



NTNU – Trondheim
Norwegian University of
Science and Technology

OpenMP loop scheduling & threads vs. caches

Today's topic

- We've gone through some OpenMP directives
 - Enabling/disabling a team of threads
 - Atomicity directives: *atomic*, *critical*, and *master*
 - Worksharing directives: *sections*, *for*, and *single*
 - Data sharing clauses: *shared*, *private*, *reduction*, *firstprivate*, *lastprivate*
- We also suggested that `for` directive has a clause to direct how it partitions the iteration space
 - Let's look at it today



schedule(kind,blocksize)

- This is it.
- 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



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 do 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
```

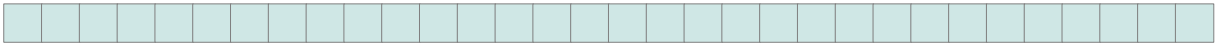
```
export OMP_SCHEDULE="dynamic,16"
```

```
./my_program    # program runs with schedule(dynamic,16) instead
```



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

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:

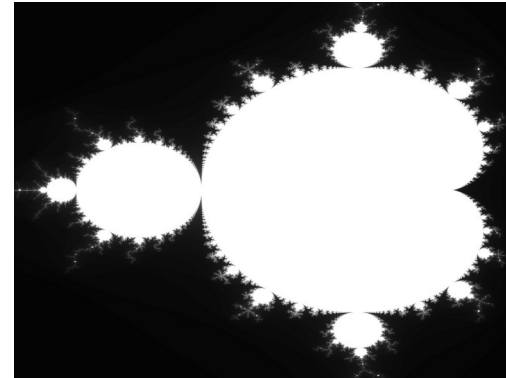


- Near the end, everyone gets the smallest available block sizes, to minimize the inevitable wait



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

While we are on the topic

- Since we're talking about distributions of loop iterations, there is another effect I would like to demonstrate
- Today's experiment #2 is found in 02_advection
 - It's our 2D advection equation solver from before
 - Now with OpenMP instead of MPI
 - (and a slight y-velocity added, just to make the pictures a little different for a change)
 - It's also parallelized along the y-axis to begin with



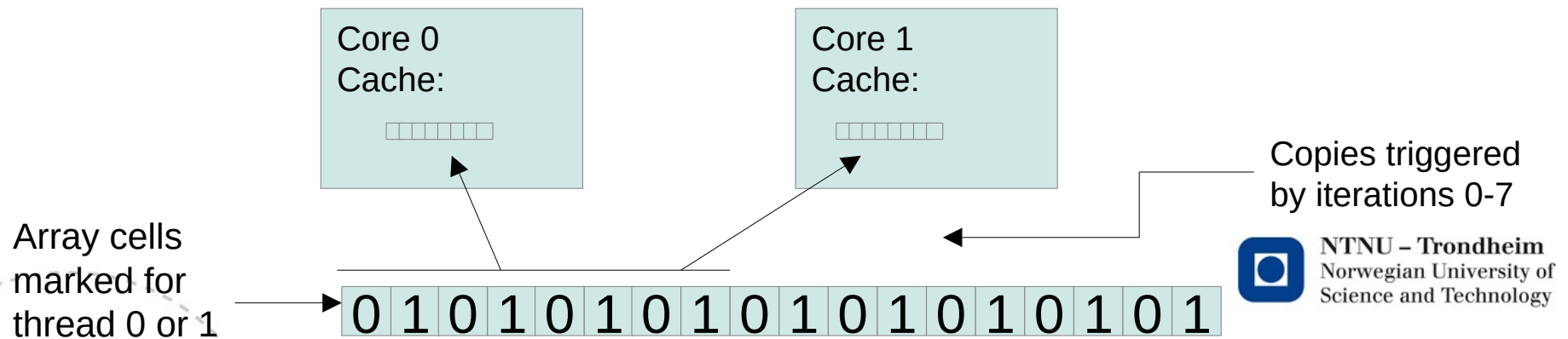
Here's what I will do

- Move the *parallel for* directive to the inner (x) loop
 - The program slows down quite a lot
 - Is it because of the constant thread starting/stopping?
- Separate the *parallel* from the *for*
 - Slowdown was *partly* from the start/stop behaviour, but it's still slow
 - Is because of the incessant barriers?
- Try it with *nowait*
 - Performance is almost up to snuff again, but it's not spectacular
