Philip Strachan

CSCI 551

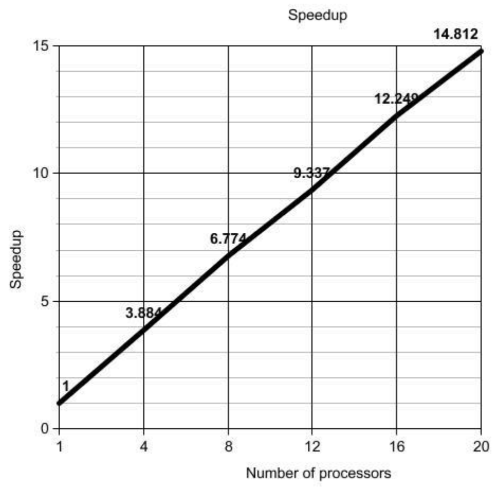
Dr. Judy Challinger

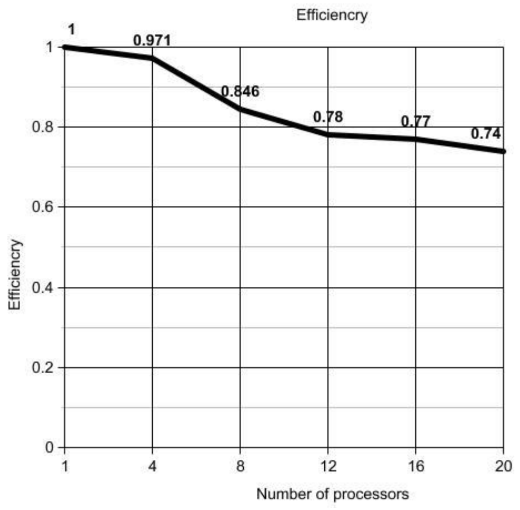
Parallel Programming: Matrix Multiplication

For my program, I use a row style partition of the solution. Each process is responsible for an equal number of rows in the solution matrix. Matrix A is scattered in the same manner. Matrix B is broadcasted over all processes. Since each process calculates at least an entire row of the solution matrix and each spot in the solution matrix needs the entire row of A and the entire column of B, each process needs the entirety of Matrix B. I followed the same pattern in all three forms: “ijk”, “ikj”, and “kij”. I used this pattern because I knew we would not need to do matrices with a smaller row count than the number of processes, and this method of partitioning minimized the communication required by dividing the size of the first array and the solution array in each process by the comm size. This data partitioning seemed to work best in the “kij” form. This could be because the k loop reads all addresses in the local portion of c, causing more cache hits.

IJK

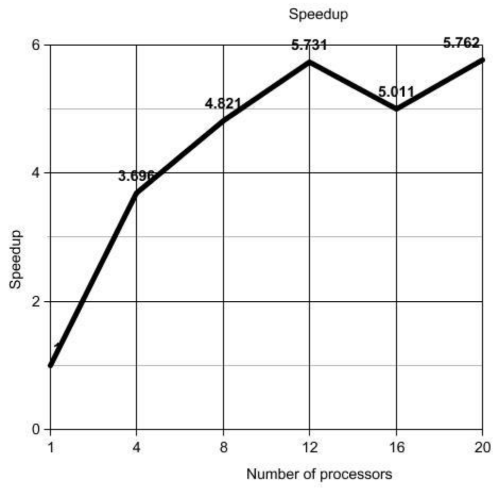
Processors	Run 1	Run 2	Run 3	Run 4	Run 5	Speedup	Efficiency
1	1485.44 s	1614.75 s	2101.6 s	2161.5	1849.7	1.0	1.0
4	720.253	382.413	669.947	582.313	593.905	3.884	0.971
8	230.45	266.04	235.153	219.275	220.435	6.774	0.846
12	207.222	287.928	206.729	291.825	158.415	9.377	0.78
16	253.519	121.271	122.377	143.725	233.68	12.249	0.77
20	160.116	100.288	97.3213	103.866	185.295	14.812	0.74

