NTNU – Trondheim
Norwegian University of Science and Technology

OpenMP worksharing directives

Guest lecture for TDT4200

Jan Christian Meyer 24.09.2024

Partitioning shared work

- We have seen that we can assign work to threads based on their index in the thread pool:

  int tid = openmp_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

NTNU – Trondheim
Norwegian University of
Science and Technology

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

Partial products roll past in this direction

Throughput doubles when the pipeline is full

NTNU – Trondheim
Norwegian University of Science and Technology

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

NTNU – Trondheim
Norwegian University of Science and Technology

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 */ }        Two threads here
        #pragma omp section
        { /* Section #2 */ }
    }                               Stop
    #pragma omp sections
    {
        #pragma omp section
        { /* Section #3 */ }        Two threads here
        #pragma omp section
        { /* Section #4 */ }
    }
}
```

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        <- Skip the barrier
    {
        #pragma omp section
        { /* Section #1 */ }
        #pragma omp section            <- Two threads here
        { /* Section #2 */ }
    }
    #pragma omp sections
    {
        #pragma omp section
        { /* Section #3 */ }
        #pragma omp section            <- Two more threads here, right away
        { /* Section #4 */ }
    }
}
```

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)

NTNU – Trondheim
Norwegian University of
Science and Technology

Loops

- Parallelizing loops by thread indices amounts to 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)

The *schedule*(kind,blocksize) clause

- Parallel loop directives allow you to control how they partition the iterations between threads
- The *blocksize* is an optional, positive integer which we shall discuss imminently
- The *kind* is one of these:
  - static
  - dynamic
  - guided
  - auto
  - runtime

A unit of work

- When OpenMP partitions an iteration space between threads, it has some leeway with how many iterations to include in “one work unit”
- The absolutely smallest unit available/possible is to distribute 1 iteration at a time
  - Units of 1 iteration gives the round-robin assignment we’ve worked out manually
- Depending on how much work each iteration contains, 1 iteration can easily be a bit on the short side
- Increasing the unit size makes the work distribution more coarse-grained

Small vs. big work units

- Big blocks:
  - Fewer units to distribute, and hence, less scheduling to do

    BUT
  - There's a limit to how big the blocks should be
  - At the extreme end: if the entire iteration space is one big block, we've taken away all the parallelism again
- Small blocks:
  - More units to distribute, more disruptions in memory access pattern

    BUT
  - Greater flexibility to assign work to unemployed threads

The block size

- It's easy to assume that the block size parameter of the schedule is the number of iterations handed to each thread
- This is not always true
- It is the minimal number of iterations handed to a thread
  - Some of the schedule kinds take the liberty to hand out bigger blocks
  - They won't hand out smaller blocks if they can help it, though
  - It's intended to be a measure of how few iterations it can make sense to lump together
  - This number depends on the details of your program, so you can set it

NTNU – Trondheim
Norwegian University of Science and Technology

Automatic schedule

- This is the default kind
- It doesn't have to be a particular fixed kind, it's the one that your OpenMP implementation nominates as most likely to the best job in the greatest number of cases
- It tends to be a good guess for nested loops that sweep over multidimensional arrays with approximately equal workloads per element
  - That's a very common use case

Runtime schedule

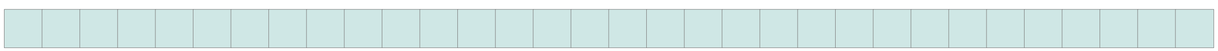
- This is for when you don’t want to embed the choice of schedule into your program
- With a runtime schedule, your OpenMP program will search the calling shell’s environment variables for a specification of what to use on each run:

```
export OMP_SCHEDULE=”static,4”
./my_program      # program runs with schedule(static,4) as default
export OMP_SCHEDULE=”dynamic,16”
./my_program      # program runs with schedule(dynamic,16) instead
```

NTNU – Trondheim
Norwegian University of
Science and Technology

Static schedule

- This is the schedule we've been calculating by hand throughout all this threading stuff
- Given an iteration space

0  max

schedule(static,6) will assign iterations to *e.g.* threads 0, 1, and 2, thus:

0  max

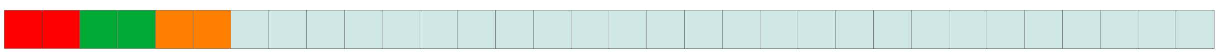
The master/worker pattern

- We haven't made much use of it, but a common way to implement work sharing in queued systems is to
  - Keep an active *thread pool* of available worker threads
  - Keep a queue of finite work packages (which may or may not grow/shrink while the program is running)
  - Assign the next package in the queue every time a worker thread becomes available
- Web servers, transactional databases, and other on-line request processing systems tend to do this
  - HPC programs rarely have infinite streams of incoming requests
  - They still use this pattern to achieve some measure of *load balancing* when the amount of work in each package is unevenly distributed

