CS426 HW1

Barış Can 21501886

1 Implementation and Design Choices

Serial parts of the homework were fairly easy and there was not much to decide. However, there are many things that I think was not clear enough for the parallel implementations. First of all, in the homework, it says, "The master must also perform computation". However, I could not decide if the "computation" word stands for calculating the max from the start or calculating the min from the outputs that come from each processor. Therefore, in V1 of part A, since master also calculates the min of the outputs that come from processors, I did not give it work for calculating the local minimums, I divide the calculating local minimums job between n-1 processors and the master calculated the global minimum after outputs came from n-1 processors. Furthermore, in V2 of part A, since the master does not calculate any global min from outputs, I divided the work between n processors including the master. For both versions, and V1 of part B, I divide the array into equal sizes and if the dividing does not give an integer number, I gave the remaining items to the last processor. This generally gives a good partitioning if there are several numbers of processors as in real life. In addition, in V2 of part B, I could not understand if we must use both Allreduce and Bcast together or one of them is sufficient. As I could not understand the question and after thinking to broadcast the whole array to all of the processors would be a bad idea, I sent the arrays after dividing into pieces like I did in the V1 then collected the results using Allreduce method. The observations of this part can be seen in discussions.

For part B, I read the entire file and save it in an array. Then, for the serial part, I multiplied the RGB values with the desired numbers and saved them line by line. For the parallel part, I again read the entire file, decide which rows processors will process and sent them again line by line. Processes multiplied the RGB values with the desired numbers and sent the grayscale values to the master line by line. Master then combined them and write the outputs to a text file. For the V1, there was not much to decide. I just could not understand the sentence "You also have to send neighbour rows for the borders of row patches", but I believe I did this part. For the V2, since it should be a perfect square, I changed the input file and reduced the number of rows to 900 to make it divisible by 4, 9 and 16. Those numbers were my test numbers. There were several small differences between V1 and V2 in my code but the most important difference was two for loops that arrange the rows and columns to

send them as squares. Instead of sending the rows directly in increasing order, I sent them like squares again line by line. After sending the lines I applied the same procedure as V1. For the V3, I tried many things, first I tried to write the file in processors in rank order to eliminate the overhead of sending the distributed arrays to master, but I could not manage this since after searching a way to make processors work in order, I realized that there is no shared memory feature in MPI, Each processor has its own memory. Therefore, I discarded this idea. Then, I tried a different partitioning algorithm because I thought by sending the RGB values as squares we try to send similar pixels to use the cache more, but since the processor number is limited in our computer, we still cannot use this feature as much and sending them rows by rows were much simpler and efficient in my code. Therefore, I tried to find a better partitioning algorithm for preventing this, but after searching on the internet and thinking about it, I could not come up with a better algorithm. Finally, I decided to not sending the partitions to processors to eliminate the sending cost. They all have access to the same array and I calculated their borders. I also did not send the borders since each processor has access to them. After calculating the grayscale values, I again sent them to master to write them in a text file. I again tried to write the file in processors, but since there is no shared memory mechanism, they were just overwriting each other's data including the master, so I decided to send the values to master.

As for the general of the code, I could not find any MPI library designed for Windows, I found some versions but there were old and has no support. I found some software that can emulate Linux and use some features of MPI, but I could not trust it. Therefore, I used Visual Studios MPI library, MS-MPI, however, there was no C feature on it and I wrote my code on C++ and changed it to C. I could not find any problems with it and it worked the same. I wrote my code on Visual Studio and saw many users that use MS-MPI. Mr Ozturk also said it is ok as long as it is MPI, so I continued with it. I will upload both C and C++ codes in case there could be problem.

2 Discussion

For Part A, since it is a very simple and straightforward algorithm. I think using parallel programming created a big overhead. As can be seen from Figure 1, the running time of the algorithm increased as I increased the number of processors. Moreover, V2 had a longer running time since we are not only sending the outputs to master but all the processors and they calculate the minimum together. Therefore, instead of each processor sending a local minimum to master which makes $n-1$ sends, every processor sent their local min to every other processor making n^2 sends which increased the running time.

