Homework 6

We will write a striped matrix multiply C = AxB in OpenMP and MPI. Code for a basic matrix multiply is provided in the file mm.c, you may use and modify this for your program, but you don't have to.

To make life simpler, the arrays to be multiplied are all square, i.e., the number of rows and columns are the same, and the number of threads or processes should evenly divide the number of rows and columns. Use 1, 4, 8 and 16 threads and processes for timing the program. The programs will also be self-initializing, i.e., you do not need to read input data from a file but rather each MPI process will initialize its B and C array elements. I initialized them with the process id for MPI so that I could check the value being sent and know which processor it started on, to help with debugging.

For the OpenMP program, the conversion should be straightforward, trivial even. Use omp_get_wtime (https://www.openmp.org/spec-html/5.0/openmpsu160.html) for timing.

For the MPI program, let *stripes* be the number of rows of the A matrix that each process multiplies times stripes columns of the B matrix. *Stripes* is equal to the number of rows and therefor process p will compute the p*stripes:(p+1)*stripes-1 rows of the C matrix.

Initially, each process should have its *stripes* rows of the A matrix, and the corresponding columns of the B matrix. It will have all columns of the A matrix, and all rows of the B matrix. It will have *stripes* rows of the C matrix and all columns of the C matrix. After it computes the C elements corresponding to these rows and columns, it will send its B matrix columns to the left (process p-1 for all processes except for process 0, and process P-1, the highest numbered process, for process 0) and receive more B matrix columns from its right process (process p+1 for all processes except for process P-1, which will receive from process 0).

After the program is completed, each process will send the rows of C it computed to process 0, which will form and print the entire result C matrix.

For timing the program, use MPI_Wtime (https://www.mcs.anl.gov/research/projects/mpi/tutorial/gropp/node139.html).

Instructions on how to compile an MPI job to run on scholar can be found here: https://www.rcac.purdue.edu/knowledge/scholar/compile/mpi. I would suggest using the Intel compiler as they make very good compilers.

Instructions on how to run an MPI job on scholar can be found here: https://www.rcac.purdue.edu/knowledge/scholar/run/examples/slurm/mpi

If you used the MPI compiler to compile your job, change the .sub script as follows

```
#!/bin/bash
# FILENAME:  mpi_hello.sub
#SBATCH --nodes=4
#SBATCH --ntasks-per-node=10
#SBATCH -t 00:01:00
#SBATCH -A scholar

srun -n 40 ./mpi_hello
```

srun -n 40 ./mpi_hello should become *srun **--mpi=pmi2** -n 40 ./mpi_hello*.

Change the *–nodes=40* in the line above to –nodes=1 (and 4 and 8 and 16) to run with different numbers of processes. We will only need one node, and you will get faster turnaround on your jobs.

Some hints/caveats:

One of the challenges of the MPI part of the assignment is keeping track of which parts of the C matrix is being computed with the most recently received B data. As you get data from different processes, you will need to keep track of what parts of the C matrix to fill in. I debugged my code without communication first, on my laptop, and then added the MPI stuff and re-debugged it using Scholar. I think it went faster that way. When programming the solution this was the hardest part of the program, in my experience.

When doing your sends of the B array data, remember that you *cannot* have every processor

Send data to its left processor
Receive data from the right process

You will likely deadlock. Rather you need to have all of the even processes send to the odds, which receive, followed by all of the odd processes sending to the even, which receive. Of course, you can have the odd processes send first if you want, just don't have everyone send at once. You could also post non-blocking receives for every process, following by sends and then a wait to make sure the receive finished.

You will need to use separate buffers to send data from the B array and to receive B array data. I.e, in the above, if the odd processors receive even processor's sections of the B array into a single B array, they will not have their B array elements to send to the odd processors.

I used an MPI_Gather to gather all of the C arrays into the global C array on process 0. When doing this, don't use the same C buffer for the sendbuffer and recvbuffer.

Finally, when doing a send, specify 1 as the tag, and then specify MPI_ANY_TAG for the receive. Specifying MPI_ANY_TAG in a send will get an error in a standard conforming MPI implementation.

What to turn in:

Turn in two directories, OpenMP and MPI, which contain their respective programs, and a .txt file that contains the timing for 1, 4, 8 and 16 threads or processes. The timings should be before any printing you might do.

For MPI, you can run all processes on a single node – each process will be assigned a core. If necessary, scale your problem size so that the 1 thread/processor time runs in under a minute. I found that 2400x2400 matrices give me a running time of about 45 seconds on 1 process on Scholar with MPI.

For MPI, use a barrier before getting the initial time and immediately before getting the final time, and print out the process 0 time rather than the times for all processes