AMS 530: Parallel Computing **Name:** *Sophia Nolas*
Final Project
11/28/22

Problem 4.4

- Project description.

  In this project, I am asked to use the Strassen's Algorithm to multiply two large matrices, A and B. The Strassen's algorithm is a way to make multiplying two large matrices more efficient. When multiplying two matrices using the naive method, we will need to use 8 matrix multiplications of size N/2, which, when N is large, can be really time and operation consuming. Strassen's algorithm gives us a way to cut out one of these multiplications, for a total cost of only 7 multiplications and 18 matrix additions.

  This is achieved by dividing the large matrix into four quadrants; then performing 10 matrix sums of these sub-matrices of size N/2; then doing 7 products on these sums and quadrants.

  Then, to reconstitute the solution matrix C, we perform a series of simple matrix sums on these products.

  This essentially saves us one costly matrix multiplication. And the really exciting part is that we can then further split each of these products into another set of quadrants, and then do the Strassen algorithm again recursively, saving us more matrix multiplications. So the second level of the algorithm would require $7\hat{2}$=49 matrix multiplications; a reduction from $8\hat{2}$=64 in the naive method. And once again we can split each of those products into Strassen algorithm method for solving, giving us $7\hat{3}$=343, a reduction from $8\hat{3}$=512 matrix multiplications. So the further this algorithm is applied, the matrix multiplications required both reduce in quantity and in size, which can really save us time and operation cost.

  Now the question becomes how to parallelize this process. A clear way to do this is to use 7 processors, and divide the 7 matrix products among them. We can further save time by using 7*p processors, where p¿=1. Not only does this divide up the work and save time from the single processor implementation, but because of the way the sums and products partition the data, we can save on communication costs by only sending the data to the processor that will be using it.

- Algorithm description and pseudo-code describing the main structure of your programs. Source code with compile- and run-time options to enable grading tests.

  In naive multiplication, we will need to perform 8 matrix multiplications; if the full solution matrix is divided into quadrants C_11, C_12, C_21, and C_22, then these are:

  C_11 = A_11 * B_11 + A_12 * B_21

  C_12 = A_11 * B_12 + A_12 * B_22

  C_21 = A_21 * B_11 + A_22 * B_21

  C_22 = A_21 * B_12 + A_22 * B_22

  When writing the code to implement this algorithm on one processor, the first level is simply the process of computing the 10 sums, then the 7 products, and then combining

them back together elementwise to create the final matrix. The sums and products, as well as the final sums, are the following:

For the simple case of a 8x8 matrix, I visualized the subdivisions like this:

A11	A12
A21	A22

A11	A12	A13	A14
A21	A22	A23	A24
A31	A32	A33	A34
A41	A42	A43	A44

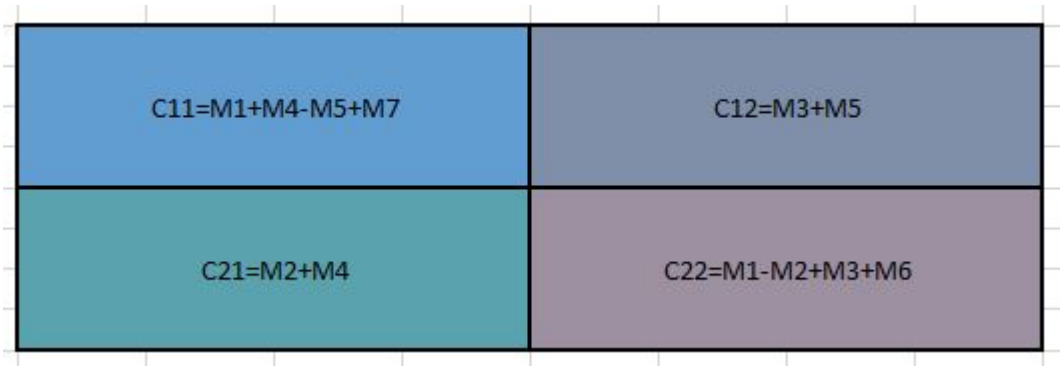
A11	A12	A13	A14	A15	A16	A17	A18
A21	A22	A23	A24	A25	A26	A27	A28
A31	A32	A33	A34	A35	A36	A37	A38
A41	A42	A43	A44	A45	A46	A47	A48
A51	A52	A53	A54	A55	A56	A57	A58
A61	A62	A63	A64	A65	A66	A67	A68
A71	A72	A73	A74	A75	A76	A77	A78
A81	A82	A83	A84	A85	A86	A87	A88

