Chris Keefe
Assignment 3 - Distribution Sort

Activity 1: Code description

My implementation captures a start time, calculates bucketsize and start/end indices for the local bucket, and then iterates over the data nprocs times, sorting values that belong in the "destination bucket" into the send buffer (or directly into myDataSet if no communication is necessary).

Data is sent with non-blocking MPI_Isend, which allows ranks to immediately begin receiving any data sent their way. MPI_Get_count allows me to memcpy only the amount of data actually sent from the receive buffer into myDataSet. At this point, we MPI_Wait, to ensure that the data in the send buffer isn't overwritten before the send has completed.

Each rank "sends" first to itself (actually memcpy), and then sends to the rank (my_rank + i) % nprocs. By staggering in this way, ranks never have to send or receive more than one message per iteration, which probably removes some blocking behavior over a more naive approach.

```
global distrib time: 5.234949, global sort time: 6.412118, global total time: 11.630869
Number of ranks not sorted properly: 0
Global sum 499937769104586 == unsorted global sum 499937769104586
~
```

Some clear opportunities for improvement include the following:

- receiving data directly into myDataSize (rather than a receive buffer) would reduce the overall memory allocation by N*sizeof(int). This would be a little fussy, but there are no barriers to implementing it but time.
- Re-organizing the loop to delay the MPI_Wait. In this implementation, this is probably a micro-optimization, and the MPI Docs and common tutorials don't discuss how/if one might initialize a request object into a state in which MPI_Wait will pass without an actual send occurring. (This would allow us to proceed without re-organizing the loop behavior).
- It is possible to iterate over the data to be distributed only once (rather than nprocs times) without increasing the memory required, but this behavior would have to be conditional on N >> nprocs, and might still be fragile. It could be useful in production, but didn't make sense for this implementation.

Jobscript/run description

As in assignment 2, I used BASH to coordinate the running of all jobs. The sbatch.command script passes job-specific output parameters, including --job-name, --ntasks, and --output. I've included the `sbatch.command` script with my code.

Note: Please consider synchronizing the number of ntasks parameters we are asked to check across assignments. This assignment adds a two-processor run that didn't exist in the prior assignment. This is a trivial change, but it added some unnecessary friction for those of us who overlooked it, (re-running additional jobs manually).

- **Q1: Based on the problem description, should distribution sort have low or high load imbalance?**

Assuming we have a uniform distribution of points (as we do here), distribution sort should have low load imbalance. Each rank should have a roughly even distribution of values to re-distribute and to sort.

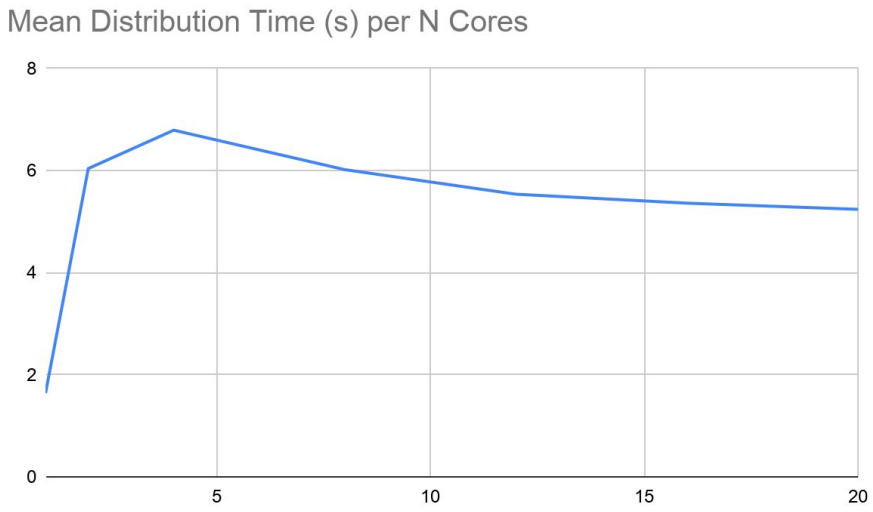
- **Q2: Is there anything different about the algorithm when p=1?**

When P=1, the algorithm can skip the distribution step, moving directly to quicksort (or whatever nextsort you're using). For simplicity/readability, my implementation still copies values from data to myDataSet when p=1, but this copy could also trivially be avoided by simply sorting and reporting on the initial data when p=1.

