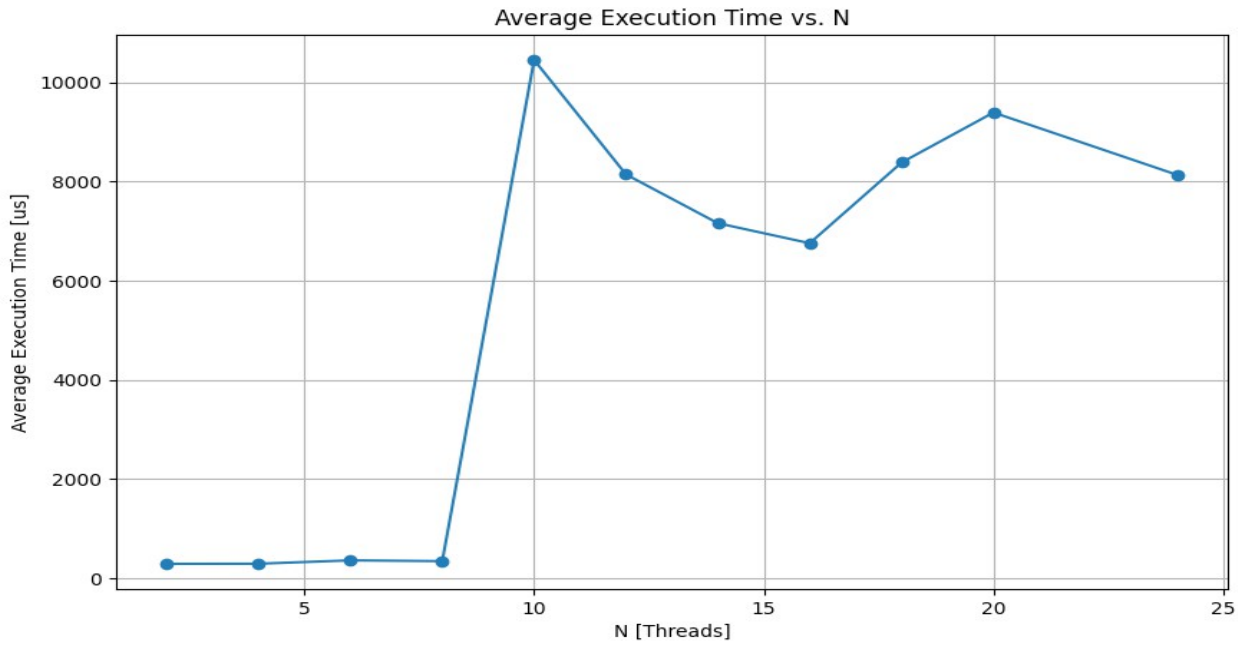
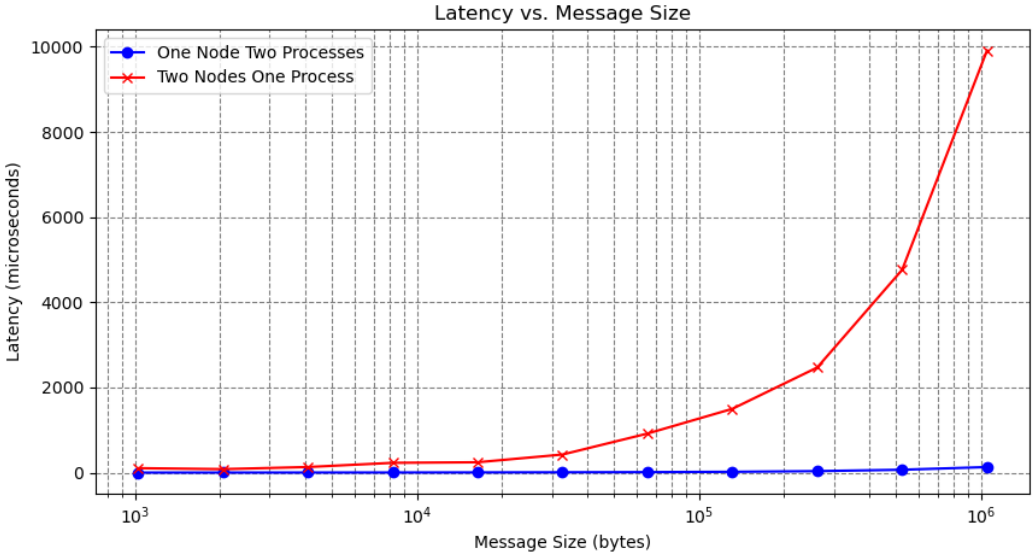
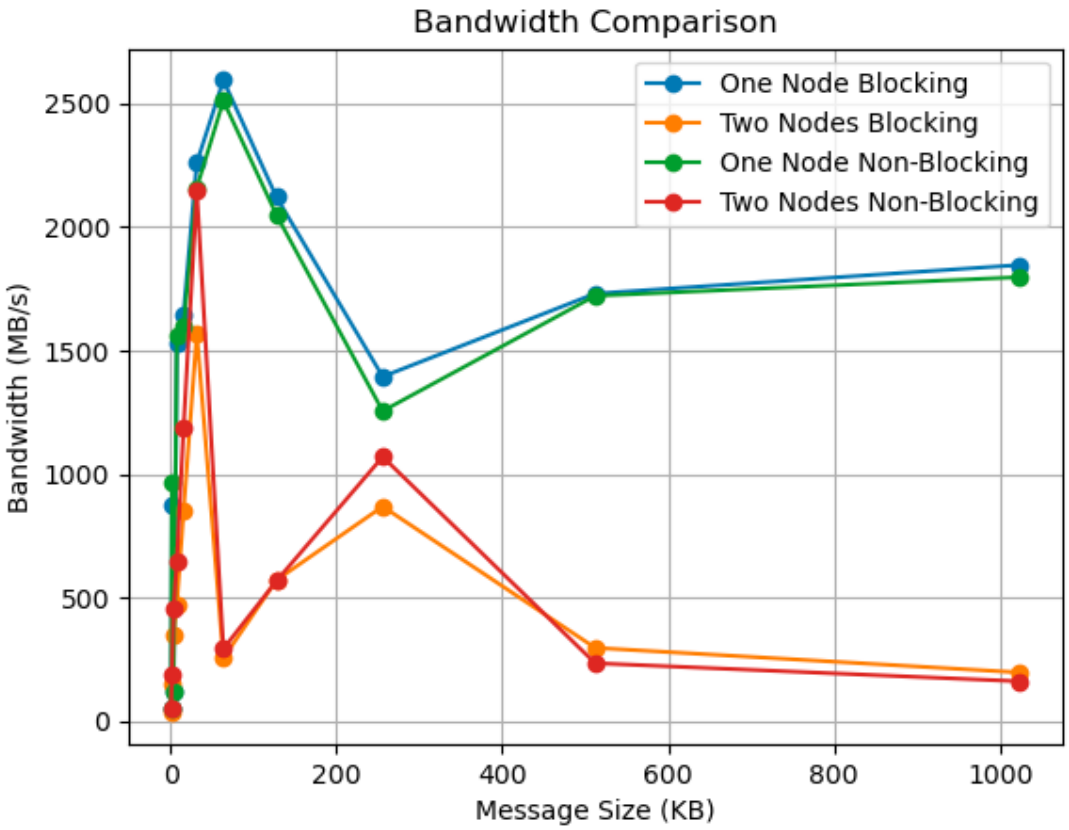
Ex 2.1:
Diagram 1 shows the average time per message over the number of cores involved in MPI ring communication. It is clear that the average time per message increases significantly as soon as more than one node is involved (10 threads, 2 nodes, 5 threads per node). The same applies to 18 threads (3 nodes, 6 threads per node) and 20 threads (4 nodes, 5 threads per node). The code is optimized so that the MPI message is first sent within a node and then passed on to the next node. This saves time, as the forwarding of messages within a node is relatively slow (see data points for 2, 4, 6 and 8 threads) and the forwarding of messages from node to node is much more time-consuming.



