Rüzgar Ayan 21801984

2020-2021 SUMMER CS426 PROJECT 1 Report

Implementation and Design Choices

Part A. Count the ACGT

The serial part was straightforward. In the parallel parts, I was initially reading the file line by line using the getline function and sending these lines to the other processors one at at time by char arrays. To indicate that all the lines are finished, I was sending them a special char array so that they can start sending the accumulated counts. But sending a message for each line was causing a lot of overhead. I tried to do following to reduce this problem: First, find out the total size of the file. Then compute the size of the part that each processor should be assigned. Send this size to the processors so that they can allocate the required memory. Finally send the whole parts of the file to the processors that they are assigned to in a single message per processor. Doing this resulted in better execution times. However, since the PDF document was asking for the initial solution (reading line by line), I commented out this faster one in all the files.

Only change of the MPIv2 from MPIv1 was that instead of all processors sending their information to the master process, I have added an MPI_Allreduce operation to the end that sums up all the counts.

Part B. Neural Network

In this part too, I wanted to send the weights data to the corresponding processors in single messages. So, I have parsed the Weights file into a 1D array such that numbers in a column of file is parsed contigious to the array. After finding out the number of columns in the Weights file (by counting the commas in a line), the program decides how many columns will be sent to each processor and then sends this number and the data. Workers find the output to the given columns and finally, master collects all the outputs from the workers and writes them to a file.

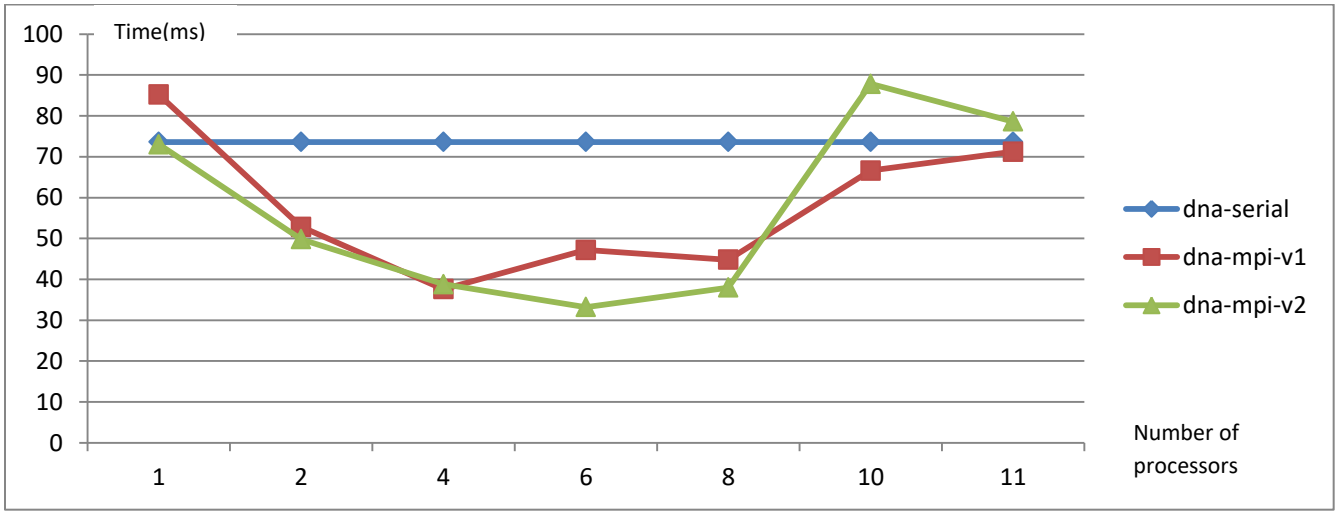
My implementation for MPIv1 part was general enough to work with any number of processors. So I just changed one line from **int numWorkers = 10;** to **int numWorkers = numprocs - 1;** for MPIv2.

Performance

Since the execution times were varying a lot in the parallel ones, I ran each instance of the program 5 times with the same arguments and calculated the average. Before plotting the averages, I will also give the maximum and minimum execution times in tables to get even more information. Note that I have assigned 11 of my 12 cores to the virtual machine that I have ran the tests on.

