Introduction

The goal of this project was to implement a repeated histogramming algorithm to sort a variable data input in parallel utilizing different paradigms and to compare the overall performance of those methods. The project itself is broken up into 3 separate function tasks to accomplish the algorithm, making it possible to analyze sections of the implementation independently. After the histogramming function has identified the global key splits for the data, each rank sends data to the correct histogram bin.

The method used to move the data between the processes was to be varied in order to do a comparison. For this project, I chose to compare the performance of a Point-to-Point communication approach and One-Sided communication approach when distributing the data amongst the processes.

Function Implementation Methods

Rebalance:

Following the methodology used by Kale, Harsh and Solomonik in the 2018 paper “Histogram sort with sampling”[1], the overall method used to rebalance the dataset among the ranks used a root processing node which collected global knowledge and distributed instructions to the other nodes. I chose to use a queueing method, where all nodes used One-Sided communication to inform the root node of their data parameters, and the root node computed the data surpluses and deficits per node to build a queue of data swap instructions. These instructions where then distributed to all the ranks via Point-to-Point communication, and based on these instructions, each node sent their indicated data where the root determined was optimal.

Find Splitters:

The general method I used to implement the histogramming algorithm used only One-Sided communication and broadcasting, as by this point, I found it to be much more intuitive to open segments of memory up to be shared or altered then to “send” data around like messages. After getting each rank back on the same page as to the state of the global data size and key structure, I chose to implement the entire histogramming algorithm in a single large loop, where the loop would terminate if the keys were within the target margin, or the loop had run to a variable maximum iteration (to avoid potential infinite looping). The histogramming itself also followed the papers[1][2] in that the root node was in charge of creating the split keys and distributing them to all the nodes every iteration. At the end of each iteration, each node would share the results of the current splitter keys on its’ data, and the root would determine if the keys were adequate. If not, it would broadcast new splitter keys for the next iteration.

Move Data:

General methodology of this function starts very similarly to the last two. There is a collection of One-Sided communications where, based on the split keys determined by the findSplitters function, calculate the indexes in their arrays that correspond to the split keys, and then put them in the root's global indexing array to be broadcast to all ranks. This cheaply gets all ranks on the same page as to where each rank's data is to be partitioned, and is easily indexable by their rank. After this information is available to all processes, it is merely a matter of taking the indicated partitions of each processes data and moving them to the correct process/bin. This section was given an if-then switch to test distribution using One-Sided Communication and Point-to-Point communication. Each of the two methods uses the exact same information to determine where the data should go, the only real difference is between the MPI primitives used to move the data making it ideal to test the performance differences.

Test Method

The method used to test the performance of the One-Sided Communication vs the Point-to-Point communication approaches was to solely alter the number of nodes equally, keeping all else constant, and comparing the results for each approach. This also increases the amount of overall data to be moved as the nodes increase, so keeping the other variables constant makes sense for comparing the performance of the actual communication paradigms. The nodes were adjusted to be 2, 4, and 8 with the tasks and cpus per task fixed at 4.

