MIDDLE EAST TECHNICAL UNIVERSITY

Project Report

CENG478 - Introduction to Parallel Programming
Parallellizing Matrix Multiplication

July 5, 2021

Student Name/Number: Ersel Hengirmen / 2468015

Contents

1 Introduction

In this problem we will focus on N×N matrix multiplication with 2 N×N matrices. Let's say we have A and B matrices (A and B being N×N matrices) and want to find A.B=C matrix. For example:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} * \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}$$

To find any c_{xy} in a matrix multiplication like the one above, we would need to calculate the below equation:

$$c_{xy} = \sum_{z=1}^{N} a_{xz} * b_{zk}$$

Which would require O(N^3) amount of time complexity.

There are also algorithms like Strassen's algorithm which has a complexity near O($N^{2.8074}$) and as of December 2020 there has been a new algorithm of complexity O($N^{2.3728596}$) which is said to be best complexity ıntil now. This algorithm was created by Josh Alman and Virginia Vassilevska Williams. However this algorithm is a galactic algorithm.[1]

In this project we will not focus on the other algorithms but try to design a Paralel model for O(N^3) model to reduce its complexity because of its convenience to adapt into parallel models. We will use Cannon's algorithm to parallelize it[2].

Note: A galactic algorithm is one that outperforms any other algorithm for problems that are sufficiently large, but where "sufficiently large" is so big that the algorithm is never used in practice.[3]

2 Proposed Model

2.1 Assumptions

- We are asssuming that we have an hpc network with cut through routing.

- We are asssuming that only one of the processes has the Initial A and B matrices stored in it.
- Because of the first assumption we will assume that message transfer cost is t_s+mt_m
- We are assuming that the network is in 2D-mesh topology.
- We are assuming that multiplication and addition operations take only 1 unit of time.

2.2 Restrictions

- To use cannon's algorithm easily we will restrict process count to be a square number(like 1 4 9 16)
- Also to make the matrix blocks homogenous we will only accept N when it is square root of process count's multiple.(for example when N is 12 the process count can be 4,9,16,36,144 because 12 is divisible to their square roots (2,3,4,6,12))

2.3 Parallelizing Matrix Multiplication

For parallel matrix multiplication of 2 N^2 matrices A and B to create a N^2 matrix C there are 3 basic methods that comes to mind.

1. Every process will calculate rows of the C matrix
2. Every process will calculate columns of the C matrix
3. Every process will calculate a block of the C matrix

Even though in the end there will be exactly the same number of calculations the first 2 methods has an inefficient side opposed to the 3rd method. The problem with the first 2 methods comes from the way the matrix multiplication works. If we are using matrix row method then every process will only need that corresponding row from the A matrix but since the elements in a rows are in every column then every process will need all of the B matrix to process their rows.

so every process(assume there are P number of processes) would need to get N/P rows from matrix A and all of the B matrix which means to send the matrices we ould need to make 1 Broadcast operation and 1 scatter operation from process 0. Broadcast would take O($log(p)(t_s$+N^2t_m)) time (since it is the whole matrix the message size is N^2). On the other hand scatter operation would be done on rows and would take O($log(p)t_s+(p-1)\frac{N^2}{p}t_m$) time ($\frac{N^2}{p}$ is the message size every process recevives) so in the end just sending the matrices would take O(N^{2}log(p)) operations(because we need to broadcast the whole matrix)

On the other hand for the block approach we will only need to send rows from A matrix and columns from B matrix normally and since we chose cannon's algorihm we will only send **blocks** of $\frac{N}{\sqrt{p}} \times \frac{N}{\sqrt{p}}$ instead. Which would need 2 scatter operations of complexity O($log(p)t_s + (p-1)\frac{N^2}{p}t_m$)). so the complexity of initialization would be O(N^2) for this case.

2.4 Cannon's Algorithm

The problem with Block based matrix multiplication is the fact that when we try to calculate an entry of the result matrix's value(for example c$_{ik}$) we tend to start from a_{i0} and b_{0k}. Which means that every process would need to start with left-most A group and upper-most B group. (For example in the matrix below the Group 1 2 and 3 of C matrixes would need to start with Group1 of A.)

$$\begin{bmatrix} Group1 & Group2 & Group3 \\ Group4 & Group5 & Group6 \\ Group7 & Group8 & Group9 \end{bmatrix}$$

Cannon's algorithm uses a way to bypass this problem. By their method instead of sending every group of block of matrix to their correspongind process we would instead rotate every block of A by its block row and every block of B by its block column according to their block row number and block column number. In this way at any given time, each process would be using a different block in every iteration(while keeping the computation accurate).

