APMA 2822b Homework 2

Ryan Greenblatt

February 2019

1

Code is attached in the email. I have included a CMakeLists.txt file which can be used to compile the code on the ccv. The MPI must be loaded. Note that there is a hard coded path to the OpenBLAS library because cmake did not automatically find the library. I also added the SLURM scripts I used. There are 3 executables in the project, one which tests the code, one which times the matrix multiplication, and one which times MPI message passing.

Algorithm and Implementation

I used a grid data decomposition across all matrices. Any grid dimensions can work, but if the dimensions are closer to being equivalent the algorithm will be more efficient. As such, the closest integer factors of the MPI world size are used. The code doesn't handle the case where a matrix dimension isn't divisible by the corresponding grid dimension, but this would be simple to implement use padding and would have a negligible effect on performance. Figures 1 and 2 show the decomposition for different grid dimensions. The approach of the algorithm is to loop over the blocks in the shared dimension between A and B. At each step, each process adds to its section of matrix C the product of the sections of A and B which it currently possesses. Then, each row of matrix A rotates and each column of matrix B rotates. This allows each process to only send data to the same two other processes each step. Because matrix multiplication is performed at each step, 3 level blocking techniques can be used to improve cache utilization. The intuition behind this algorithm is that the value of the top left block of C is computed by multiplying the first row of A by the first column of B. This operation can be decomposed into the multiplication of the top left block of A and the top left block B as well as every remaining block in the top row of A with its corresponding block in the left most column of B. A similar decomposition can be done for every block in C. Note that the data in row and columns other than the first must start rotated in order for the data to be multiplied by the correct other block. The grid pattern will not always be square, so it is potentially necessary further divide the shorter dimension. In my implementation, there are always more column blocks. Multiple approaches can be used for further dividing the data. I assigned the additional divisions as blocks in matrix B (they are always in matrix B because there are always more column blocks) that some process contains but aren't used in that iteration. This can be seen in figure 2. Note these blocks require extra rotations of the next row in A. This can be seen in the bottom row of A in figure 2. This approach has the advantage of being simple and computationally light, but it does result in a slightly uneven distribution of data.

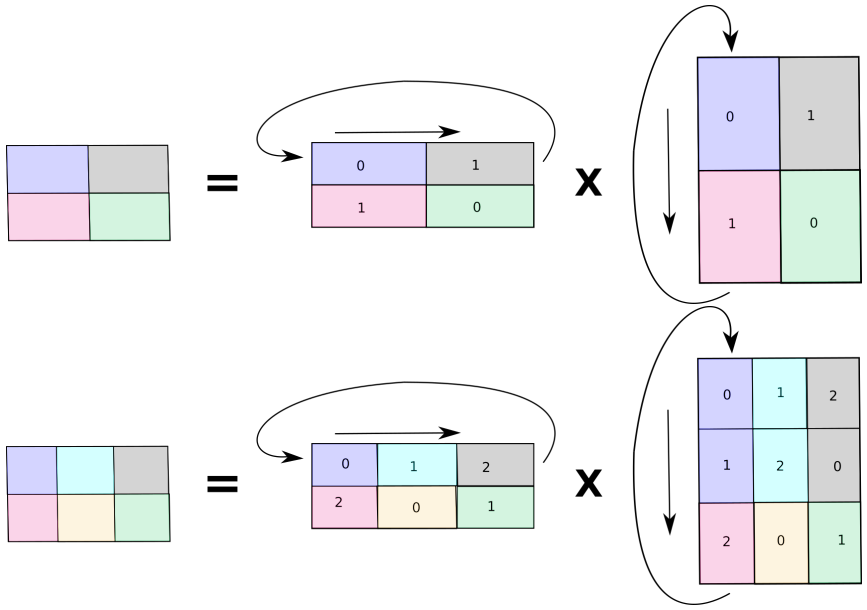


Figure 1: Examples of the data decomposition. Each color represents a different process. The numbers indicate the initial data placement in terms of which block along the shared dimension the data is from. For instance, if the numbers in a column in matrix B are 2, 0, 1, the column has been rotated once.

