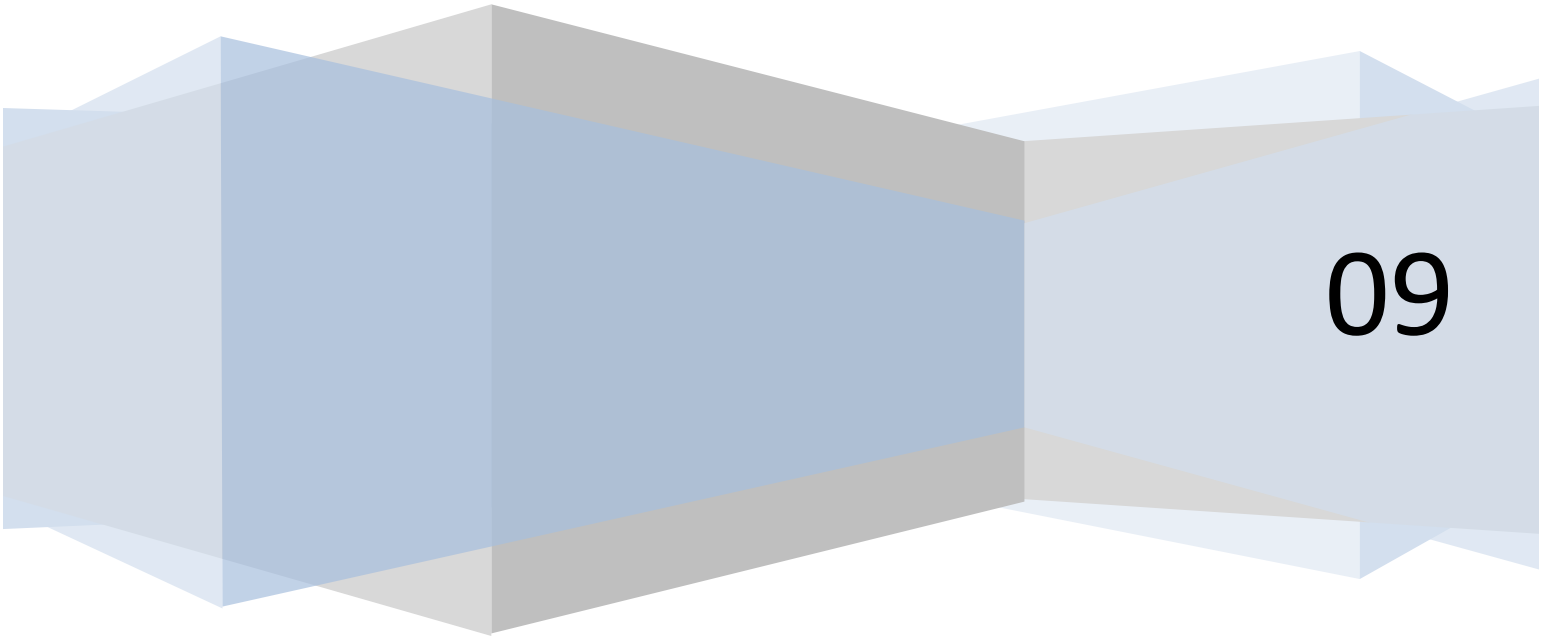
MPI

CS 732

Joshua Hegie



09

The goal of this assignment was to get a grasp of how to use the Message Passing Interface (MPI). There are several different projects that help learn the ups and downs of creating parallel projects in MPI. The first project helps show how timing in MPI works, at least on the grid. After that is the staple of counting the number of occurrences of the value 3 in an array. The other standard test is a matrix multiplication.

Test Bed:

This code was all run on the 2 core rack 40/41 machines.

Ping Pong Test:

This test is similar to the Posix Threads ping pong test. the goal of the test is to see how much time is required to pass a message between two instances. For threads it was the time to get one thread to respond, for MPI it is passing a message over the network and getting a response back. This took 13.7 µs over ethernet on the rack 40/41 machines. If the same code was to be tested on rack 4, where InfiniBand is available, the time will likely go down.

