Jason Lai
204995126
DIS 1A

CS133 Lab 2 Report

Computation and Data Partitioning:

In terms of the data, the data is partitioned among the processors as follows: for matrix A, we first divide the rows by the number of processors, so that each processor gets the same number of rows to do their portion of the computations. To find the exact number of bytes each portion is, we multiply the number of rows each processor gets by the column size of matrix A (the number of elements in each row). We use the MPI_Scatter API to then distribute the allocated chunks of matrix A to the “receive buffers” of each processor grouped in MPI_COMM_WORLD from the processor ranked 0. For matrix B, every processor needs the entirety of matrix B to do their calculations, since each processor has complete rows of A that they need to find the product for when multiplied with B. To send every processor matrix B, we broadcast the data using MPI_Bcast, where processor 0 broadcasts matrix B to all processors grouped by MPI_COMM_WORLD.

For the computations, every processor received the same number of rows of matrix A to multiply by matrix B. This means that every processor will compute that many rows of the final product matrix C, based on how the data was partitioned. They then store the final computed data in their respective product buffers allocated to them. A call to MPI_Gather then collects the computed data from every processor and builds the final matrix C at processor 0.

Theoretical Impact of Communication APIs:

On the sender side, we have APIs for buffered and unbuffered sends, blocking and non-blocking sends, and combinations of the two categories. First, we compare MPI_Bsend and MPI_Send (default), which are both blocking APIs, however the former is buffered while the latter is unbuffered. In theory, we would expect buffered sends to perform better than unbuffered sends, as the MPI_Bsend call does not need to wait for the receiver to verify it is okay to send data and will return immediately, allowing the sending processor to reuse the “data buffer” (buffer where the data to be sent is stored). This is because MPI_Bsend creates a copy of the data buffer and will send the data to the receiving processor at a later time when it is okay to send, allowing the buffered call to return immediately.

Now comparing MPI_Isend and MPI_Send, which are both unbuffered, but non-blocking and blocking respectively, we would expect MPI_Isend to perform better than MPI_Send. This is because the MPI_Isend call returns immediately without needing confirmation from the receiver that it is okay to send data. This allows for the sending processor to execute concurrent computation and communication, which is why performance is expected to be better than the blocking MPI_Send call. However, unlike the buffered send case, MPI_Isend does not allow for the sending processor to reuse the data buffer immediately; this is because there could be updates to the data buffer, while the data is still being transferred to the receiver. However, in the case of gemm, I do not think this poses an issue, since we are not updating matrices A or B.

The final send API we consider is MPI_Ibsend, which combines the non-blocking and buffered concepts. We expect this API to have the best performance, as the MPI_Ibsend works just like the buffered MPI_Bsend, however MPI_Ibsend will not block while the data buffer is being copied. This means that implementations using this must be wary of updates to the data buffer while it is being copied, but otherwise can execute computation concurrently with copying the data buffer (although MPI_Wait would be needed to check the status of the copy). After the data buffer has been copied, just like with MPI_Bsend, the sending processor can overlap computation with communication to absorb some of the overhead of the data transfer.

On the receiver side, we have MPI_Recv and MPI_Irecv for blocking and non-blocking receive's respectively. According to the MPI specification, non-blocking receive calls may allow for concurrent computation while the data transfer is still in progress, but this may be hardware dependent. If this were the case, then I believe MPI_Irecv would be more efficient than MPI_Recv. Otherwise, I think the performance difference would be negligible, as MPI_Irecv would need an MPI_Wait call to check the status of the data transfer, which blocks until MPI_Irecv returns. This blocking would probably have the same effect as just calling MPI_Recv.

Instead of these point-to-point communication APIs, I used collective communication APIs: MPI_Scatter, MPI_Bcast, and MPI_Gather, since it seemed like the most natural implementation (one group of processors with just one-to-all and all-to-one communication). From my understanding (using the resources from lecture and online searching), their routines are blocking with other caveats such as having buffers of the same size across the processors.

Performance Comparison:

Matrix Size / Trial	1	2	3
1024^3	0.0343728 s 62.4762 GFlops	0.0354199 s 60.6292 GFlops	0.033356 s 64.3808 GFlops
2048^3	0.131658 s 130.489 GFlops	0.134269 s 127.951 GFlops	0.132522 s 129.638 GFlops
4096^3	0.877629 s 156.603 GFlops	0.878455 s 156.455 GFlops	0.879841 s 156.209 GFlops

All of the above performance tests were done using make np=4, since that is the number of processors that the performance evaluation is based on.

