Matrix Multiplication ijk Forms with Open MPI
Lab #2

Jason Boutte

April 20, 2015

1. Partitioning

ijk form

For the ijk form I evenly distributed the rows from matrix A to each process and broadcasted matrix B. Each element of the resultant matrix can be calculated with the row from matrix A and the column of matrix B that contain the element. Since an entire row is needed to calculate its elements results it made sense to partition by rows. Each element in the rows also needs its corresponding column from matrix B, thus requiring us to broadcast all of matrix B.

ikj form

For the ikj form I evenly distributed the rows from matrix A to each process and broadcasted matrix B. Each element of the resultant matrix can be calculated as long as the elements entire row from matrix A and the entire matrix B are present. Since each row is needed to calculated its elements result, it made the most sense to partition by the rows of matrix A. In order to find the result of each element in a row, their corresponding rows in matrix B are needed so the entire matrix B is broadcasted.

kij form

For the kij form I evenly distributed the columns from matrix A and the related rows from matrix B. Each element of the resultant matrix can be calculated by multiplying each value in the column of the matrix A by each value in the corresponding row of matrix B. Thus each column in matrix A will create an entire new matrix that must be summed to find the result. Since an entire column and row are needed it made sense to partition by the columns of matrix A and their related rows of matrix B, in order to accomplish this matrix A needed to be transposed before being partitioned.

2. Timings

a. ijk

Run	Cores					
	1	4	8	12	16	20
1	824.19	219.17	117.44	79.77	60.71	49.43
2	823.62	221.35	117.88	79.89	60.19	49.38
3	823.79	227.99	117.61	79.46	60.87	51.43
4	823.84	222.38	118.57	80.10	60.87	49.28
5	823.96	219.48	118.75	79.98	60.75	51.76

b. ikj

Run	Cores					
	1	4	8	12	16	20
1	774.60	196.57	99.85	67.55	51.62	41.79
2	775.21	196.68	99.61	67.51	51.51	41.84
3	774.64	196.05	99.57	67.58	51.48	42.31
4	774.33	196.39	99.54	67.54	53.00	41.70
5	775.11	196.46	99.50	67.67	51.38	41.78

c. kij

Run	Cores					
	1	4	8	12	16	20
1	777.11	225.75	154.56	127.49	117.93	116.39
2	777.13	225.79	154.22	126.42	119.44	115.25
3	777.96	224.47	158.46	127.16	174.51	113.71
4	777.70	225.76	158.95	127.09	138.14	115.23
5	776.70	226.13	160.18	127.45	118.46	116.06

3. Speedup and Efficiency

a.

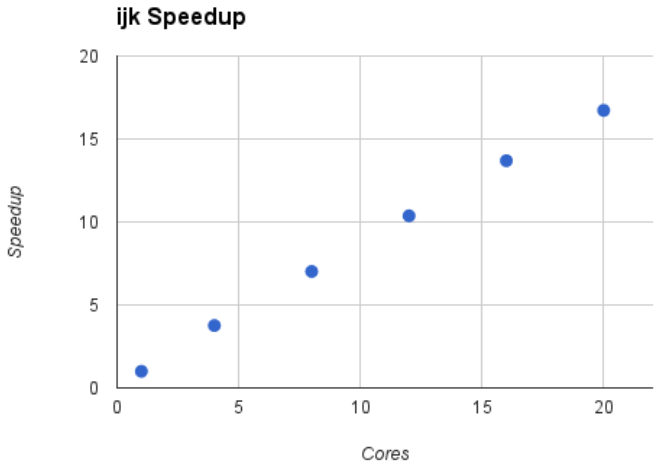
Cores	**Speedup**		
	ijk	ikj	kij
1	1	1	1
4	3.76	3.95	3.46
8	7.02	7.79	5.04
12	10.37	11.48	6.14
16	13.69	15.09	6.59
20	16.72	18.59	6.83

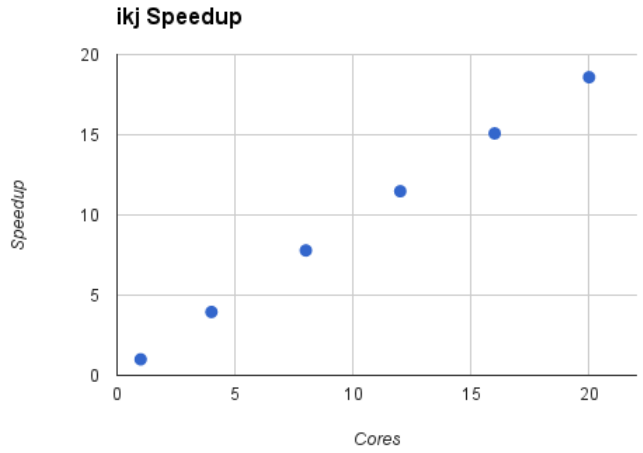
b.