Then the sums and products are:

- S_1 = A_11 + A_22
- S_2 = B_11 + B_22
- S_3 = A_21 + A_22
- S_4 = B_12 - B_22
- S_5 = B_21 - B_11
- S_6 = A_11 + A_12
- S_7 = A_21 - A_11
- S_8 = B_11 + B_12

- S_9 = A_12 - A_22
- S_10 = B_12 + B_22
- P_1 = S_1 * S_2
- P_2 = S_3 * B_11
- P_3 = A_11 * S_4
- P_4 = A_22 * S_5
- P_5 = S_6 * B_22
- P_6 = S_7 * S_8
- P_7 = S_9 * S_10
- T_1 = P_1 + P_4
- T_2 = P_7 - P_5
- T_3 = P_1 + P_3
- T_4 = P_6 - P_2
- C_11 = T_1 + T_2
- C_12 = P_3 + P_5
- C_21 = P_2 + P_4
- C_22 = T_3 + T_4

In other words, the way we divide up the matrices gives us 10 sums and 7 products; then these are combined back together to create the solution matrix. I visualize this like this:



Separately, just to be safe, I wrote the function to perform the naive matrix multiplication, so that I could compare solutions to make sure the algorithm really works (it does!).

As mentioned above, the parallelizing of this algorithm really lets us save on time and operation cost, as well as the way the algorithm is partitioned lets us save on communication costs between processors. Every processor will not need the data from the full matrix; rather, each will only need the sub section of the matrix, as well as the sub section in the sum (or sums). So in my implementation as I take this from a single processor algorithm to parallel, I will stipulate via the MPI rank of each processor which one should do which

sub-matrix multiplication. Then at the end, I will use MPI_Send to send each of these products to the root processor; and MPI_Recieve to take in the information at the root; so that I can combine them back together into the solution matrix C. That way each processor only has to do the one product, and then only has to store the information from that product.

For the first level of Strassen's Algorithm, all I need is a function which performs naive matrix multiplication; then I will simply write the code to compute the matrix sums, call this function on each pair of matrices to generate the products, and then add these back up, by quadrant piecing together the solution matrix C.

Taking the algorithm to the second level is straightforward as well. I will simply need to write a function that, rather than performing a naive matrix multiplication, will perform the Strassen's algorithm for matrix multiplication, and call it for each product in the previous step. The function will take as arguments the two parts of the product, and return the relevant product. Then once I have these, I will as in the first level, sum them back up to create the four quadrants of C.

And I will take the same tactic for level 3 of the algorithm; write another function that takes the even smaller arrays as arguments, and returns the product using the Strassen algorithm. But for the products in this function, I will once again return to the naive multiplication algorithm.

Unfortunately it is at this point that my progress on this project was limited by my knowledge of C++, which I am currently still learning, and so while I was able to implement the first level of the Algorithm quickly, the second and third level were accomplished without the level of elegance I would have liked. However, the algorithm itself is quite straightforward. What I would have liked to do was basically write a function to perform the Strassen's algorithm, inside of the Strassen algorithm function from level two. What I needed for this was to figure out a way to call a function recursively from inside itself.

The other aspect that was limited by my knowledge of programming was using multiple processors. My conception was to label each processor by rank, and then send the products to them 1 through 7 in modulus; that is, I'd send the same part of the work to 1 and 8, and split the tasks between them. This would become even more useful as N becomes larger; however I was not able to implement it here for this project.

