Athit Vue

CSCI 551

Report on The Performance of Matrix Multiplication Using MPI

My approach in partitioning the work and data among the processors is as follow. First, I used MPI_Scatter(...) to scatter all the rows of the matrix A to all the processors. I created a variable in rank 0 that would handle the input (data) of this matrix from the user. A local variable for each rank (processor) was also created within each rank to store the partial data of matrix A that it will receive from rank 0. Once the data for matrix A was distributed, I decided to use MPI_Bcast(…) to send all the whole matrix B to all the processors. The processors will then do their partial dot product calculation and store their results to a dynamically allocated local variable. Then to gather the data, MPI_Gather() is called. This will gather all of the partial results from each processors and store it in a local variable in rank 0. This variable is the resulting matrix C.

It worked just fine, however, after testing it with different number of processors and sizes, there was one question in mind: what about for the special case when the number of rows is not divisible by the number of processors? The scatter function does not provide such data partition that would allow me to send irregular size of data to all processors. This is when I discovered MPI_Scatterv(). This function allowed me to do such. It would also allow me to define the stride. However, the importance in this function is that it allowed me to give a processor an extra row of data for matrix A, if the number of rows was not divisible by the number of processor.

After changing MPI_Scatter(…) to MPI_Scatterv(…) and testing the extra row that a processor would have to take on, it seems to not have worked. I kept on getting the wrong resulting row after the first one. These numbers were big! This lead me to think that the numbers in that placeholder are just a bunch of garbage. After troubleshooting, I realized that I needed to zero all the dynamically allocated variables. Using memset(…) would cost me a runtime of o(n) which would add to the overhead, but I needed to call memset(…) to get the correct resulting matrix. I felt that this was the more efficient way than having too many send and receive call. Calling send() and recv() was what I had in mind at first but decided that these numerous number of calls can create a lot of overhead and penalize our performance a lot, especially when as we increase the number of processors. In this case, partitioning with a collective routine works better.

Tables for Run Times (Random 6000x6000 matrices):

IJK FORM

P	Run_1 (secs)	Run_2 (secs)	Run_3 (secs)
1	2386.122	2356.133	2540.233
4	1260.322	1250.300	1294.805
8	860.1566	859.9554	861.5501
12	768.2879	767.5123	775.5590
16	590.1504	590.1599	592.6604
20	465.5571	469.8891	472.9901

IKJ FORM

P	Run_1 (secs)	Run_2 (secs)	Run_3 (secs)
1	2180.150	2260.160	2160.957
4	1150.395	1154.995	1160.785
8	785.1550	789.4852	782.1145
12	645.9958	646.8520	642.1120
16	559.8625	558.9952	561.9547
20	459.2278	460.4156	458.0024

KIJ FORM

P	Run_1 (secs)	Run_2 (secs)	Run_3 (secs)
1	2250.848	2013.624	2169.336
4	1159.696	1158.8869	1161.9952
8	856.4558	857.2454	958.5047
12	603.4456	604.8526	603.1458
16	590.9546	585.6971	586.7456
20	431.8859	430.8752	432.5541

Speedup and Efficiency:

IJK FORM: Minimum Time Run

P	Elapse Time (secs)	Speedup	Efficiency
1	2356.133	1	1
4	1250.300	1.84454	0.461135
8	859.9554	2.73983	0.342479
12	767.5123	3.06983	0.255819
16	590.1504	3.99243	0.249527
20	465.5571	5.06089	0.2530445