Part A. Count the ACGT

Program	Number of processors	Minimum (ms)	Average (ms)	Maximum (ms)
dna-serial	-	72	73.6	80
dna-mpi-v1	1	69	85.2	104
	2	42	52.8	73
	4	32	37.6	43
	6	35	47.2	58
	8	29	44.8	67
	10	34	66.6	112
	11	36	71.2	109
dna-mpi-v2	1	72	73	74
	2	44	49.8	56
	4	34	38.8	47
	6	29	33.2	41
	8	29	38	47
	10	47	87.8	165
	11	37	78.6	114



First, we see that increasing the number of processors decrease the runtime. But still, doubling the number of processors does not really halve the execution time. Since there is not much computation for the data we send, I was already expecting this (If we were doing some more complex computations like pattern searching, I believe parallelization would result in much better runtimes). And increasing the number of processors even higher results in execution times even worse than the serial case. This is probably because some processors are busy with other background tasks when we assign most of our processors. These processors will bottleneck the whole program since the master process waits for

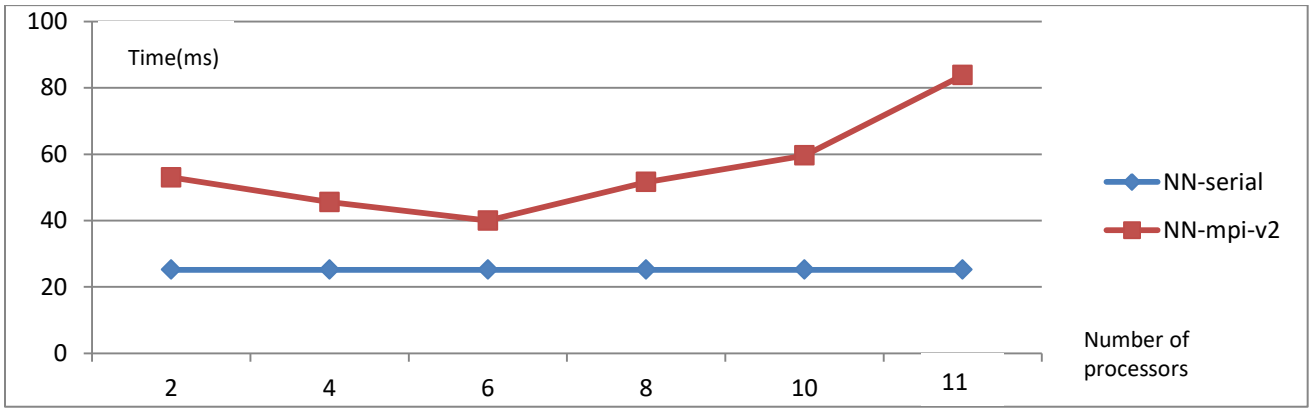
sending the data those slower ones. As I explained in the beginning, I have also written another version where only 1 message is sent per processor. With that implementation, this bottleneck is quite reduced.

Finally, it seems that dna-mpi-v2 runs slightly faster than dna-mpi-v1 which suggests Allreduce operation is faster than making Send and Recv with all worker processors one by one.

Part B. Neural Network

The given data for this part was very small, so I did the tests with a larger Weights file. Since the NN-mpi-v1 is just the special case of NN-mpi-v2 when it is ran with 11 processors, I just give the execution times for NN-mpi-v2. Also, all the file reads were done at the beginning for this part's programs. So I didn't include the parsing times in the following table.

Program	Number of processors	Minimum (ms)	Average (ms)	Maximum (ms)
NN-serial	-	20	25.2	32
NN-mpi-v2	2	39	53	67
	4	26	45.6	66
	6	27	40	56
	8	25	51.6	76
	10	34	59.6	75
	11	52	83.8	139



The results for this part was surprising for me. After seeing the reduction of runtime in the previous part where the only job of workers was counting the letters, I thought the same would happen in this part as well. However, unlike the part A, just running the parallel program with 2 processors caused a lot of overhead compared to the serial one. The overall trend is similar with part A: Increasing the number of processors up to 6 reduces the runtime, but after that the runtime also increases because of the discussed reason. But even at its best case, parallel program could not beat the serial one for this part.

I would expect the results of the Part A and Part B to be similar but I could not find out the reason of this inconsistency.