Vivek Sivakumar
UID: 303652195

Lab 2 Report

In order to perform blocked matrix multiplication in MPI I partitioned the matrix A into chunks of (n / np) rows and sent each processor a chunk, then I sent all of matrix B to each processor. Once this was done each processor computed the local product (local C) by multiplying each of its rows of A with B. Once a processor finished, it sent back its local product to the root process (process 0) to be combined into the output C matrix.

To accomplish this I used the MPI_Scatter routine to scatter (n/np) chunks of A to each processor. I sent B to all processors using the MPI_Bcast routine, once both of these steps were done I performed the blocked matrix multiplication that I used in lab1 to compute the local C. Once all local C's were computed I used the MPI_gather routine to gather all local C's into the final C matrix.

Since I utilized the MPI_Scatter and MPI_Gather routines, I did not use a blocking and non-blocking send/receive. My intuition is the non-blocking would be faster if useful computation was done before a process attempted re-send or re-receive. For instance if I was using send to broadcast B instead of MPI_Broadcast I might have a continuous while loop through all processes and attempt a non-blocking send of B to the first process in the loop. If the send works I remove it from the list of processes in the while loop, if not I attempt to send to the next process instead of simply blocking and waiting for the send to complete. In the case where different processes could finish computing the local matrix multiplication at very different times, I would expect the non-blocking approach to be significantly faster.

Scaling With Number of Processors for Matrix Size: 2048, block size = 64

np	GFlops
4	12.32
8	19.21
16	19.43
32	21.95

Scaling With Number of Processors for Matrix Size: 4096, block size = 64

np	GFlops
4	16.16
8	28.25
16	31.37

32	43.14

I did not see a linear speedup (doubling of performance) when I doubled the number of processors. This is what I expected, doubling the number of processors: increases program overhead (for spawning & finalizing more processes in the MPI runtime), it increases the communication time (for MPI_Scatter, MPI_Broadcast, and MPI_Gather), it can also increase the process waiting time at barriers (such as after MPI_Scatter/Broadcast/Gather). With all of these increasing costs, increasing the number of processors had diminishing returns on increasing overall speedup.

When optimizing for block size I used the same thought process I had for lab 1, mainly I wanted blocks of A,B, and C to fit in fast memory so one iteration of the outermost loop would have no cache misses. From using lscpu I saw that the L1 cache is 32 kB or 32768 bytes and that the L2 cache is 256 kB or 262144 bytes. The formula for optimal block size should be:

(BLOCK_SIZE ^ 2) * (sizeof(float)) <= bytes in fast memory

Case 1: L1 is the only fast memory:

(BS^2) * 4 <= 32768
BS <= 90.50
BS = 64 B → to fit one block in L1
3*BS <= 90.50
BS <= 30.0
BS = 16 B → to fit 3 blocks in L1
BS = 32 B → to fit 2 blocks in L1

Case 2: L1 and L2 are fast memory:

(BS^2) * 4 <= 32768 + 262144
BS <= 271
BS = 256 B → to fit 1 block in L1 & L2
3*BS <= 271
BS <= 90.5
BS = 64 → to fit 3 blocks in L1 & L2
BS = 128 → to fit 2 blocks in L1 & L2

Case 3: L1, L2, and L3 are fast memory:

Unlikely because L3 is shared memory but will try 512 just in case

Performance with Block Size for Matrix Size: 4096, np =16

Block Size	GFlops/s (1-5 trials)	Avg GFlops/s
16	7.90, 7.69, 7.35	7.55

32	23.04, 23.08,26.79	24.30
64	31.37, 31.98, 32.09, 31.56, 38.14	33.03
128	37.28, 58.47, 48.46, 44.42, 44.31	**46.588**
256	40.32, 32.77, 44.59, 40.50, 64.80	**44.596**
512	16.65,10.13,13.38	13.39

Interestingly the results do not seem to support fitting 3 blocks in the cache at once, instead favor either 2 or 1. The results for a block size of 128 and 256 are very close but I decided to use 256 based on the general trend I saw in all trials I ran.

The MPI implementation obviously required more programming effort because I had to handle the sharing of resources between machines before matrix multiplication instead of relying on an OpenMP pragma to handle the parallelization for me. For this relatively small-sized problem that easily fits on one machine I expected OpenMP to perform better because the MPI implementation adds significant communication costs that are not required in the OpenMP implementation. OpenMP also parallelizes at a much finer grain that MPI, each row of A was parallelized in my OpenMP implementation instead of chunks of rows like in MPI.

After looking at the data from the experiments I was highly surprised to see my MPI implementation outperform my OpenMP implementation. This suggests that the communication APIs in MPI are very efficient. It also suggests that the execution time for each iteration is so similar that having a granularity of chunks of iterations vs each iteration does not matter, in fact the increased data locality of the chunked version could explain some of the better performance. Finally in the MPI version each process had its own copy of C to operate on and hold in its cache for the entire local matrix multiplication. In my openMP version C was a shared resource and the appropriate row had to be freshly pulled into the cache at each iteration. Perhaps if I had optimized this portion of my OpenMP program it would have outperformed my MPI version.