I evaluated this algorithm with OpenBLAS as well as the matrix multiplication implementation that I did for the last homework. This is so the efficacy of the distributed matrix multiplication algorithm can be assessed independently from the shared memory matrix multiplication algorithm used. I used only the non-blocking MPI functions for both send and receive. Both are started before the matrix multiplication each step, and the wait for completion happens after the multiplication. Additional memory is required to do this. In the case where there is just one process in the data communication ring, I still use MPI to communication with itself. This case could be further optimized at the cost of some simplicity.

2 Analysis

The roof-line model will be evaluated for square grids of processes for simplicity though the analysis will generally apply to other cases. The total number of FLOPs used by this algorithm is the same as the number of FLOPs used in standard shared memory matrix multiplication. The number of times the data is passed between processes is $\sqrt{d} - 1$ if d is the number of processes. The total memory bandwidth will be the number of times data is passed between processes multiplied by the combined size size of matrices A and B and the memory bandwidth for standard matrix multiplication. This counts memory bandwidth associated with MPI communication. If the dimension of A is $n \times n$, the dimension of B is $n \times n$, and the bigger of the two previous block dimensions is d, the total number of FLOPs used is $2n^3$ and the total memory that must be transfered in bytes is

$8*3*(\sqrt{d}-1)*n^2+8*3*n^2 = 24*\sqrt{d}*n^2$. The arithmetic intensity is $\frac{2n^3}{24*\sqrt{d}*n^2} = \frac{n}{12*\sqrt{d}}$ The CPUs used in the CCV are Intel Xeon model number 6126. According to the Intel specification found here: the CPU has 652.8 peak GFLOPS. The peak memory bandwidth should be about 100 gigabytes per second based on the memory bandwidth of other similar CPUs. As such, the ratio of peak FLOPS to peak memory bandwidth is approximately ($\frac{652.8}{100} = 6.528$). So long as $\frac{n}{12*(d+1)}$ is greater than 6.528, the computation will ideally be FLOP limited. For values of n in the range 2,048 to 16,384 and d between 1 and 64, the computation should be FLOP limited.

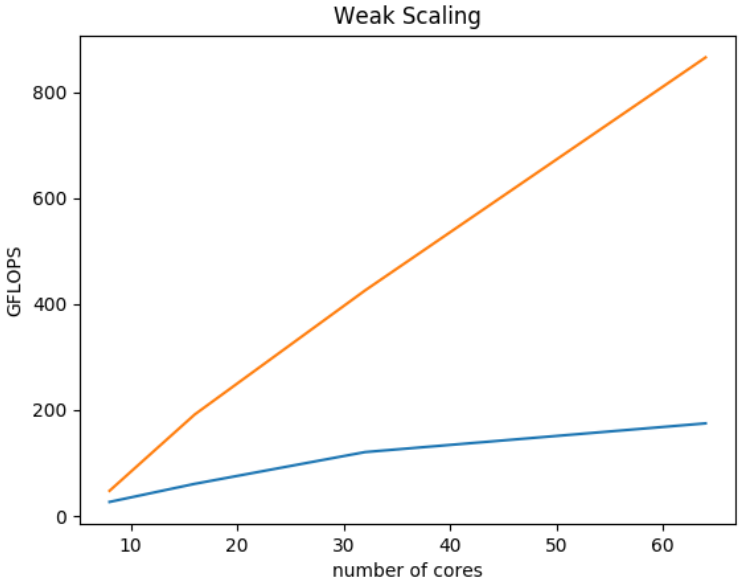
This previous computation doesn't factor in that MPI message passing will be substantially slower than the peak memory bandwidth. The total MPI memory bandwidth is $24*(\sqrt{d}-1)*n^2$. I attempted to test the between process memory bandwidth, but the bandwidth seems too low (around 4 GB/s). If this is assumed to be accurate, then the computation has an effective MPI ratio of peak FLOPS to peak memory bandwidth of ($\frac{652.8}{4} = 163.2$) The effective MPI arithmetic intensity is $\frac{n}{12*(\sqrt{d}-1)}$, so MPI communication will be limiting for some values of n and d within the scope of this computation. For instance, the computation will theoretically be MPI bandwidth limited for $n = 2048$ and $d = 9$ and $n = 4096$ and $d = 16$.