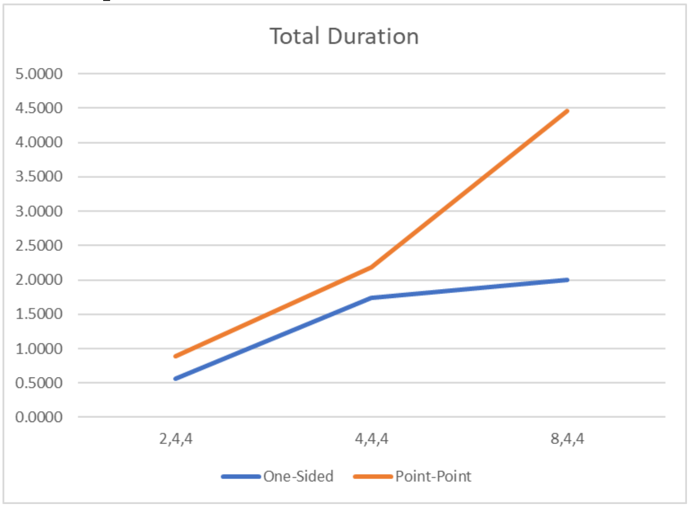
Test Results

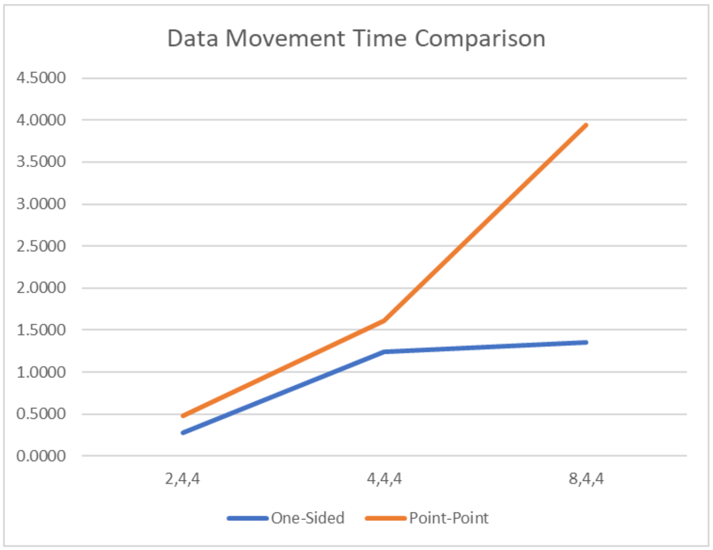
Movement Method	Configuration	Rebalance	Sort_1	Splitters	**MoveData**	Sort_2	**Duration_Total**
One-Sided	2,4,4	0.022	0.133	0.044	**0.284**	0.075	**0.558**
One-Sided	4,4,4	0.167	0.174	0.065	**1.238**	0.101	**1.744**
One-Sided	8,4,4	0.336	0.157	0.064	**1.354**	0.093	**2.004**
Point-Point	2,4,4	0.097	0.161	0.059	**0.485**	0.088	**0.890**
Point-Point	4,4,4	0.230	0.173	0.066	**1.615**	0.095	**2.178**
Point-Point	8,4,4	0.203	0.157	0.063	**3.940**	0.092	**4.454**

*Note: Configuration is in format (Nodes, nTasks-Per-Node, cpus-per-task) indicating the slurm parameters

Although I think it would be reasonably expected, the results of this test were overwhelmingly in support of the One-Sided communication method. Although the only alteration was the manner with which the data was moved, the One-Sided communication approach is consistently much faster than the Point-to-Point. I tried to ensure that both methods were equally efficient and used the same indexing for communication as well as sending/getting defined areas of the arrays, not single points at a time in a loop.

As you can see from the above table, the rate at which the MoveData function's processing time (shown in seconds) increases much faster for the Point-to-Point than the for One-sided, resulting in an overall run time duration difference surpassing a factor of 2 for the 8 node test. The graph to the right shows this comparison, and you can see the Point-to-Point method (orange) taking on a much steeper slope after the introduction of 4+ nodes.

