Implementation details

rebalance()

- Use MPI_Allgather to let each process know the number of elements in each of the processes, store in sub_counts.
- Figure out the global number of elements and what is the rebalanced datasize *rCount in each process
- If a process currently has less load than *rCount, receive data from other processes until it reaches *rCount
- If a process has more load than *rCount, it analyizes the sub_counts array and figure out how many data it needs to send to which processes. All processes follow the algorithm as follows:
    - Starting from rank 0, scan the sub_counts array. Consider rank i:
      If a processes j needs data, send min(sub_counts[i] - *rCount, *rCount-sub_counts[j]) to it. Add the amount sent over to sub_counts[j] and remove from sub_counts[i]
      Rank i continue scanning the array and send data until sub_counts[i] == *rCount

findSplitters() // assume data is already sorted

- Send local data size, local max to rank 0 using MPI_Reduce
- For rank 0 do the following:
    - Get an initial set of PROBE_MULTIPLE*(numSplitters-1) probes by using PROBE_MULTIPLE*(numSplitters-1) local values evenly spaced
    - Broadcast the number of probes to all processes
    - Broadcast the probes to all processes
    - Calculate ideal bin size = global_N/ numSplitters, from the binsize we can derive the ideal splitter locations.
    - Count the number of elements in each bin separated by the probes
    - Get the global number of elements in each bin using MPI_Reduce
    - If any of the probes fall within 1% of the ideal splitter location, mark it as achieved.
    - For unachieved splitters, count how many splitters there are in each of the intervals separated by the probes. For an interval having m splitters, generate PROBE_MULTIPLE * m new probes evenly spaced between the interval
    - Broadcast the number of probes and the probes to all processes
    - Iterate until all splitters are located.

- For all other ranks do the following:
    - Receive broadcasted number of probes from rank 0.
    - If number of probes > 0:
        - Receive broadcasted probes from rank 0
        - Calculate local probe ranks, send to rank 0 using MPI_Reduce
        - iterate
    - If number of probes == 0, it means all splitters has converged:
        - Receive broadcasted splitters array from rank 0

- Receive broadcasted global bin size from rank 0
- End function

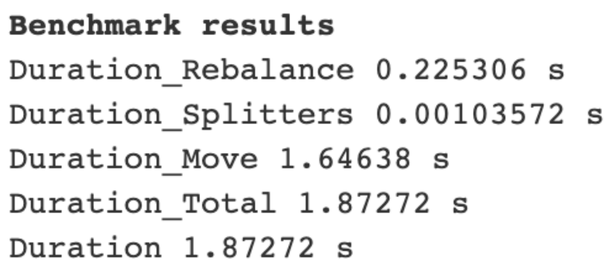
moveData()
- All to all collective (MPI_Alltoall(), MPI_Alltoallv())

I also experimented with:
- One to all collective(MPI_Gather())
- Point to point messaging (MPI_Isend, MPI_Recv)

But they are all slower than all-to-all, possibly because collective communications are highly optimized in the MPI implementation.

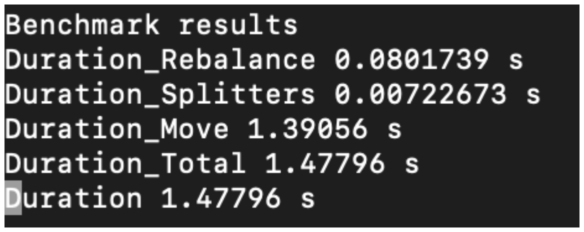
Benchmarking

Target performance:



```
Benchmark results
Duration_Rebalance 0.225306 s
Duration_Splitters 0.00103572 s
Duration_Move 1.64638 s
Duration_Total 1.87272 s
Duration 1.87272 s
```

