Yaseen Aniss

Matrix Multiplication ijk Forms with Open MPI

I used the same method of partitioning for each of the three forms. My initial method that I started to implement was to broadcast the A matrix to each processor and then scatter portions of the B matrix to each process, and then from there I can do the partial matrix multiplication on each process and send the results back to process 0 with a gather command. When I started implementing this method, I realized that it would be very complicated to scatter columns of B to each processor (unless I read it in by column major). So, I changed my method of partitioning and decided I would broadcast all of B to each process and then scatter rows of A to each process. This was a simpler method because if the programmer follows the scatter command documentation, it is easy to send contiguous array values (rows) without any extra work that the programmer needs to do (such as reading in a matrix in column major order). Here are the steps for how my program works:

1.) Create a derived data type for the “Details” object that holds the matrix form (ijk, kij, ikj), the flag (I, R) and the n value.
2.) If my_rank is 0, then read in the matrix form, the flag and the n value into the details object…then read in the two matrices in row major order. Next, start the timer.
3.) Broadcast the initialized details object from processor 0 to all the processors in the comm world.
4.) Now that every processor has the value of n (through the details object), they use this value to allocate the appropriate amount of space for their matrices.
5.) Scatter ((N*N) / comm_sz) matrix A values from processor 0 to all the other processors.
6.) Broadcast matrix B values from processor 0 to all the other processors.
7.) Calculate local matrix multiplications in each processor.
8.) Gather the final matrix by sending to processor 0.
9.) If my_rank == 0, stop the timer.

This method ended up being successful in reducing the timing of the work as the number of processors increased. After I did my runs using this approach, I realized something interesting. I found out that the “IJK” form was the slowest. When I began to look at the for loop for this form, I realized that at the innermost loop, we are accessing values in matrix B by column. This is important because this means that each index into the matrix is 4800 slots away from the previous index. This causes many cache misses and wasted clock cycles. I was able to notice this fairly easily because I chose to index my matrices using multiplication instead of brackets (which helped me easily visualize what the CPU was really doing). After I came to this realization, I soon came to an epiphany…that if I was to read in Matrix B via column major order and change my innermost for loop for IJK to access B by using this line of code:
“(*(B + (j * n) + k))”, I would get the best possible cache performance out of any of the other forms because I would be looping through by row in all 3 matrices (the resulting matrix that we are writing to (C), the first matrix (A) , and the second matrix (B)).

(*(C + (i * n) + j)) += (*(A + (i*n) + k)) * (*(B + (j * n) + k));

The fastest method with my current implementation was the IKJ form. This form as well as the KIJ form are very similar in cache efficiency for the right hand side of their innermost loop, but the left hand side for IKJ is more efficient because it only indexes a new column in the resulting matrix (C) N times while the KIJ matrix does this N^2 times. This operation is inefficient for the caches and that's why IKJ was a tad bit faster.

I also began to think about how I would handle the situation if the work was NOT divided up evenly like it was in this assignment. I did some research and found that ScatterV takes care of this issue for us, so I would have used that along with GatherV.