	A1 Total Time (s)	**A1 Dist Time**	**A1 Sort Time**	**Speedup**	**A1 Par Effic**	**Global Sum**	**Jobscript**
1	155.154	1.64094	153.5133	1	1	499939833260556	jobscript.sh + sbatch.command
2	79.67015	6.04017	73.63371	1.9475	0.97375	499934584339325	jobscript.sh + sbatch.command
4	42.0463	6.79155	35.2784	3.6901	0.92253	499937155109296	jobscript.sh + sbatch.command
8	23.9669	6.01865	17.9691	6.4737	0.80921	499925398982548	jobscript.sh + sbatch.command
12	16.5272	5.53739	11.0271	9.3878	0.78232	499924679946291	jobscript.sh + sbatch.command
16	13.5382	5.36197	8.1906	11.4605	0.71628	499932728507365	jobscript.sh + sbatch.command
20	11.6323	5.24118	6.4119	13.3382	0.66691	499937769104586	jobscript.sh + sbatch.command

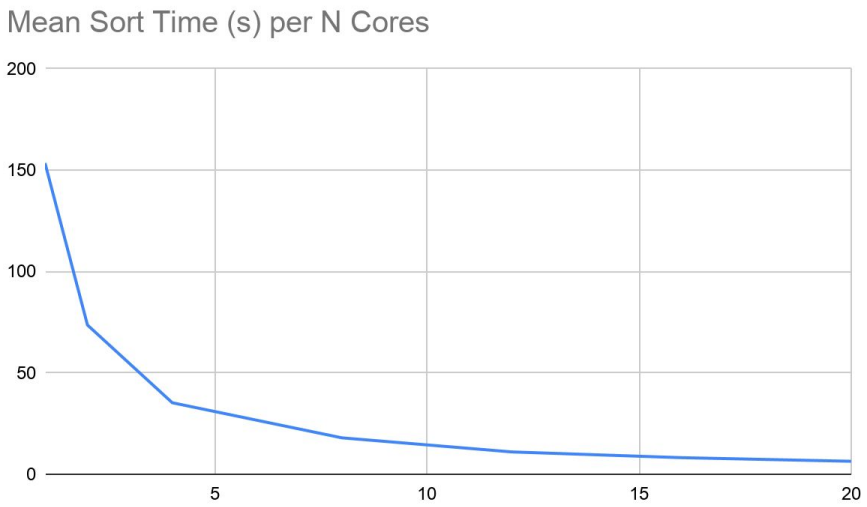
- **Q3: Does the time to distribute the data vary as a function of p?**

  Yes. Dist time at p=1 is very short, climbs rapidly when distribution is actually required, and appears to decrease very slightly (perhaps logarithmically or exponentially) towards an asymptote as the number of cores increases from four to 20.



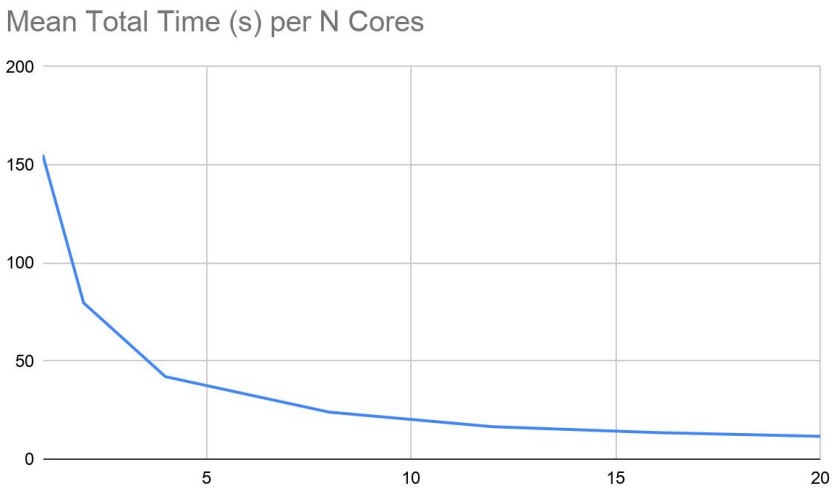
- **Q4: Does the time to sort the data vary as a function of p?**

  Yes. Sort time seems to drop exponentially as the number of cores increases.