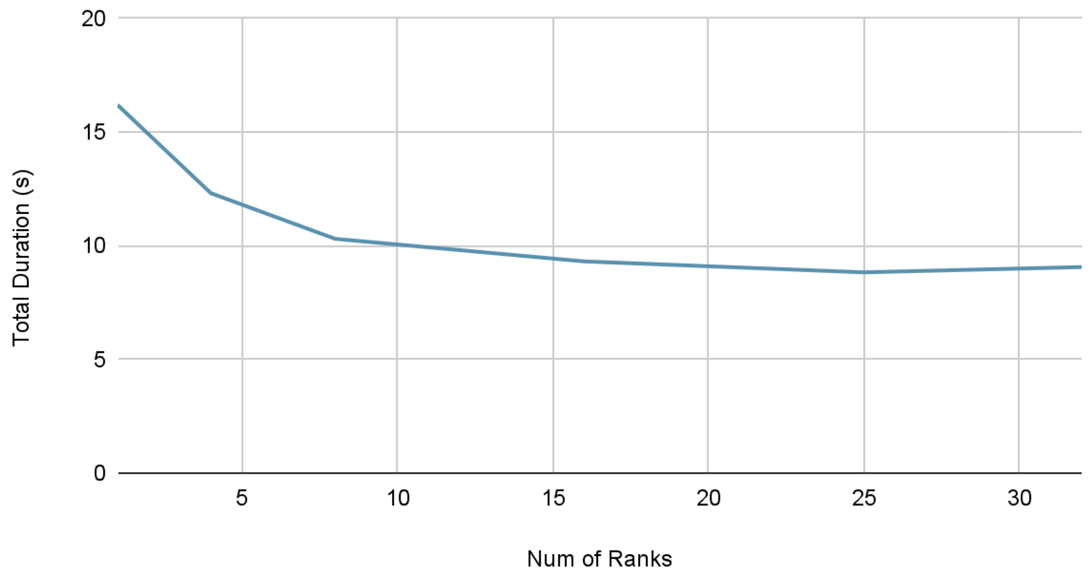
My performance running default batch.slurm



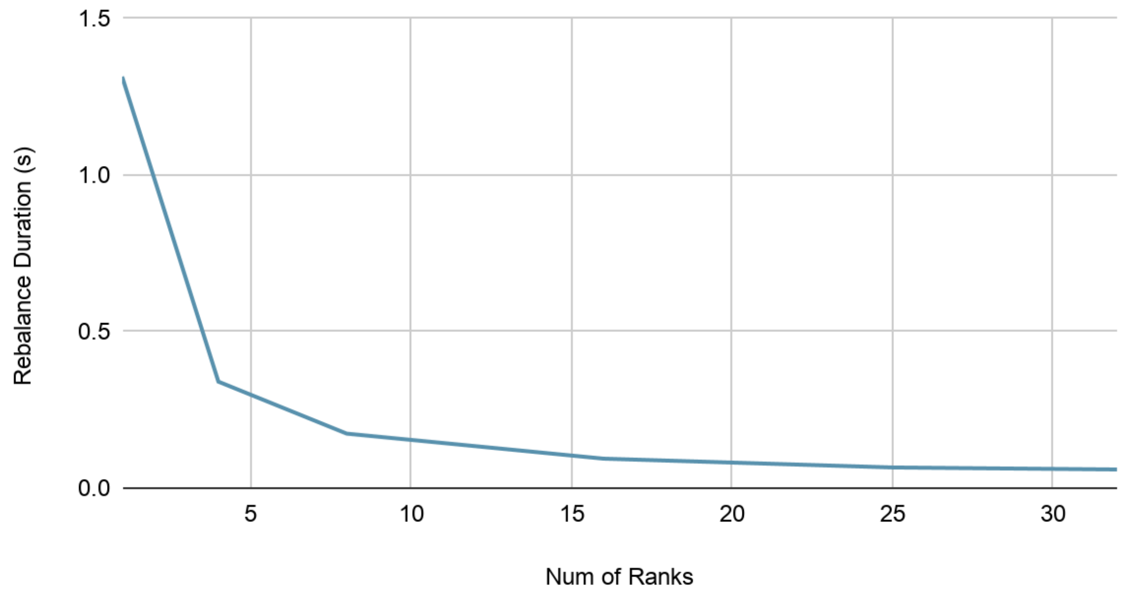
```
Benchmark results
Duration_Rebalance 0.0801739 s
Duration_Splitters 0.00722673 s
Duration_Move 1.39056 s
Duration_Total 1.47796 s
Duration 1.47796 s
```