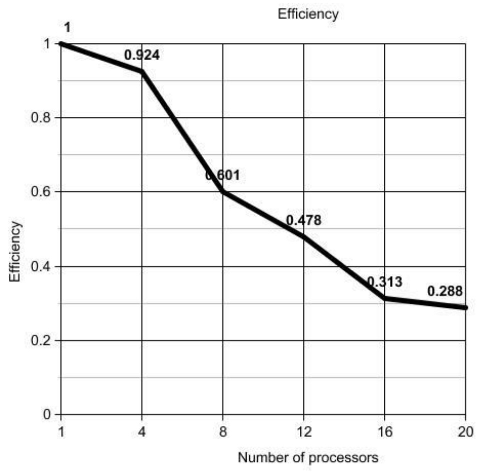




IKJ

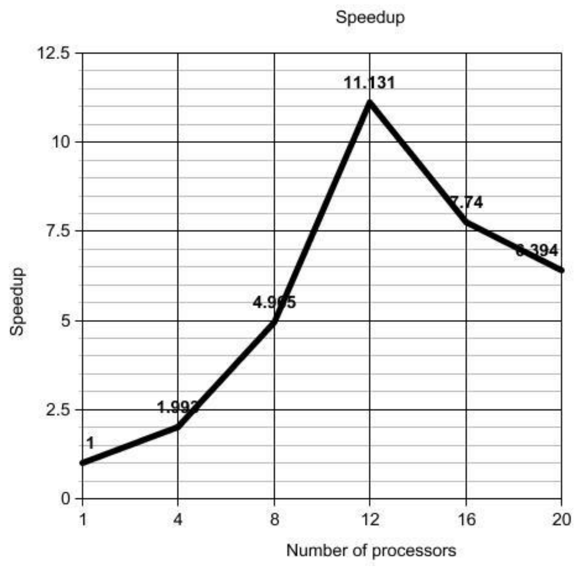
Processors	Run 1	Run 2	Run 3	Run 4	Run 5	Speedup	Efficiency
1	795.9	543.858	519.929	561.653	605.32	1.0	1.0
4	187.888	140.68	140.693	140.79	251.104	3.696	0.924
8	107.854	134.642	127.643	193.29	128.392	4.821	0.601
12	105.933	95.3635	90.7188	119.155	106.817	5.731	0.478
16	153.007	166.709	137.888	105.096	103.744	5.011	0.313
20	106.78	108.088	144.398	90.228	165.623	5.762	0.288

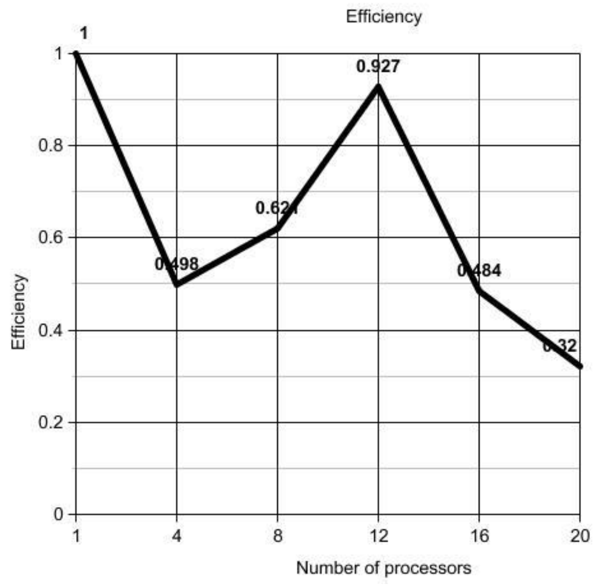




KIJ

Processors	Run 1	Run 2	Run 3	Run 4	Run 5	Speedup	Efficiency
1	791.255	597.59	548.123	542.275	610.33	1.0	1.0
4	554.119	272.017	306.561	369.038	298.802	1.993	0.498
8	224.293	138.801	304.378	109.222	141.767	4.965	0.621
12	103.901	66.2416	48.7133	76.5165	76.4371	11.131	0.927
16	130.746	118.957	70.03	119.174	112.837	7.74	0.484
20	104.951	107.777	107.921	108.002	84.8133	6.394	0.320





Conclusion

There are many factors that can go into how much speedup you will gain from adding more processors to a cluster. The algorithm used and exactly how it accesses the data will affect speedup depending on how many cache hits it gets. Form “KIJ” appeared to run the fastest with my data partitioning. This could be because the algorithm accesses all indexes of local_c and local_a before returning to previous indexes. Different algorithms could also require different communication patterns, which will require different overhead. My communication pattern is a composed of a broadcast and a scatter, as array B in A*B=C is needed across all processes, if A is scattered by row. B is broadcasted, while A is scattered. Finally, the solution is gathered. It could be possible to have each processor calculate a block of values in the solution matrix. This would mean that each processor would be able to split A by row and B by column. However, if this method is used, we must duplicate data on different processors, which causes extra overhead. Each processor would be required to have twice as many rows of A, but would only take the same amount of columns of B as it did rows of A. Even more overhead would be added to decide which rows go where, as a simple scatter would not get what we need. The final factor that goes into speedup running on a cluster is what other tasks are being processed on the cluster. In my “KIJ” runs, when I went from 12 to 16 processors, it appeared to do much worse. This could be because these new processors had jobs running on them already, causing them to slow down.