Assignment 3 (Open MPI)

Daniel Wisner (daw268)

Part 1 (MPI Communication Overhead)

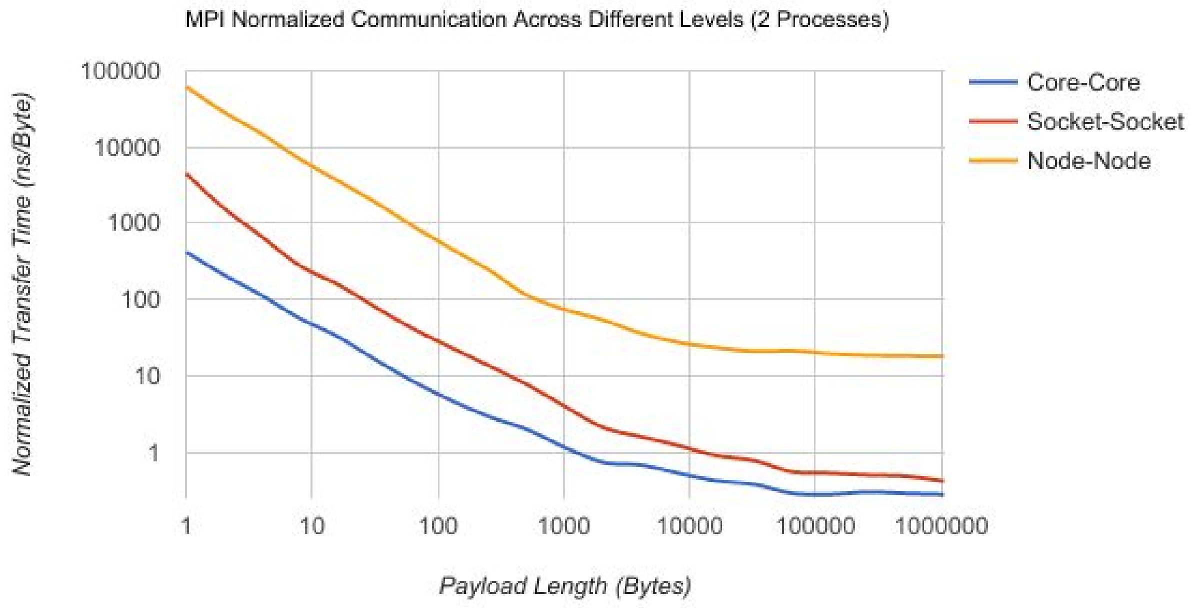
Three different levels of communication between cores was benchmarked: core-core (same socket), socket-socket, node-node. Since the most sockets on any one node (en-openmpi00) was two, to make data comparable across the different levels, only two processes were run. Otherwise, it would be an unfair comparison as multiple processes for the socket-socket test would be running on the same socket, causing the QPI link between the two sockets to be loaded with multiple pairs of processes communicating.

One process would send a message with n bytes, the other would receive the data then send it back. The number of bytes, n, was varied from 1 byte to 2^{20} bytes, and each ping test for each particular n was run several thousands of times and averaged. This test effectively measured the round-trip ping time between process running with different payload sizes. The results were normalized by n (number of bytes send) to determine the round-trip time per byte of data. All tests were tests were run on en-openmpi00, except for the node-node test where en-openmpi01 was also used. The results are shown below:

MPI Normalized Communication Time Across Different Levels (2 Processes)

Payload Length (Bytes)	Core-Core (ns/Byte)	Socket-Socket (ns/Byte)	Node-Node (ns/Byte)
1	419.739081	4486.726539	62125.96782
2	209.796781	1569.358574	28602.02221
4	112.870566	665.961579	14861.57998
8	56.482349	277.085974	6967.646186
16	32.621415	157.980935	3618.733899
32	16.178149	79.115864	1876.738679
64	8.36576	41.026965	909.478331
128	4.724612	23.363057	463.417564
256	2.905487	13.439511	241.2512
512	1.965969	7.627506	113.261194
1024	1.14572	3.952891	73.666115
2048	0.73161	2.097122	53.776944
4096	0.676006	1.585279	35.950194
8192	0.525215	1.210977	27.260256
16384	0.418506	0.893624	23.313405
32768	0.374605	0.768436	20.927263
65536	0.287254	0.548179	21.212973
131072	0.278244	0.526746	19.416541
262144	0.299567	0.497602	18.513957