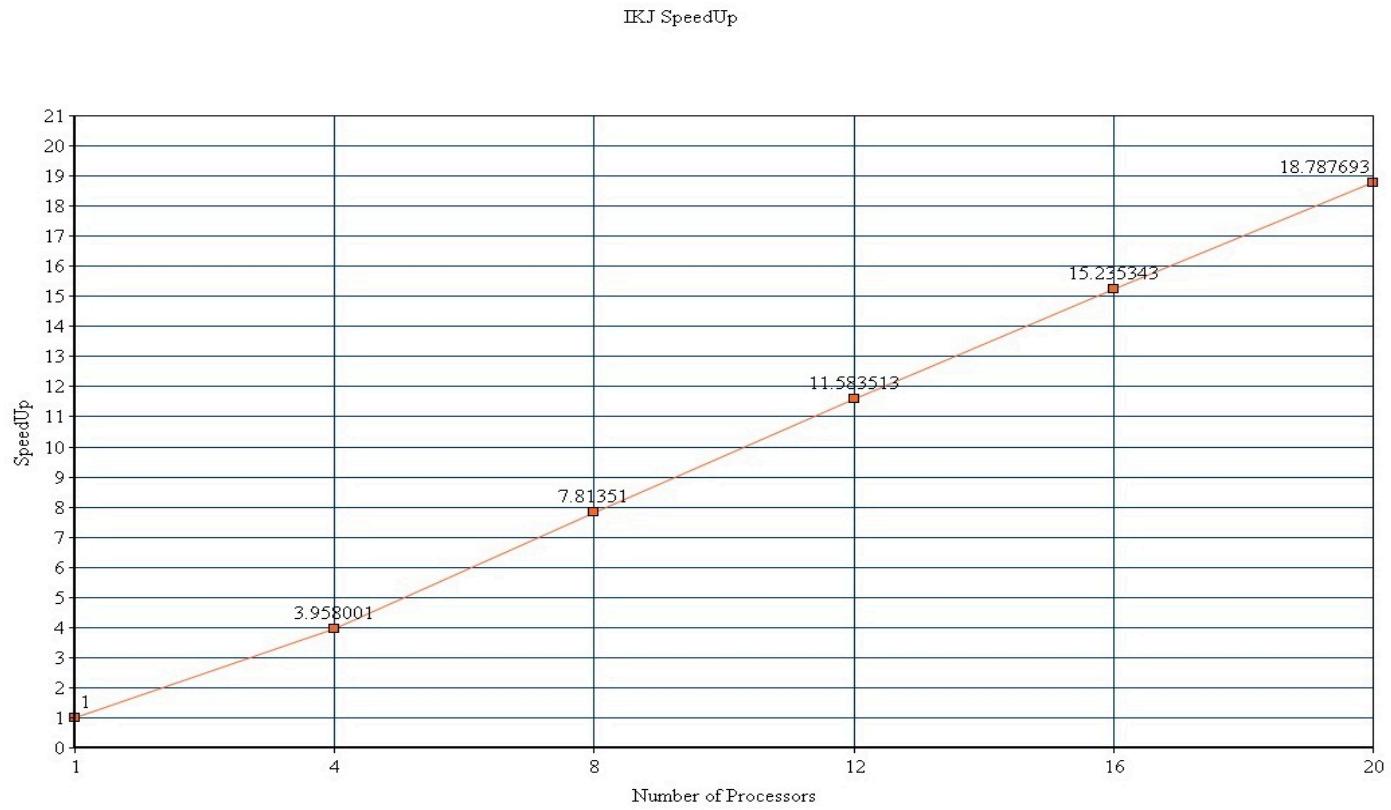
In the end, we can conclude that the method you use for matrix multiplication does actually matter. Initially, I did not think performance differences would be so major between the different IJK forms, but this assignment taught me otherwise. It is easy to see that there is a major difference if we simply look at the timing runs from 1 processor running IJK (min = 1249.0245 seconds) vs IKJ (min = 928.05212) vs KIJ (min = 931.24939). The difference in timing is about 5 minutes, which is huge.