Count 3's Test:

#of Nodes	Time (us)
0	627334.625
2	15.15
3	128.762
4	13.9
5	136.475
6	65.863
7	209.837
8	80.213
9	213.212
10	106.463
11	157.363
12	37.662
13	59.912
14	116.887
15	291.388
16	52.963

Table 1: Average time for count 3's with the given number of threads, 0 threads is sequential

# of Nodes	Speedup
2	1.57584824
3	2.40946477
4	2.5624892
5	2.06990075
6	4.16536912
7	4.98595323
8	3.46171184
9	3.33192681
10	3.29418175
11	3.36390507
12	0.97451371
13	1.16851234
14	3.64663439
15	6.08434724
16	2.31071823

Table 2: Speedup for Count 3's

# of Nodes	Efficiency
2	0.78792412
3	0.80315492
4	0.6406223
5	0.41398015
6	0.69422819
7	0.71227903
8	0.43271398
9	0.37021409
10	0.32941818
11	0.30580955
12	0.08120948
13	0.08988556
14	0.26047389
15	0.40562315
16	0.14441989

Table 3: Efficiency for Count 3's

This code was provided in the book, as a means of showing how the cache sizes on a CPU can affect how parallel code is run. For this code an integer array of 1,000,000 integers 0 to 9, inclusive, s processed by the given number of nodes. In the case of 0 nodes the code is being run as a single process. The MPI code did not do very well. This is likely because there was not enough work for all of the nodes to do, as well as a high cost for passing messages between the nodes. The amount of data definitely did not justify adding nodes to the system. If the size of the array is increased, there will probably be an increase in the time required for the sequential version, and an increase in speedup and efficiency for the parallel version. Table 1 shows the times required for the various numbers of nodes. As more nodes were added the times decreased, but overall speedup remained well below the linear

line, as can be seen in Chart 1. Chart 2 shows the efficiency of the program. With 2 nodes, up to 3, the efficiency achieved by adding another node remains above 75%, after that it just tanks. This seems to be related to the problem size, as will be seen in the matrix multiplication section.