524288	0.286771	0.480604	18.139999
1048576	0.27811	0.413382	17.927899



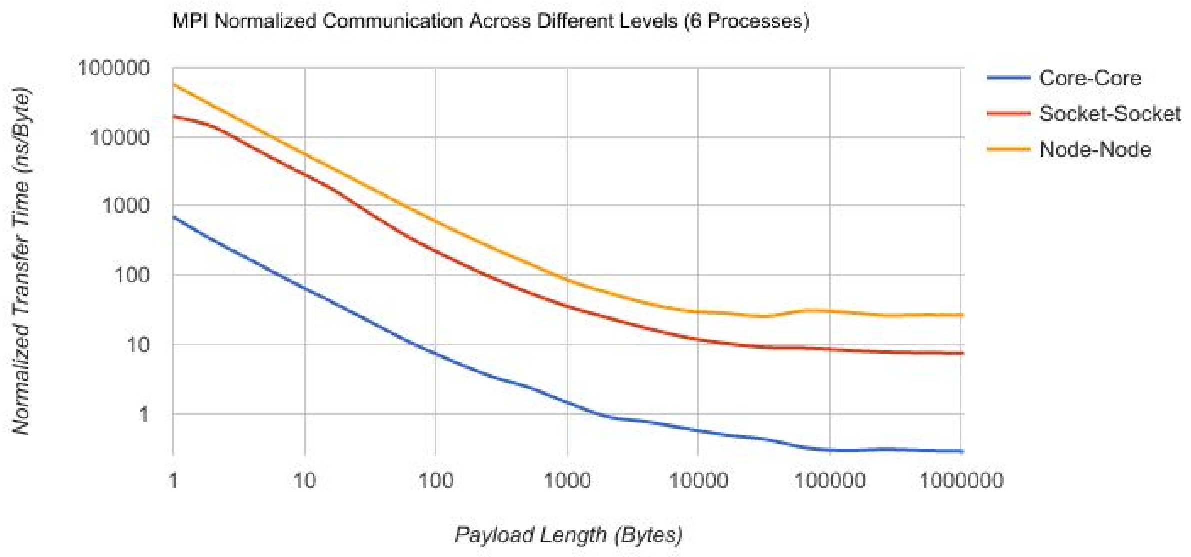
The tests was run with much large n to confirm the normalized transfer times did not substantially decrease further than the collected data. The round-trip latency can be approximated by the round-trip time for a payload on one byte. This came out to be ~420 ns for core-core, ~4500 ns for socket-socket, and ~62,000 ns for node-node. This works out to roughly one order of magnitude between each higher level (i.e. node-node latency is ~100 times longer than core-core).

Additionally, bandwidth can be roughly approximated by taking two times the reciprocal of normalized round-trip transfer times for large n. For core-core this worked out to be ~7.2 GB/sec, socket-socket was ~4.8 GB/sec, and node-node was ~112 MB/sec. These results make sense for the node-node and socket-socket, given that DDR4 memory bandwidth is on the order of 30 GB/sec and each send and receive likely needs several memcpy() between kernel and userspace buffers. The node-node data especially makes sense given that the nodes are connected via ethernet and 1 Gigabit ethernet has a maximum of ~125 MB/sec. The ethernet connection was confirmed with “ethtool em1” which reported the connection as running at 1000 Mb/sec. The results for bandwidth this time show that core-core and socket-socket bandwidth are comparable, whereas node-node bandwidth is 1 to 2 orders of magnitude slower.

As an extension to the original benchmarks, the tests were rerun with six processes simultaneously pinging each other, with the even nodes sending first then receiving, and the odd nodes receiving first then sending. The “--map-by ppr:16:socket” flag was used for the core-core test to make sure all processes were on the same sockets. The “--map-by socket” flag was used for the socket-socket test to make sure each pair of processes communicating were on different sockets. Finally, the “--map-by node” flag was used for the node-node test to make sure each pair of processes communicating were on different nodes. Of course, the results are a bit skewed for the socket-socket test, due to all the interprocess traffic having to be serialized across the QPI link that connect the two sockets. The results are shown below:

MPI Normalized Communication Time Across Different Levels (6 Processes)