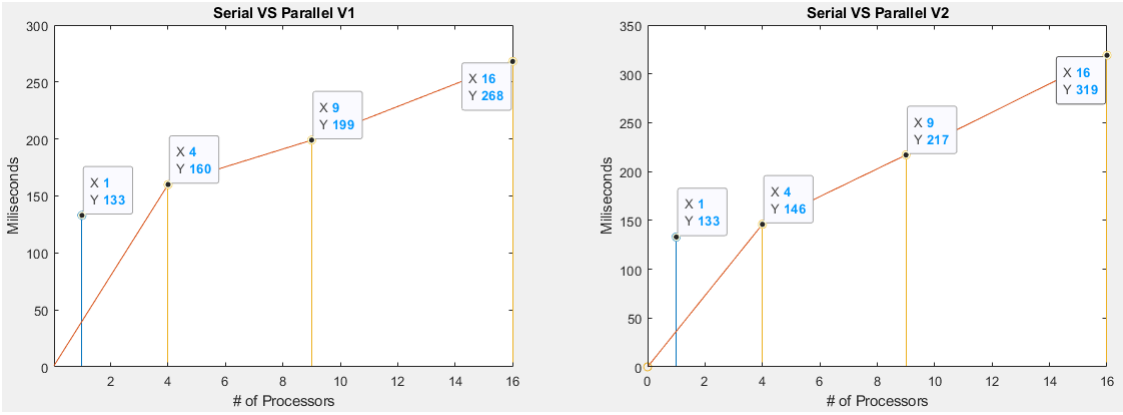

Figure 1: Comparison of the Serial and Parallel Implementation for Part A

For Part B, the running time of parallel programming decreased since it utilized the processors better and the size of the image was average as it can be seen from Figure 2. It was not a very big change, but the running time was quicker regardless. I felt like there could be better implementations that have better results since I could have used some loops that do useless work and increase the complexity. As I mentioned above, I think the running time of the V2 should be less than V1 since it sends the similar pixels at the same time assuming similar pixels are near each other and there will be less cache misses, but I think partitioning an $900X900$ array to 9 is not enough to group similar pixels. I think if I had 30 processors and obtain an $30X30$ array when I divide them as squares, I could have been much better. However, after 30 processors it would not mean anything since the groups have a small of pixels. For the V3, not sending the arrays to processors made a big change on the running time. The running time of the four processors for the V3 was the least among all versions. However, I could not understand the rapid increase in the running time while increasing the number of processors. I thought since there are a lot of processors, they access the whole array the same time and this caused a rapid increase in the running time. However, considering many computers does not have 16 processors, the running time of the 9 processors was still better than all other versions.

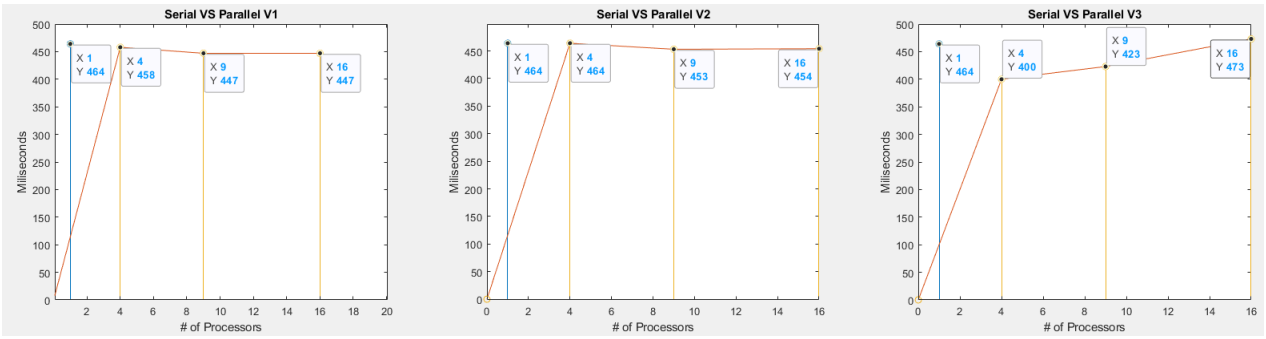

Figure 2: Performance Comparison of the Serial and Parallel Implementation for Part B