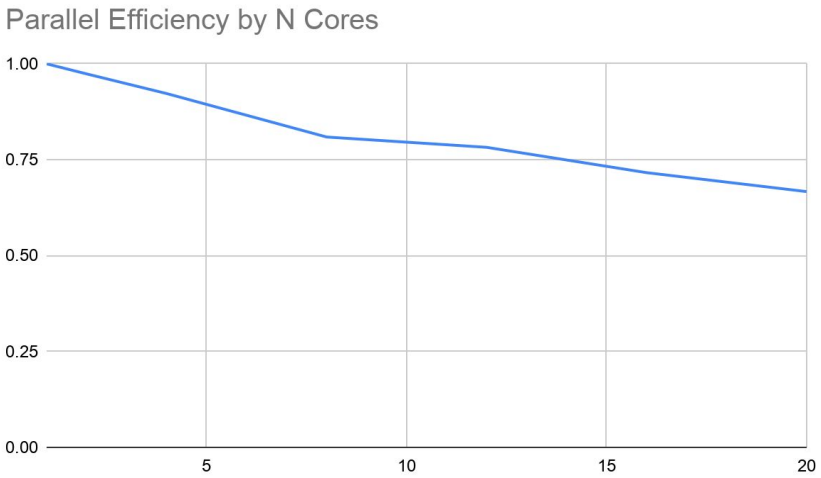
- **Q5: How does distribution sort scale with increasing p?**

Roughly exponentially, as with the sort time, though the curve is slightly shallower because distribution time is nearly constant relative to the larger-scale change we see in sort time. This near-constant factor mitigates speedup as we scale.



- **Q6: How does the parallel efficiency scale with increasing p?**

  It appears to decrease roughly linearly as the number of cores increases. This ~linear reduction lines up with the near-constant time cost imposed by the distribution step.



- **Q7: What is the bottleneck in distribution sort as p increases?**

The distribution step. It has a high enough fixed cost (asymptote) that parallel efficiency suffers as we increase p. During this step, the computer must traverse all N values, assigning each to the appropriate bucket. Though it's possible to decrease the size of the fixed compute cost here (at the cost of increased memory use, the adoption of fragile assumptions about the distribution of the data, or through improvement of communications efficiency), that cost is likely to remain fixed *at some level*, mitigating the decay of the sort time with increases in p.

Activity 2: Code description

The code here is identical to that in Activity 1, with the exception of the change in data distribution.

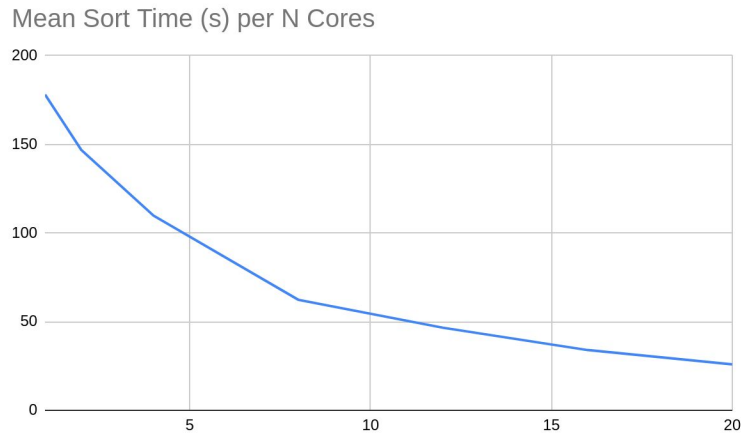
Jobscript/run description
The jobscripts have been refactored slightly here to reduce breakage by:
- compiling once, in the local script, rather than in the jobscript itself
- exporting variables in the calling script rather than setting them in the jobscript. This is experimental, but may solve some very puzzling intermittent errors I ran into with activity 1.

