PHPC : Project

Aiday Marlen Kyzy

Sciper : 283505

Due on June 13th 2021

1 Introduction

In this project, I used the CUDA and MPI library in order to speed up the execution of the code given in class which computes a solution to a system of linear equations using the conjugate gradient method. In the following section I discuss first the use of the MPI library.

2 MPI Library

2.1 Strong Scaling

To run the code, type the following in the terminal : `srun -n number_tasks ./cgsolver lap2D_5pt_n100.mtx`. The speed up is defined as the ratio of the execution time for one processor to the execution time for p processors :

$$S(p) = \frac{T_1}{T_p} \tag{1}$$

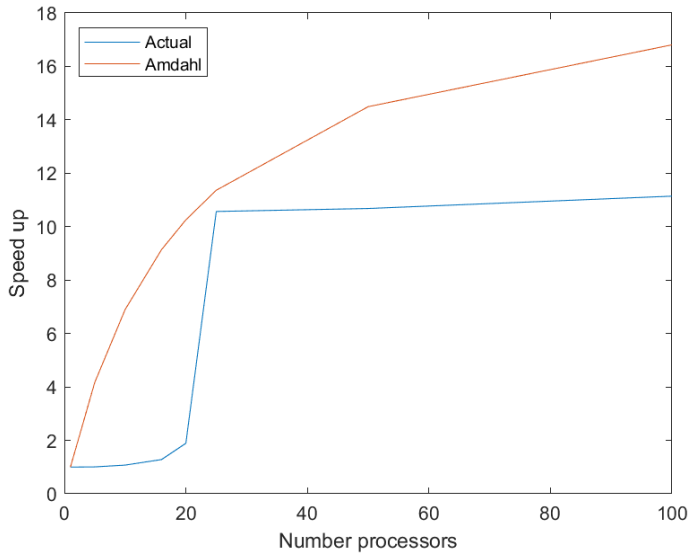
Amdahl's law states that the maximum achievable speed up given that a fraction f of the code can not be parallelized must obey the following inequality :

$$S(p) \leq \frac{p}{1 + (p-1) \cdot f} \tag{2}$$

Now in the code given, I approximate f to be around 0.05 (also confirmed by teacher). I find the following execution times for different numbers of processors. Since the matrix I use is 10000 by 10000, I choose numbers that divide 10000.

Number of processors	Execution time in seconds
1	35.0095
5	34.7741
10	32.5960
16	27.3066
20	18.5132
25	3.31237
50	3.27791
100	3.14202

Using this data above I am able to draw the following strong scaling graph with Amdahl's prediction.



In the beginning when we increase the number of processors from 0 to 20, we do not see a large increase in the speed up. This is perhaps due to the architecture of the machine (we do not see a large change until we use more nodes despite using more threads). When the number of processors increases from 20 to 25, there is a large speedup of almost by a factor of 10, which is close to the maximum speedup predicted by Amdahl's law. However from that point on, although the speed-up increases, it does not change a lot. This is perhaps due to the fact that when each thread works on a small set of rows, it executes the code so quickly, that making it execute on an even smaller set of rows has a marginal effect. In this case it would mean that the plateau of 3 seconds represents the execution of the code which was not parallelized, and that the speed-up is not going to be in practise much higher than this. Since my graph is still far from Amdahl's prediction, it means that I can further attempt to parallelize my code and improve the execution time.

2.2 Weak Scaling

In this section, I plot the weak scaling curve. In this case we consider the efficiency which is defined as follows.

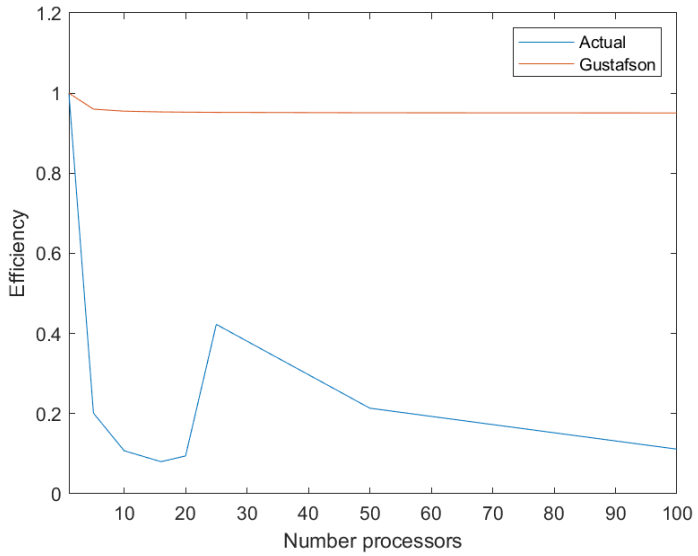
$$E(p) = \frac{S(p)}{p} = \frac{T_1}{p \cdot T_p} \tag{3}$$

Now Gustafson's law says that for a code for which a fraction f of the code is necessarily serial, then the maximum efficiency one can achieve is as follows :

$$E(p) = \frac{S(p)}{p} = \frac{p - f \cdot (p-1)}{p} = 1 - f + \frac{f}{p} \tag{4}$$

We plot the efficiency with Gustafson's law as follows below.

From the figure below, we see that the maximum achievable efficiency according to Gustafson's law is close to 1. This is because the fraction of serial code is low. The actual efficiency I get, attains a peak of around 0.4 when 25 threads are used. This is indeed inline with the previous strong scaling graph. Following this the efficiency falls off to 0.1/0.2. This is a very low value, and it indicates that adding more threads, does not proportionately decrease the execution time. This was already discussed in the previous section, and it was hypothesized that in practise the effect of adding more threads becomes marginal past a certain threshold. From the graph, I see that the efficiency and the speed-up could theoretically be greatly improved.