From the results above we see that there are significantly different throughput numbers for different problem sizes, where the throughput increases dramatically as the problem size increases. I believe that this is primarily due to the communication overhead as a result of using MPI. As the problem size gets smaller, there are less pertinent computations performed i.e. less floating point operations executed in the matrix multiplication. This means that the proportion of time spent executing floating point operations to time spent communicating between processors is smaller than for larger problem sizes. Because a larger percentage of the total time for execution is spent on communicating, we see that this lowers our throughput. In contrast, as the problem size gets larger, we see that throughput increases, as there are more floating point operations that each processor must execute, so a larger portion of the total execution time is spent on pertinent computations. This effectively increases our throughput, as not nearly as much of the total execution time is used on communication as it is for the smaller problem sizes.

However, there is noticeably less of a jump in performance from 2048^3 to 4096^3 than for 1024^3 to 2048^3, and I suspect that this is due to some diminishing returns as the problem size increases past 4096^3. This might be caused by other sources of overhead that are also increasing as the dimensions of the problem increase. For example, copying the entire matrices to aligned buffers and then read from these buffers in the communication process may be contributing some nontrivial overhead as the problem size increases, which detracts from the percentage of the time spent on floating point operations, which may be diminishing throughput.

Scalability:

np / Trial	1	2	3
1	2.84239 s 48.3533 GFlops	2.84472 s 48.3137 GFlops	2.8444 s 48.3192 GFlops
2	1.51197 s 90.9003 GFlops	1.51392 s 90.7834 GFlops	1.51565 s 90.6801 GFlops
4	0.882462 s 155.745 GFlops	0.879025 s 156.354 GFlops	0.8773 s 156.661 GFlops
8	1.08322 s 126.879 GFlops	1.07709 s 127.602 GFlops	1.07323 s 128.061 GFlops
16	8.83286 s 15.56 GFlops	5.56878 s 24.6803 GFlops	11.2773 s 12.1872 GFlops
32	27.388 s 5.01822 GFlops	19.8354 s 6.92897 GFlops	22.295 s 6.16457 GFlops

My program appears to scale linearly up until 8 processors and beyond are used. When using more than 4 processors, we observe that there is performance dropoff, especially so at the 16 and 32 processor counts. I surmise that this is due to communication overhead when dealing with large numbers of processors, as managing and transferring data most likely contributes significant overhead when so many processors are used. Put another way, the root processor must scatter/broadcast matrices A and B to more processors and gather final results of matrix C from these processors, with each communication between the root and different ranked processors contributing its own overhead. And because there is a large amount of overhead from communicating, this increases our total execution time not spent performing pertinent computations, so the throughput and therefore the scalability decreases.

MPI Implementation vs. OpenMP Implementation:

In terms of programming effort, I observed that my Lab 1 OpenMP implementation was much easier than my Lab 2 MPI implementation. This is of course barring the loop transformations that I did that are consistent in both. For the OpenMP implementation, I found it much easier, since it was simply a single line of code: the OpenMP pragma with appropriate clauses that was used in order to utilize parallelism and obtain massive gains in performance. This is in contrast to the MPI implementation where we must first consider the different buffers needed to be sufficient for the MPI communication APIs, and while we did not need to implement it ourselves, the usage of aligned memory allocation also suggests there are more scenarios to consider with the MPI implementation. Then, for the MPI implementation, we needed to figure out what routines we would use to execute the communication between the root processor and all other higher ranked processors. This led to considerably more code having to be written for the MPI implementation than the OpenMP one. However, I will say that I did find the MPI implementation easier to understand in principle as to how parallelism is achieved, most likely because the data transfer concept felt familiar after having taken a networking course (CS118). This is opposed to the difficulty I had understanding how OpenMP and its clauses work when parallelizing for loops, as I was looking for different scheduling clauses and other methods to gain performance in Lab 1.

In terms of performance, the MPI implementation's performance is best when using 4 processors (~160GFlops), which is relatively close to the OpenMP implementation's best performance (~175GFlops), which is when 4 or 8 threads are used (I observed negligible difference between the two). However, the MPI implementation still performs worse than the OpenMP implementation, and scales significantly worse in regards to both the problem size and number of processes (threads in OpenMP's case). I think this is again because of the setup and communication overhead associated with using MPI. The MPI implementation not only takes time to allocate memory, but also spends time communicating data between the root processor to each of the higher ranked processors, since the assumption is that each processor has its own memory that initially does not have data like in separate machines. This is unlike the OpenMP implementation, where the threads have access to the same shared arrays in memory, so time doesn't need to be spent copying the arrays to a buffer then sending them to the buffers of each processor. As a result, not only does the MPI implementation spend a lot of time communicating, but also scales poorly as each added processor means more communication and more overhead. In contrast, OpenMP's sources of overhead are context switching, scheduling, etc, but is still significantly less than MPI's overhead. In the first place, MPI was designed for computations across different systems, whereas OpenMP is contained to a single system. However, in this lab, we use MPI on a single system, so in some ways it is just a less efficient OpenMP.