Parallel Computing &
OpenMP Introduction

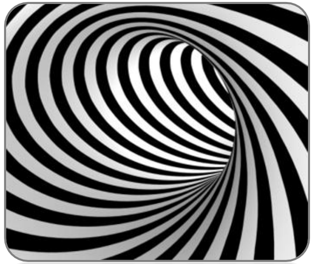
Luca Tornatore - I.N.A.F.

"Foundation of HPC" course

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

OpenMP Outline

Parallel Regions



Parallel Loops

NUMA AWARENESS

Advanced Parallelism

Parallel Regions


```
#pragma omp parallel _some_clauses_here_
single-line-here
```

A parallel region can be as short as a single line

```
#pragma omp parallel _some_clauses_here_
{ … }
```

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

```
#pragma omp parallel for _some_clauses_here_
{ … }
```

The specific construct about `for` loops

```
#pragma omp sections _some_clauses_here_
{ … }
```

A more general work-sharing construct

```
#pragma omp task _some_clauses_here_
{ … }
```

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

The threads factory

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"
         "&i is           : %p\n"
         "   rbp - &i     : %td\n"
         "   &i - rsp     : %td\n"
         "\n\n",
         getpid(), syscall(SYS_gettid),
         (void*)base_of_stack,
         (void*)top_of_stack,
         &i,
         (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\tmy 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);
  }
```

parallel_regions/
00_stack_and_scope.c

The threads factory

serial section

```
main thread (pid: 26291, tid: 26291) data:
base of stack is: 0x7ffe6f51efc0
top of stack is : 0x7ffe6f51ef60
&i is           : 0x7ffe6f51ef7c
   rbp - &i     : 68
   &i - rsp     : 28
```

parallel region

```
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 )
        my i address is 0x7f7342c82b60
                -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
```

- 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 `i`s are actually in the threads stacks, and so different memory regions than the shared `i`.

parallel_regions/
00_stack_and_scope.c

The threads factory

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: '`B`', '`K`', '`M`', '`G`' mean bytes, kilobytes, megabytes and gigabytes respectively (if no letter is specified, the number is interpreted as kilobytes).

The threads factory


```
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
thread 3: bp @ 0x7f18b4bd0b70, tp @ 0x7f18b4bd0ad0: 160 B        --> -40800 B from top of thread 2

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
```

The threads factory

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.

You can check this limit by

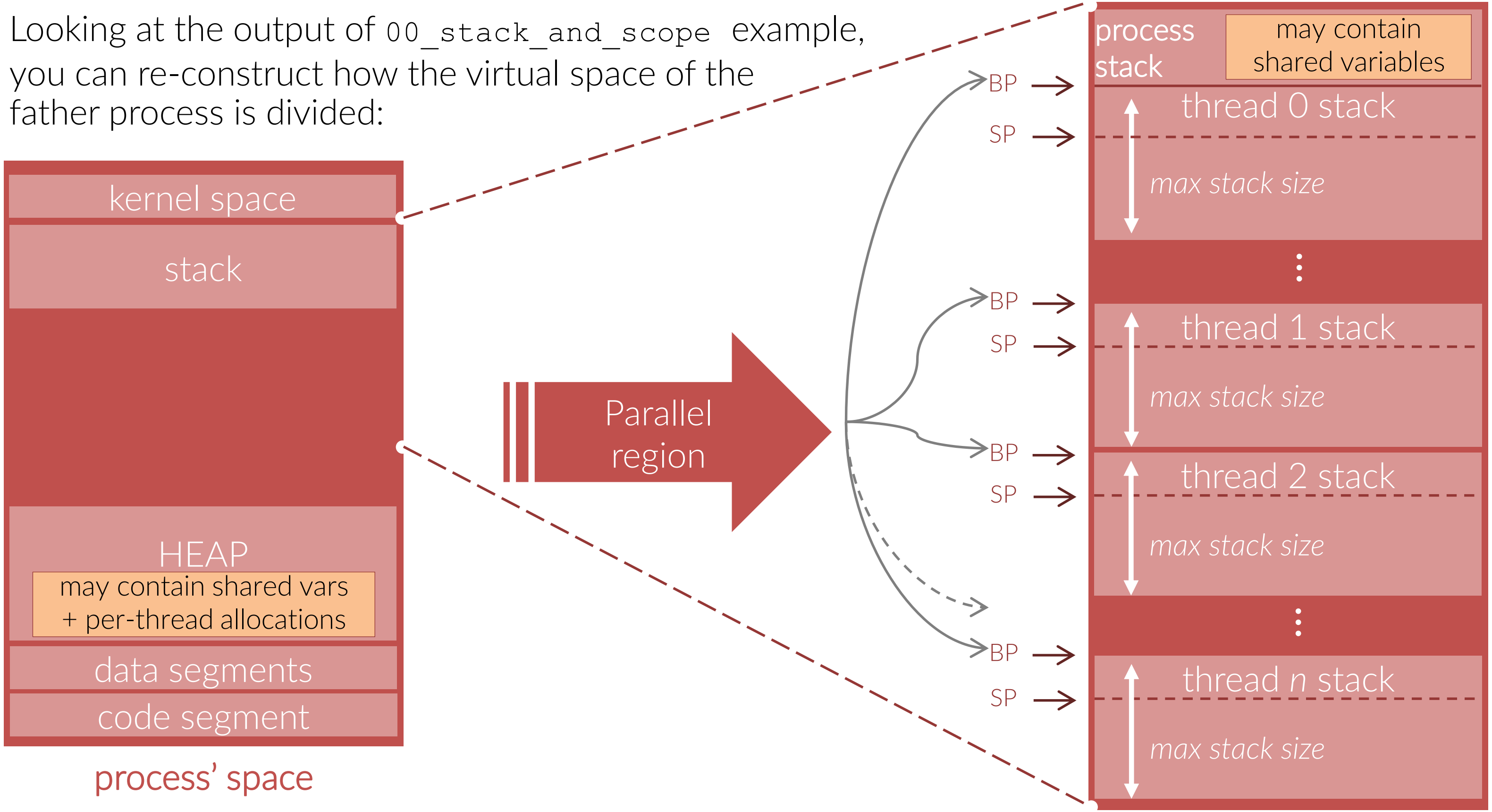
```
cat /proc/sys/kernel/threads-max
```

or calling the omp library function

