Parallelle beregninger - TDT4200 - Problem Set 2

Ola Kristoffer Hoff

19. September 2022

2 Questions

2.1 How did you avoid a deadlock during the border exchange?

I began by sending data north and receiving from south. This way all the data would be sent north, execept for at the north boundary. Since the processes at the north boundary do not send data, they go straight to receiving data, so a deadlock would be impossible. Imagine if all elements "south" of it was stuck (blocking) waiting on the process directly above itself to receive their data, then this would propogate through to the top. However, since the top doesn't send data, it would receive data, allowing the process below to receive the data from below it. This would propagte through the processes, and hens a deadlock is impossible. Then I do the same thing with sending south and receiving north. Than again for east-west and west-east.

Since every process do the same steps, and all the sub-steps are impossible to deadlock, then the steps put together cannot deadlock either. This due to the fact that the first sub-step (sending north, receiving south) cannot deadlock, so it is must go on to the next step, which also cannot deadlock, and this continues.

2.2 How could creating derived MPI datatypes be useful for the border exchange?

It would be beneficial to only specify the datas mapping once, when creating the type, and everywhere else just give the start address and the type. I did create types for the row and column.

Another benefit of having derived MPI datatypes would be to limit the amount of messages sent between the processes. If you could wrap all the data into one structure and send it in one message that would reduce the amount of traffic on the network, this would speed up the application, given that the network transfers are slower than the overhead of putting the data together.

I tried to create one (one for rows and one for columns) wrapping datatype to store the values from all three matrices (mass, mass_v_x and mass_v_y). I used a *hindexed*-type and found the *displacement*-values be getting the difference from the mass arrays address to the others.

```
const int NUM_BLOCKS = 3;
const MPI_Aint displacements[NUM_BLOCKS] =
{0, &PNU(0, 0) - &PN(0, 0), &PNV(0, 0) - &PN(0, 0)};
```

However, it turned out that the processes had non-deterministic allocation of the addresses. This meant that not all processes was guaranteed to have the same displacement values. This caused a lot of problems which I did not see an end to.

2.3 Describe the pros and cons of using MPI Cart shift in connection with cartesian communicators?

The benefit of using the *shift*-function was that one could easily get the rank of the processes neighbouring. This way one could address the correct process in the border exchange part. This required little overhead, just two funciton-calls, (the overhead lays inside those funcitons, but our code is cleaner), and even if the process did not have a neighbour in a directoin the value return would just be ignored by the *send*-calls, hens we did not need to check if we could send data in the direction, this was implicitly taken care off.

The downside to using this is the confusion. At least in my program the x- and y-axis did not algin with the north-south-east-west-values in an intuitive way. Having north point in the negative x-axis was at least not what I expected.

However, these troubles could have been caused by the handout code saving to file in a "y by x" matrix and not a "x by y" matrix, so things was a bit flipped on their head.

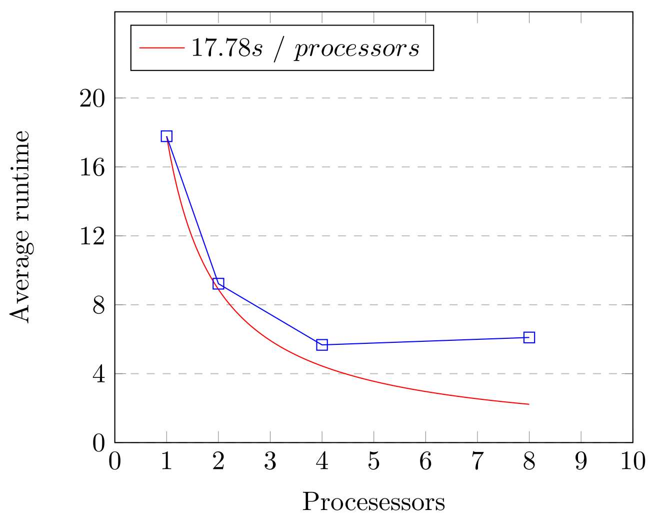
2.4 Look at the execution time of the time loop when running with 1 processes. Look at the execution time of the time loop when running with 2, 4 and 8 processes and document the speedup you get compared to running with a single process. Is the result as you would expect? Why/why not?

Processors	Time_1	Time_2	Time_3	Time_4	Time_5	$\text{Time}_{average}$
1	17.77s	18.08s	17.76s	17.65s	17.62s	17.78s
2	9.25s	9.17s	9.13s	9.30s	9.24s	9.22s
4	5.48s	5.44s	5.55s	5.73s	6.17s	5.67s
8	5.91s	5.90s	6.13s	6.18s	6.38s	6.10s

The runtimes for the different number of processes are given in the table above, and the average time is plotted in the graph below. As we can see, the average time decreases with the number of processes, hens more processes, equals faster execution. I glossed over the last average time for now.
This is about what we expect, but we are more intrested in how much faster the runtime is. If we estimate that almost all of the execution time is in the parallel part, hens ignoring Amdahl's law for now. We would expect the runtime to be cut in half when we double the number of processes. This is also what happens roughly in our results.
The runtime cannot reach zero, so when the number of processes goes twards infinity we get a constant value, this is the code running in the serial part, this is just as Amdahl's law says. It could look like we have reached that point when we run with eight processes, since we have flattened out (out more accurate, we use more time with eight than with four processes).
However, I think this is due to the fact that the machine I am running the code on only have four physical cores. This means that i need to use the *–oversubscribe*-flag when running with eight and this means that they are not actually running in parallel, but the OS switches between them.

We can see the red continoues line in the plot represents the theoretical value for cutting the runtme in half when doubling the number of processes.

Average runtime per number of processors


$17.78s$ / $processors$
Average runtime
Processessors