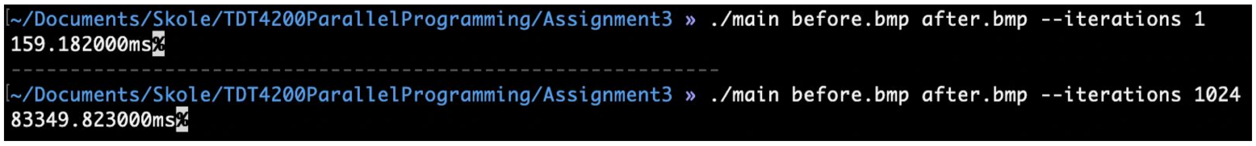
Task 0.3a

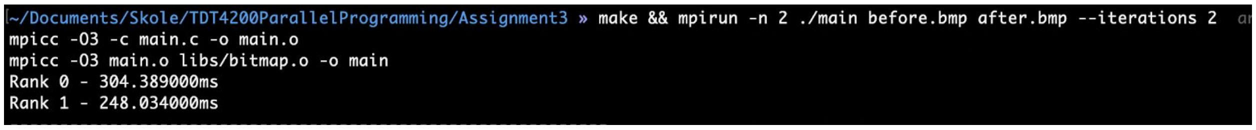
I use clock() and CLOCKS_PER_SEC for calculating the elapsed time for each rank when executing the programs. My baseline readings are as follows for 1 and 1024 iterations in milliseconds
1 iteration: 159ms
1024 iterations: 83350ms



```
[~/Documents/Skole/TDT4200ParallelProgramming/Assignment3 » ./main before.bmp after.bmp --iterations 1
159.182000ms%
------------------------------------------------------------
[~/Documents/Skole/TDT4200ParallelProgramming/Assignment3 » ./main before.bmp after.bmp --iterations 1024
83349.823000ms%
```

Task 1e

When running with mpi using my 2 cores, the time taken for 1 iteration was as follows.



```
[~/Documents/Skole/TDT4200ParallelProgramming/Assignment3 » make && mpirun -n 2 ./main before.bmp after.bmp --iterations 2
mpicc -O3 -c main.c -o main.o
mpicc -O3 main.o libs/bitmap.o -o main
Rank 0 - 304.389000ms
Rank 1 - 248.034000ms
```

Here we see a significant decrease in performance, this is likely due to the setup of the MPI environment, and the distribution av the data taking considerable amount of time.
Without MPI 1 iteration used 160ms, and with 300ms. I look at Rank 0 as this is the one loading and storing the processed image and as this takes additional time is the most accurate. So this took about double the time than running without MPI.

Task 2b

I read the question so that the data and communication is only for the border exchange not the setup or gathering before/after the iterations.

First one exchange the border southwards.
Rank 0 sends border to Rank 1
Rank 1 sends border to Rank 2
…..
Rank n - 1 sends border to Rank n

After the southwards exchange, the borders are exchanged northwards.
Rank n sends border to Rank n - 1
….

Rank 2 sends border to Rank 1
Rank 1 sends border to Rank 0

Each border contains one row of the image. This means that each exchange transmit WIDTH bytes between the nodes. Since in the southwards Rank n only recieves the total number of bytes transmitted = (n - 1) * WIDTH for southwards. The same also for northward since Rank 0 only receives data. This makes that:

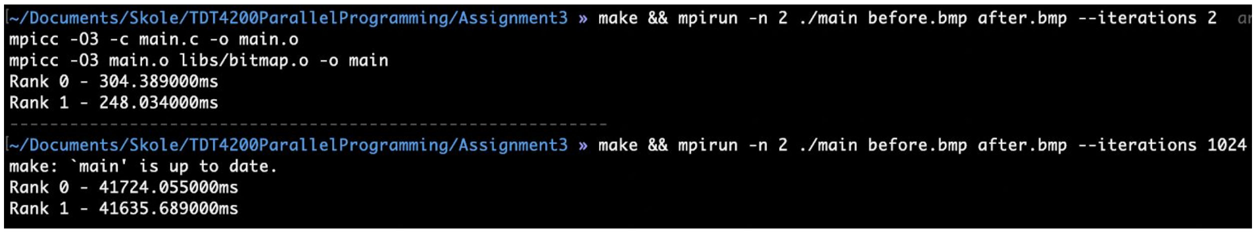
1 iteration (bytes) = (2n - 2) * WIDTH bytes.
3 iterations (bytes) = (6n - 6) * WIDTH bytes.

For mye runs (2 processes) amount of data transmitted.
3 iterations (bytes) = 6 * WIDTH bytes.

For the example image this equals
3 iterations (bytes) = 6 * 2560 bytes = 15360 bytes.

When calculating the number of communications, since every communication is of the size WIDTH, then for 1 iteration 2 communications used if 2 processes. This means that for a total of 3 iterations 6 communications are needed (2n - 2 communications where n in number of processes).

Task 2c



```
[~/Documents/Skole/TDT4200ParallelProgramming/Assignment3 » make && mpirun -n 2 ./main before.bmp after.bmp --iterations 2  ar
mpicc -O3 -c main.c -o main.o
mpicc -O3 main.o libs/bitmap.o -o main
Rank 0 - 304.389000ms
Rank 1 - 248.034000ms
------------------------------------------------------------
[~/Documents/Skole/TDT4200ParallelProgramming/Assignment3 » make && mpirun -n 2 ./main before.bmp after.bmp --iterations 1024
make: `main' is up to date.
Rank 0 - 41724.055000ms
Rank 1 - 41635.689000ms
```

Baseline was 160ms for 1 iteration, with MPI 1 iteration over 2 processes takes 300ms. Baseline for 1024 iterations is measured to 83350ms, with MPI 10224 iterations over 2 processes takes 41720ms. Here one can see that the extra computation for setting up of MPI and initial distribution of the data is benefited with about half of the computation time saving 41,5 seconds of computation.

Task 2d

First I initialized the MPI environment, then gave rank 0 the task of loading the image and creating the imageChannel. I then broadcasted the size of the original image for count and displacement calculation for MPI_Scatterv/MPI_Gatherv. Then I Sent the ranks the data with Scatterv giving the last rank the remaining data caused by image height not being divisible by number of ranks. (Sent entire rows, not rectangles) Then I made image channels with space for the ghost rows. Then started the iteration, exchanged southwards first, than northwards with special cases for rank 0 and rank n. For southwards rank 0 only sends and the send/receive chains down to rank n that only receives. This is opposite for the northward with rank n only sending and rank 0 only receiving. Then when the processing finished the data is gathered with Gatherv. This is placed back into the imageChannel originally created on load of image, now containing the new image values and rank 0 create the rgb channels and write the image to disk.