- **Q8: Does performance differ between the uniformly distributed (Programming Activity #1) and exponentially distributed data? If so, explain the performance discrepancy.**

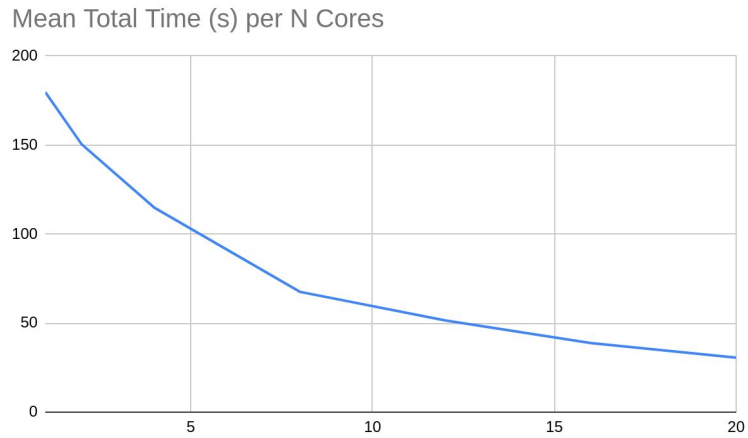
Because our data fits an exponential distribution, and our bucket ranges are uniform, some buckets will receive and sort far fewer values than others. This load imbalance results in significant losses in performance:

	A2 Total Time (s)	A2 Dist Time	A2 Sort Time	Speedup	A2 Par Effic	Global Sum	Jobscript
1	179.728918	1.61666	178.11229	1	1	231347569226388	jobscript.sh + sbatch.command
2	150.386686	3.47358	146.9131	1.1951	0.5976	231345039566716	jobscript.sh + sbatch.command
4	114.812213	5.0257	109.78652	1.5654	0.3914	231348974505153	jobscript.sh + sbatch.command
8	67.582108	5.38491	62.35358	2.6594	0.3324	231346368407353	jobscript.sh + sbatch.command
12	51.519698	5.21173	46.58284	3.4885	0.2907	231344540852148	jobscript.sh + sbatch.command
16	38.801884	4.97446	33.9737	4.632	0.2895	231346601181947	jobscript.sh + sbatch.command
20	30.608124	4.77254	25.89796	5.8719	0.2936	231347819319091	jobscript.sh + sbatch.command

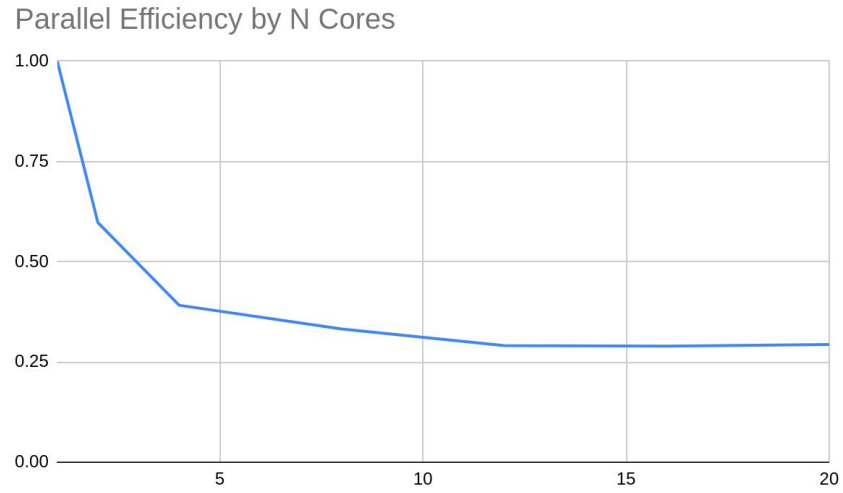
The decay in our sort time curve occurs at a far shallower rate than with the evenly distributed data, and our timings at N=20 are nearly 3x as long.



Because sort time represents the most computationally complex part of the algorithm, this trend bears out in total time.



