TDT4200: Parallel computing

Assignment 3: Image Edge Detection with MPI

Name: Martin Rebne Farstad

Task 0.3a

Iterations	*Time (s)*
1	0.262
1024	47.463

Table 1: Serial

Task 1e

Iterations	*Time (s)*	*Speedup*
1	0.348	0.753

Table 2: Parallel (2 cores)

Compared to the serial application, parallelism with only one iteration seems to add too much overhead.

Task 2c

Iterations	*Time (s)*	*Speedup*
1	0.325	0.806
1024	30.054	1.579

Table 3: Parallel (2 cores)

Same as mentioned in task 1e, parallelism with one iteration adds to much overhead. For 1024 iterations however, we see a significant speedup.

Task 2

(b) The communication pattern consists of two broadcasts, one scatter, one gather and the *sendrecv* function calls each process performs each iteration in the for loop. The broadcasting consists of the process with rank 0, from now on called process 0, distributing the image width and height to all the other processes. This will be $2(n-1)$ communications. The data in two broadcasts each consist of one integer (4 bytes), which makes a total of $8(n-1)$ bytes, where n is the amount of processes.

The scattering consists of distributing smaller separate chunks of the original image data to different processes so they can be processed in parallel. The gathering consists of taking the results of each parallel computation from each process and gather them into the resulting image of the original size. Let $process\_0\_rows$ be the amount of rows process 0 is processing. $2(n-1)$ communications are needed between process 0 and the other processes to scatter the original image data and gather the sub image data, and the amount of bytes exchanged is the following, where the amount of bytes in a pixel struct is 3:

$$2 * (image\_height - process\_0\_rows) * image\_width * 3$$

The communication using *sendrecv* works as follows. First each process except process 0 will send their south border to the process with one rank lower than themselves. They will have to wait until process 0 calls *sendrecv* and sends it's north border to process 1. Then process 1 can receive it's data from process 2. Now process 1 sends it's north border to process 2 and so on until every process except process $world\_rank - 1$ have sent their north border to the next process with one rank higher, and properly received the south border from the process with one rank above themselves. This is repeated for 3 iterations

Every process except process 0 and process $world\_rank - 1$ will do two sets of send and receives. They will therefore each contribute with 2 communications. Process 0 and process $world\_rank - 1$ will only do a single send and receive, and therefore only contribute with one communication. Therefore, the following equation counts the number of communications for each iteration:

$$2(n-2) + 2 = 2n - 2$$

For 3 iterations the communication count will be:

$$3(2n-2) = 6(n-1)$$

If $image\_width$ is the variable containing the size of each row, and each row consists of unsigned characters of one byte, then the total amount of bytes exchanged in the loop is:

$$6(n-1) * image\_width$$

.

The total amount of communications in the program will then be:

$$2(n-1) + 2(n-1) + 6(n-1) = 10(n-1)$$

.

The total amount of bytes transferred:

$$8(n-1) + 6 * (image\_height - process\_0\_rows) * image\_width + 6(n-1) * image\_width$$

.

(d) My approach to solve this problem was to first think about which rows each process should send to each other. I noticed that process 0 and process $world\_rank - 1$ could not exchange both their halo rows. Process 0 could only exchange the north border and process $world\_rank - 1$ could only exchange the south border.

To be able to receive rows from the neighbouring processes, two additional rows had to be appended to the original sub image data. I allocated a new memory region with extra space for two extra rows, and moved the previous sub image data in between the rows.

I figured out that I could use the *sendrecv* function, because each process needed to do both a send and a receive. This was also noted in the lecture note about Ghost Cells. Since the communication was required each iteration, I moved the function calls inside the for loop.

I made sure to draw on paper the flow of the communication and see that it should work. I also noted that the communication could be optimised further, since every process has to wait for several processes to receive their borders after sending initially. I decided that this was out of the scope of the assignment. I did no changes to the kernel computation.