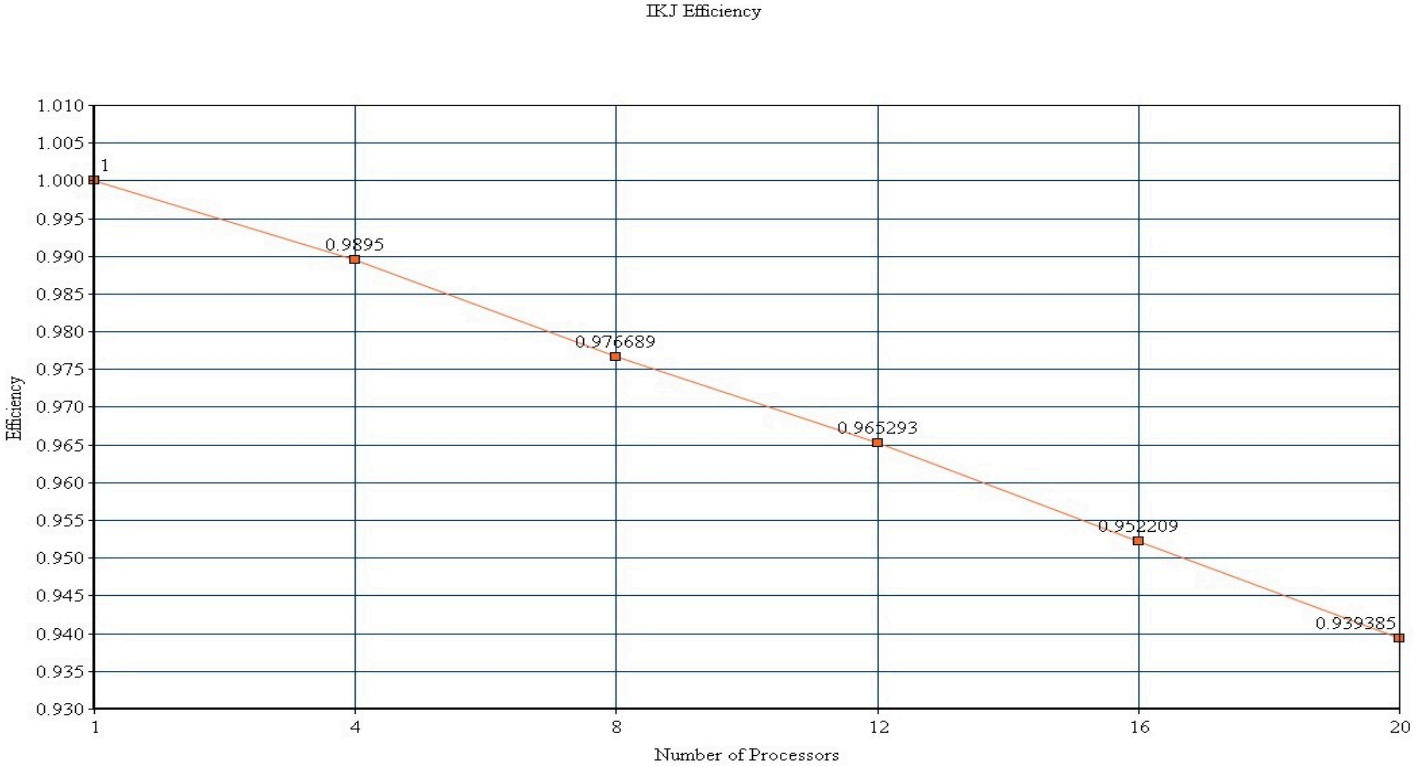
For Loop Form	Number of Processors (Times are in Seconds)					
	1	4	8	12	16	20
IJK	-> 1249.0245 <-	323.3315	161.19017	-> 107.17693 <-	81.387831	66.405539
	1250.2384	322.20503	158.82207	107.22562	-> 81.301377 <-	66.367359
	1249.1558	-> 315.8317 <-	160.07774	107.25106	82.161611	66.218579
	1253.6686	316.21107	158.5909	107.42812	81.610694	-> 65.577977 <-
	1267.586	318.46949	-> 158.56746 <-	107.80458	82.23929	65.7224
IKJ	928.55334	234.65949	118.81671	80.37642	61.008533	49.400135
	928.16964	-> 234.47495 <-	-> 118.77531 <-	-> 80.118363 <-	60.965482	49.619495
	928.27876	234.69507	118.80128	80.299319	61.029474	-> 49.396811 <-
	-> 928.05212 <-	234.60153	118.89058	80.150053	61.079599	49.397762
	928.15172	234.6494	118.94237	80.447911	-> 60.914421 <-	49.409853
KIJ	931.31674	234.85126	118.92132	80.606154	-> 61.127414 <-	49.660248
	932.4446	235.55411	119.24383	80.61748	61.25233	49.622691
	-> 931.24939 <-	235.47941	-> 118.90616 <-	80.644684	61.300891	49.539667
	931.66506	-> 234.8344 <-	119.26628	-> 80.596913 <-	61.263628	49.499813
	931.53473	235.18775	119.19419	80.617037	61.247006	-> 49.444644 <-