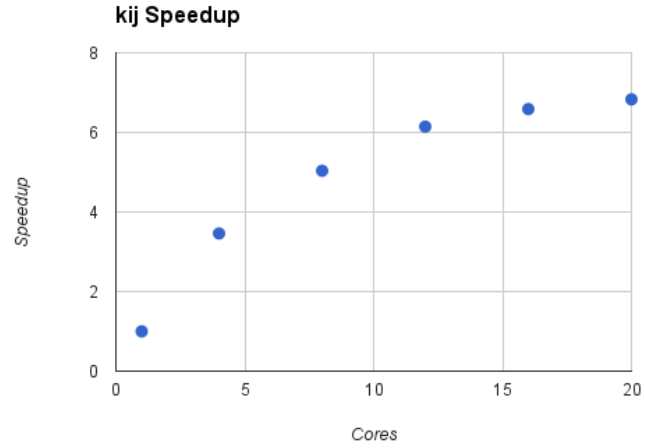
Cores	Efficiency		
	ijk	ikj	kij
1	1	1	1
4	0.94	0.99	0.87
8	0.88	0.97	0.63
12	0.86	0.96	0.51
16	0.86	0.94	0.41
20	0.84	0.93	0.34

4. Graphs

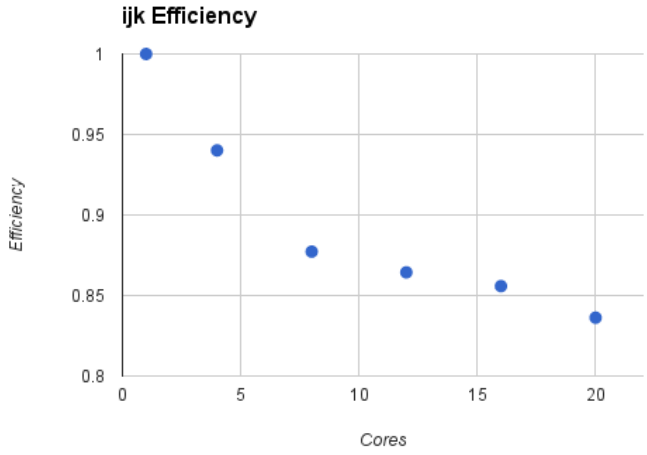
a. Speedup

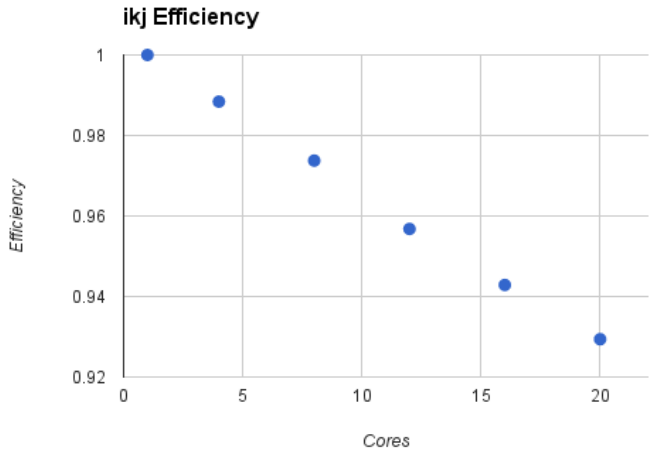


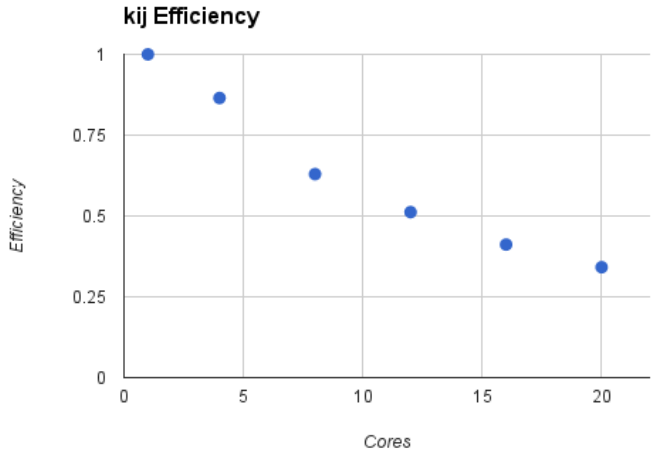




b. Efficiency







5. Conclusion

Out of the 3 ijk forms ikj turned out to be the fastest. At first I wasn't sure why this was because I partitioned ijk and ikj the same. After looking at the algorithm I discovered why, in the most inner loop of the ikj algorithm one of the values from the matrices doesn't change. If this value that doesn't change and gets assigned outside of the loop then the OS will have less cache misses and improve the performance when processing large data sets. The worst performance was from the kij form, and this mostly makes sense. While we originally transferred less data to each process by the end we're sending a full matrix from each process. To restore some of the performance rather than sending the matrices and summing them on the root process, I utilized the MPI_Reduce function on each element of the resultant matrix.