For example assuming the above matrix is A it would become

$$\begin{bmatrix} Group1 & Group2 & Group3 \\ Group5 & Group6 & Group4 \\ Group9 & Group7 & Group8 \end{bmatrix}$$

and if it was the B matrix it would become

$$\begin{bmatrix} Group1 & Group5 & Group9 \\ Group4 & Group8 & Group3 \\ Group7 & Group2 & Group6 \end{bmatrix}$$

Note: We are not actually swapping the blocks then sending them. Instead we are sending corresponding A and B blocks to neccessary processes.

After Initial block sending from process 0 is done every program will need to calculate their given blocks. But to calculate C block of for example group 9 the program would need to make $\sqrt{p}$ block matrix multiplications (for Group9 it would calculate AGroup8*BGroup6+AGroup7*BGroup3+AGroup9*BGroup9). To do this every process will send its received blocks(A and B blocks) to the next process and receive 2 blocks from the previous one for $\sqrt{p}-1$ times($\sqrt{p}$ blocks in every row and column but since the first one was received from process 0 we only need to make $\sqrt{p}-1$ iterations.)

Sending these messages would take $t_s + \frac{N^2}{P}t_m$ for every iteration so in total sending and receiving messages to next and from previous partners would take $2(\sqrt{p}-1)(t_s + \frac{N^2}{P}t_m)$

And for every $\frac{N^2}{P}$ entry we would need to make N multiplications and N-1 additions so in total $\frac{2N^3-N^2}{P}$ operations would be done paralelly for every matrix and the time complexity of computations would become O($\frac{N^3}{P}$)

The last part is receiving these matrices. To do this the program would only need to make a gather operations to receive blocks from every process to one of the specified processes. And that would take O($log(p)t_s + (p-1)\frac{N^2}{p}t_m$)) time

So in the end Cannon's Algorithm(Parallel) would make 3 scatter/gather operations(2 scatters for distributing A and B matrixes and 1 gather to gather calculated C matrix) in $3log(p)t_s + 3(p-1)\frac{N^2}{p}t_m$ time. Parallely sends blocks to each partner processes which takes $2(\sqrt{p}-1)(t_s + \frac{N^2}{P}t_m)$ time and calculates the C block of the matrix in $\frac{2N^3-N^2}{P}$ time which comes to $\frac{2N^3-N^2}{P} + t_s(3log(p) + 2\sqrt{p} - 1) + t_m\frac{N^2}{P}(3p + 2\sqrt{p} - 4)$ or O$(max(\frac{N^3}{P}, N^2))$

Ant the sequential algorithm's time only depends on computations which comes up to N^3-N^2 or O(N^3)

Note: Because of the scatter and gather operations time complexity will never exceed O(N^2). And increasing the number of processes is not always efficient.The problem is that just sending the matrix itself would take O(N^2) operations so thinking that this is the best case.

Important Note: Instead of scatter and gather operations I sended and received blocks of A,B and C because of to send Blocks I needed to send them with derived data types[4] because Collective Communication Routines can only be used with primitive data types. Because of that the actual complexity of sending A and B and gathering C became O$(p * t_s + N^2t_m)$) instead of O$(log(p)t_s + (p-1)\frac{N^2}{p}t_m)$). So in the end the total complexity of my implementation is $\frac{2N^3-N^2}{P} + (3p + 2\sqrt{p} - 2)(t_s + \frac{N^2}{p}t_m)$ which stil comes up to O$(max(\frac{N^3}{P}, N^2))$

3 Discussion

I have used mpi libraries to check efficiency with maximum N being 1200,2400,4800 with process count changing between 4,9,16 and 25.

While the parallel version was faster then the sequential it has not reached the expected efficiency of P times speedup.

