Luke Johnson

CS555

Lab 2

1. **Answer the following questions that relate to your performance program. Be specific in your answers; Excel line graph plots are a convenient way to visually display the results. To answer the following questions run your performance program using different numbers of message (increasing by powers of two), different message lengths (four of five lengths), and interacting with different processors.**
   1. **What is the message latency (measured by 0 length messages) when using two cores?**

About 30-50 microseconds.

* 1. **What is the bandwidth (bytes per second excluding latency) when using two cores?**

I get about 1 billion bytes/second – tested by passing one message with 5,120,000 bytes

* 1. **What are the differences in your latency or bandwidth estimates when sending data of differing lengths?**

It seems to be fairly steady.

* 1. **Do the estimates vary from run to run, and how much?**

Estimates do vary between runs, but not incredibly so. This is on a machine running other processes, with different users.

1. **Answer the following questions that relate to your collective communication program. Run your program varying n for 2, 4, 8, 16, and 32.**

**a. How do the operations scale when running with different numbers of cores?**

Communication takes longer as more cores are added...

**b.What does this say regarding how MPI implements collective operations?**

There is a bottleneck somewhere...

* 1. **One could implement this program only using MPI\_Send and MPI\_Receive operations. What are the disadvantages of using this approach? What are the advantages?**

It would be advantageous, because you could explicitly specify exactly which piece goes to each processor, it is a problem because there ends up being a bottleneck at the master processor.

1. **In your collective operation program, how did you measure elapsed time? More specifically, how did you determine which processor had the longest elapsed time.**

Initialization steps were performed before an MPI\_Barrier() call, and the MPI\_Wtime() operation was called before and after before communication steps were performed. An MPI\_Barrier() call was again performed after the end time was stored to prevent printf interfering with the process.

1. **MPI has two pairs of collective scatter/gather operations. The first is MPI\_Scatter, and MPI\_Gather; the second is MPI\_Scatterv and MPI\_Gatherv**
   1. **What is the major differences between these operations?**

Scatterv and Gatherv specify a stride distance between blocks of values in an array, rather than passing along consecutive blocks.

* 1. **When would MPI\_Scatterv/MPI\_Gatherv be useful?**

If you're only interested in certain portions of an array that contain information you need, or you are performing an operation that requires operating on separate pieces at different times.

* 1. **Suppose that the processors don’t know in advance how much data should be distributed to each processor. How could that be implemented?**

Get the size of the data to be distributed and divide it by the number of processors to which it will be distributed.

1. **Suppose that we needed to send an integer, a double, a character, and a string to each processor. We could do four collective broadcast messages, but this would be quite slow.**
   1. **How could we use a derived data type to do this in one communication? Show the code for deriving such a datatype.**

Wrap it all up in one.

Int lengths[4] = {1, 1, 1, 300};

MPI\_Datatype types[4] = {MPI\_INT, MPI\_DOUBLE, MPI\_CHAR,

MPI\_CHAR};

int displacements[4];

displacements[0] = 0;

displacements[1] = sizeof(int\*length[0]);

displacements[2] = sizeof(double\*length[1]) + displacements[1];

displacements[3] = sizeof(char\*length[2]) + displacements[2];

MPI\_Datatype\* myType;

MPI\_Type\_create\_struct(4, lengths, displacements, types, &myType);

MPI\_Type\_commit(myType); //commit it.

//and now myType is an MPI structure.

* 1. **We could also use the MPI\_Pack/MPI\_Unpack calls. Show the code describing how this could be done.**

int myInt = 42;

double myDouble = 3.14;

char myChar = 'a';

char myString[30] = “Lorem ipsum dolor sit amet”;

char myBuff[1000];

int position = 0;

MPI\_Pack(&myInt, 1, MPI\_INT, buff, 1000, &position,

MPI\_COMM\_WORLD);

MPI\_Pack(&myDouble, 1, MPI\_DOUBLE, buff, 1000, &position,

MPI\_COMM\_WORLD);

MPI\_Pack(&myChar, 1, MPI\_CHAR, buff, 1000, &position,

MPI\_COMM\_WORLD);

MPI\_Pack(&myString, 30, MPI\_CHAR, buff, 1000, &position,

MPI\_COMM\_WORLD);

MPI\_Send(buff, position, MPI\_PACKED, 1, 0, MPI\_COMM\_WORLD);

… and then a receive and unpack each item.

1. **Suppose each processor has an array of doubles, all of the same size. Show the code that shows how the master could determine the maximum value of each location in the array and which processor holds that value. Hint: It is a collective operation.**

do MPI\_Reduce, passing in MPI\_MAX as the operation.

double \*myArray = ….

double max;

MPI\_Reduce(myArray, &max, 1, MPI\_DOUBLE, MPI\_MAX, 0,

MPI\_COMM\_WORLD);

1. **What is a barrier? When is it needed?**

A barrier is a point in the program where each processor will halt execution until every processor reaches the barrier. When there is a need to synchronize processes. I used it to prevent prints from occuring before communication was completed.

1. **Define the following and describe the differences between them: Standard send, synchronous send, asynchronous send, buffered send, ready send, blocking send, unblocking send, collective send.**

Standard send: A standard send blocks until the message is received, or is copied into a buffer.

Synchronous send: A synchronous send means that the data is sent directly to the receiving processor, and returns when the transfer is complete.

Asynchronous send: An asynchronous send copies the data into a buffer, leaving the receiver to pick it up when it gets to it.

Buffered send: A buffered send is completed when the message data is copied into a buffer. The sender only cares that its contents get copied.

Ready send: A ready send is a blocking send that waits for the receiving processor to finish receiving data before returning.

Blocking send: In a blocking send, execution continues when the call is complete.

Nonblocking send: In a non-blocking send, the call returns immediately. Even if the send is not complete, execution can continue.

Collective send: A collective send is any send that reaches multiple endpoints. This could be a broadcast or scatter, among others.

1. **What is a major difference between gather and reduce?**

Gather does not call an operation on the data being gathered.

1. **Explain how deadlock could occur if all processors implement a send immediately followed by a receive operation. Explain why deadlock might not always occur when running a program written this way. What are two ways to avoid this problem?**

All the processors could be waiting for a buffer to clear. If there are enough buffers available, then this would not cause deadlock.

1. **Derive a subgroup of the odd processors in MPI\_COMM\_WORLD and form a communicator. Show the code to do this.**

MPI\_Group world;

MPI\_Comm\_group(MPI\_COMM\_WORLD, &world);

int ranks[size/2];

for (i=1; i<size/2; i+=2) { ranks[(i-1)/2]=i; }

MPI\_Group odd\_group;

MPI\_Group\_incl(world, size/2, ranks, &oddGroup);

MPI\_Comm odd\_group\_comm;

MPI\_Comm\_create\_group(MPI\_COMM\_WORLD, odd\_group, 0, &group\_comm);

1. **What are some common errors that can cause collective operations to fail? Review the rules in week 2 slides to answer this question.**

Specifying the wrong size for message lengths!

Specifying the wrong receiving or sending processor.