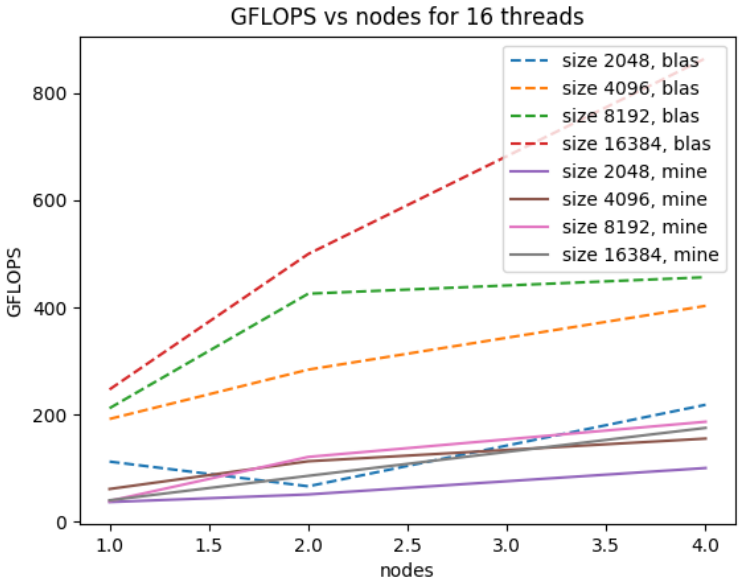
3 Results

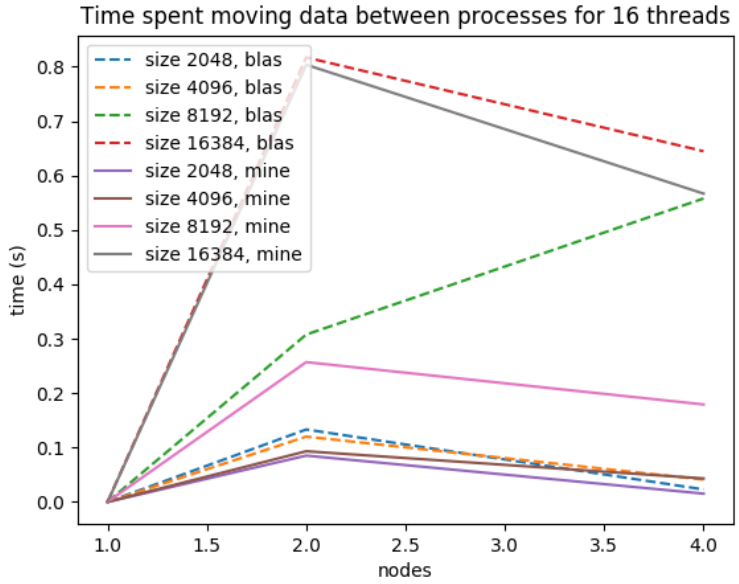
A table with data from all runs can be found in the appendix.

