Distributed Matrix Multiplication

Ryan Jones

2019 - 10 - 14

For this project we were tasked with performing matrix multiplication on 2 matrices using `MPI` to distribute the computation. There were three main parts to solving this problem:

1. Partitioning the matrix data
2. Distributing the matrix data to the various processors with `MPI`
3. Combining the computed data back into a final "result" matrix

All code used for testing can be found on my GitHub: github.com/LeftShiftJones.

1 Partitioning

For this project I decided to partition the A and B matrices along A's rows and B's columns. I chose this strategy because it afforded me the ability to keep the same number of rows of the completion, *'C'* matrix, which I ultimately used to write the results of the multiplication to disk.

The methodology of this partitioning is that given P cores with which to compute the matrix multiplication, the number of rows which will be completed of A (and therefore C) would be approximately equal to $\frac{A_{Rows}P}{}$. I partitioned B along its columns, transposing the matrix to ensure better cache performance, so that each processor got $\frac{B_{Cols}}{P}$

Because I was planning on running my program on 1, 2, 4, and 8 cores for testing, my implementation has only been configured to run on those number of processors. Ideally, all configurations would be run on matrix sizes that are evenly divisible by 8 to handle the load effectively. I generated my matrices using Dr. Nurkkala's in-class example code. If only one processor is specified to be run, then it will only run on that processor.

2 MPI Usage

To distribute the matrices, I used the Message Passing Interface (MPI) to partition the matrices: A by rows and B by columns. The matrices were loaded by the program `generatematrices.c`, where processor 0 takes the read data and uses a loop to iterate over the data at certain points where the other processors would start, were this a shared-memory system. I used an `MPI_Send` and `MPI_Recv` to transfer this data back and forth.

In the matrix multiplication method itself, I created a new `int *` for the partition of the result matrix based on the number of rows from A and the number of columns in B (though in my implementation I transposed B when reading it in so that I could have more efficient use of the cache). After computing the values for C that I could with my data, I had each processor send its B "columns" data to its adjacent processor and repeat the process. After the loop had run one less time than the number of processors, all the data from B had been passed around, each processor that wasn't 0 sent its C matrix to processor 0, which then received the data and wrote them to disk.

My program also has a debug feature which notifies the user where in the program it is currently executing, and writes the sequential solution matrix to an output file using just the triple-nested (nondistributed) for-loop solution.

3 Data

For all tests, I had my program generate two matrices with values for `m, n,` and `p` that were all equal.

Table 1: Data for 512x512 matrices on multiple cores

Matrix	p	t_p	s	e
512x512	1	0.801	1.0	1.0
	2	0.413	1.94	0.97
	4	0.219	3.66	0.92
	8	0.217	3.69	0.46

