Parallel Program 2

Jennifer Kulich

November 16 2020

1 First Program- Ping Pong Communication

1.1 Description

This program will send a message from process 0 to process 1 and then straight back to process 0. There are variable message lengths and variable time the message passing for a particular length is run.
There was some help from: `https://edoras.sdsu.edu/~mthomas/sp17.605/lectures/MPI-Comms-Perf.pdf` on slide 14 AND
`http://www.csl.mtu.edu/cs4331/common/PPong.c`

1.2 What was Submitted and How to Run

Just ping_pong.c and a Makefile were submitted. To run, you can do 'make clean;make all'. Then, you type in 'mpiexec -np 2 ./ping_pong ¡number of times to pass the message¿ ¡size of the message in bytes¿
Side note: please do not put more than 2 processes.... You will get the timed result, but the program will hang.

1.3 Algorithms and Libraries Used

There was nothing super complicated used in this program. The mpi.h library was used which allowed for MPI_Send, MPI_Recv, MPI_Isend, MPI_IRecv, and MPI_SENDRECV.

1.4 Functions and Program Structure

There was only one function- main. It starts by initializing MPI and all that fun MPI jazz. A gather is done which will do a gather from all processes. If the rank was 0, this meant the message has to be sent to process 1 and received back. I started with the blocking, started recording the time, did a send a receive, did this as many times as the user specified, stopped recording the time, and then printed the elapsed time. I did the same for the non blocking except I had to throw waits in there. I then did a MPI_Sendrecv. If the rank was non 0, we could assume we were first receiving a message and then sending it back to

process 0. This was similar to what I did if the rank was 0, but there was no timing involved- this was part of the timing already. I then did a MPI_Finalize to shut down MPI.

1.5 Testing and Verification

Testing started with using the message passing code given in class (blocking) and then modifying it so only messages were passed from process 0 to 1 and then back to 0. Once I knew that was happening properly, I added timing and loops with variable message lengths. Once I had that working, I was able to move onto the non-blocking message passing.

1.6 Data

I went through a recorded different lengths and sending it a variable amount of times (results in Figure 1).
I eventually had to stop because the data I was getting was super inconsistent. I would have to run it multiple times before I could even consider what the timing most likely was. This made collecting data pretty difficult.

Blocking	Length->					
	Time run	2	4	8	16	32
	2	9562	8154	13995	30263	22997
	4	16992	21492	23061	15859	19744
	8	16748	20775	16501	16249	16441
Non blocking	Length->					
	Time run	2	4	8	16	32
	2	11025	8912	6994	13656	9992
	4	4196	9989	8940	10624	9995
	8	8535	10215	5965	7499	6795
MPI_Sendrecv	Length->					
	Time run	2	4	8	16	32
	2	8497	4209	5998	13884	9006
	4	10	9985	15228	7373	9467
	8	9853	2774	8644	7019	5992

Figure 1: Timing of blocking, non-blocking and MPI_Sendrecv in microseconds

1.7 Analysis of Data

Based on the data, MPI_Sendrecv seems to be the fastest, then the non-blocking send and receive. The slowest seems to be the blocking. One things I noticed when running tests was that the more I ran things, the 'faster' the sending/receiving seemed to be- this makes sense because the data seems to show that bigger messages sent multiple times take 'less' time. For example, in the right most column of each table, the average time it takes to send and receive messages decreases as more runs are made.
So I know you asked us to calculate the latency and bandwidth using linear regression, but I was having issues with excel, so I wasn't able to do that.

2 Second Program- Game of Life

2.1 Description

The Game of Life simulates cellular automation. Each cell in the grid is either alive or dead. If a cell has less than 2 or greater than 3 alive neighbors, it will die. If a dead cell has exactly 3 alive neighbors, it will suddenly become alive. All cells are updated simultaneously.

2.2 What was Submitted and How to Run

Two files were submitted: life.c and the Makefile. To run, you can do 'make clean;make all'. Then, you type in 'mpiexec -np ¡number processes¿ ./life ¡number of live cells¿ ¡iterations to play the game¿ ¡what iteration to print the board¿ ¡number of rows¿ ¡number of columns¿' (that's a lot to put in!)
Side note: for the number of processes, please put in the number of rows!

2.3 Algorithms and Libraries Used

I found some super helpful code `http://www.shodor.org/petascale/materials/distributedMemory/code/Life_mpi/` that helped me get an idea for how things worked. I was able to use a lot of concepts from this. In the end though, the way it was done didn't really sit right with me. I realized it was sort of convoluted, and we were shown a better approach in class.This meant that I sort of redesigned it- using MPI_Barrier to make sure things were done in the correct order, the graph was still m x n, etc. It took a lot of reworking haha, and I have a feeling it could be done better.
To create a random graph, I would loop through the graph of cells. If it was a dead cells, a random number would be generated, and if it was even, the cell would become alive. This was done until the number of alive cells in the graph matched what the user input for number of alive cells.

2.4 Functions and Program Structure

The program will first initialize and randomize a graph with the size the user passed in. It will then print the initial graph so the user can see what they started with. From there, the program enters a for loop which runs for the number of iterations the user input. If the iteration is a multiple of the kth graph to print, the graph will be printed. Then, the graph will be evaluated based on the game of life rules. Once all of the iterations have been completed, the graph will be printed again so the user can see the end result, the graphs are freed from memory, MPI is 'finalized', and that's the end.

2.4.1 initializeGraph

This function will initialize a graph with the correct M and N input with buffers in rows and columns.

2.4.2 createRandomMatrix

This will go through the graph and fill it with the correct number of alive cells. This is done by looping through the graph, and if the cell is dead, a random number is generated. If the random number is even, the cell will be alive. If not, the cell will be dead. We loop through the graph until there are the correct amount of alive cells in it.

2.4.3 getNewGraph

This function will

2.4.4 printSolution

This function will print out the graph without the dead boarders.

2.4.5 freeGraph

This function will free up a 2D graph passed to it.

2.5 Testing and Verification

I made a 5x5 graph with 20 alive cells, and I ran it for 2 iterations printing each iteration. This worked perfectly because it almost always tested the "bringing a cell back from the dead" and killed off a bunch of cells showing a simultaneous update. I guess that's all of the verification I really had to do since testing it this method kind of made sure everything was done.