To achieve the desired results, code changes were minimal. The hardest part was to change the program to send messages to both its neighbors, since it had to wait for one message to come in before it could send it and then wait for the next message and send that one.

Code changes are included in the electronic version, along with the tabulated data.

1) In the hello program, the processes respond in no particular order because they all have different startup and run times associated with the particular node they are running on. Although, they all start the same printf statement at the same time, there is no guarantee who will finish executing and reporting the message back first.

In the “message” program, process 0 waits to receive from any source (MPI_ANY_SOURCE), so the ordering is arbitrary. If we specify the rank in MPI_Recv, then it is possible to output in increasing order.

```c
    cout << "Message from process 0: 0" << endl;
    for (int i=1; i<size; i++) {
        MPI_Recv(msg, 2, MPI_INT, i, tag,MPI_COMM_WORLD, &status);
        cout << "Message from process " << msg[0] << ": " << msg[1] << endl;
    }
```

2a) As message size increases, there is a direct relationship with the time it takes to send the message. The relationship is linear for sizes larger than 64 bytes. Although not clearly visible in the plot, for sizes up to and including 64 bytes, the time is nearly the same (11 us), which is most likely due to the minimum allowable packet size. For these small sizes, the data is probably being padded out to reach a minimum of at least 64 bytes, making the time to send any small packet the same.

2b) As message size increases, bandwidth levels out, as demonstrated in the following plot. The logscale helps to show how bandwidth reaches it’s peak around 512KB sized messages. This means that the amount of data that can be sent in one second has reached it’s limit. Prior to 512KB, there was additional room in the communication channel for more data to be sent in a
second, allowing for an increasing trend as message size increased. This is usually a limitation of the routing devices and hardware connected to each computer, but can also be affected by the operating system.

2c) The message startup time is the measure of how much it costs to initiate a network connection and for data to be passed from the application to the OS to the network to the OS to the application. This can be approximated using a message of the smallest possible size, since it incurs no extra cost (it does not send more than the bare minimum number of packets). This startup time was measured at 11.3 us.

2d) The half power point can be calculated with the following formula.

\[
n = 2MB, \quad Bw = 232.90, \quad T(n) = 9004.4us
\]

\[
\alpha = \frac{1s}{(232.90*10^6)B} \times 1MB = 4502.258us
\]

\[
n/2 = 4502.258us \times ((232.90*10^6)B) = 104,857 \text{ B (about 100KB)}
\]

2e) The initial ring messages specifying name and node number are jumbled together, as they all reach the fprintf statement at probably equal times. With fprintf and cout there seems to be an implementation matter where all the data between parameters in fprintf, or between “>>” operators in cout, is sent to the output stream separately. Thus, in fprintf, the first part of the message is sent, then when %d is reached, it makes a new request to send that data, which allows for interleaving of output with different nodes. The printf seems to ensure that all data in a single call is assembled before being outputted.

3a) If we modify the program so that the nodes copy the message after receiving and before sending, bandwidth drops when the 512KB message is sent (from 140KB/s at 256KB to 131KB/s at 512KB). The most obvious reason is that the Valkyrie cluster only has a 256KB L2 cache and that it’s maxed out at 512KB, forcing most calls to go to main memory which slows down bandwidth. In fact, there are slow downs from 64KB to 128KB to 256KB, which in these cases seem to indicate that the L2 cache is not devoted entirely to the data of this particular program. The bandwidth is stable after dropping to 130KB/s at a message size of 512KB, because the time to access RAM is relatively constant. Had we sent messages that were larger than 1GB, the
process would have had to reach into virtual memory, affecting an even larger slowdown.

3b) When time is measured as a function of message size, there is a very close correlation to the program that does not copy messages (except that the times are generally doubled). To send a 64MB message without copying takes 284,253 us, while if copying is employed, that value jumps to 508,182 us. This demonstrates that care needs to be taken whenever data is moved around in a program through copy operations.
4a) If the program is modified so that each node sends and receives a message to and from their previous and next neighbors, the time to send grows significantly, approximately doubling from the original experiment. Since twice as much data is sent, it makes sense that the time also doubles.

![Time vs. Message Size (with double passing)](image1.png)

4b) If we pass twice as much data, the bandwidth behaves interestingly, in that it quickly reaches its peak bandwidth, but begins to decline as message size increases, which is most likely a function of network congestion, where so much data is being sent and received by each process that there are data collisions on the network and senders need to back off and slow down their sending rate to accommodate the amount of data. Indeed, with 4 nodes, each sending two 64MB messages, a total of 512MB is being moved around, which is well over the 1Gb/s (or possibly 100Mb/s) rate of the network.

![Bandwidth vs. Message Size (with double passing)](image2.png)