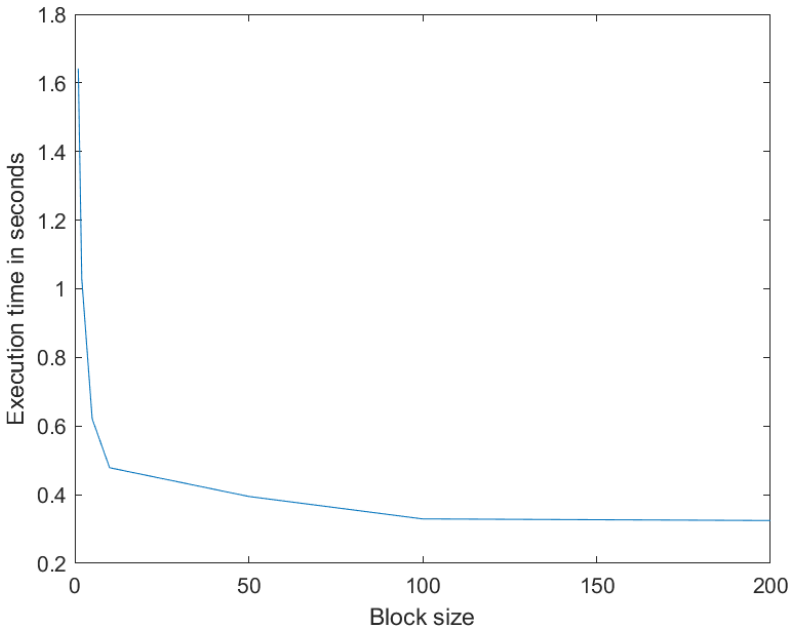


3 CUDA library

In this section we use the CUDA library in order to optimize the code. To run the code one needs to go into the folder `project_cuda` and type in the terminal : `srun --gres=gpu:1 --qos=gpu_free --partition=gpu ./cgsolver lap2D_5pt_n100.mtx`. I made the choice that each thread will work on groups of complete rows only. We will study the effect of changing the block size on the computation time. Choose a grid of 10 blocks arranged vertically. Now each block is in charge of 1000 threads. Consider the following block sizes : 1, 5, 20, 100, 200, 500, 1000. We have the following execution times.

Block size	Time in seconds
1	1.64167
2	1.02973
5	0.621331
10	0.478634
50	0.394957
100	0.329465
200	0.324736
500	0.319286
1000	0.322244

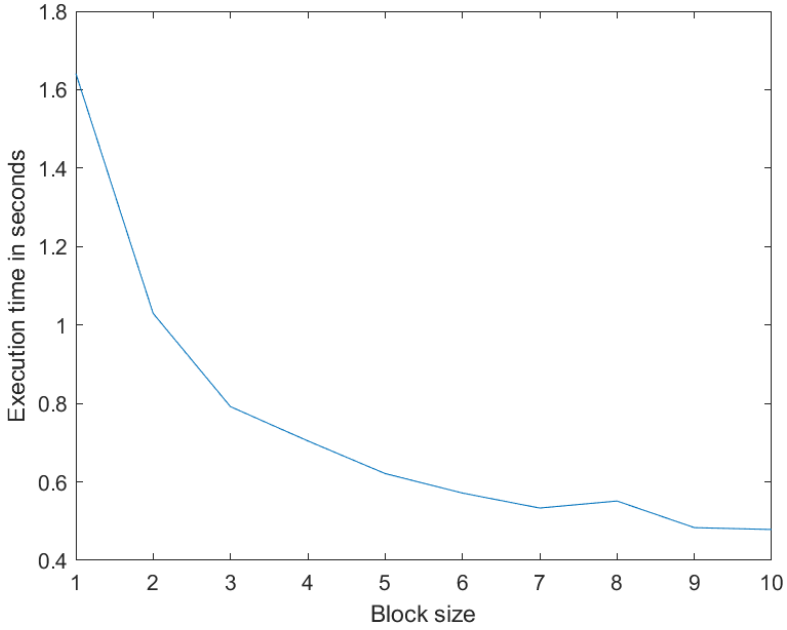
The plot is as follows. From the figure below and from the data, we notice that increasing the block size from 1 to 5, decreases the execution time by more than a factor of 2. Similarly by going from a block size of 5 to 500, we decrease the execution time by more than a factor of 2. However the decrease in execution time becomes marginal as the block size increases, and one requires bigger and bigger block sizes to attain the same speed up. The biggest change is in the beginning. The plot shows block sizes from 0 to 200, because for higher block sizes, the decrease is not significant.



I provide the next zoomed in graph where we limit our attention to block sizes from 1 to 10, with all intermediate integer values :

Block size	Time in seconds
1	1.64167
2	1.02973
3	0.791903
4	0.704583
5	0.621331
6	0.571626
7	0.533325
8	0.551264
9	0.483394
10	0.478634

In this way, we obtain the following plot on the next page with the above data. We notice once again that as the block size increase, the execution time decreases, but the rate of decrease is slower. From the two graphs in this section, I conclude that the optimal choice of block size is around 50 to 100. Past this, the execution time does not change much.


1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
Execution time in seconds
1
2
3
4
5
6
7
8
9
10
Block size