```
int omp_get_max_threads()
```

The threads factory

Looking at the output of `00_stack_and_scope` example, you can re-construct how the virtual space of the father process is divided:



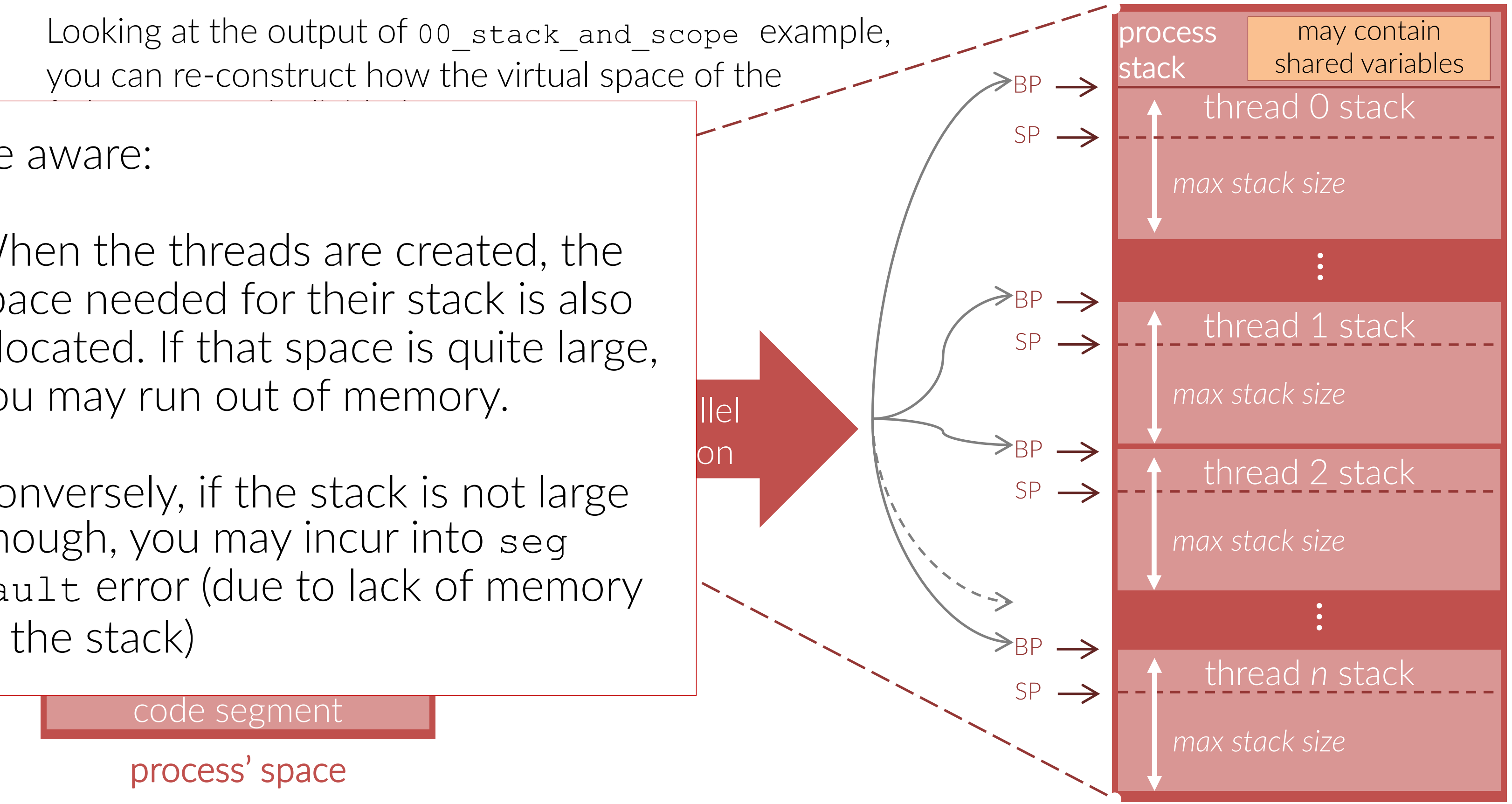
The threads factory

Looking at the output of `00_stack_and_scope` example, you can re-construct how the virtual space of the

Be aware:

When the threads are created, the space needed for their stack is also allocated. If that space is quite large, you may run out of memory.

Conversely, if the stack is not large enough, you may incur into `seg fault` error (due to lack of memory in the stack)



Parallel Regions

returns the ID of the calling thread

