Parallelle beregninger - TDT4200 - Problem Set 3

Ola Kristoffer Hoff

4. October 2022

1 Pthreads tasks

1.1 Serial pthreads

Done, se code.

1.2 Parallel pthreads

1.2.1

Added an args-object to the thread, holding the a given ID for the thread and the number of total threads.

1.2.2

Created an array of threads and arguments for the given threads.

1.2.3

Spawned the corresponding number of threads.

1.2.4

I joined the threads back togheter after the work was done. After they had executed their portion of the data (in time step).

1.2.5

I divided the area in slices (in the y-axis) based on the number of threads.

1.2.6

For the speedup we can look at table 1. I added timing in the serial code in the same way it was used in problem set 2. We can see that we got the greates speedup with 8 threads givin a speedup of 3.06 times the serial speed.

Program	Time_1	Time_2	Time_3	Time_4	Time_5	$\text{Time}_{average}$	Speedup
Serial	16.96s	17.68s	17.48s	16.77s	16.60s	17.10s	1.00x
Pthreads (2)	9.15s	9.38s	9.26s	9.32s	9.27s	9.28s	1.84x
Pthreads (4)	5.51s	5.59s	6.03s	6.20s	6.18s	5.90s	2.90x
Pthreads (8)	5.19s	5.35s	5.80s	5.76s	5.84s	5.59s	3.06x
Pthreads (16)	5.87s	5.88s	6.60s	6.44s	6.75s	6.31s	2.71x

Table 1: Table showing speed of execution with pthreads (2, 4, 8 and 16 threads) and serial. The speedup column is relative to the serial speed and uses the average values.

2 OpenMP Tasks

2.1

Added the "*pragma omp parallel for*" above all the outer for-loops in the *time_step*-function.

2.2

In table 2 I have extended table 1 with the *OpenMP* values as well. We can see that *OpenMP* matched on the 100th of a second the time achieved by the *Pthreads* with 8 threads, a 3.06 times speedup. To wondered if 8 threads was optimal for the first solution so I tried with 6. This reuslted in the overall best performance with a speedup of 3.13 times the serial speed.

Program	Time_1	Time_2	Time_3	Time_4	Time_5	$\text{Time}_{average}$	Speedup
Serial	16.96s	17.68s	17.48s	16.77s	16.60s	17.10s	1.00x
Pthreads (2)	9.15s	9.38s	9.26s	9.32s	9.27s	9.28s	1.84x
Pthreads (4)	5.51s	5.59s	6.03s	6.20s	6.18s	5.90s	2.90x
Pthreads (6)	5.14s	5.47s	5.86s	5.25s	5.57s	5.46s	3.13x
Pthreads (8)	5.19s	5.35s	5.80s	5.76s	5.84s	5.59s	3.06x
Pthreads (16)	5.87s	5.88s	6.60s	6.44s	6.75s	6.31s	2.71x
OpenMP	5.19s	5.46s	5.42s	5.84s	6.03s	5.59s	3.06x

Table 2: Table showing speed of execution with pthreads (2, 4, 6, 8 and 16 threads), OpenMP and serial. The speedup column is relative to the serial speed and uses the average values.

3 Questions

3.1 What is the difference between a thread and a process? How does this relate to the difference betweem MPI and Pthreads?

The difference between a thread and a process is that threads share memory while processes do not. This means that for *Pthreads* one have to be careful when using variables, since they are shared and can easily lead to udefined behaviour if not accessed correctly. With *MPI* one have to send the data that is to be shared manually, there is no chance of messing with the same memory at the same time, but now there is a lot of overhead in the greater seperation of execution.

3.2 Why is it not necessary to implement border exchange for the Pthreads solution?

As described above, *Pthreads* have shared memory which means that they already have access to the data the other threads have calculated. Therfore it is not needed to send it explicitly as with *MPI* where the memory is not shared.

3.3 What type of parallelizations are much easier to do using MPI instead of Pthreads?

Pthreads are in general better to used when there is less computation, because of the much lower overhead of spawning threads compared to processes. *MPI* is more useful when we want a distributed network of computers working on the same problem, that is much harder to do with just *Pthreads*. However, if we look at the case of just one computer as well, *MPI* can have the upper hand in some situations. Since *MPI* spawns processors they are regarded, by the OS, as different programs and are given qn equal amount of time, with *Pthreads* it is one process which gets no more time dispite the number of threads spawned in that process (here I ignore the fact that it would get more an more time from the OS as it executed, if the OS uses a MLFQ or something similar).

So if the workload per sub-task is heavy/long enough then it would be beneficial to use *MPI* over *Pthreads*.

3.4 Which registers are associated for AVX instructions?

The AVX instructions are associated with the *YMM* and *XMM* registers.

3.5 Run the serial code with -O2 and -O3. Which is faster? Can you see usage of AVX instructions in the assembly file?

As we can clearly see in table 3 we have quite a significant speedup with both -O2 and -O3, respectively 3.50 and 4.36 times faster than the unoptimised serial runtime.

Program	$Time_1$	$Time_2$	$Time_3$	$Time_4$	$Time_5$	$Time_{average}$	Speedup
Serial	16.96s	17.68s	17.48s	16.77s	16.60s	17.10s	1.00x
Serial -O2	5.01s	4.90s	4.77s	4.73s	4.98s	4.88s	3.50x
Serial -O3	3.84s	3.98s	3.96s	3.97s	3.86s	3.92s	4.36x

Table 3: Table showing speed of execution with no "O"-flag, -O2 and -O3 for the serial solution.

In figure 1 we can se a snippet of the code from running the command: "*gcc src/shallow_water_serial.c src/argument_utils.c -fopenmp -lm -pthread -O3 -S -fverbose-asm*". We can see some different AVX instructions such as: *mulpd*, *divpd* and *addpd*. These can be read as for example: addpd - add packed double (precision). Note that we can see that the instructions are using the *XMM* registers.

```
# src/shallow_water_serial.c:153:                       + 0.5 * gravity * (PN(y, x) * PN(y, x) / density);
    mulpd   %xmm0, %xmm1    # tmp928, vect__55.316
# src/shallow_water_serial.c:152:             DU(y, x) = PN(y, x) * U(y, x) * U(y, x)
    mulpd   %xmm6, %xmm0    # tmp929, vect__53.314
# src/shallow_water_serial.c:153:                       + 0.5 * gravity * (PN(y, x) * PN(y, x) / density);
    divpd   %xmm5, %xmm1    # tmp1302, vect__56.317
# src/shallow_water_serial.c:152:             DU(y, x) = PN(y, x) * U(y, x) * U(y, x)
    mulpd   %xmm6, %xmm0    # tmp929, vect__54.315
# src/shallow_water_serial.c:153:                       + 0.5 * gravity * (PN(y, x) * PN(y, x) / density);
    mulpd   %xmm4, %xmm1    # tmp1303, vect__57.318
# src/shallow_water_serial.c:153:                       + 0.5 * gravity * (PN(y, x) * PN(y, x) / density);
    addpd   %xmm1, %xmm0    # vect__57.318, vect__60.319
```

Figure 1: Snippet of assembly code for serial code with -O3.