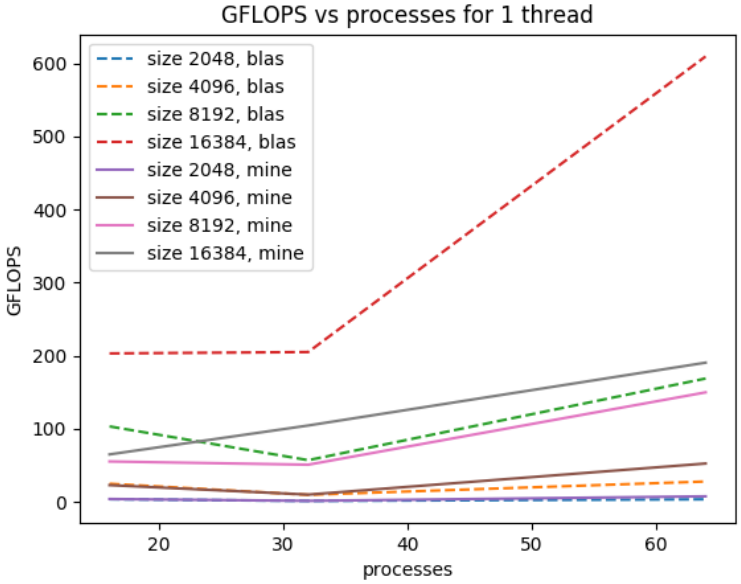
All testing was done on the CCV. The distributed matrix multiplication was evaluated using the code I wrote for the last project and using BLAS (OpenBLAS specifically). The performance is far better using BLAS. I am not entirely sure why my code of shared memory matrix multiplication is so much worse. A variety of experiments were conducted, but the figures show the result of using 16 threads and different numbers of nodes and 1 thread and different numbers of processes. Note that 16 processes corresponds to 1 node, 32 processes corresponds to 2 nodes and 64 processes corresponds to 4 nodes. I was unable to get SLURM to allocate threads which run on different cores and processes on a single node at the same time. Note that the BLAS matrix multiplication gets substantially faster with increases in the matrix size. As such, adding more processes is problematic even without considering the overhead of the message passing. Note that square grids were more effective than none square grids. The distributed matrix multiplication scales quite well. 4 nodes with 16 threads is just less than 4 times faster than 1 node with 4 threads. The optimal model seems to be using 1 MPI rank per CPU and a thread for each core. The maximum FLOP rate I achieved with 4 nodes was 865 GFLOPS using BLAS and 194 GFLOPS using my shared memory matrix multiplication implementation. Strong scaling can be seen in the plots which display node/process count vs GFLOPS.

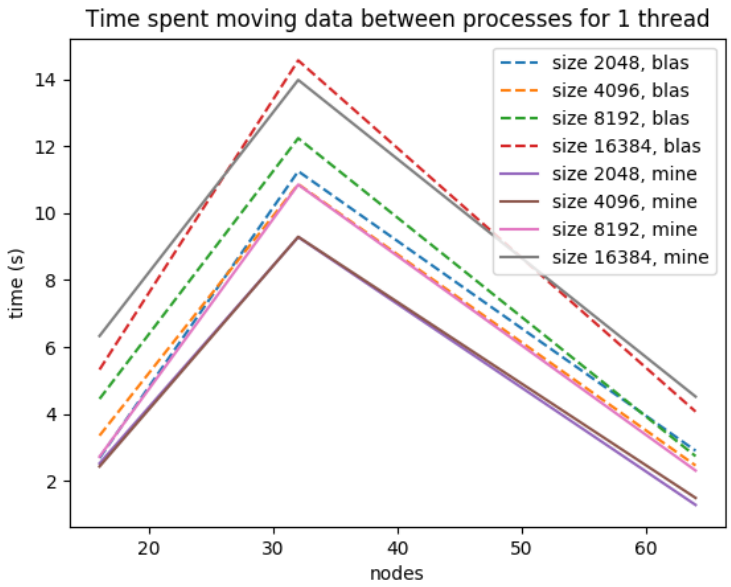
Figure 2: The first value is using 8 threads and 1 node. Other values use increasing numbers of nodes and 16 threads






Time spent moving data between processes for 16 threads
size 2048, blas
size 4096, blas
size 8192, blas
size 16384, blas
size 2048, mine
size 4096, mine
size 8192, mine
size 16384, mine
time (s)
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
1.0
1.5
2.0
2.5
3.0
3.5
4.0
nodes


GFLOPS vs processes for 1 thread
size 2048, blas
size 4096, blas
size 8192, blas
size 16384, blas
size 2048, mine
size 4096, mine
size 8192, mine
size 16384, mine
GFLOPS
0
100
200
300
400
500
600
20
30
40
50
60
processes



A Additional data

The full data can be seen in the below two tables. The column used refers to which shared memory matrix multiplication algorithm was used. The setup, wait, copy, and dgemm columns refer to the total spent doing each of those tasks.

