Ryan Devaney

CS415

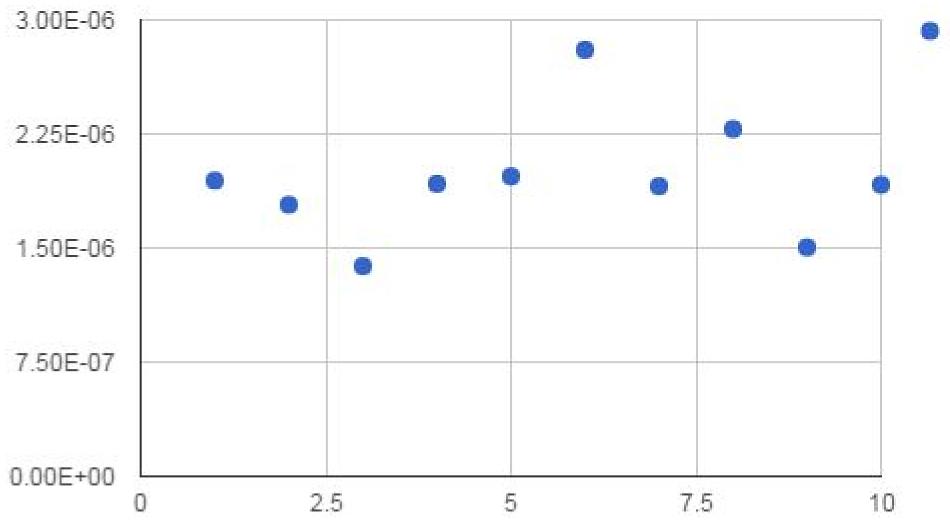
2/23/17

Project 1 - Ping Pong

This project had us send and receive messages from different processes. There were three different parts to this assignment. The first was sending a message from one processor to another processor in the same box. Then sending a message back to the original processor. The second part was the same as the first, but sending the message to a different box. The third part was sending more and more integers until we could find a jump in the time it takes for a processor on a different box to receive them.

Part One-

For the first part of this project, I sent a message using MPI_SEND and MPI_RECEIVE between two processors on the same box. Since they were on the same box, it used the shared memory between them and the time it took to send a receive a message back was very short. In order to achieve the message sending and get a proper timing, I started a timer (WTIME) and sent a message from the master processor to a slave processor. Then the slave received the message and sent one back. Once the master received the message back another timer was started. The second timer was then subtracted from the first in order to get the time it took between them. This process was repeated 1000 times and an average was taken in order to prevent outliers from skewing the time it took to ping pong a message. Some problems that I ran into were understanding some of the MPI datatypes and what they were used for such as MPI_STATUS.

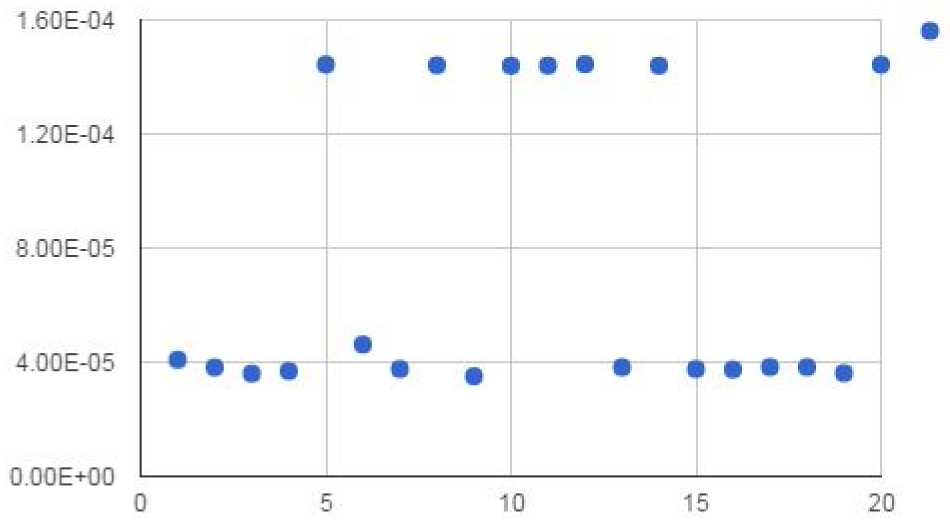


Scatterplot showing 10 trials of ping ponging messages on the same box

This graph shows several trials of sending data back and forth between two processors on the same box. As we can see, the data is pretty uniform with a few outliers. If we remove the high and low numbers in this graph we get an average time of 1.89 x 10^{-6} seconds. This number is about the time it takes to sent data to a processor and have that processor send it back on the same box.

