Lab 11 Daniel Hjelm

My computer does not seem to support openMP so I will need to use the school computers (Fredholm).

Task-0:

Using Valgrind without OpenMP works just fine. No memory leaks.

When using OpenMP we get some possible and still reachable.

Task-1:

When compiling without OpenMP we get: a = 120 and b =20

When compiling with OpenMP we get: a = 20 and b = 20 16 times.

When using firstprivate: a = 120 and b = 20.

firstprivate is the **clause that contains the variables that each thread in the OpenMP parallel region will have an identical copy of**. These copies are initialised with the value of the original variable passed to the clause. By contrast, private variables do not.

|  |  |
| --- | --- |
| [lastprivate](https://docs.microsoft.com/en-us/cpp/parallel/openmp/reference/openmp-clauses?view=msvc-170#lastprivate) | Specifies that the enclosing context's version of the variable is set equal to the private version of whichever thread executes the final iteration (for-loop construct) or last section (#pragma sections). |

Task-2:

The sections construct is a non-iterative worksharing construct that contains a set of structured blocks that are to be distributed among and executed by the threads in a team. Each structured block is executed once by one of the threads in the team in the context of its implicit task.

We can let two threads do two completely different things by letting two threads use two different functions. This we can do with sections.

We have two sections and two threads and the different sections do two different functions.

If we change to one thread the thread will have to do both sections and we should have double the time but both functions will be runned.

Two threads time: real 0m2,689s, user 0m5,337s , sys 0m0,004s

One thread time: real 0m5,314s, user 0m5,312s, sys 0m0,000s

Three thread time: real 0m2,690s, user 0m5,350s, sys 0m0,004s

So only real time is changed between runs and it is depending on the number of tasks.

If we have three sections and three functions one thread can take each function.

It works, it only takes three seconds but three functions are evaluated.

Task-3:

Without parallelization: real 0m5,311s, user 0m5,310s, sys 0m0,000s

With three threads: real 0m1,794s, user 0m5,342s, sys 0m0,004s

So we have a speed-up of 3, which makes sense and the sum is still correct.

We can also use #pragma omp parallel for num\_threads(3) instead of having two blocks, which is clean.

Rewriting the other-loops to “#pragma omp parallel for” we get different results every time.

We have to be sure that the loops can be parallel, otherwise we can get weird results and also different results every time. In the “other-loop” file we can see this since y will be used and changed by every thread, the loops are no longer able to be parallelized.

Task-4:

I think this is independent, I don’t see why it would not be.

Since it gives out the same answer every time it is probably right.

Time for 2 threads: 9.128294 sec

Time for 4 threads: 5.347399 sec

Time for 8 threads: 2.863283 sec

Let’s try different schedules where chunck\_size is set to default (not specified) with 8 threads:

* Static: 2.884700 s
* Dynamic: 1.761585 s
* Guided: 1.760577 s

Here Dynamic and Guided gave the same speed, so we could probably use both for load balancing.

This could be done to increase the speed of the prime-counter!

Task-5:

Time 1 thread: 0m1,860s

Time 5 threads: 0m0,383s (5x speed up)

Time 10 threads: 0m0,324s

Time increases with more threads.

The reduction clause can be used to skip using “private” and “critical” which makes the code easier.

Task-6:

For N = 100 0000:

1 thread: 0m2,221s

2 threads: 0m1,113s

4 threads: 0m0,561s

8 threads: 0m0,283s

Task-7:

Case A:

* One thread: real 0m8,344s, user 0m8,311s
* Four threads: real 0m2,339s, user 0m8,909s
* 4x speed up for real time

Case B:

* One thread: real 0m10,260s, user 0m10,240s
* Four threads real 0m3,027s, user 0m11,654s
* 4x speed up

Case C:

* One thread: real 0m6,328s, user 0m6,310s
* Four threads: real 0m3,779s, user 0m14,480s
* 2x speed up

So for case C we can see this phenomena clearly. This we need to be careful of.

The user time (cpu-time) also skyrockets for case C, which should not normally happen, it should stay the same.

Task-8:

With parallelization I will get a 8x speed up for 16 threads in comparison to one thread. The only weird thing is that the sum is not exactly the same, but that might be because of other things.

The great thing is that I only had to insert one row.

Task-9:

For n = 40:

1 thread: real 0m2,367s

2 threads: real 0m1,454s

4 threads: real 0m0,892s

8 threads: real 0m0,559s

Not quite ideal speed-up. Don’t really understand why not.

Task-10:

Do this!