Lab 2 - CPU w/ MPI: General Matrix Multiplication (GEMM)

Dennis Zang
704766877

Please briefly explain how the data and computation are partitioned among the processors.
Also, briefly explain how the communication among processors is done.

In summary, to perform the matrix multiplication, I took the original 2 matrices (A and B), and partitioned them into different powers of two (1/2, 1/4, and so on). The total number of partitions is dependent on the total number of processors I am to work with (ie if I am to work with 8 threads, I will partition one in 1/2 and the other in 1/4). THe partitioning assumes that the number of processes is a power of 2. For the matrices, as A and B are originally only accessible on process 0, I divided them up on process 0 (and transposed B for easier communication and locaility for blocked matrix multiplication later on). This was done by allocating local buffers for each submatrix and manually using non-blocking send and recv functions to transport them to "edge matrices" (if we are to image the processes as a grid topology, we would have the topmost and rightmost edges adjacent to 0 be the edge to which I scattered the partitions of the matrices A and B). Then, I broadcasted these submatrices from these edge processors to other processors in the same row (for A) and column (for B). With each process having a sub-partition of A and B, I had each process get a "sub-product" of the matrices, which I them concatenated back together using gather and send.
(Note: there were a few functions that I had troubles using, especially with scatter [due to segmentation faults from memcpy internally.)

Please analyze (theoretically or experimentally) the impact of different communication APIs
blocking: MPI_Send, MPI_Recv
buffered blocking: MPI_Bsend
non-blocking: MPI_Isend, MPI_Irecv
Attach code snippets to the report if you verified experimentally. Please
choose the APIs that provides the best performance for your final version.

(blocking)

Blocking would definitely be the least ideal of the three, as this would force the communication operations to other processes to be sequential. Although this would guarantee order, this also means that the process would progress in discrete steps, and wouldn't be able to work as efficiently (like through pipelining or in parallel).

(buffered)

Buffered would allow the sending operation to work asynchronously (as the send functions can return without being forced to finish their operation). However, note that the recv operation would still be blocking. This can be a huge issue if buffered data from later operations manage to arrive first, but the recv function will block and wait for the current data to arrive, adding a bottleneck to the program in that it cannot work asynchronously completely.

(non-blocking)

Non-blocking would basically be like buffered, except that the recv operation would not be forced to block. The only issue with this that it will not be easy to check if a send and its complementary recv function have both succeeded. However, due to the truly asynchronous nature, this would be the most time efficient approach for the program.

(I did try out using buffered operations to some degree, as shown in some of the commented out lines of code.)
(For instance...)

```
/*
        int bufsize;
        MPI_Pack_size(kI * kK, MPI_FLOAT, colComm, &bufsize);
        bufsize += kI * MPI_BSEND_OVERHEAD;
        float *fbuffer = (float*) malloc(bufsize * sizeof(float));
        MPI_Buffer_attach(fbuffer, bufsize);
        */
        if(RANK == 0)
        {
                for(int i = 0; i < kI; i++)
                {
                MPI_Request request;
                MPI_Isend(a[i], kK, MPI_FLOAT, (i / ARows), i, colComm, &request);
                }
        …..
```

Please report the performance on three different problem sizes (1024^3, 2048^3, and 4096^3). If you get significantly different throughput number for the different sizes, please explain why.
(using n = 4)

(1024^3)
Run parallel GEMM with MPI
Time: 0.336346 s
Perf: 6.38475 GFlops

(2048^3)
Run parallel GEMM with MPI
Time: 2.65177 s
Perf: 6.47864 GFlops

(4096^3)
Run parallel GEMM with MPI
Time: 22.1668 s
Perf: 6.2002 GFlops

There doesn't seem to be a significant difference in my case (I was limited in time and couldn't get some of the MPI functions I wanted to use to work). In this case, this can likely be attributed to a skewed partitioning of the matrices (not very square), the transposing of matrix B (which can be very costly in time), and not much use of locality (as we partitioned all parts of A and B among all the processors to manage at once).

Please report the scalability of your program and discuss any significant non-linear part of your result. Note that you can, for example, make np=8 to change the number of processors. Please perform the experiment np=1, 2, 4, 8, 16, 32.
(using 4096^3)

(np = 1)
Run parallel GEMM with MPI
Time: 80.7671 s
Perf: 1.70167 GFlops

(np = 2)
Run parallel GEMM with MPI
Time: 41.1953 s
Perf: 3.33628 GFlops

(np = 4)
Run parallel GEMM with MPI
Time: 24.1828 s
Perf: 5.68332 GFlops

(np = 8)
Run parallel GEMM with MPI
Time: 14.0609 s
Perf: 9.77452 GFlops

(np = 16)
Run parallel GEMM with MPI
Time: 19.5509 s
Perf: 7.0298 GFlops

(np = 32)
Run parallel GEMM with MPI
Time: 23.592 s
Perf: 5.82567 GFlops

There does seem to be a slight amount of non-linearity around np = 16 and greater. This would likely be attributed to the overhead from running so many threads at once.

Please discuss how your MPI implementation compare with your OpenMP implementation in lab 1 in terms of the programming effort and the performance. Explain why you have observed such a difference in performance (Bonus +5).

[MPI]
programming effort:

The gist of MPI is meant to visualize how all the processes act at all points in time, and have each process perform different tasks by communicating different information with each other and designating different groups to different tasks at once. In a way, this is much more difficult that OpenMP, in which the main goal would be to parallelizing repetitive action in loops. However, one advantage over OpenMP would be that different tasks can be performed at the same time.

performance:

In the scope of this lab, because we can use MPI to allocate different processes to work on different parts of the data, we probably could have been able to get a bit more performance out than OpenMP (as we can then more easily combine said data through reductions, gathering, scattering, etc...). However, I wasn't able to do that in this lab (I was barely able to get send and recv working :\ ). If I were able to, the performance would have likely been much better.

[OpenMP]
programming effort:

The gist of OpenMP would be to take a long, repetitive task, and use parallelism to shorten it. As mentioned above, this is more intuitive that designating different tasks to different processes at once. However, one advantage would be that it is easier to make use of locaility (as each process performing similar tasks, meaning data sharing is a lot more likely).