Parallel efficiency drops rapidly on what may be a logarithmic curve (unlike the ~ linear curve in activity 1), resulting in a parallel efficiency at n=20 greater than 2x worse than with uniform data.



Activity 3: Code description
The distribution sort code here is nearly identical to that in Activity 1, with the exceptions that bucketSize is set in a more semantically correct way, and ranks select bucket start/end ranges from arrays of histogram-calculated values, instead of calculating those ranges on the fly.

The histogram code iterates over rank 0's local data (which is a representative sample of the data at large), populating a "histogram" array of NBINS bin counts. We set a "rough" bucket size to indicate the optimum (even) number of values each rank should calculate, and then iterate over our histogram, capturing an "end index" every time the cumulative sum of values (over all processes) meets or exceeds a multiple of our rough bucket size. This array of threshold values (describing the maximum value to be sorted by the rank where my_rank == array index) is broadcast to all ranks, which construct a corresponding startIndex array locally. Because these arrays are nprocs long, and non-root ranks are blocked until they have startIndex values, I'm betting that computing locally costs less time than computing once and sending.

```
chris@tumbleBox:~/src/hpc599/a3/a3 ((cd606b9...))> mpirun -np 10 -hostfile ../myhostfile.txt ./sort
 CumSum: 1025
 CumSum: 2002
 CumSum: 3015
 CumSum: 4015
 CumSum: 5015
 CumSum: 6013
 CumSum: 7001
 CumSum: 8003
 CumSum: 9002
End indices: 25000 54000 88000 126000 168000 226000 298000 393000 541000 1000000
Start indices: 0 25001 54001 88001 126001 168001 226001 298001 393001 541001
End indices: 25000 54000 88000 126000 168000 226000 298000 393000 541000 1000000
Start indices: 0 25001 54001 88001 126001 168001 226001 298001 393001 541001
End indices: 25000 54000 88000 126000 168000 226000 298000 393000 541000 1000000
Start indices: 0 25001 54001 88001 126001 168001 226001 298001 393001 541001
End indices: 25000 54000 88000 126000 168000 226000 298000 393000 541000 1000000
Start indices: 0 25001 54001 88001 126001 168001 226001 298001 393001 541001
End indices: 25000 54000 88000 126000 168000 226000 298000 393000 541000 1000000
End indices: 25000 54000 88000 126000 168000 226000 298000 393000 541000 1000000
Start indices: 0 25001 54001 88001 126001 168001 226001 298001 393001 541001
End indices: 25000 54000 88000 126000 168000 226000 298000 393000 541000 1000000
Start indices: 0 25001 54001 88001 126001 168001 226001 298001 393001 541001
End indices: 25000 54000 88000 126000 168000 226000 298000 393000 541000 1000000
Start indices: 0 25001 54001 88001 126001 168001 226001 298001 393001 541001
Start indices: 0 25001 54001 88001 126001 168001 226001 298001 393001 541001
End indices: 25000 54000 88000 126000 168000 226000 298000 393000 541000 1000000
Start indices: 0 25001 54001 88001 126001 168001 226001 298001 393001 541001
global distrib time: 0.091863, global sort time: 0.001893, global total time: 0.093362
Number of ranks not sorted properly: 0
Global sum 23231572943 == unsorted global sum 23231572943
```

Test run screen capture on 10 ranks, showing that roughly 1/10 of all values are going to each rank (CumSum lines) and displaying the bucket value bounds each rank would have assigned. This confirms that the implementation is distributing values to ranks for sorting as expected.