threads	processes	nodes	n	used	gflops	setup	wait	copy	dgemm
8	1	1	2048	blas	47.7667	0	0	0	0.278573
8	1	1	2048	mine	26.7809	0	0	0	0.506932
8	1	1	4096	blas	38.6369	0	0	0	3.09188
8	1	1	4096	mine	29.9626	0	0	0	4.05046
8	1	1	8192	blas	44.9202	0	0	0	21.914
8	1	1	8192	mine	19.9453	0	0	0	50.4504
8	1	1	16384	blas	72.9442	0	0	0	108.798
8	1	1	16384	mine	20.7888	0	0	0	390.516
16	1	1	2048	blas	112.098	0	0	0	0.0883248
16	1	1	2048	mine	36.4969	0	0	0	0.383815
16	1	1	4096	blas	191.71	0	0	0	0.458686
16	1	1	4096	mine	60.9018	0	0	0	1.89523
16	1	1	8192	blas	211.796	0	0	0	4.03631
16	1	1	8192	mine	38.566	0	0	0	25.7391
16	1	1	16384	blas	246.826	0	0	0	29.9783
16	1	1	16384	mine	39.8028	0	0	0	202.615
8	2	2	2048	blas	45.9791	0.00413686	0.0906128	0.0353779	0.0695233
8	2	2	2048	mine	50.8592	0.00405215	0.0533599	0.00123115	0.227584
8	2	2	4096	blas	125.556	0.0152528	0.258129	0.0046053	0.504755
8	2	2	4096	mine	61.3704	0.0158789	0.137663	0.00471361	1.81844
8	2	2	8192	blas	216.728	0.126833	0.0931871	0.0211473	3.75363
8	2	2	8192	mine	61.1919	0.0629568	0.123547	0.0192578	16.0993
8	2	2	16384	blas	260.433	0.499814	0.196414	0.0772465	28.9567
8	2	2	16384	mine	40.922	0.50382	0.196392	0.0780313	197.686
16	2	2	2048	blas	65.9625	0.00929961	0.0203101	0.103526	0.0688881
16	2	2	2048	mine	50.7611	0.00404378	0.0162771	0.0648065	0.201195
16	2	2	4096	blas	283.817	0.0164215	0.0536486	0.0499838	0.216928
16	2	2	4096	mine	112.667	0.0150878	0.0471517	0.0309087	0.937368
16	2	2	8192	blas	425.66	0.0576426	0.229659	0.0202813	1.57966
16	2	2	8192	mine	120.798	0.0578863	0.116619	0.0824271	7.82413
16	2	2	16384	blas	499.753	0.252816	0.45899	0.105163	13.9953
16	2	2	16384	mine	85.4951	0.231671	0.470755	0.101114	93.4414