Payload Length (Bytes)	Core-Core (ns/Byte)	Socket-Socket (ns/Byte)	Node-Node (ns/Byte)
1	697.366583	19534.84025	57592.63877
2	322.062988	14087.39869	28209.15931
4	161.041195	7039.110642	14110.80787
8	79.914268	3514.578566	7047.322773
16	41.317738	1783.36874	3576.657036
32	20.953545	763.910066	1803.907556
64	10.628355	343.1475	913.482533
128	5.904136	175.849891	475.651783
256	3.492914	95.03492	257.845992
512	2.364914	55.560046	146.494358
1024	1.412369	34.857294	83.31934
2048	0.891608	24.050166	55.535698
4096	0.751186	16.908767	38.770348
8192	0.600049	12.484515	30.3489
16384	0.484536	10.257324	27.908795
32768	0.417521	8.993264	25.284175
65536	0.316879	8.789565	30.607898
131072	0.288601	8.180201	28.956947
262144	0.301531	7.685294	26.079815
524288	0.289532	7.477008	26.512281
1048576	0.282693	7.380976	26.189885



As expected the core-core and node-node results remained relatively the same for both latency and throughput measurements with three pairs of pinging processes. The core-core results saw an ~50% increase in latency, and no change in throughput. Similarly, the node-node saw relatively unchanged latency and a 30% reduction in throughput. However, the socket-socket, as predicted, saw a dramatic increase in both latency and throughput. The socket-socket results, on average, show nearly a 20 times decrease in throughput for large n's. Additionally, latency increased by nearly a factor of five. Again, this is was expected and due to the processor interconnect (QPI) that connects the two separate sockets having to handle three times the amount of data.

Conclusions

Overall, these results reveal that socket-socket communication throughput is comparable to core-core throughput for single processes communicating, with latency being roughly one order of magnitude larger. However, communication across sockets with multiple pairs of processes does not scale well. If any more than two processes are communicating across sockets, both the latency and throughput seem to have a much worse than linear slowdown for each additional pair of processes that are communicating across sockets. Therefore keeping the number of processes requiring inter-socket communication to a minimum is critical, especially for small transfers.

Node-node communication over ethernet is roughly 100 times slower in terms of throughput, and has latencies that are roughly 150 times as large compared to core-core communication on the same socket. Again, keeping the amount of data transfers down, especially for small transfers is critical for good performance. Additionally, although not explicitly tested, it is obvious that minimizing the number of processes on one node that need to communicate to another external node is also important, as the node interconnect (ethernet) is shared across all processes running on any particular node.

Macros to Play With:

- **NUM_RUNS:** Sets the number of times to run a ping test for each particular payload size. The results are all averaged. **Recommended: (1 << 10)**

- **MAX_MSG_LEN:** Sets the max payload size to use in the benchmark. All payloads from 1, 2, 4,…, MAX_MSG_LEN bytes are tested. **Recommended: (1 << 20)**

Part 2 (Using MPI To Parallelize SVD)

Jacobi Rotations

Parallelization of computing the SVD of a MxN matrix A was done by implementing Jacobi rotations that would zero out the off diagonal elements of A'*A. This rotation was also applied to a matrix V that initially started out as a NxN identity matrix. When the sum of the squares of the off diagonal elements of A'*A (off(A'*A)) are less than a set threshold, the method terminates. At this point the normalized columns of A are orthonormal eigenvectors (left-singular vectors), the elements on the diagonal of A'*A are the corresponding singular values, and the columns of V are the right-singular vectors. Due to the fact that a Jacobi rotation only affects two columns of A and two columns of V the method is a good candidate for parallelization.

Implementation

Initially, the algorithm was implemented in a single threaded program that did not use MPI (daw268_hw3_part_2_single_threaded.c). This initial program performed sweeps across the columns of A, performing rotations with each pair of columns until off(A'*A) was less than a set threshold. Pseudo code for one sweep is as follows:

```
// A = MxN, V = NxN
for (i=0; i < N; i++)
        for (j=i+1; j < N; j++)
                jacobi_rotatation(A[i][], A[j][], V[i][], V[j][]);
```

Once, this implementation was confirmed working by comparing the results with Matlab, moving onto parallelizing the algorithm came next. Given that this program would be running on an MPI cluster, the design goal for this parallelized algorithm was to scale well when running on multiple nodes, not just for running on the same node where communication overhead between processes is minimal. Taking into account the results form part 1, it was clear that inter-node communication had to be kept to infrequent large transfers, as the latency between processes on different nodes was on the order of 150 times longer than between processes on the same processor. Thus, the algorithm was designed to increase the granularity over the suggested Brent-Luk column permutation method which requires each process to exchange data roughly every N/(2*NUM_PROCS) jacobi rotations N times to perform one sweep.