IKJ FORM

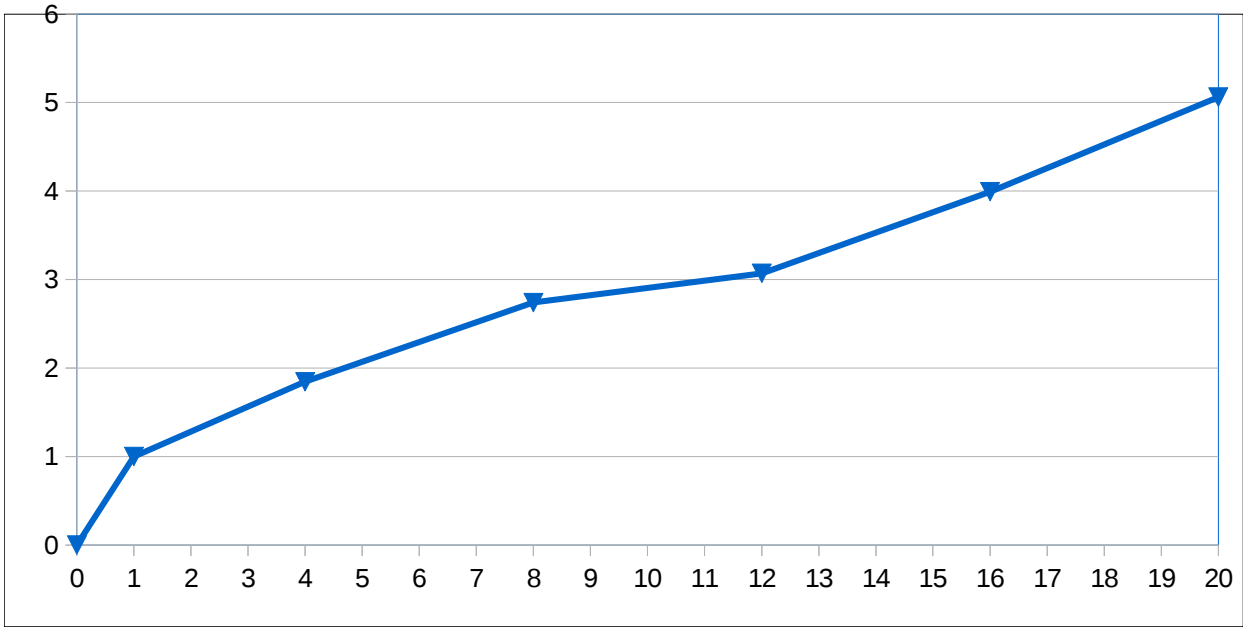
P	Elapse Time (secs)	Speedup	Efficiency
1	2160.957	1	1
4	1154.995	1.870967	0.467742
8	782.1145	2.762961	0.34537
12	642.1120	3.365390	0.28045
16	558.9952	3.865788	0.24161
20	458.0024	4.718222	0.23591