Graph 1: 512^2 Matrices

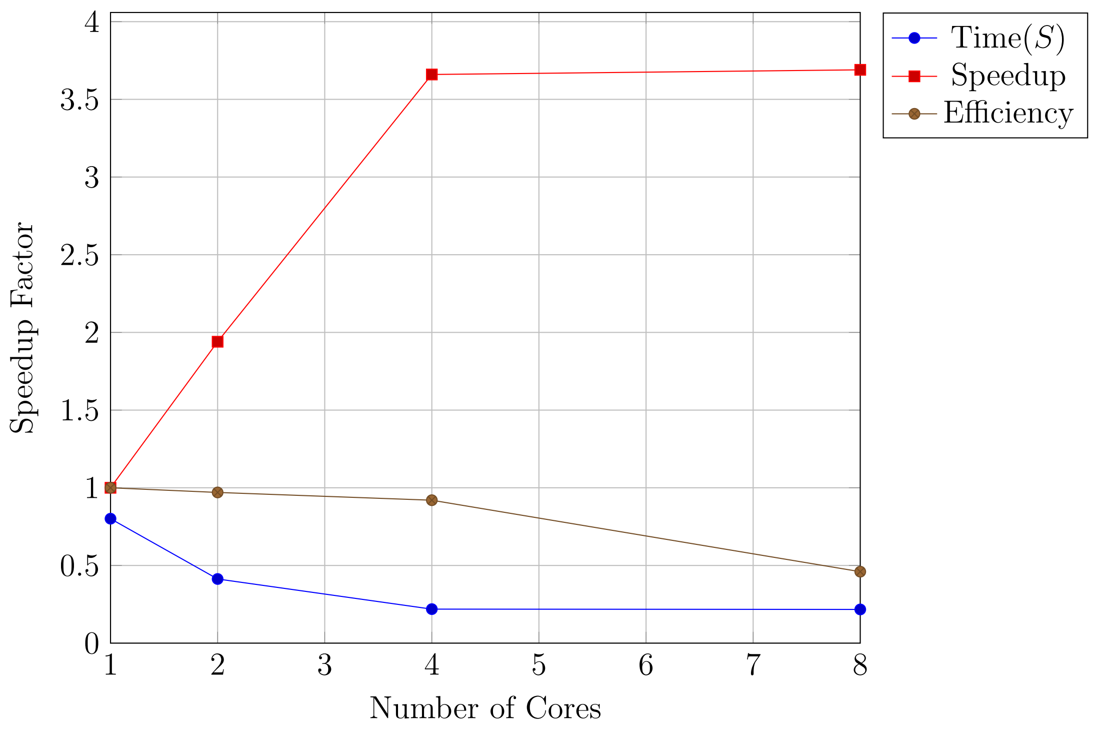


Table 2: Data for 2048x2048 matrices on multiple cores

Image	p	t_p	s	e
2048x2048	1	51.192	1.0	1.0
	2	26.271	1.95	0.97
	4	13.579	3.77	0.94
	8	12.656	4.04	0.51