Keeping the global data set at 500M and vary the number of ranks

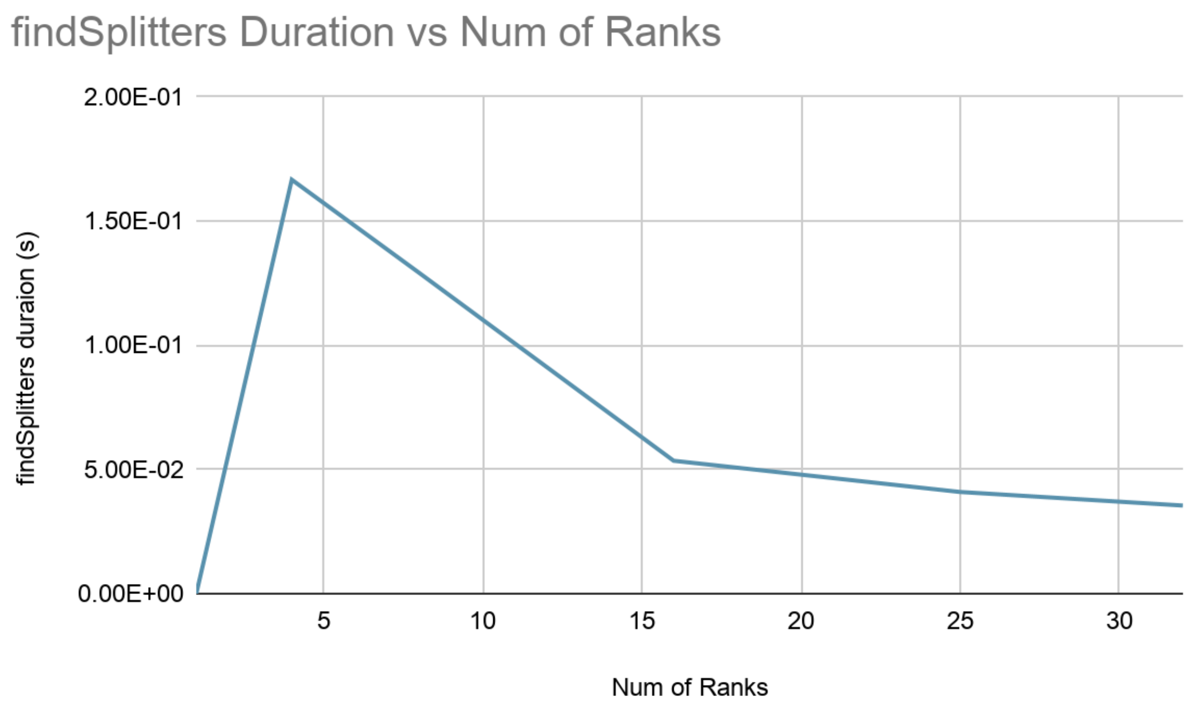
Total Duration vs. Num of Ranks



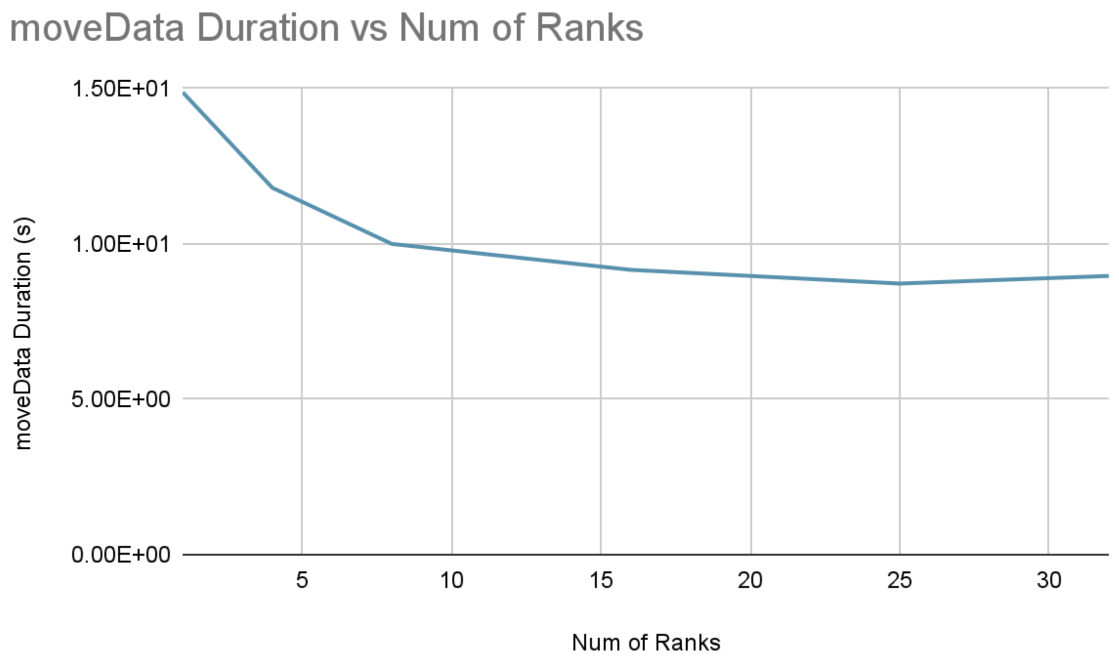
Rebalance Duration vs Num of Ranks



\



findSplitters function ran really fast when only have 1 rank.
This is due to the advantage of not needing to communicate between different ranks, and the input data is already sorted.


moveData Duration vs Num of Ranks
1.50E+01
1.00E+01
5.00E+00
0.00E+00
moveData Duration (s)
Num of Ranks