Part Two-

The second part of the project used the exact same code as part one. The only difference was using -N2 in the srun statement. This causes SLURM to use two processor on different boxes. Since there had to be some network communication between the two boxes, the time increased.

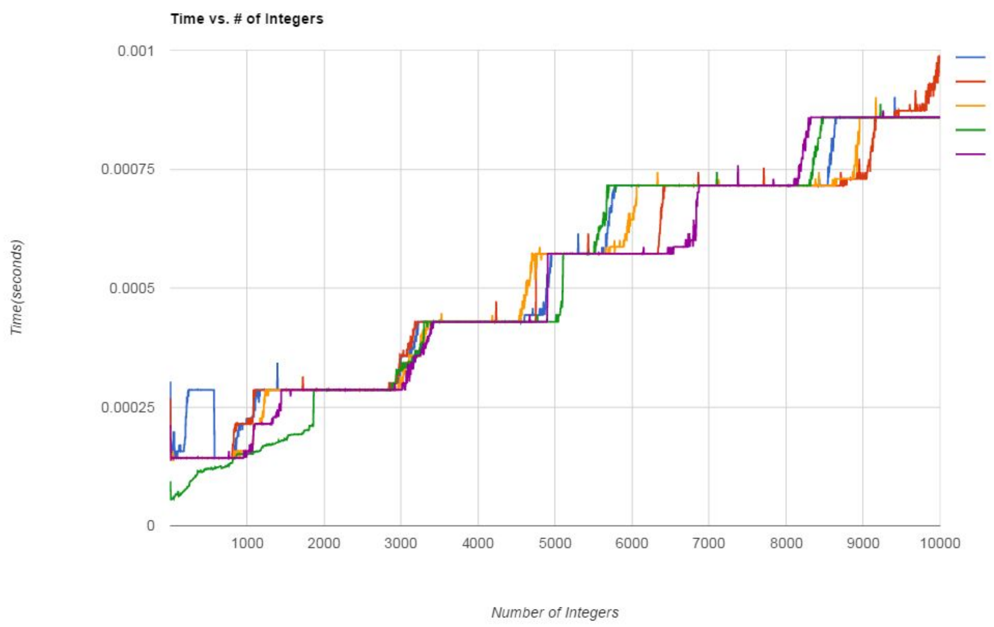


Scatterplot showing 20 trials of ping ponging messages to a different box

This graph shows 20 trials of sending a message to a different box and then getting the message back. There seems to be two sets of timings that are close together. The first one is around 4.00 x 10^{-5} and the second one is around 1.35 x 10^{-4} seconds. These times could be explained due to network congestions where it completes faster when there is no network congestion, and takes longer when there is. This means that it would not be accurate to average the data in this set and call that the time it takes to send a message back and forth between two processor on different boxes. Instead I will split it into two different averages, one in which there is little activity on the cluster, and the other where there is activity on the cluster. For little activity the average comes out to 3.82 x 10^{-5} seconds. And when there is higher activity on the cluster the averages comes out to 1.44 x 10^{-4} seconds.

Part Three-

The third part of this project was seeing how many integers could be passed before the packet could no longer hold them in one communication process. The code I wrote increased the amount in the buffer by five every iteration. It went until it reached 10000 ints. In order to get a better timing and remove some outliers, the same amount of integers would be sent ten times and then those times were averaged. This helped create a much better graph and better represented where the jumps were. Some things I did incorrectly was adding up all the timings instead of resetting the timer between each test.



Graph showing the time it took for various amount of integers to be sent to processor on a different box

This graph shows the amount of time taken for five different trails. The first 1500 numbers is a bit messy. This could be due to interference between the network as many other people were testing out their code. There is still a somewhat uniform jump in the first few numbers. This jump occurs around 1000. There are additional jumps around 3000, 5000, 6500, and 8500. Averaging these numbers, we get 1875. Therefore we can conclude that the max size that a packet can hold is 1875 integers before a new packet has to be made and sent in order to accommodate anymore integers. In order to get a better graph I think it would have been beneficial to average more than ten integers between every increase in sending size. This would have allowed for a more normalize timing throughout the entire graph.