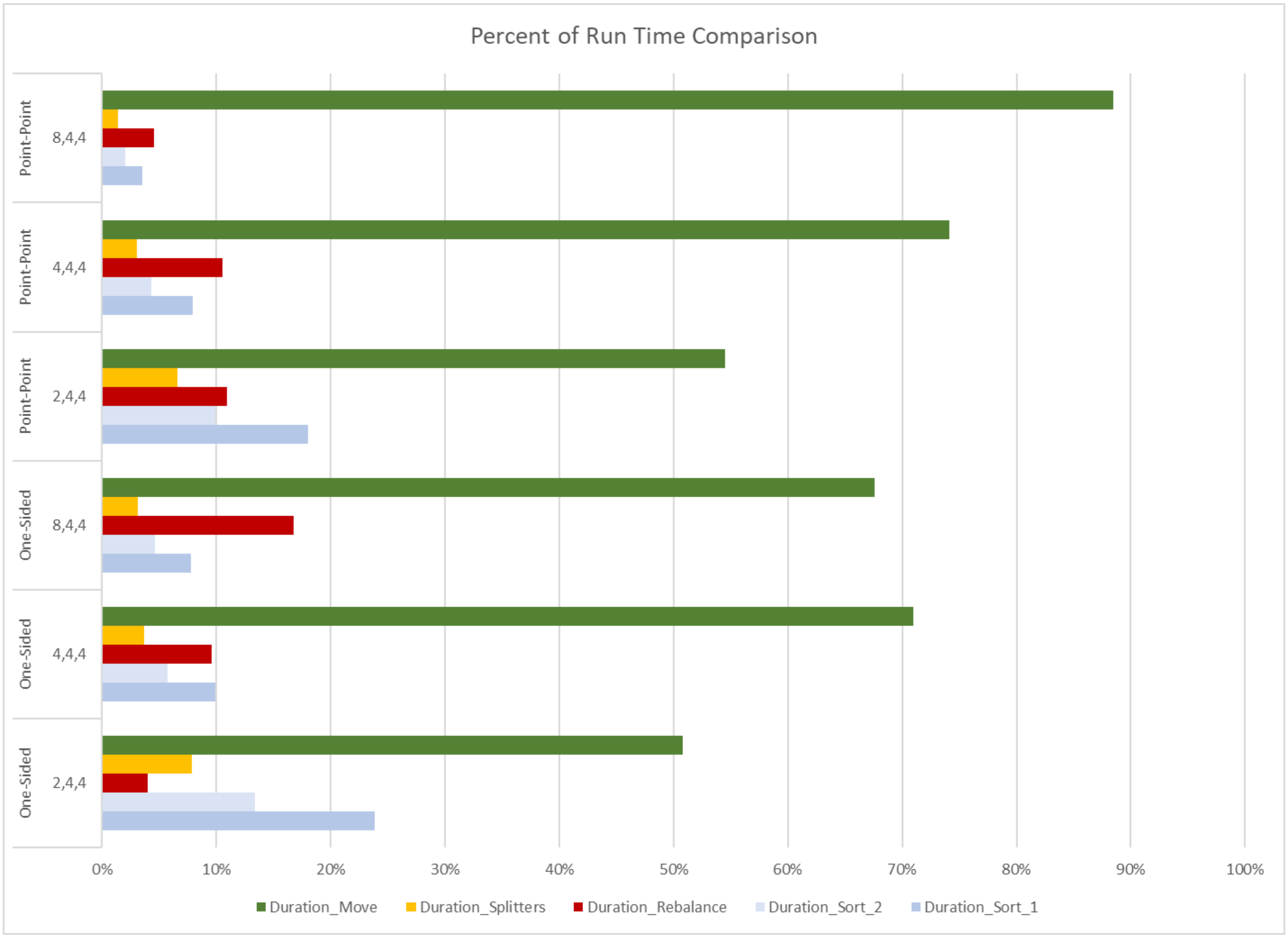




Further, you can see from the comparison graphs to the left, as the number of nodes increases, both implementations increase not only in the amount of time total but also in the time to move the data as well, with very high correlation.

The Point-to-Point method becomes increasingly inefficient after 4 nodes, presumably because of the exponential increase in communication needed. Interestingly, the One-Sided communication method seems to be doing the exact opposite, and appears to be leveling off.

I would suspect the difference in result time is because the manner with which MPI opens up memory windows is more efficient than multiple sends and receives. Being able to open up memory strategically to other processes and allowing them to get what they need is intuitively more efficient than sending information via communication primitives. It is interesting to note that when comparing the function runtimes as a percentage of the total run time (graphed below), not only does the One-Sided communication method outperform the Point-to-Point method in moving the data, but the percentage of the run time to move the data actually began to decrease, and would presumably find a balance while the run time increased.



References

[1] V. Harsh, L. V. Kale, and E. Solomonik, “Histogram sort with sampling," *CoRR,* vol. abs/1803.01237, 2018.

[2] L. V. Kale and S. Krishnan, “A comparison based parallel sorting algorithm," *in 1993 International Conference on Parallel Processing - ICPP'93*, vol. 3, pp. 196-200, Aug 1993