CS 484 – Spring 2021 – Final Project

William Sayer – wsayer2

Overview of Approach

Rebalance()

My approach in this function was to redistribute as simply as possible with the least amount of messages required. I do that by using four primary attributes: **prefix sum**, **myDataCount**, **average data** count (or **target**), and **my_rank**.

This function starts with **MPI_Scan** on myDataCount to calculate a prefix sum at each rank followed by an **MPI_Allreduce** on myDataCount (aggregated with sum) so each rank knows the total data count across all ranks.

The **prefix sum** is used to inform each rank how many data items are stored in the ranks that come before it. This can be used to compare against the *expected* prefix sum (i.e. *expected* at that rank if all items were balanced) to tell the rank how many items the rank has to send/receive to/from the rank that comes after (forward rank).

Delta_pre = actual prefix sum – expected prefix sum

If **delta_pre > 0** then it must send to the forward rank, and if **delta_pre < 0** then it must receive from the forward rank. The absolute value determines the number of data items to send/receive. If my_rank == (comm_size – 1) then delta_pre == 0 must be true.

The **average/target** data count is then used in conjunction with delta_pre to determine how many data items are needed to send/receive with the backwards rank.

Delta_avg = my_count – average count

bkwdCnt = delta_avg – delta pre

If **bkwdCnt > 0** then this rank must send to the backward rank, and if **bkwdCnt < 0** then this rank must receive from the backward rank. The absolute value determines the number of data items to send/receive. If my_rank == 0 then bkwdCnt == 0 must be true.

Each of these corresponding values is calculated consistently at each rank (independently) and MPI_Isend/MPI_Irecv calls are made. This makes it such that each rank only has to communicate *at most twice* – once with their forward neighbor and once with their backward neighbor. This also keeps communication local and simplifies the question of "who do I send my data to?".

Checks are in place such that if a rank must receive data before sending (i.e. send data count > my original data count) then it will wait and then use that receive buffer if needed. If total data count across all ranks is not divisible by the comm_size then remainders are assigned through round-robin starting with lowest rank.

FindSplitters()

My approach in this function was pretty much in line with the guidance the paper provided. First I use **MPI_Allreduce** on **data_size** so each rank knows the total data count across all ranks. Next I calculate the **minimum and maximum data values** on each rank and then use **MPI_Allreduce** again so each processor knows the global minimum and maximum values (however an MPI_Reduce to rank 0 probably would have sufficed).

I then run a **do-while loop** that has the following steps:

1) If rank == 0 then either **initialize splitters** (if phase == 0) or **recalculate splitters** (phase > 0)
    a) Initialization of splitters is done with a **linear interpolation** from **global min** to **global max**
    b) Recalculation of splitters is done by **proportional linear interpolation** as described in the paper, using the **global min** as the **lower bound** (instead of 0) for faster convergence

2) **Broadcast** new splitter values from rank 0
3) **Each rank counts** how many data items they have in each bin
4) **Aggregate count totals** using MPI_Reduce to rank 0 with MPI_SUM
5) **Rank 0 checks bin counts** to see if all bins have a count that is **within +/- 0.01 deviation** from average
    a) If yes return TRUE, if no return FALSE – this variable is used
6) **Increment phase value** and **broadcast the returned boolean** value from step 5 so all ranks know whether they should continue on to next iteration or escape the do-while loop

Once we have escaped the do-while loop, **rank 0 broadcasts the final global counts** for all bins using the latest splitter values.

MoveData()

My approach in this function is first have rank 0 **broadcast splitter values** to ranks. I then have **each rank count how many data items** they have to send to each rank. **My implementation generalizes** this step such that **if number of bins > number ranks** then the data movement is handled such that ranks are associated with their corresponding bins (i.e. if there are 3x number of bins as there are ranks then rank 0 gets data items for bins 0-2).

Next I **broadcast global bin counts** (from function input, not MPI_Reduce) from rank 0 to all ranks so that each rank knows how many items to receive and **allocates** the appropriate amount of memory.