```
int nthreads     = 1;
int my_thread_id = 0;

#ifdef _OPENMP
#pragma omp parallel
{
    my_thread_id = omp_get_thread_num();
    #pragma 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
02_simple_pr.c

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
  ```

parallel_regions/
03_num_of_threads.c
  ```

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_dynamics( true )`

parallel_regions/
03_num_of_threads.c

The ordering of the threads


```
#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 here on the left ?

The ordering of the threads


```
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 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 0` will print the message and increase `order`.
For instance, if `thread 2` is queued 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.

The ordering of the threads


```
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; } }
}
```

parallel_regions/
05_order_of_threads.c

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++` to be 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.

Specializing execution in PR


```
#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

Specializing execution in PR


```
#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.

No explicit synchronization at all (but see relative tips in the following slides).
Only the executing thread is inside the region, the other ones can be everywhere else.

Specializing execution in PR

TIPS

Rationale of

```
#pragma omp atomic
assignment instruction
```

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.

Specializing execution in PR

TIPS

Rationale of

```
#pragma omp atomic
assignment instruction

Clauses:
read, write, update,
capture
```

Special clauses for **atomic**

read	`var = mem`	causes an atomic read of the location designated by *x* regardless of the native machine word size
write	`mem = var`	causes an atomic write of the location designated by *x* regardless of the native machine word size

update	`mem++, ++mem,`
	`mem--, --mem,`
	`mem binop= expr`
		Causes an atomic update of the location designated by *x* using the designated operator or intrinsic

capture	`val = mem++,`
	`val = ++mem,`
	`val = mem--,`
	`val = --mem,`
	`val = mem binop= expr`
	`{val = mem; mem binop= expr}`
	`{mem binop= expr; val = mem}`
		Causes an atomic update of the location designated by *x* 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 *x* is written in the location designated by *v*, depending on the form of the `atomic` construct, structured block, or statements, following the usual language semantics

Specializing execution in PR

TIPS

```
#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).

Specializing execution in PR

TIPS

```
#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.

Specializing execution in PR

TIPS

```
#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..):

```
…
results[0] = heavy_work_0();
results[1] = heavy_work_1();
results[2] = heavy_work_2();
…
```

```
#pragma omp parallel
{
    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;
}
```

dynamic extent

`parallel_region/06_assign_work.c`

`run with a second argument >0 to check about it`

`parallel_region/06_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/06_assign_work.c

if the first argument, that determines the amount of work, is ≤10, the parallel region is **not** created

Conditional creation of PR

TIPS

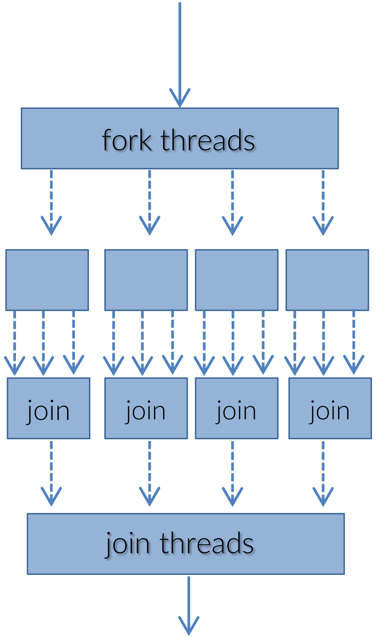
What is this useful for?

The overhead of the creation of a parallel region is of the order of ~10 μseconds (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



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

OMP_NESTED=<TRUE | FALSE>   (*) default value is 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.

07_nested_regions.c

that’s all, have fun