Below are the calculations with different matrix sizes and number of processors.

calculation(process=4,size=1200×1200)	Parallel time	Sequential time	speedup
1	23.116102	36.413473	1.575243
2	23.155564	36.350138	1.569823
3	23.083859	36.355981	1.574952
4	23.143224	36.860351	1.592705
5	23.124652	36.881687	1.594907
mean speedup			1.581526

calculation(process=9,size=1200×1200)	Parallel time	Sequential time	Speedup
1	14.723486	37.130589	2.521861
2	13.604734	36.407562	2.676095
3	14.824080	36.372106	2.453582
4	14.044482	36.405685	2.592170
5	14.767953	36.410339	2.465496
mean speedup			2.541841

calculation(process=16,size=1200×1200)	Parallel time	Sequential time	Speedup
1	14.302692	37.062425	2.591290
2	15.699542	36.390916	2.317960
3	14.494124	36.373989	2.509567
4	14.112281	36.328825	2.574270
5	14.106683	36.403323	2.580572
mean speedup			2.514732

calculation(process=25,size=1200×1200)	Parallel time	Sequential time	Speedup
1	12.045814	37.014805	3.072835
2	11.861033	36.401433	3.068993
3	11.591858	36.319211	3.133165
4	11.933886	37.024892	3.102500
5	12.255832	36.404675	2.970396
mean speedup			3.069578

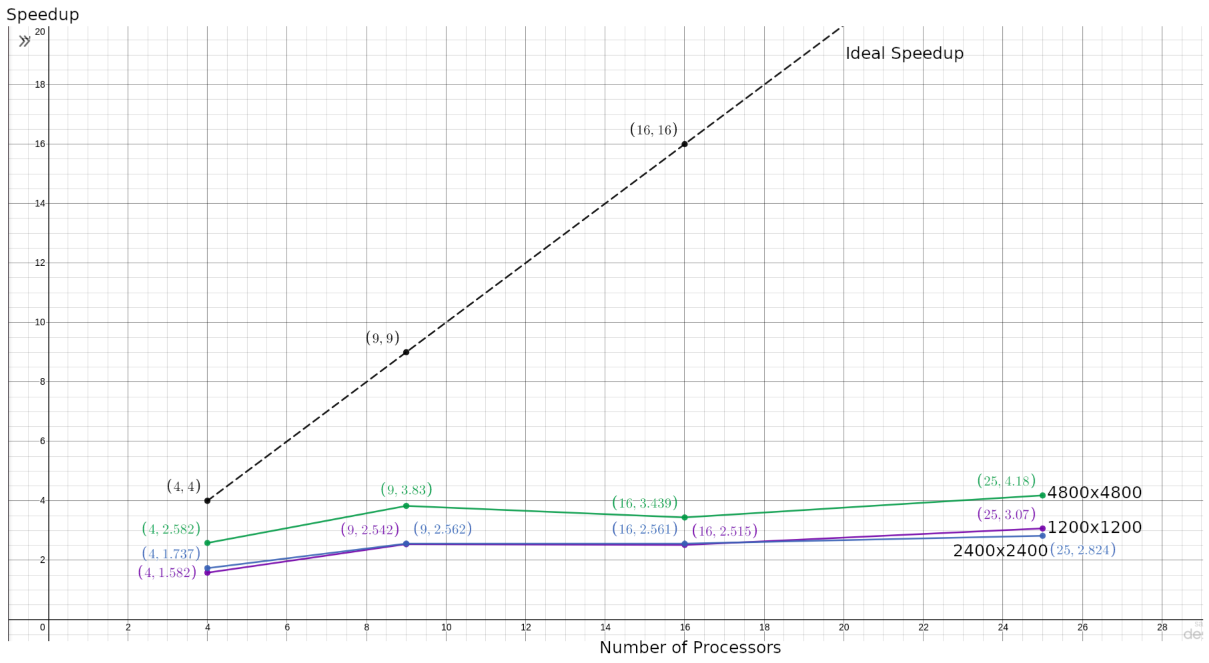
calculation(process=4,size=2400×2400)	Parallel time	Sequential time	Speedup
1	192.699241	318.211109	1.651335
2	184.211859	322.091272	1.748482
3	183.900926	321.474902	1.748087
4	181.326152	324.037898	1.787044
5	183.490697	321.113274	1.750024
mean speedup			1.736995

calculation(process=9,size=2400×2400)	Parallel time	Sequential time	Speedup
1	114.463123	319.798532	2.793900
2	134.886585	321.363734	2.382473
3	125.484979	331.610642	2.642632
4	129.826397	332.186361	2.558696
5	131.554516	319.788523	2.430844
mean speedup			2.561709

calculation(process=16,size=2400×2400)	Parallel time	Sequential time	Speedup
1	125.227370	319.848924	2.554145
2	127.693532	319.822852	2.504612
3	130.524845	320.510458	2.455551
4	123.627115	320.358177	2.591326
5	122.936653	331.839329	2.699270
mean speedup			2.560981

calculation(process=25,size=2400×2400)	Parallel time	Sequential time	Speedup
1	116.102483	332.032042	2.859818
2	116.985520	319.799840	2.733670
3	115.738467	331.603987	2.865114
4	113.796615	320.154603	2.813393
5	112.264850	319.957491	2.850023
mean speedup			2.824404

process count(size=4800×4800)	Parallel time	Sequential time	Speedup
4	1410.640666	3642.143507	2.581907
9	952.684290	3649.187653	3.830427
16	1070.134237	3680.21908	3.439025
25	880.903901	3682.348018	4.180192