The Parallel Jacobi Algorithm

The devised algorithm was inspired from the Brent-Luk column method. However, rather than send and receive a single column, an entire block of columns is sent and received. With this new algorithm, there are now N^2/(2*NUM_PROCS^2) Jacobi rotations for each process between data exchange events. Furthermore, data needs to only be exchanged NUM_PROCS times to perform one sweep. Overall, this is a significant improvement in the granularity and the number of message exchanging events compared to the single column exchange method. The Open MPI 1D virtual topology was used to implement the Brent-Luk ordering with left and right shifts. The algorithm is as follows (Note: the operations on V are not mentioned for simplicity in the explanation, but are intuitive and included in the program):

1. Split the MxN matrix A into Mx(N/(2*NUM_PROCS)) column block matrices. If the number does not divide evenly, put the extra into one of the column blocks.
2. Give each process two column blocks.
3. Each process in parallel merges its two column blocks together into one Mx(N/NUM_PROCS) matrix and perform one Jacobi sweep across all the columns.
4. Each process breaks the merged matrix back into its original two column blocks.
5. Use Brent-Luk ordering to permute the column blocks between the processes.
6. Repeat, starting at step 3 until one sweep is performed. After one sweep each process determines whether the terminating off() criteria has been satisfied for its set of column blocks. The results are reduced and distributed with an Allgather command.
7. Keep sweeping until all processes report the off() criteria as being satisfied for its two column blocks or a MAX_NUM_SWEEPS condition is reached.
8. Gather all column blocks on the root node and assemble matrix A from its gathered column blocks.

Results

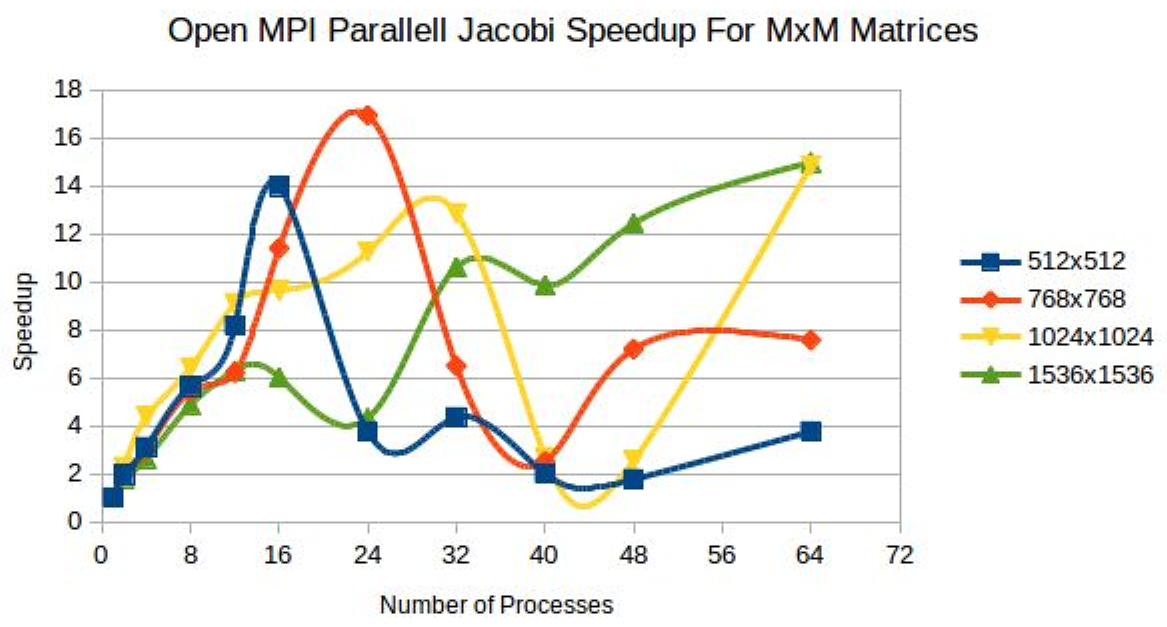
The results were collected by testing the programming with different number of processes for progressively larger MxM matrices. To truly have an accurate speedup measurement, the NUM_PROCS = 1 case was tested on the single threaded version of the algorithm that did not use MPI. Thus, the results represent the true speedup over a single threaded implementation. All tests were launched from en-openmpi00. When the number of launched processes exceeded the number of local cores (32), nodes en-openmpi02, en-openmpi03… were used in that order (see p2_hostfile). The results are shown below:

Raw Results

Number of Processes	Runtime (Seconds 256x256)	Runtime (Seconds 512x512)	Runtime (Seconds 768x768)	Runtime (Seconds 1024x1024)	Runtime (Seconds 1536x1536)
1	1.444987	15.130571	50.277724	185.830761	468.518157
2	0.632998	7.710844	25.38423	81.430562	259.121005
4	0.342429	4.8824	16.713011	42.485187	178.036148
8	0.193364	2.677001	9.284866	28.968714	96.20597
12	0.375445	1.854721	8.094351	20.42556	74.559116
16	0.086115	1.082475	4.414367	19.30424	77.97759
24	0.384752	4.039005	2.970752	16.521671	108.632612
32	0.292405	3.486918	7.748912	14.469394	44.185579
40	2.366483	7.573947	20.033486	69.246096	47.550633
48	7.674109	8.556138	6.994191	73.18017	37.729784
64	1.973865	4.029026	6.64766	12.533168	31.299525

Speedup

Number of Processes	Speedup (256x256)	Speedup (512x512)	Speedup (768x768)	Speedup (1024x1024)	Speedup (1536x1536)
1	1	1	1	1	1
2	2.282767086	1.962245767	1.980667682	2.282076366	1.808105665
4	4.219814911	3.099002745	3.008298385	4.374013018	2.631590058
8	7.472885335	5.652060272	5.415018806	6.414877823	4.869948892
12	3.848731505	8.157869027	6.211458337	9.097951831	6.283848068
16	16.7797364	13.97775561	11.38956593	9.626422019	6.008369289
24	3.755632199	3.746113461	16.9242414	11.24769771	4.312868377
32	4.941731503	4.339239122	6.48835914	12.84302307	10.60341785
40	0.6106052737	1.997712817	2.509684236	2.683627984	9.853037225
48	0.1882937811	1.768387911	7.188497426	2.539359515	12.41772699
64	0.73205969	3.75539175	7.563221344	14.82711801	14.96885838



In all cases, the parallelized Jacobi performed faster than the serial version. However, the speedup results initially seemed rather random. The same test was run multiple times, and the seemingly random results were found to indeed be accurate. Several explanations for the unpredictable performance speedup for additional processors are proposed below:

- **The drop in performance between 16-24 processes**: Given that en-openmpi00 has two 16 core processors, it becomes clear while some results showed a hit to the speedup during the transition from 16 to 24 process. After 16 processes, any additional processes must be run off socket on the other processor. As noted in part 1, the latency for inter socket communication is on the order of a magnitude longer than core-core communication on the same socket. The result is processes communicating off socket take longer to send and receive their column blocks, thus they begin processing their new column blocks later than all the other processes. Effectively, this causes the other processes to eventually have to block while they wait for the other socket-socket pair of processes to catch up.

- **The drop in performance between 32-40 processes:** All the tests show a drop in performance from 32 to 40 processes. This is due to the fact that additional processes past 32 are being run off the node en-openmpi00, and message passing between certain processes is now happening through Ethernet, which was shown in part 1 to be several orders of magnitude slower in both latency and raw bandwidth. However, as more processes are added beyond this 32-40 process transition, the additional computational resources compensate for the increased message transfer overhead.

Conclusions

As mentioned earlier, the design goal of the algorithm was to create one that scaled well across multiple nodes, not just for processes running on the same node. Given the huge communication penalty between nodes, utilising multiple nodes had to be done with careful considerations to actualize speedups. With the Jacobi rotation column block ordering algorithm that was devised, this goal was impressively achieved. Not only did the algorithm scale well when running multiple threads on the same node, but for large matrices, the largest speedup occurred when multiple nodes were participating. Overall, the Jacobi rotation method for computing the SVD lended itself well to parallelization as double digit speedups over the single threaded version could be achieved when picking the optimal number of processes.

Macros to Play With:

- **N:** The number of columns in the randomly filled matrix A to compute the SVD of.
- **M:** The number of rows in the randomly filled matrix A to compute the SVD of.
- **NUM_PROCS:** The number of processes to use. Note: The program must be recompiled every time you wish to run with a different number of processes.
- **MAX_NUM_SWEEPS:** If the off() threshold is never reached, this will limit the number of sweeps.
- **PRINT_MATRIX:** Enable printing of the matrix A (before and after) to stdout.