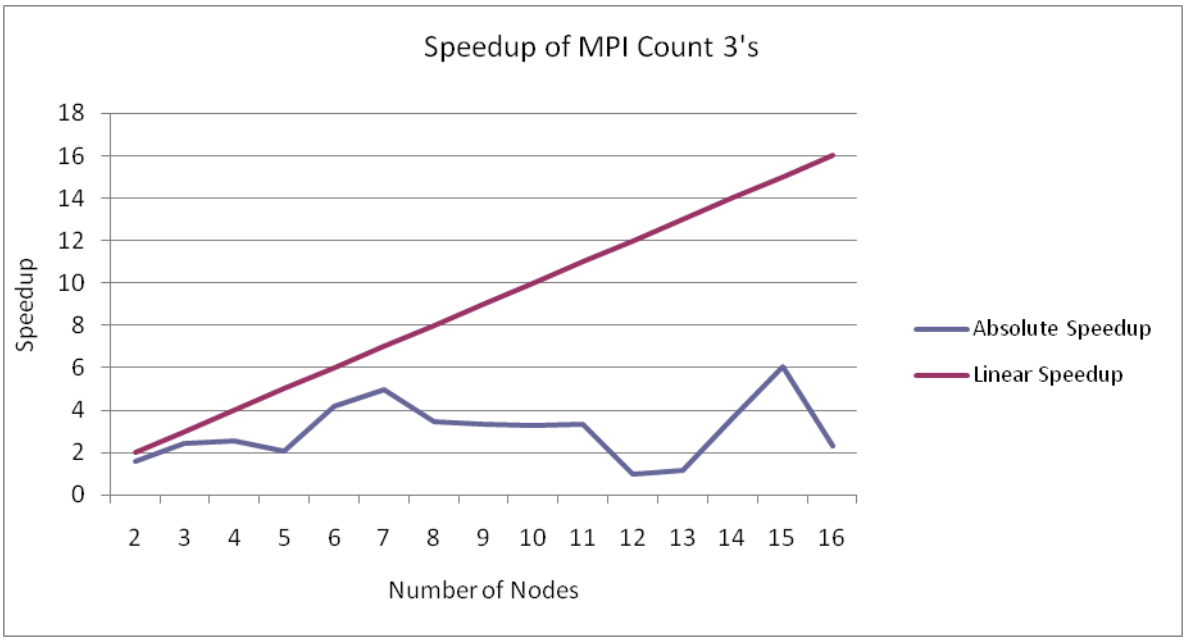


Chart 1: Speedup for the count 3's

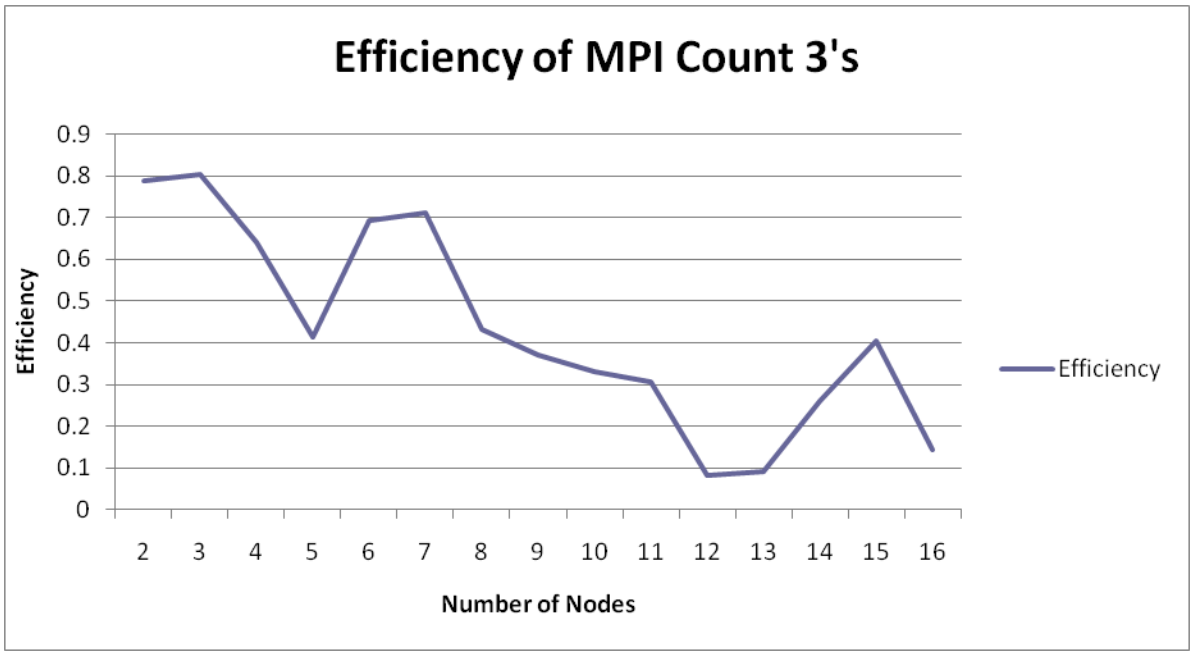


Chart 2: Efficiency of the count 3's program

Matrix Multiplication:

# of Nodes	Time (us)
0	29808371.2
2	11330883.7
3	7339176.6
4	5250285.97
5	4348382.49
6	3744523.05
7	3440641.39
8	2705110.65
9	2698073.23
10	2256387.63
11	2067661.93
12	1798610.37
13	1827401.95
14	1617482.56
15	1654994.17
16	1320284.31

Table 4: Average time for Matrix Multiplication with the given number of Nodes, 0 threads is sequential

# of Nodes	Speedup
2	2.63071902
3	4.06154162
4	5.67747574
5	6.85504812
6	7.9605255
7	8.66360887
8	11.0192798
9	11.0480216
10	13.2106606
11	14.4164628
12	16.5730008
13	16.3118854
14	18.4288671
15	18.0111639
16	22.5772366

Table 5: Speedup for Matrix Multiplication

# of Nodes	Efficiency
2	1.31535951
3	1.35384721
4	1.41936894
5	1.37100962
6	1.32675425
7	1.23765841
8	1.37740998
9	1.22755795
10	1.32106606
11	1.31058753
12	1.3810834
13	1.25476042
14	1.31634765
15	1.20074426
16	1.41107729

Table 6: Efficiency for Count 3's

This part of the project was to write a matrix multiplication program. For this section the size of the matrices is 1000 x 1000. Table 4 shows the times required to run this code. The sequential time (0 nodes) is substantially longer than the parallel times, for the opposite reason from the previous section. The data set for this was far too large. The test machines have 4 GB of RAM each. The data in this case is greater in size than the memory size, once the OS is taken into account. So the single box is stuck, at this point, page faulting and wasting time swapping memory. The MPI, however, has the memory decentralized, so each node only stores what it needs. This makes it so that there is substantially less swapping. The other benefit to MPI in this case is that as more nodes are added into the system, smaller messages are required, so if a message had to be broken up before, it may not when one more node is added. Table 5 shows numerically, as does Chart 3 visually, that the speedup for adding one more node increases, at least for the data set provided. The speedup is always greater than linear, which is explained by the swapping issue, discussed above. Table 6 shows the efficiency for the 2 node, and up through the 16 node, configurations. There is always more than 100% efficiency, see the swapping discussion above, with all of the fluctuations averages to about 130%.