The network I was using was the halley network which has 32 nodes. Because of its node count being 2^5 I assume that it's topology is 5d-hypercube. It's network topology might be the reason why the parallel version was running better for 9 processors intead of 16 processors (unlike how it should be according to Amdhal's law). But in general increasing number of processors increased speedup of the program(The increase of speedups after changing the number of processors was always a lot less then the expected increase).

Note: Even if the algorithm was working as fast as the expected speedups we would never reach the speed of amdhal's law because to after some point the communication overhead would actually decrease the efficiency of the algorithm.

According to Gustaffson's law speedup should approach to ideal speedup with the increase of workload. But this wasn't the case between 1200×1200 and 2400×2400 input cases even though the workload has increased by 4 times. The speedup increased when we increased input matrices to be of size 4800×4800. But the reason for the speedups to not change between

1200×1200 and 2400×2400 inputs still remains a mystery.

Speedup did not meet the calculated expections. (assuming t_s and t_m are little numbers (1)) for N=1200 and p=4. I was thinking of the theoritical model of using scatter/gather in my calculations which led me to believe that I should have been getting 4 times faster for the parallel algorithm(since both of their gamma time complexities were same).

$$\frac{\frac{2N^3-N^2}{p} + 3log_2(p) + 2\sqrt{p} - 1 + \frac{N^2}{p}(3p + 2\sqrt{p} - 4)}{2N^3 - N^2} =$$

$$\frac{96440001}{383840000} = 0.25125052365 \text{ for p=4 N=1200}$$

$$= 0.1124542 \text{ for p=9 N=1200}$$

$$= 0.0638547 \text{ for p=16 N=1200}$$

$$= 0.0413505 \text{ for p=25 N=1200}$$

Even when I changed my calculations to the model I have implemented there wasn't a big difference in the calculations for N 1200 and p being 4 it should have been 4 times more efficient yet it was only 1.6 times efficient.(the speedup increased to 2.5 times when we increased the matrix dimensions to be 4800 which support Gustaffson's law)

$$\frac{\frac{2N^3-N^2}{P} + (3p + 2\sqrt{p} - 2)(t_s + \frac{N^2}{p}t_m)}{2N^3 - N^2} =$$

$$\frac{434340007}{1727280000} = 0.2514589 \text{ for p=4 N=1200}$$

$$= 0.1125469 \text{ for p=9 N=1200}$$

$$= 0.0639068 \text{ for p=16 N=1200}$$

$$= 0.0413839 \text{ for p=25 N=1200}$$

4 Conclusion

Since the best known sequential algorithm for matrix multiplication has a time complexity of O($N^{2.3728596}$) with enough processors Cannon's algorithm can be a great method for square matrices.

5 How to use the code

There are 2 c files in the src directory **serial.c** and **solution.c**. **solution.c** has the parallel algorithm while **serial.c** has the sequential one.

To use the codes use **script.sh** inside the src directory.It works in this order:

- It will compile **solution.c**(with 4 processors u can change it by going inside script.sh) and **serial.c**
- Then it will start parallel algorithm

  Parallel algorithm will first create A and B matrices

  Then copy them inside **input** file(for sequential algorithm's use)

  At this point it starts Parallel timer

  After calculating the solutions it will first write the time it took to calculate C matrix in seconds.

  After that it will record the C matrix inside the **output** file
- Then it will run sequential algorithm with the inputs from parallel algorithm

  Sequential algorithm will start the timer after getting inputs and stop the timer and print it after calculation is done

  after printing the timer sequential algorithm will save the C matrix inside **output2** file.
- then the script will call diff command with output and output2

Important Note: Before using them make sure serial. **solution.c** and **serial.c** both has the same defined value for N or they will produce different results because of their different matrix sizes.

References

[1]https://en.wikipedia.org/wiki/Matrix_multiplication_algorithm
[2]https://en.wikipedia.org/wiki/Cannon%27s_algorithm
[3]https://en.wikipedia.org/wiki/Galactic_algorithm
[4]https://hpc-tutorials.llnl.gov/mpi/collective_communication_routines/