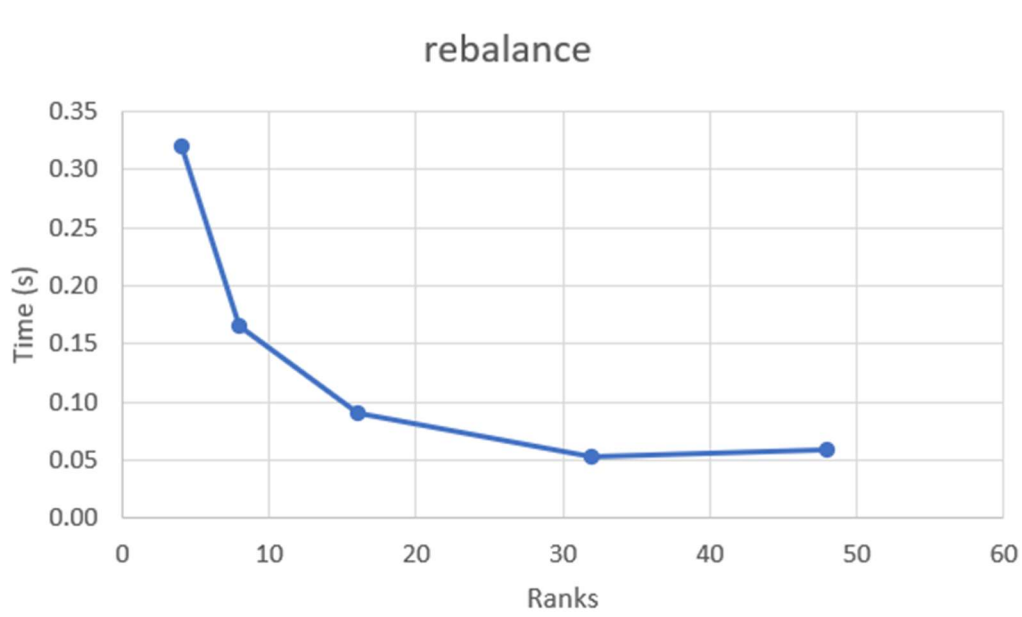
My first attempt at the data movement was using **one-sided communication**, where I use **MPI_Scan** on the recently calculated local bin counts so that each rank knows *where* in each corresponding rank's pubic array it is supposed to place the data items it needs to send to that array. Then I used **MPI_Put** to transfer data between nodes.

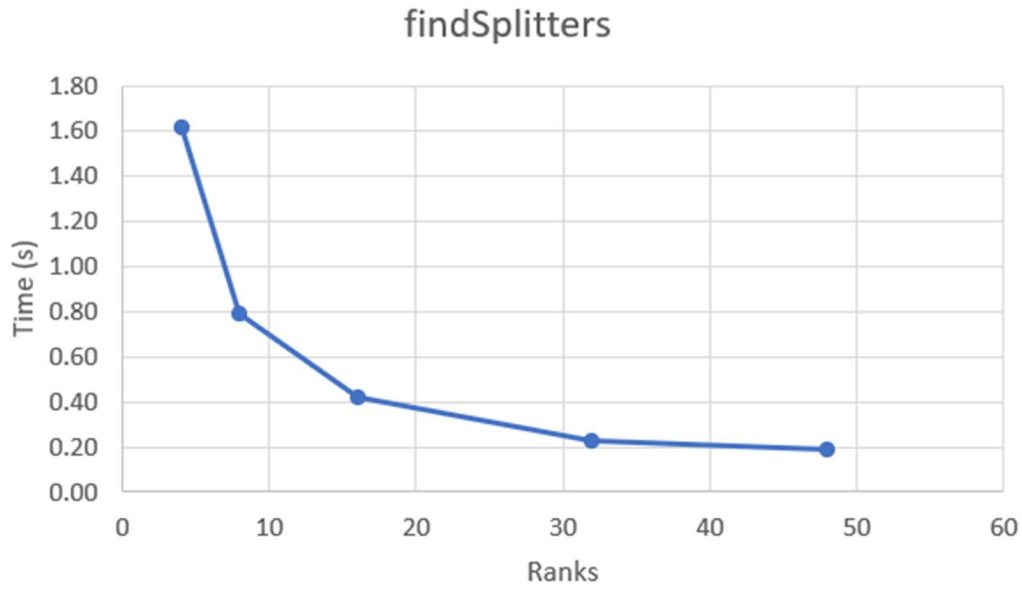
My second attempt was **point to point communication**. Here I used an **MPI_Alltoall** to communicated to each rank the number of data items they should expect from each rank. I then follow this with corresponding **MPI_Isend** and **MPI_Irecv** calls to distribute the data. **Ultimately I ended up choosing the p2p communication** because it ran in 1.50 sec while the one-sided communication ran in 3.7 sec. Using the one-sided communication method, think there was too much blocking with the fences around the MPI_Put, but it is possible I could have optimized to remove these and make the function more efficient, but I did not have time to try this.