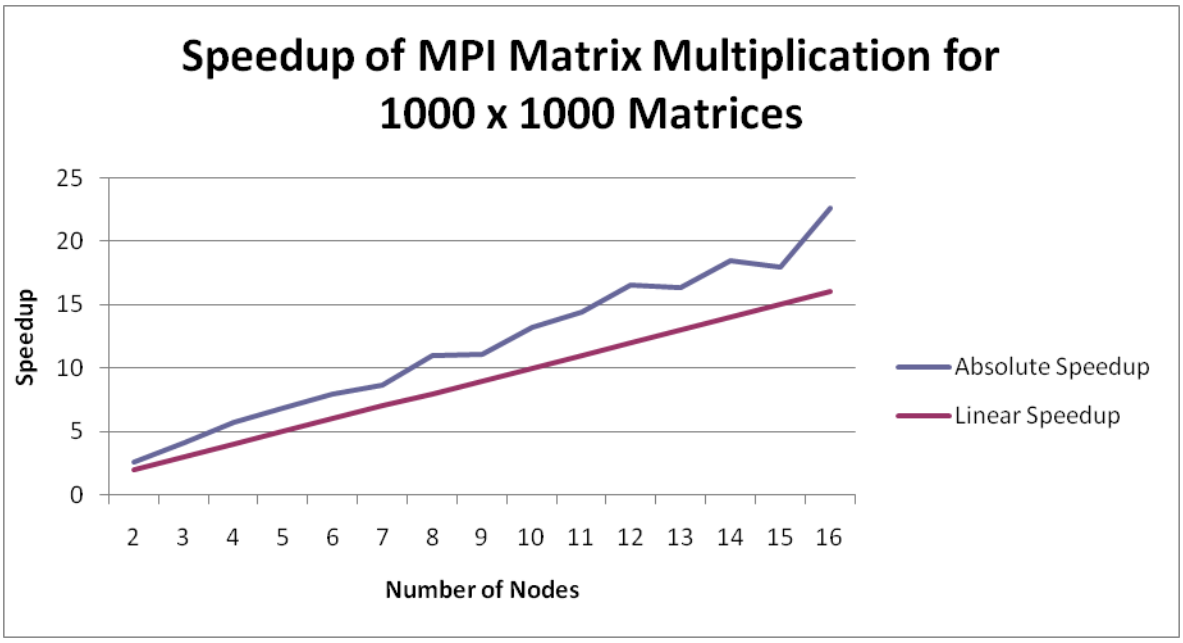


Chart 3: Speedup for 200 x 200 matrix multiplication

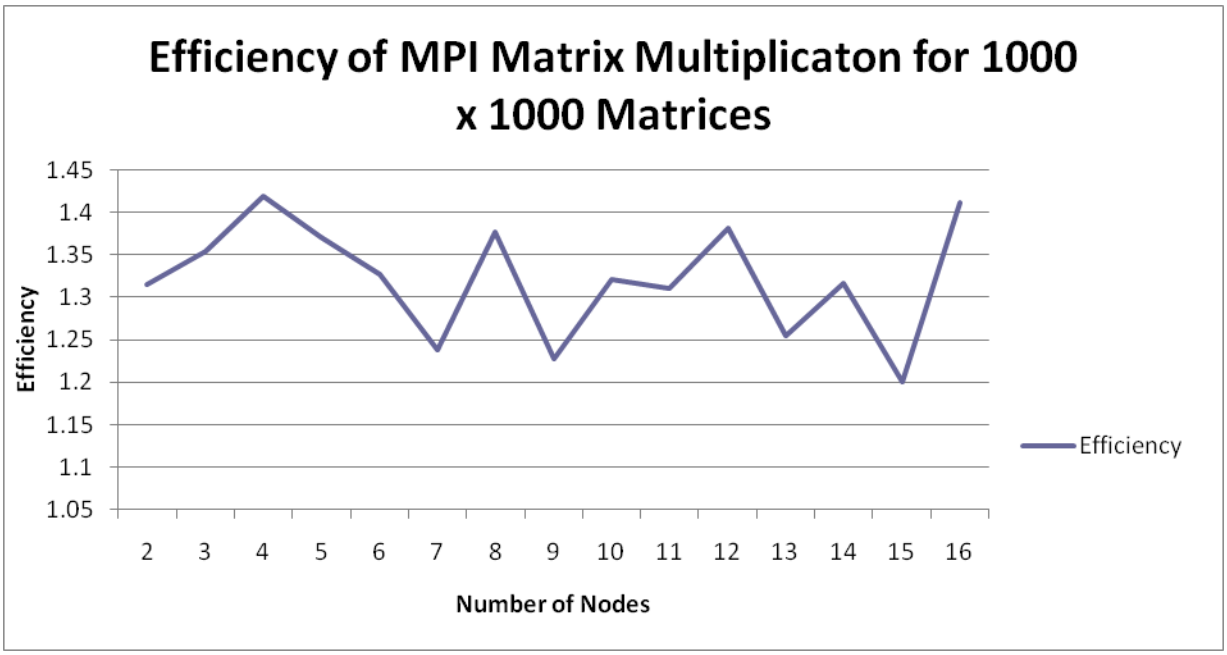


Chart 4: efficiency for matrix multiplication