Ex.: 2.2

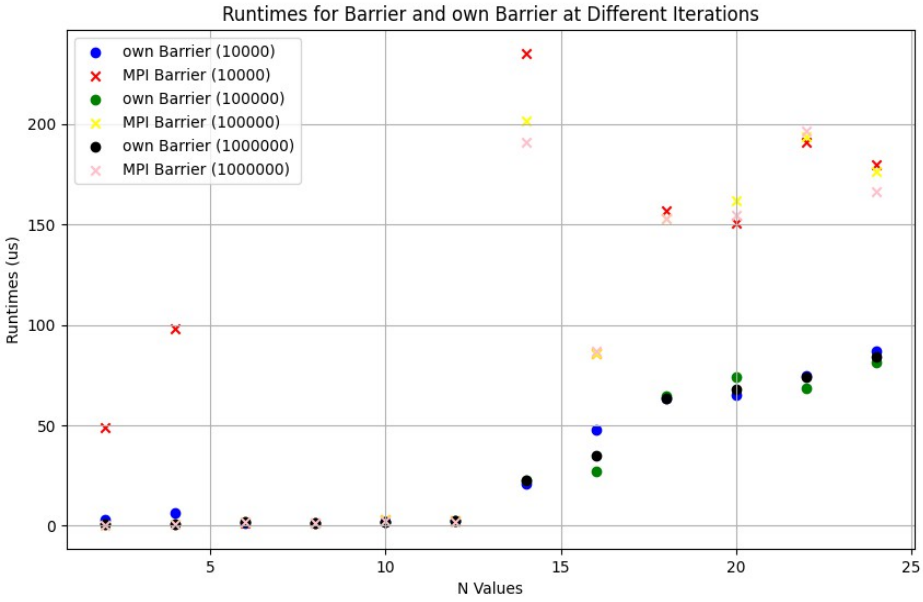

Latency vs. Message Size
One Node Two Processes
Two Nodes One Process
Latency (microseconds)
Message Size (bytes)
0
2000
4000
6000
8000
10000
10^3
10^4
10^5
10^6

Exercise 2.3:



Comparing the bandwidth values for the same message sizes, we can observe that non-blocking MPI sends generally achieve higher bandwidth compared to blocking MPI sends. This is particularly noticeable in the "One Node Blocking" vs. "One Node Non-Blocking" and "Two Nodes Blocking" vs. "Two Nodes Non-Blocking" comparisons.

E 2.4:
Diagram 4 shows the differences between a custom MPI Barrier implementation and the MPI Barrier function built into the mpi library. For the comparison, the average time over 10,000, 100,000 and 1000,000 iterations was measured. It can be determined that the own implementation is faster overall than the built-in function. It can also be seen that the average time for the built-in function increases significantly with an increasing number of threads and the involvement of more than one node (14 threads).



We share our willingness to present the exercises