



NTNU – Trondheim
Norwegian University of
Science and Technology

OpenMP worksharing directives

Partitioning shared work

- So far, we've been assigning work to threads based on their index in the thread pool:

```
int tid = omp_get_thread_num();
```

- This is a little bit of a hassle

- For thread-specific blocks of code, we need something like this

```
if ( tid == 0 ) { /* Do one thing */ }  
else if ( tid == 1 ) { /* Do another thing */ }  
...
```

- For loops, we need to combine the index with the induction variable to work out a selection of iterations

```
for ( int x=tid; x<x_max; x+=n_threads )           ← round robin  
for ( int x=bottom[tid]; x<top[tid]; x++ )       ← consecutive range
```

- It is not super difficult, but it's repetitive to type every time
 - Also extremely common, so it can be automated

Worksharing directives to the rescue!

- These are OpenMP directives that can split a given workload between threads for you, without requiring you to do anything based on the thread id#
- We'll look at three flavors
 - Sections
 - Loops
 - Single



Functional decomposition

- This is when we split the work by the function of its sub-tasks
 - We've talked about it in terms of *pipelining*

This thing only
installs seats →



← This thing only
slaps on doors

Throughput doubles
when the pipeline is full

Partial products roll past in this direction

Data decomposition

- This is when we split the work by the input/output of its sub-tasks
 - Pretty much all we've been doing so far, because you don't have to design additional code in order to increase the number of participants

Everyone does
the same thing →



↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑
They do it to individually assigned parts of a field

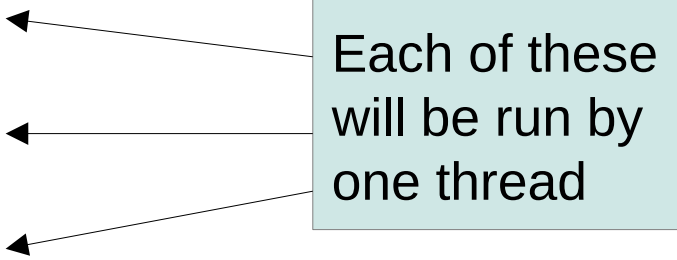


NTNU – Trondheim
Norwegian University of
Science and Technology

Sections

- For functional decomposition, OpenMP has *sections*

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        { /* Section #1 */ }
        #pragma omp section
        { /* Section #2 */ }
        #pragma omp section
        { /* Section #3 */ }
    }
}
```



Each of these
will be run by
one thread



Implicit synchronization

- Worksharing directives have an implicit barrier at the end

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        { /* Section #1 */ }
        #pragma omp section
        { /* Section #2 */ }
        #pragma omp section
        { /* Section #3 */ }
    }
}
```

All threads will
synchronize here
by default



Implicit synchronization

- Because they very often occur just after each other

```
#pragma omp parallel
```

```
{
```

```
  #pragma omp sections
```

```
  {
```

```
    #pragma omp section
```

```
    { /* Section #1 */ }
```

```
    #pragma omp section
```

```
    { /* Section #2 */ }
```

```
  }
```

```
  #pragma omp sections
```

```
  {
```

```
    #pragma omp section
```

```
    { /* Section #3 */ }
```

```
    #pragma omp section
```

```
    { /* Section #4 */ }
```

```
  }
```

```
}
```

Finish these first

Synchronize

Finish these next

Synchronize again

Implicit barriers
make these two
blocks of sections
work as separate stages



Clauses

- Most OpenMP directives have an optional set of additional terms that can control details of their semantics
 - We've already seen the *num_threads* clause for the *parallel* directive
- The worksharing directives have a clause *nowait*
 - Its use indicates that you wish to omit the implicit barrier at the end



nowait in practice

- The nr. of sections limits the number of threads in use, additional threads wait

```
#pragma omp parallel
```

```
{
```

```
  #pragma omp sections
```

```
  {
```

```
    #pragma omp section
```

```
    { /* Section #1 */ }
```

```
    #pragma omp section
```

```
    { /* Section #2 */ }
```

```
  }
```

```
  #pragma omp sections
```

```
  {
```

```
    #pragma omp section
```

```
    { /* Section #3 */ }
```

```
    #pragma omp section
```

```
    { /* Section #4 */ }
```

```
  }
```

```
}
```

Two threads here

Stop

Two threads here

By default, this example will only use 2 threads at a time



nowait in practice

- If we omit the implicit barrier, additional threads will “fall through” and start working

```
#pragma omp parallel
```

```
{
```

```
  #pragma omp sections nowait
```

```
  {
```

```
    #pragma omp section
```

```
    { /* Section #1 */ }
```

```
    #pragma omp section
```

```
    { /* Section #2 */ }
```

```
  }
```

```
  #pragma omp sections
```

```
  {
```

```
    #pragma omp section
```

```
    { /* Section #3 */ }
```

```
    #pragma omp section
```

```
    { /* Section #4 */ }
```

```
  }
```

```
}
```

