Question 2:

2. (40) Develop a parallel histogramming program using C/C++ and OpenMPI. A histogram is used to summarize the distribution of data values in a data set. The most common form of histogramming splits the data range into equal-sized bins. For each bin, the number of data values in the data set that falls into that class are totaled. Your input to this program will be integers in the range 1-100,000 (use a random number generator that first generates the numbers). Your input data set should contain 8 million integers. You will vary the number of bins. You need to figure out how to assign bins to OpenMPI processes. You are suggested to use the sample batch script provided on Canvas for specifying your OpenMPI configuration and running your program (you will need to change some of the job parameters).  
a.) Assume there are 128 bins. Perform binning across nodes and processes using OpenMPI, and then perform a reduction on the lead node, combining your partial results. Run this on 2, 4 and 8 nodes on Explorer. Your program should print out the number of values that fall into each bin. Compare the performance between running this on 2, 4 and 8 nodes. Comment on the differences.

The code in folder A was run with the following node and process configurations to evaluate the performance at each. All timings took place in seconds.

|  |  |  |  |
| --- | --- | --- | --- |
|  | 2 Nodes | 4 Nodes | 8 Nodes |
| 1 process per node | 0.087 | 0.072 | 0.093 |
| 4 processes per node | 0.069 | 0.0293 | 0.029 |
| 16 processes per node | 0.026 | 0.0299 | 0.033 |

These results indicate interesting performance features associated with using nodes and processes in MPI. As the number of processes per node increases, there is an improvement in the efficiency of the histogram production from 1 process to 4 processes. This means that the overhead associated with communication amongst processes is insignificant compared to the speedup with parallel, individual histograms. However, for 16 processes per node, the overhead from synchronization delays associated with distributing chunks of data over 8 nodes, 16 processes each, means performance worsens compared to 4 processes per node. This isn’t seen in 2 nodes, as with 16 processes per node, the fewer number of nodes prevents any additional overhead, so performance is best in this case. At higher process counts, memory contention may cause some congestion, but this doesn’t take place as much with 32 total processes.

Without good parallelism, a greater number of nodes resulting in communication overhead that causes inefficiencies, as shown by the timing improvement with 2 nodes and 4 nodes compared to 8 nodes. However, between 2 and 4, though more communication associated with 4 nodes, the problem size per process is reduced sufficiently to see speedup.

For 4 nodes and 8 nodes, with 4 processes per node, performance is better than with just 2 nodes. Based on these performance differences, I can conclude that for lower numbers of processors, at an optimal amount, there is speedup due to better parallelism from more processors for the same problem size.

Based on this data, it can be concluded that this application exbibits decent strong scaling, since increasing the number of processors somewhat results in speedup for this problem size.

b.) For this part, assume you have 32 bins. Perform binning on each process using OpenMPI, and then perform a reduction on the lead node, combining your partial results. Run this on 2 and 4 nodes on Explorer. Your program should print out the number of values that fall into each bin. Compare the performance between running this on 2 and 4 nodes. Comment on the differences.

The code in folder B was run with the following node and process configurations to evaluate the performance at each. All timings took place in seconds.

|  |  |  |
| --- | --- | --- |
|  | 2 Nodes | 4 Nodes |
| 1 process per node | 0.103 | 0.079 |
| 4 processes per node | 0.082 | 0.041 |
| 16 processes per node | 0.021 | 0.027 |

What stands out with these performance results is the scalability of the number of processes per node, and its implications on performance. At 1 and 4 processes per node, using 4 nodes is more efficient than using 2. This is likely because at these number of processes, the parallelism per node is largely insignificant, as the overall total number of processes across all nodes enables parallel speedup. With a total of 16 processes, 4 nodes with 4 processes per node, there are twice as many processes running in parallel compared to 1 node with 4 processes per node. This enables the speedup.

However, at 16 processes per node, there is more data distribution, resulting in more local histograms which reduce into the main histogram. The communication associated with this introduces overhead that causes inefficiencies, and the resulting performance degradation.

c.) Compare the performance measured in parts a.) and b.). Try to explain why one is faster than the other and run additional experiments to support your claims.

In part A and part B, 8 million integers are organized into histograms, only the number of bins in each histogram varies from 32 and 128. A smaller histogram, with less options for bins, means that integers are likely to fit better into the CPU Cache, enabling more efficient memory access. Overhead is also associated with some of the performance between the same configurations of nodes and processes for each histogram.

With 1 process per node, both bin size programs exbibit similar behavior, as the efficiency of 4 nodes, despite communication overhead, results in speedup compared to 2 nodes.

For 1 process per node and 4 processes per node, the performance is better in the program with 128 bins compared to 32 bins. This, though counterintuitive, could be because the data distributed amongst bins leads to less contention for each location in memory. Contention in fewer bins causes slower memory access, leading to performance degradation.

At 16 processes per node, the 32-bin size program is more efficient than 128 bins. This is due to the scale of the communication overhead, which at this number of processes is much more noticeable.

An interesting difference between the performances is at 4 nodes with 4 processes per node, where in 128 bins, the performance is better than at 16 processes per node. This can be attributed to the overhead of communication amongst nodes, where the parallelism taking place is not valuable enough to improve performance comparatively. With 32 bins, there is normal scalability, which is likely due to distribution of data, as fewer operations are required, so the workload per process is much easier to scale.

For additional tests, the bin size was changed and tested at 4 nodes.

|  |  |  |  |
| --- | --- | --- | --- |
|  | 64 bins | 256 bins | 512 bins |
| 1 process per node | 0.08 | 0.08 | 0.080 |
| 4 processes per node | 0.04 | 0.0411 | 0.04322 |
| 16 processes per node | 0.03 | 0.032 | 0.031 |

Based on these results, it’s likely that the result from trials at 128 bins for 4 nodes and 4 processes per node produced outlier results. There is good strong scaling here, as introducing more parallelism results in continuous speedups, though the difference between 1 and 4 is greater than between 4 and 16. This demonstrates overhead associated with distributing data across additional processes.

Additionally, based on these results, it appears as though changing the bin size has a minimal effect on performance. This means that differences in performance can be attributed more to communication overhead and delays associated with data distribution moreover.