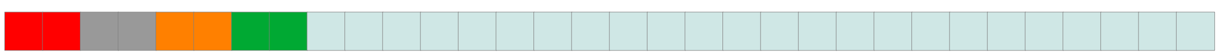
Dynamic schedule

- The dynamic kind of schedule works this way
- We can illustrate schedule(dynamic,2) with our iteration space and 3 threads again:

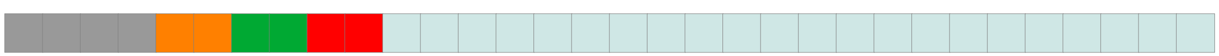
Everyone gets something to begin with

0  max

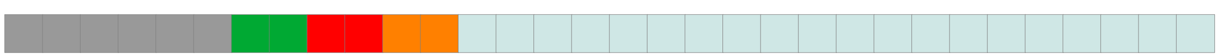
If thread 1 finishes first, it gets the next unit

0  max

Thread 0 may be the next one out

0  max

By now, thread 2 may have come around

0  max

...and so, it continues...

Guided schedule

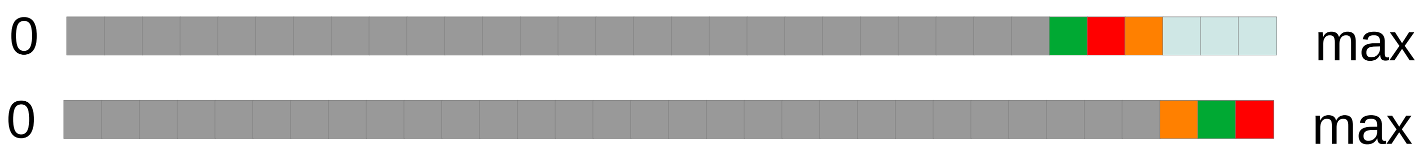
- This one is similar to dynamic, but acknowledges the observation that we probably have a barrier coming up at the end of the loop
- While the barrier remains far in the future, it doesn't matter so much how big the blocks are
  - Workers that run out of work can just pick up some more
- When the barrier is imminent, it's a mistake to hand out a giant workload to one worker
  - Everyone else will have to wait for it to finish
- Guided schedule starts with big blocks and gradually shrink them down to the blocksize

schedule(guided,1) illustrated

- Everyone gets lots of work at the beginning:

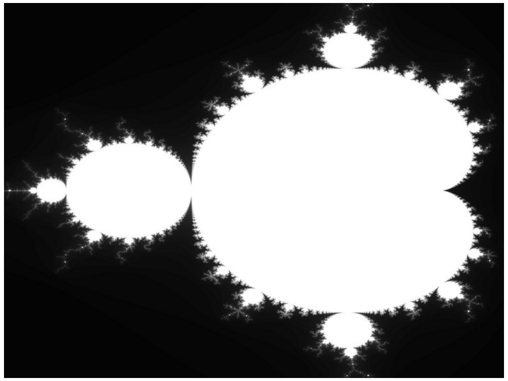
- Past the halfway mark, we should probably shrink the workloads we dispense:

- Near the end, everyone gets the smallest available block sizes, to minimize the inevitable wait



Demonstration time

- Today's code archive has a subdirectory 01_fractal
  - It has an OpenMP-enabled Mandelbrot set generator in it
- We don't have to dwell on the calculation, but the output image looks like this:
  - The important characteristic is that a black point represents a loop that has terminated immediately, while a white point has required 256 iterations



- The loop over the y-axis is the parallel one
- Clearly, some horizontal lines contain much more work than others

The experiment

- The program times its own execution
  - Conveniently, this also lets us demonstrate the function
    double omp_get_wtime (void);
  - It's exactly like MPI_Wtime(), in that it returns some number of seconds
  - It's also exactly like MPI_Wtime() in that implementations tend to use precisely the same system clock
- If our theory is correct, this program might run faster with a guided schedule than with the automatic
  - Let's try it out

Single (and master)

- If you are inside a parallel region but only want one thread to do something, you can mark a block with

  #pragma omp single

  and only one thread (but *any* thread) will go in there
- There's another flavor called

  #pragma omp master

  which makes sure that only thread 0 goes in there
- It's a flavor of mutual exclusion without having to declare locks