threads	processes	nodes	n	used	gflops	setup	wait	copy	dgemm
1	16	1	2048	blas	3.58378	4.1418e-05	1.16057	1.52322	1.03319
1	16	1	2048	mine	3.86192	2.5566e-05	1.1372	1.3869	0.96981
1	16	1	4096	blas	24.7603	6.93e-05	1.39897	1.95591	1.32715
1	16	1	4096	mine	22.8327	4.0539e-05	0.537606	1.89522	2.84192
1	16	1	8192	blas	103.192	4.1638e-05	2.54013	1.91146	5.2546
1	16	1	8192	mine	55.2286	5.2436e-05	1.67128	1.05411	14.9515
1	16	1	16384	blas	203.11	5.3044e-05	4.20376	1.12288	31.5746
1	16	1	16384	mine	64.8972	4.9335e-05	5.31518	1.01614	118.633
4	16	4	2048	blas	24.0798	0.00023221	0.362493	0.05429	0.0778591
4	16	4	2048	mine	50.1262	0.000216744	0.115041	0.000680538	0.0671567
4	16	4	4096	blas	175.215	0.000530238	0.341215	0.00414194	0.181162
4	16	4	4096	mine	157.914	0.000567374	0.150019	0.00368773	0.543925
4	16	4	8192	blas	531.028	0.0185465	0.429401	0.0159047	1.10177
4	16	4	8192	mine	189.281	0.00274346	0.342857	0.0156812	4.8044
4	16	4	16384	blas	886.162	0.0397578	0.977789	0.0632309	7.4089
4	16	4	16384	mine	194.061	0.0397732	2.71324	0.058316	38.1079
1	32	2	2048	blas	1.22459	0.00143051	3.93324	7.31769	1.30482
1	32	2	2048	mine	1.28475	0.000284275	3.48279	5.80199	1.86865
1	32	2	4096	blas	9.55717	0.000400333	5.09728	5.76809	1.90973
1	32	2	4096	mine	9.87654	0.014119	3.7044	5.57485	2.33138
1	32	2	8192	blas	57.0062	0.000258864	1.35948	10.8778	5.33454
1	32	2	8192	mine	50.8025	0.0558136	3.37863	7.41753	8.34072
1	32	2	16384	blas	204.967	0.000396412	7.66612	6.89602	23.6195
1	32	2	16384	mine	104.449	0.000522341	4.08164	9.89758	64.145
16	4	4	2048	blas	218.194	0.0049528	0.00305767	0.0149129	0.0301758
16	4	4	2048	mine	100.085	0.000552069	0.0137194	0.000945043	0.12254
16	4	4	4096	blas	402.94	0.0105919	0.0173653	0.0130606	0.209621
16	4	4	4096	mine	155.047	0.0102802	0.0246147	0.00821585	0.624417
16	4	4	8192	blas	456.375	0.0731058	0.456121	0.0285154	0.832948
16	4	4	8192	mine	186.511	0.0442442	0.107235	0.0276274	5.0225
16	4	4	16384	blas	865.319	0.175643	0.364849	0.10408	6.50175
16	4	4	16384	mine	174.993	0.0401204	0.450364	0.076162	45.3777
8	4	4	2048	blas	68.0084	0.0037108	0.0722899	0.0314687	0.0666243
8	4	4	2048	mine	74.3238	0.00282934	0.0384147	0.000488909	0.112573
8	4	4	4096	blas	284.548	0.00294532	0.105617	0.00232854	0.218599
8	4	4	4096	mine	115.627	0.00287362	0.057027	0.00240996	0.922932
8	4	4	8192	blas	375.233	0.0102069	0.552328	0.0091469	1.81112
8	4	4	8192	mine	126.912	0.0111452	0.234846	0.00908523	7.45041
8	4	4	16384	blas	525.905	0.0437044	1.03468	0.0372861	11.8671
8	4	4	16384	mine	78.7252	0.043419	0.427618	0.0370708	83.3374
1	64	4	2048	blas	3.44022	0.000858813	2.02787	0.874364	1.63681
1	64	4	2048	mine	7.50764	0.0452747	0.476482	0.761887	0.299881
1	64	4	4096	blas	27.7975	0.000193663	1.46668	0.993756	1.63706
1	64	4	4096	mine	52.4181	0.000315995	0.294509	1.19871	0.718232
1	64	4	8192	blas	168.74	0.000170699	1.8775	0.861384	2.49915
1	64	4	8192	mine	149.795	0.000183126	1.77825	0.531122	3.81381
1	64	4	16384	blas	609.544	0.000256883	3.16026	0.91117	8.81494
1	64	4	16384	mine	190.45	0.000364729	3.5987	0.920173	37.6115
8	8	4	2048	blas	44.3405	0.00325842	0.0878071	0.15857	0.0633794
8	8	4	2048	mine	65.1491	0.00102803	0.0770617	0.026423	0.0654919
8	8	4	4096	blas	234.351	0.000413269	0.0301224	0.248752	0.23706
8	8	4	4096	mine	152.592	0.00261909	0.0449692	0.0849738	0.590471
8	8	4	8192	blas	457.655	0.000947662	0.482697	0.0959676	1.37528
8	8	4	8192	mine	171.819	0.000945103	0.328512	0.0548389	5.34775
8	8	4	16384	blas	535.082	0.0752286	0.546276	0.151114	13.0185
8	8	4	16384	mine	173.633	0.0423695	2.96215	0.183347	43.419