(golden values)
         each thread uses golden values[]
```

```
The clause copyin() applies to parallel and
worksharing (e.g. for) constructs.
This clause basically provides a way to
perform a broadcast of threadprivate
variables from the master thread (i.e. the
thread 0) to the corresponding
threadprivate variables of other threads.
```

The copying happens at the creation of the region, before the associated structured block is executed





copyprivate

```
#pragma omp parallel
    double seed[2];
   #pragma omp single copyprivate(seed)
         something happens here and the
         thread that executes this block
         initializes the seed[2] array
     at this point the values of the seed[2]
     array have been propagated to all the
     other threads
```

The clause copyprivate () applies only to single CONStruct.

This clause provides a mechanism to propagate the values of private variables, including threadprivate, from a thread to the others inside a parallel region. The copying is ultimated before any threads leave the implicit barrier at then end of the construct



The OpenMP environmental variables



You can configure the OpenMP behaviour through Internal Control Variables (ICV). Those are "conceptual" variables, meaning that you, as a programmer, are not interested in where and how those internal variables are implemented; you are only interested in how to access and use them.

These variables have different scopes:

- Global scope applies to the whole program, it is fixed once at the begin (there's only one, to my knowledge)
- Device scope OpenMP, since v4.0, can deploy threads on heterogeneous platforms (CPUs, GPUs, co-processors, ...); some ICV are naturally different on different devices
- Task scope most ICVs have a local scope, i.e. each OMP task(*) holds its own values and can change them; when new tasks are created with either a parallel or task directives, they inherit the ICVs. If a task modifies the ICVs, this has effect only during its lifetime and can not be backward-propagated.

^(*)a "task" is in general a target to be executed by a thread: for instance, a section of a for loop scheduled for a thread is, in this sense. a "task.



The OpenMP environmental variables



ICV	Scope	Env. Variable	Accessibility from src
nthreads	TASK	OMP_NUM_THREADS	get / set
nested	Task	OMP_NESTED	get / set
max-active-levels	Task	OMP_MAX_ACTIVE_LEVELS	get / set
active-level	Task	-	get
dynamic	Task	OMP_DYNAMIC	get / set
stacksize	DEVICE	OMP_STACKSIZE	-
threads-limit	TASK	OMP_THREAD_LIMIT	get
waiting-policy	DEVICE	OMP_WAIT_POLICY	-
binding	TASK	OMP_PROC_BIND	set
placement	TASK	OMP_PLACES	-
cancellation	GLOBAL	OMP_CANCELLATION	get
default-device	TASK	OMP_DEFAULT_DEVICE	get / set
run-schedule	DEVICE	OMP_SCHEDULE	get / set
default-schedule	DEVICE	-	-

Affect parallel region

Affect program execution

Affect auto-parallelization of loops





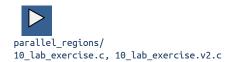


Lab exercise / 1



I propose you to write a code here and now, in the next 20 minutes, using what you know on OpenMP up to know and drawing from the codes that you find in the examples that we have discussed.

Please, ask about any doubt you may have; let's make this an interactive lab

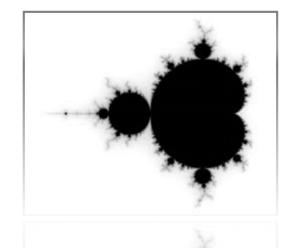


let's check together the code that is your starting point



Lab exercise / 2





Let's aim to a more ambitious and, likely, funny, exercise.

Write a code that calculates the Mandelbrot set that. as you may know, is a gorgeous fractal generated by the very simple equation

$$f_c(z) = z^2 + c$$

in the complex plane.

You find the details in OpenMP/parallel regions/11 lab exercise.pdf

that's all, have fun