Graph 2: 2048^2 Matrices

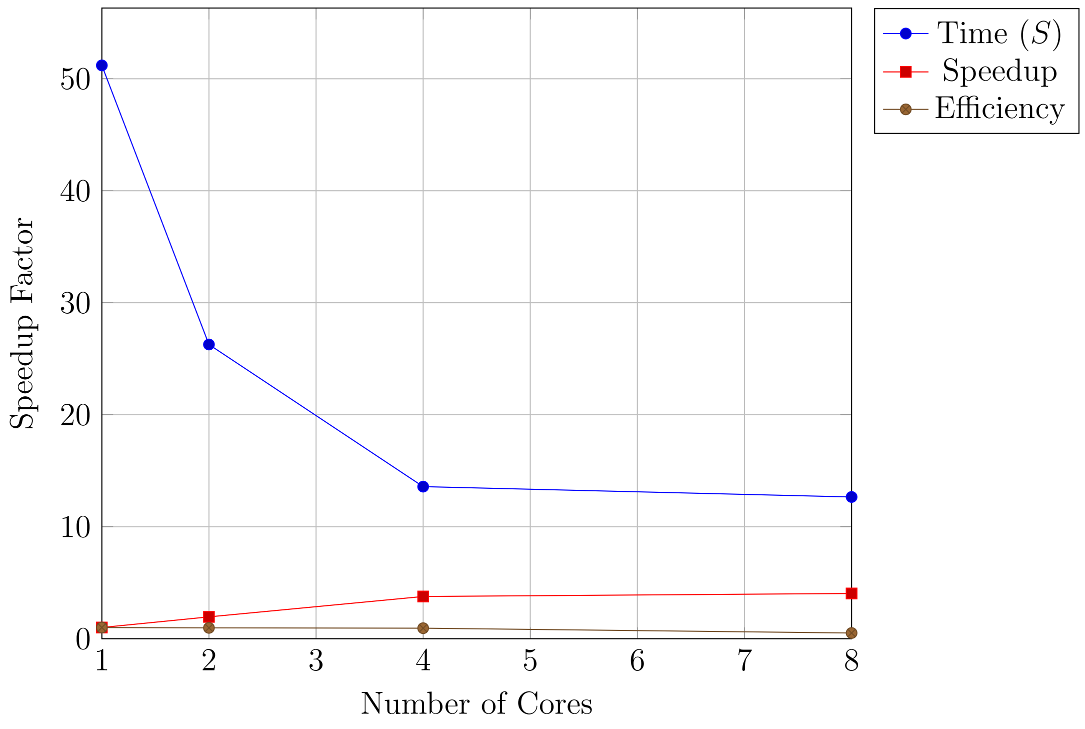
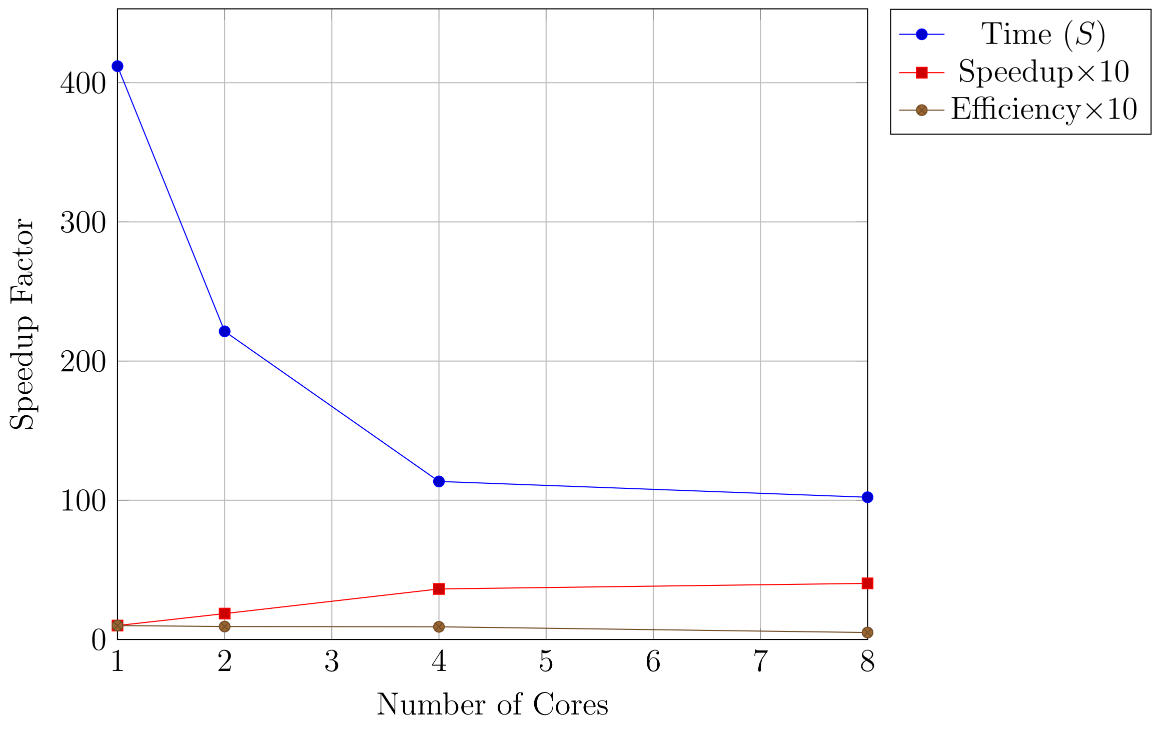


Table 3: Data for 4096x4096 matrices on multiple cores

Image	p	t_p	s	e
4096x4096	1	411.861	1.0	1.0
	2	221.352	1.86	0.93
	4	113.522	3.63	0.91
	8	102.188	4.03	0.50

Graph 3: $409x^2$ matrix processing



4 Interpretation of Data and Reflection

The data I received from my tests was very similar to other testing I've done for this class: running times tend to halve for each doubling of the number of processors used. The performance for the program is on track without any noticeable flaws in the execution or the performance.

In terms of difficulty, this project stretched me to my limits computationally and time-wise. One of the most difficult aspects was figuring out what to send and in what length. At first, I had all of the processors reading in their respective data and using only that, but that added to computation time and memory usage. I tried following Dr. Nurkkala's suggestion about sending structs via `MPI` (my final implementation involved storing a struct that contained 2 integer pointers), but in my experience despite repeated attempts this corrupted my data, so I ended up sending the integer pointers themselves. The majority of the project was little things like that which came up every time I thought I had a working solution.