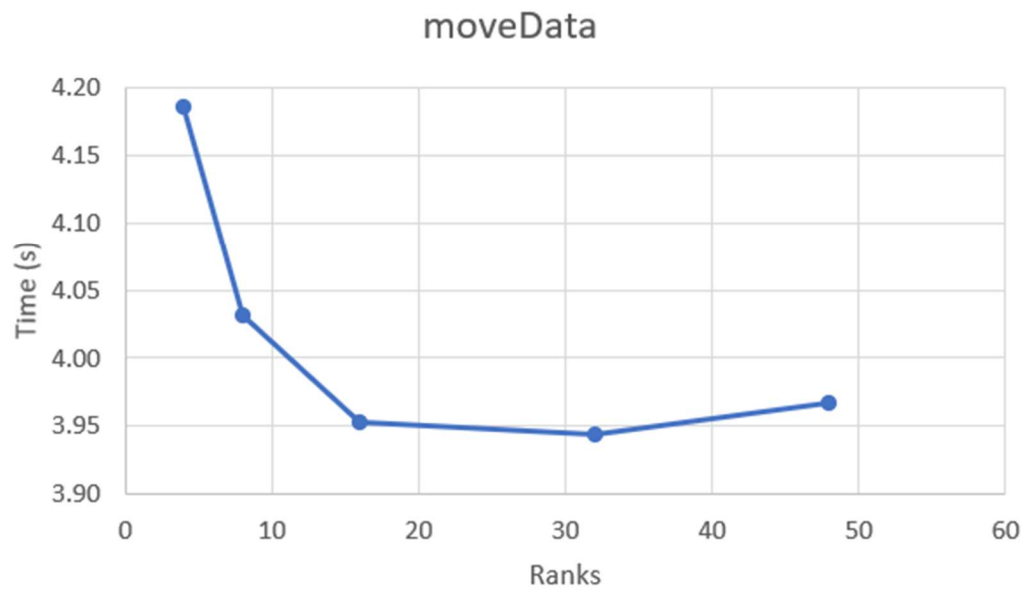
Scaling Study

As you can see from the results in the table below and on the next page, these functions scale pretty well. The moveData() function takes up the most time by far.

# ranks	datasize-a	datasize-b	rebalance	findSplitters	moveData
4	40,000,000	80,000,000	0.32	1.61	4.19
8	20,000,000	40,000,000	0.17	0.79	4.03
16	10,000,000	20,000,000	0.09	0.42	3.95
32	5,000,000	10,000,000	0.05	0.23	3.94
48	3,333,333	6,666,667	0.06	0.19	3.97


rebalance
Time (s)
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0
10
20
30
40
50
60
Ranks


findSplitters
Time (s)
1.80
1.60
1.40
1.20
1.00
0.80
0.60
0.40
0.20
0.00
0
10
20
30
40
50
60
Ranks


moveData
Time (s)
4.20
4.15
4.10
4.05
4.00
3.95
3.90
0
10
20
30
40
50
60
Ranks