The "gap" in my knowledge of C++ which caused the most trouble is in passing matrices to functions; it is simple to pass a matrix of known size of course, but passing a matrix of variable size requires some finesse using pointers and dynamic memory that it would be possible for me to learn but not in the time I have. For this reason, instead of writing just two functions (a naive multiplication function, and a Strassen's algorithm multiplication function), I managed to get what I needed done by writing functions that took matrices in decreasing size. Basically, since I knew explicitly what sizes I would be using, I just wrote the functions with that in mind; a really successful code, I believe, would allow for dynamic sizes of the arrays. Then I would only need the two functions.

In theory, we could go even further than level 3 here, and just reduce the matrix multiplications recursively in size by halves until we are simply multiplying individual elements. Of course at that point we'd have to do some work to decide whether the cost of calling

the same function, or in my code smaller and smaller functions, would be worth the time we save from using the naive matrix multiplication. Perhaps it would be more effective to simply reduce the size of the matrices down to a certain level where even the naive multiplication takes trivial time. Either way, there is more work to be done here which would save computational time for a wide variety of applications.

Another thing I'd like to do is to basically use the same code, but fill the matrix with string characters representing the elements, and then perform this algorithm in a way that will show us as a result which original elements go where and how they line up in the final matrix. Since I am using integer elements, we get a better idea of how the timing would work out, and arrive at the solution, but I think for data and algorithmic study it would be useful to see where each element is used and to which processor it needs to be sent to and from.

- Results (numbers, figures, and tables).
- Analysis of program performance and your other answers to specific questions given by the problem set.

1. Design the algorithms and implement them on your computer.

   Done! And I also computed the solutions the naive method to make sure the ones with my method were accurate. I used matrices A (all ones) and B (all twos). I'm attaching the files, ".cpp" and ".sh", to this submission, please let me know if there is any trouble downloading them!

2. Test the performance using on $7p$ with $p = 1, 2, 3, 4$ for matrix of size $N = 2^n$ with $n = 8, 10, 12$

   I used the timer MPI_Wtime() to collect time information for my tests with P=7, and clock() to collect time information on the case of P=1.

processors	matrix size	clock time	email time
7p	N		
1	256	140768	
7	256	145685291	0:00:02
7	1024	562125320	0:00:03
7	4096	1698745465	0:00:47

   The results ended up a little confusing unfortunately, but I also got runtime data from the Seawulf END emails which were sent when my programs were completed. All of these data are in the above table.

3. Collect the performance results and analyze them.
4. Plot the speedup curves.

   Since I was only able to use P=1 and P=7, my speedup results are less than illuminating. However, as we can see there is a dramatic change between the two processor counts.

# of processors	time	speedup
1	140768	N/A
7	145685291	0.000966247

5. Comment on your performance results.

   Of course it makes sense that the program would take longer to run when we increase the size of N; and this is exactly why an algorithm like this would be so useful, as we increase the size of N, the time cost also increases; so being able to decrease the amount of matrix multiplications, and size of the matrices, will be very useful.

- Further work to improve the study.

  To go further with this study, what I'd like to do is find a way to call the function recursively in itself, so that I could do as many levels of the algorithm as wanted. Barring that, I'd like to write the functions without explicitly stating the size of the matrices, so we can use it for matrices of any size (that is a power of 2, that is). Additionally, it might be useful to combine this algorithm with another matrix product algorithm, such as Cannon's Algorithm or BMR method, to save time on the matrix multiplications (which we still have to do 7 of per step; maybe we could save time by using another matrix multiplication method other than the naive one).

Programming Files Attached to this submission:

For clarity, I'm attaching the following files to this submission, each case of N has one .cpp file and one .sh batch file. They're labeled as follows:

1. Case: N=256

   twobatch.sh

   strassenNtwo.cpp

2. Case: N=1024

   tenbatch2.sh

   strassenNten.cpp

3. Case: N=4096

   forbatch.sh

   strassenNforty.cpp

References:

1. AMS 530 Lecture Notes
2. "Introduction to Parallel Computing" Grama, Gupta, Karypis, Kumar
3. "Parallelizing Strassen's Method for Matrix Multiplication on Distributed - Memory MIMD Architectures"