- There is an effect here that I want to provoke
 - Its worst possible behaviour should appear with `schedule(static,1)`



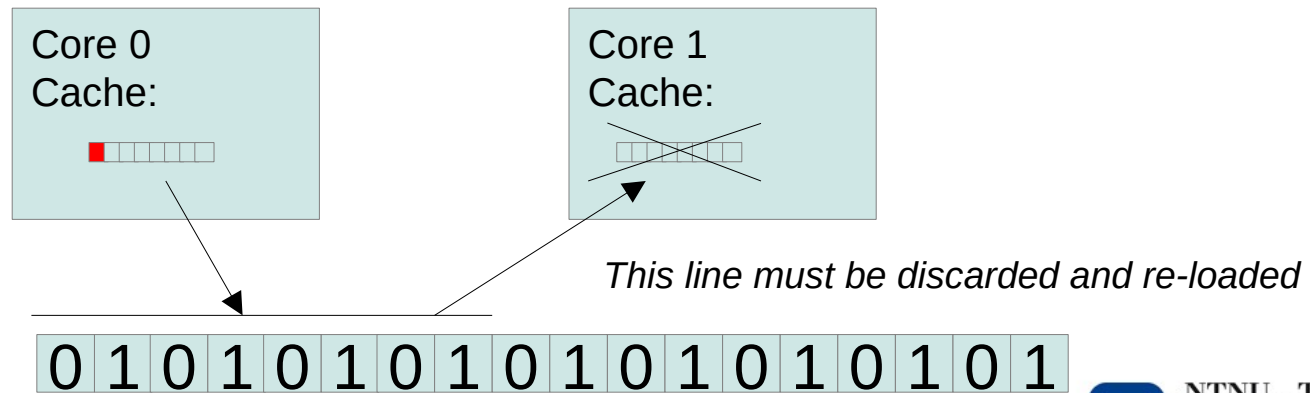
What's going on?

- x-coordinates are adjacent to each other in memory
 - Hence, the majority of neighbor values occupy the same cache line
 - We've assigned a different thread to each consecutive value
 - Different threads live on different cores
 - Different cores have different private (Level-1) caches
- It's fine as long as they *read* values that are close
 - Every core can have a copy of the cache line



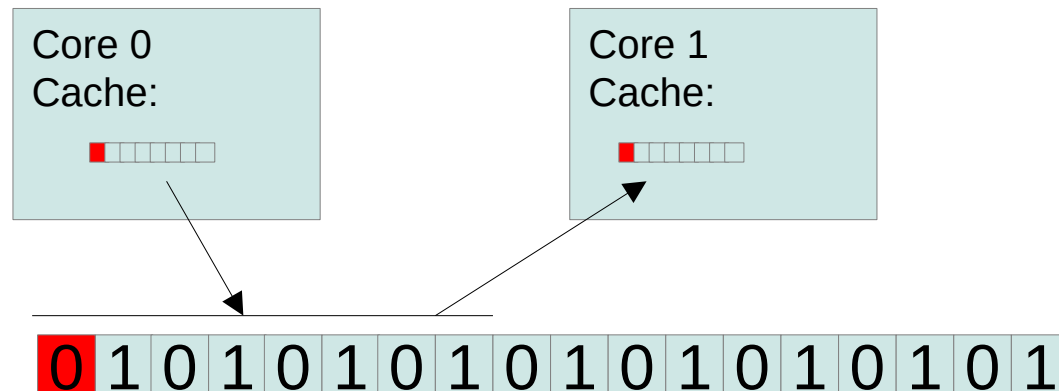
When core 0 wants to write something

- A change in the cache line's values means core 1's copy is out-of-sync with the state of memory it's supposed to reflect
- This can be detected as soon as core 0 sends its update away, but memory doesn't update as quickly



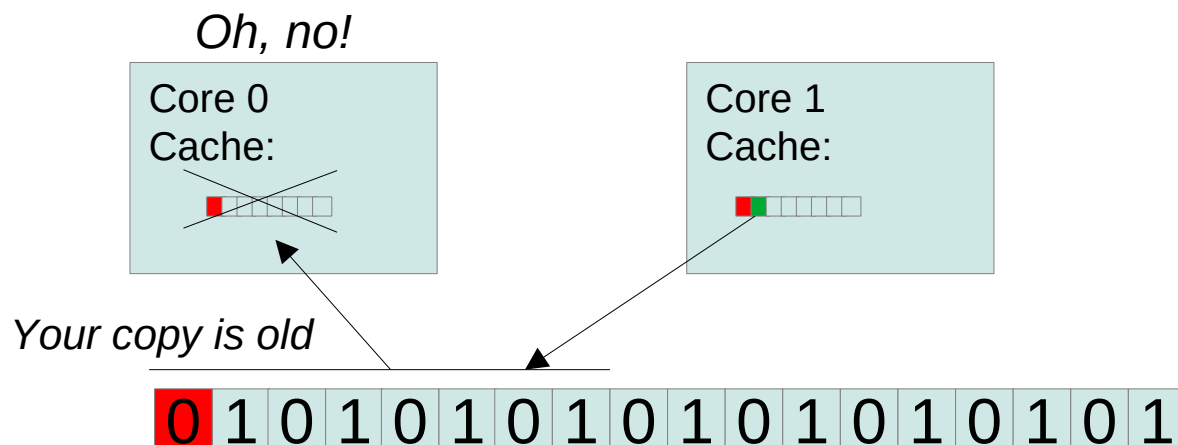
When core 0 wants to write something

- When memory finally *is* updated, core 1 still has to fetch a fresh copy
- This takes at least as many cycles as a cache miss



Then, core 1 wants to write something

- ...and the whole circus repeats



This is called *false sharing*

- It's "false" because the threads don't actually share any of the values inside the cache line
- It's "sharing" because it comes with the same overhead as synchronizing access to a shared variable
- Very expensive when it happens constantly
 - Like in that pi-example we made with the locking inside the integration loop
 - Beware, you don't need an actual shared value to create the effect
 - Two thread-assigned values that happen to sit right next to each other in memory will do



We only get this effect with threads

- MPI ranks are immune to false sharing because they don't even share memory frames, much less cache lines
- Threads don't get it with the private variables on their stacks, because those aren't right next to each other
- We can create false sharing when assigning sub-areas of a shared memory segment to distinct threads
- I propose that it's best not to create false sharing
 - Now we know it exists, so we won't create it by accident