KIJ FORM

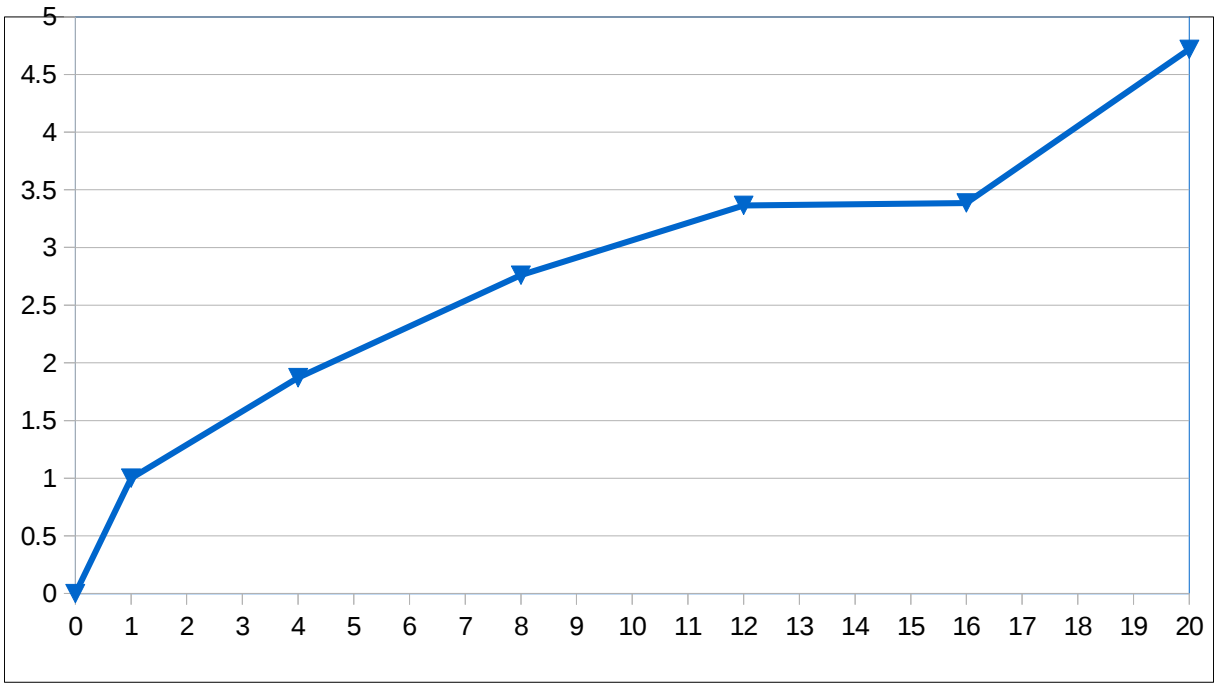
P	Elapse Time (secs)	Speedup	Efficiency
1	2013.624	1	1
4	1158.8869	1.737550	0.43439
8	856.4558	2.351113	0.29389
12	603.1458	3.338536	0.27821
16	585.6971	3.438000	0.214875
20	430.8752	4.673335	0.233667

Graphs of Speedup:

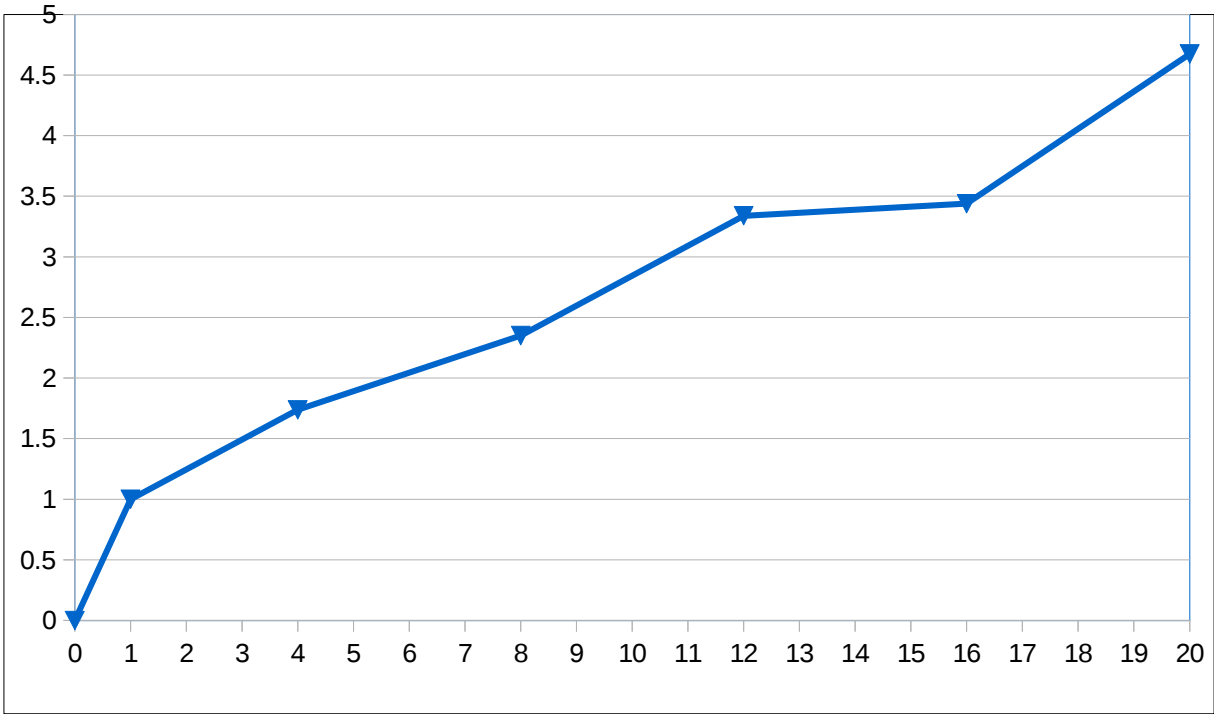
IJK FORM



IKJ FORM

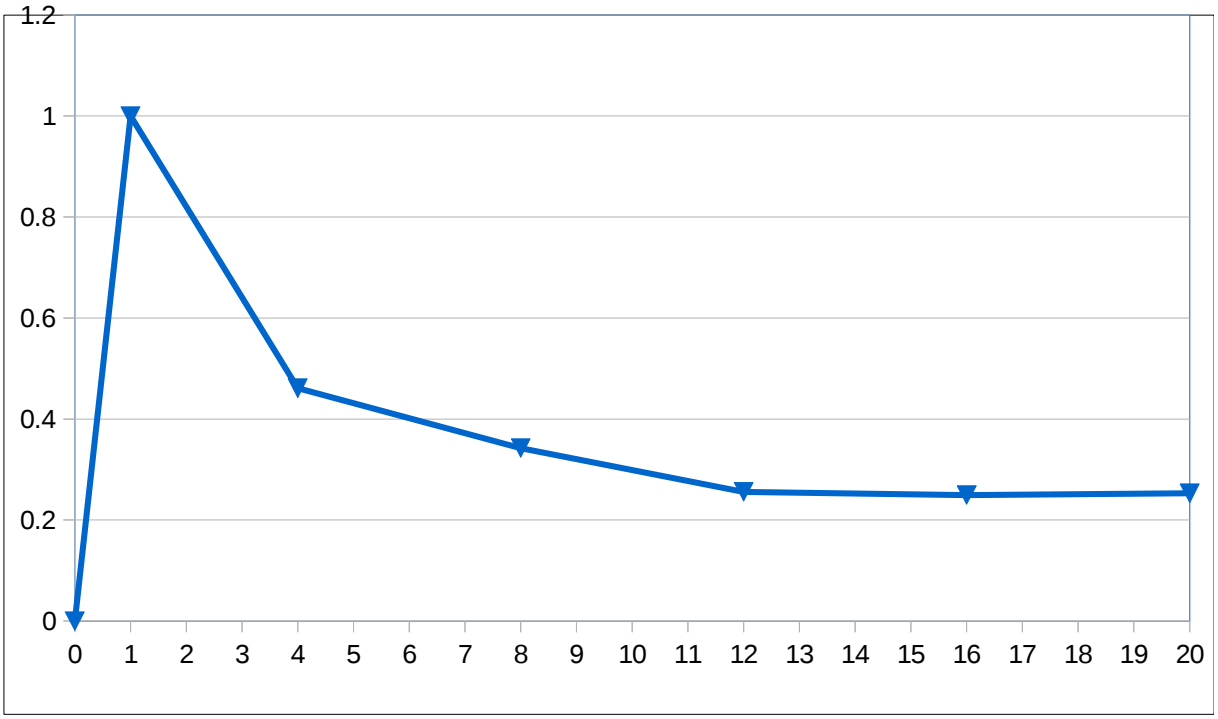