---

For Loop Form	Number of Processors (E = Efficiency, SU = SpeedUp)					
	1	4	8	12	16	20
IJK	E: 1.0	E: 0.9886788596584826	E: 0.9846160271470578	E: 0.9711546598694327	E: 0.9601809235039155	E: 0.9523200906304261
	SU: 1.0	SU: 3.9547154386339303	SU: 7.876928217176462	SU: 11.653855918433193	SU: 15.362894776062648	SU: 19.046401812608522
IKJ	E: 1.0	E: 0.9895002856381886	E: 0.9766888000544893	E: 0.9652927714794505	E: 0.9522089605021444	E: 0.9393846497499606
	SU: 1.0	SU: 3.9580011425527544	SU: 7.813510400435915	SU: 11.583513257753406	SU: 15.23534336803431	SU: 18.787692999499921
KIJ	Efficiency: 1.0	E: 0.991389453589423	E: 0.9789751325751331	E: 0.9628670993060656	E: 0.9521601367759479	E: 0.9417090655966701
	SU: 1.0	SU: 3.965557814357692	SU: 7.831801060601065	SU: 11.554405191672787	SU: 15.234562188415167	SU: 18.834181311933403

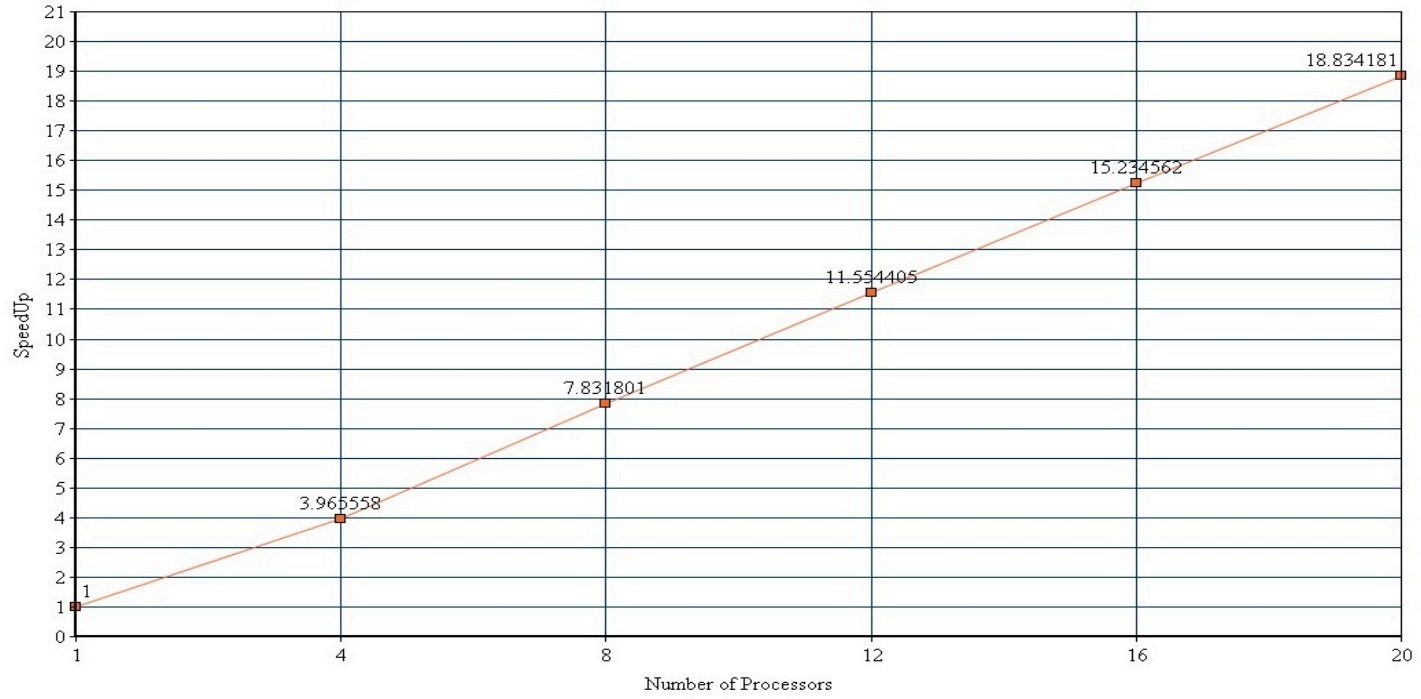
Yaseen Aniss



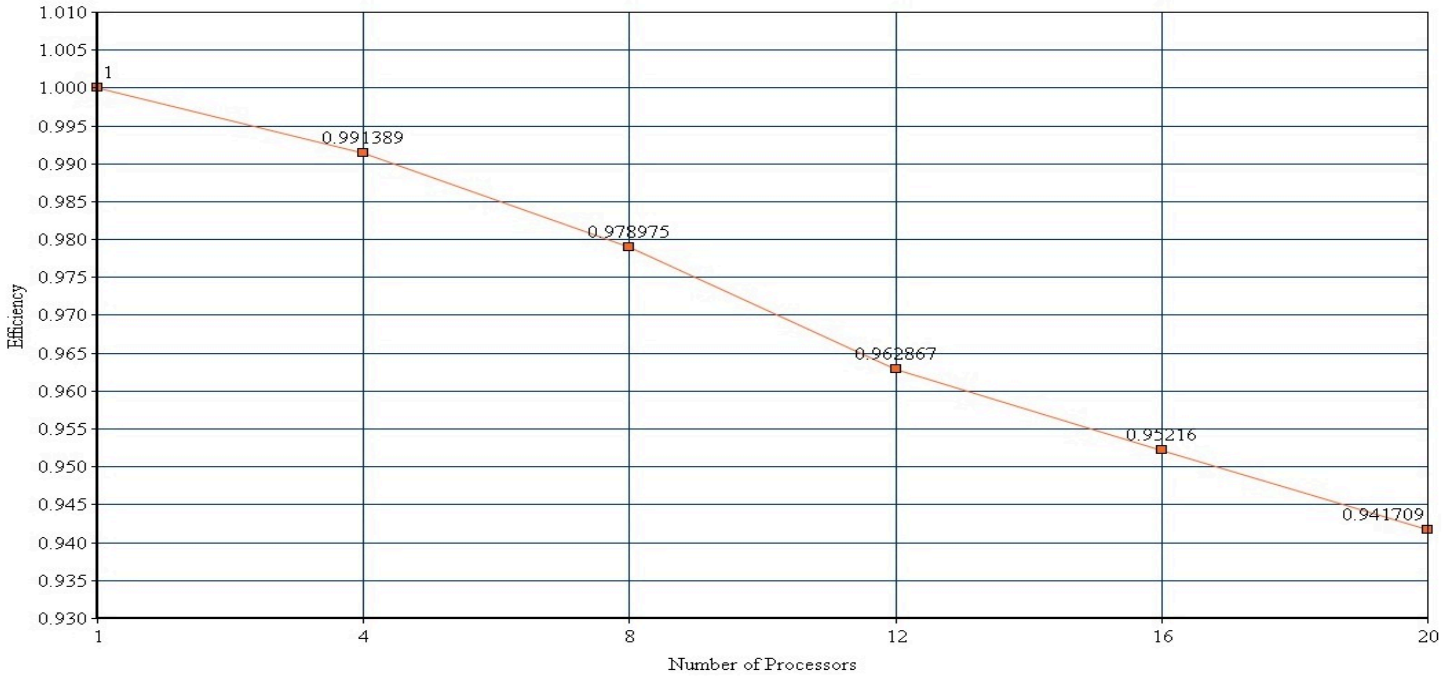


Yaseen Aniss

KIJ SpeedUp



KIJ Efficiency



Yaseen Aniss