Skip the barrier

Two threads here

Two more threads here, right away

Here, we have enough sections to employ 4 threads at a time



That's a silly example

- Yes, it is.
 - A simpler way to write the same effect would be to just include all four sections under the same `#pragma omp sections` directive to begin with
- I just wanted to make a simple illustration of the `nowait` clause
 - It applies to the other worksharing directives as well
 - It's occasionally useful



Saving some keystrokes

- A very common use case is to start some threads only to give them exactly 1 worksharing directive

- With *sections* as an example, it creates this pattern

```
#pragma omp parallel
{
    #pragma omp sections
    {
        ...
    }
}
```

- Because it's redundant to separate the thread starting/stopping directive from the work partitioning when there's only 1, we can write them together

```
#pragma omp parallel sections
{
    ...
}
```

- This means exactly the same thing as above

(But there will always be an implicit synch. at the end, because the threads join there)



Loops

- When we've parallelized loops so far, we've done it by partitioning its *iteration space*
 - for (i=tid; i<N; i+=n_threads)
 - assigns every (n_threads)th iteration to a thread
 - for (i=bottom[tid]; i<top[tid]; i++)
 - assigns blocks of top[tid]-bottom[tid] iterations to a thread
- When we have a loop with an induction variable (such as for loops in C) this assignment can be done automatically
 - `#pragma omp parallel for`
 - for (int i=0; i<N; i++)makes some default mapping of iterations to threads
 - (Note that we didn't *have* to join the "parallel" and "for" parts, you can also have several instances of `#pragma omp for` inside one `#pragma omp parallel`)
- There's no equivalent for *while* loops, because we can't predict their iteration counts in the same way
 - (There's another technique, but we'll get back to it)

Data sharing clauses

- So far, we've been discriminating between private and shared values by the scope we declare them in

```
int a = 42;                ← shared
#pragma omp parallel
{
    int tid = omp_get_thread_num(); ← private
}
```

- This works
 - The default behaviour is obvious when you see its connection to stack contexts
- It can be impractical
 - Worksharing directives have clauses that can explicitly specify what should be shared and private instead
 - Useful when your declarations are easier to read when you put them in the “wrong” scope

Shared and private

- We can revisit our running pi example once more
 - `pi_shared_private.c` in today's example archive
- I have made all the variable declarations global
 - That is a questionable decision, but we're just making a point here
- The pi value has to be shared among threads
 - The parallel directive has a clause "shared(pi)"
- The tid, n_threads, x and pi_local values should be private
 - The parallel directive has a clause "private(pi_local,x,tid,n_threads)"
- Everything still works as expected
 - OpenMP inserts the necessary cloning of space into stack frames at the directive



Reduction

- When a shared variable is the target of a global sum, product, logical-and, *etc. etc.*

(the usual bunch of operations we can make reductions out of)

OpenMP can figure out a way to coordinate local additions to the global total all on its own

- The clause

`reduction(operator:variable)`

says that *variable* is the target of a global reduction using the *operator*

(and leaves the fiddly synchronization parts to OpenMP)



The pi example in its final form

- This is the way I've been aching to write it all along:

```
#define _XOPEN_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <math.h>

#define STEPS (1e8)
#define H (1.0/STEPS)

int main ( int argc, char **argv ) {
    double pi = 0.0;

    #pragma omp parallel for reduction(+:pi)
    for ( int64_t i=0; i<STEPS; i++ )
        pi += H / (1.0 + (i*H)*(i*H));

    pi *= 4.0;
    printf ( "Estimated %e, missed by %e\n", pi, fabs(pi-M_PI) );
    exit ( EXIT_SUCCESS );
}
```



(Now we know everything that's being automatically handled for us, though)

Data sharing clauses

(there are a couple more)

- `private(variable)` doesn't actually say what the initial value of 'variable' should be
 - Your thread should take care to assign it
- If you're privatizing a shared value, its initial state makes a natural value to give to all the private copies
 - You can specify this as `firstprivate(variable)`
- When the threaded region ends, you may want to put one of the private copies back into the shared copy
 - You can specify `lastprivate(variable)` to get its "final" state



{first,last}private semantics

- The idea of ‘firstprivate’ is to initialize all local copies in a parallel for loop with the value that the global copy has in the first iteration of a sequential run
- The idea of ‘lastprivate’ is to leave behind the local copy that the global one would have in the last iteration of a sequential run
 - *i.e.* the thread that sets the lastprivate value at the end of the parallel region is the one that was assigned the final iteration
 - This is not necessarily the same as the thread that finishes first chronologically (that would produce a race condition)
- When applied to sections, the sections that literally appear first and last in the source code take on the roles of “first and last iteration” for this purpose

Single

- Our final worksharing directive today is

```
#pragma omp single  
{  
}
```

- This means that even if many threads reach this region, only one of them will execute it

Typically, the first one that gets there

- It doesn't make much sense on its own, but we can insert single-regions in the middle of parallel loops to useful effect sometimes

(Remember that it's a worksharing directive, though, so it comes with a barrier unless you disable it)