KIJ FORM

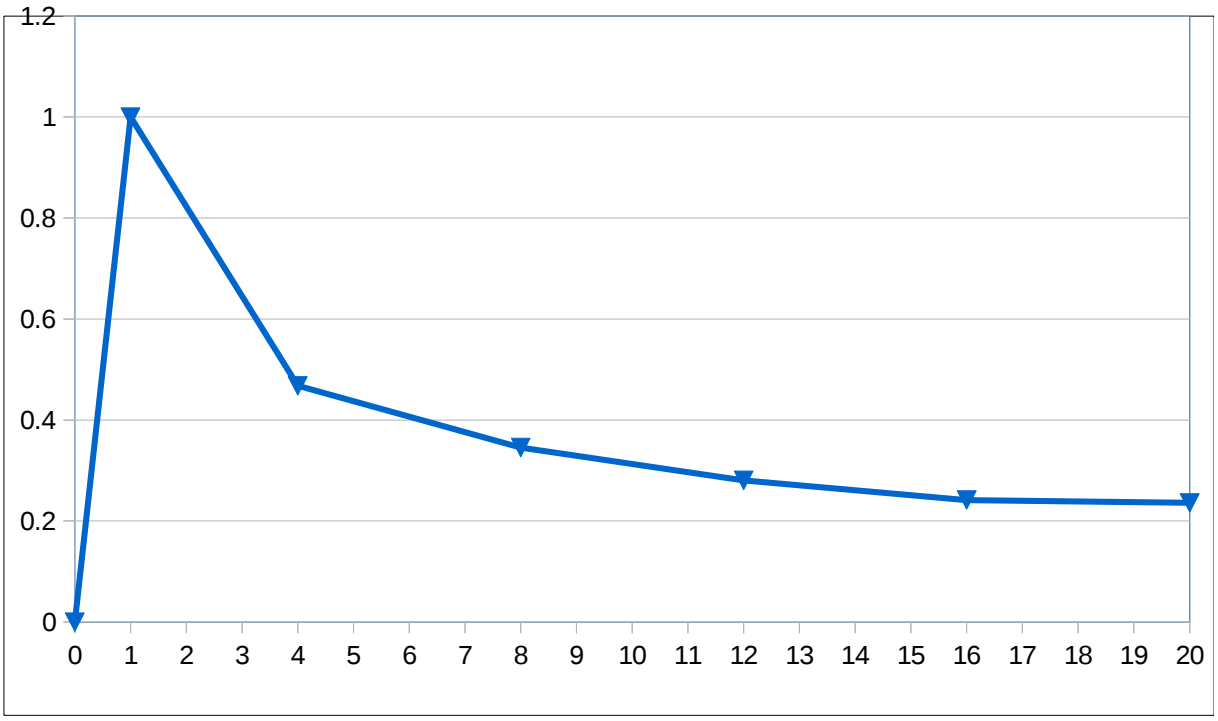


Graphs of Efficiency:

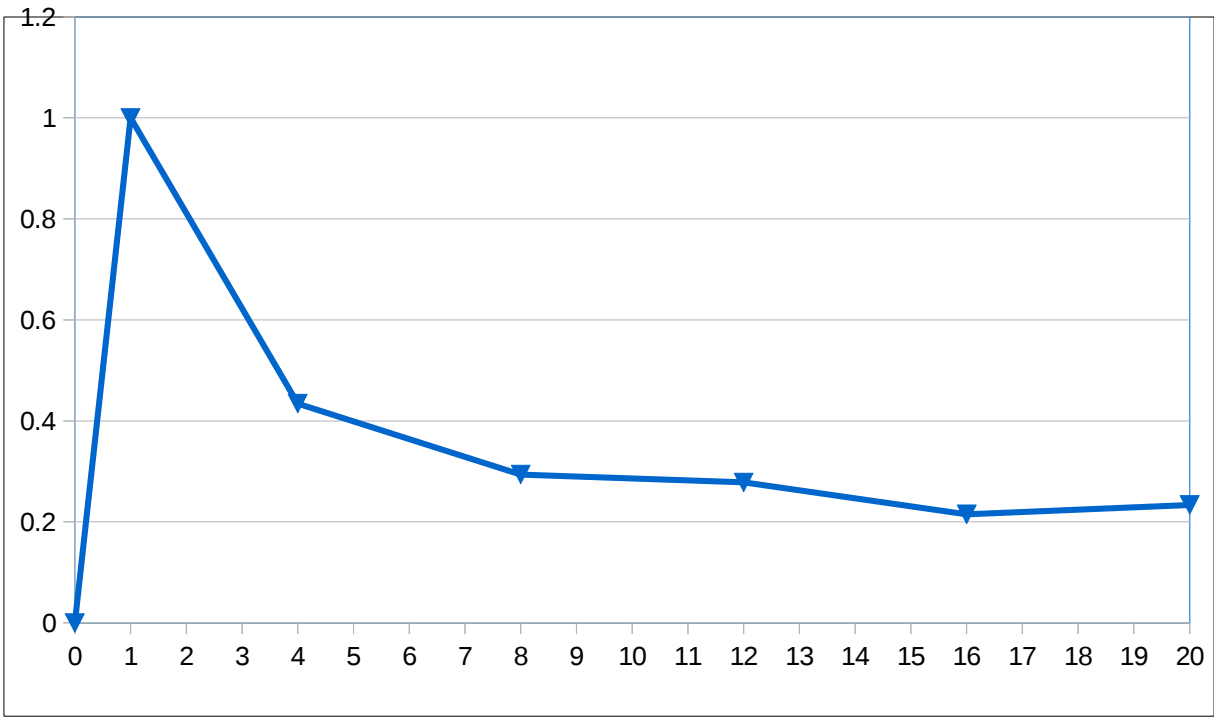
IJK FORM



IKJ FORM



KIJ FORM



Data Analysis:

Looking at the graphs, the speed up, and efficiency, it's a little bit hard to tell which one is the most optimum to use due to the lack of data consistency within each form. I spent a whole day on testing the data and waiting for the results to come back, yet consistency is a problem. I think this lack of consistency is manly due to the processors handling someone else's computing, while the program is being executed.

However, we can kind of draw the line where the data would be if there are no outliers. According to the data that I have collected, I would say that IKJ form is the most optimum. I had to use a lot of calls to memset() which also explains all the overhead that is associated with the elapsed time. I disregarded implementing point-to-point communication and decided to use collective communication. I am glad I did, because the overhead of sending and receiving will cause a lot of performance issue getting into the higher up number of processes.

With ijk we are utilizing the row major order. The i'th row on the first matrix get's reused while currently being operated on until all the columns have been used. With ikj, one element in the first matrix gets used n times, utilizing the major row order again, until moving on to the next element. This helps the program utilize the cache greatly and thus this may explain why ikj seems to be the most optimum choice (in regards to my data). With kij, the elements in the first matrix is used one after another before moving on to another element as well, again this helps with cache. However, the next element is in the next row, thus the cache is not used for long.

I have observe that my efficiency seems to be dropping as number of processors increased. This may be due to the overhead of allocating and deallocating along with Bcast and Gather. It seems the more processors we add to it the more the efficiency drops.

Conclusion:

Due to having test my program at a busy time of the lab machines, or it seems, my program took a really long time for it to finish executing. I was surprised that it took that long to return. Also, I was surprise at the results of the runtime when more processors were added. I was expecting a good near constant efficiency rate at least 60 percent. I got a fluctuating efficiency rate at almost 10 percent for every other increase in processors. This leads to me think again that maybe I have created too much overheard somewhere in my program, that penalizes me when adding another process.

My method to scatter matrix A across all processors and have matrix B sent fully to all processors proved to be useful. However, the associated overhead that was needed in order to make the program run seems to have cost me expensively on time.