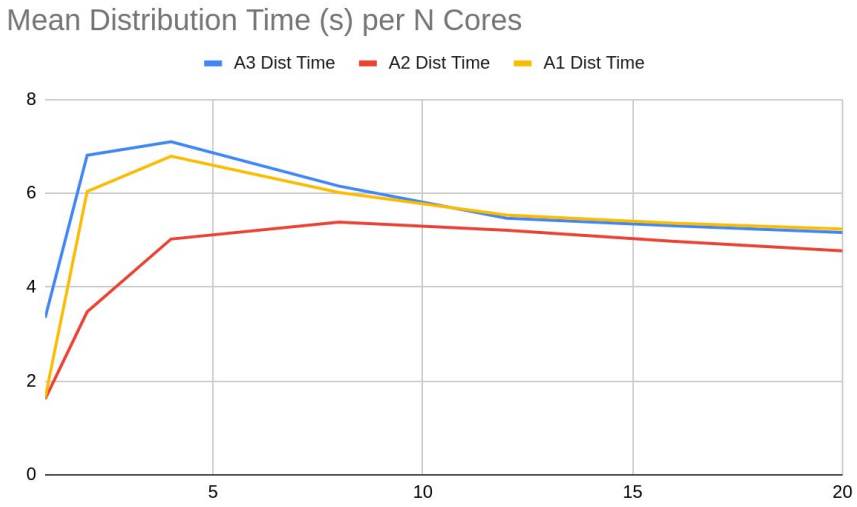
Jobscript/run description - Same as for activity 2.

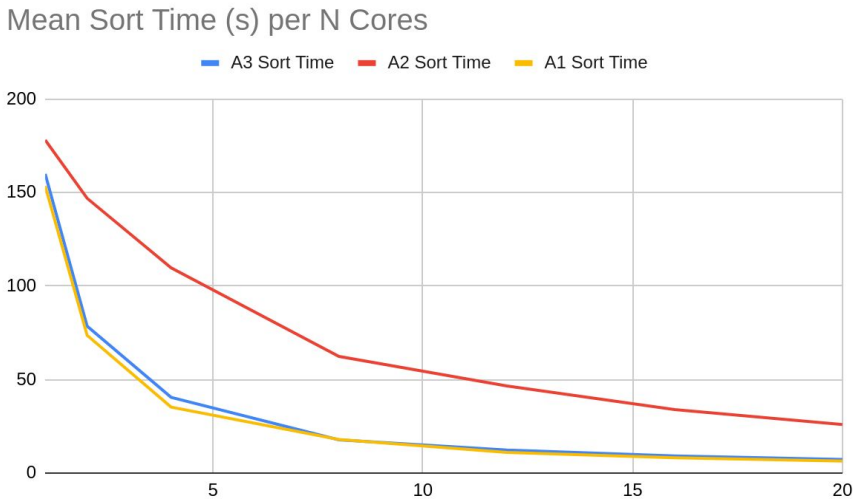
- **Q9: Based on the description above, are there any additional overheads to the algorithm?**

	A3 Total Time (s)	A3 Dist Time	A3 Sort Time	Speedup	A3 Par Effic	Global Sum	Jobscript
1	163.32098	3.3466	159.97438	1	1	231347569226388	jobscript.sh + sbatch.command
2	85.36942	6.81092	78.55849	1.9131	0.9566	231345039566716	jobscript.sh + sbatch.command
4	47.54125	7.10133	40.51413	3.4354	0.8589	231348974505153	jobscript.sh + sbatch.command
8	23.62343	6.15486	17.73197	6.9135	0.8642	231346368407353	jobscript.sh + sbatch.command
12	17.68912	5.4698	12.30675	9.2328	0.7694	231344540852148	jobscript.sh + sbatch.command
16	14.22052	5.30646	9.16168	11.4849	0.7178	231346601181947	jobscript.sh + sbatch.command
20	12.20009	5.16584	7.23948	13.3869	0.6693	231347819319091	jobscript.sh + sbatch.command

Yes. This algorithm adds localN calculations/incrementations to construct the histogram array, NBINS slightly more costly computations to set end indices, and nprocs small calculations to compute start indices, as well as NBINS * sizeof(int) and 2 * nprocs * sizeof(int) arrays of memory overhead, and the communications overhead of MPI_Bcasting one nprocs-long array of integers. As implemented with NBINS = 1000, and in comparison with our data size, the memory overhead is trivial. This may not be the case for much higher bin counts or much smaller data. Optimizing this algorithm for small data seems counterintuitive, and further study of the effect of bin size on performance would clarify whether we need be concerned with high bin counts.

- **Q10: How does the histogram solution compare to the performance achieved in Programming Activity #2?**



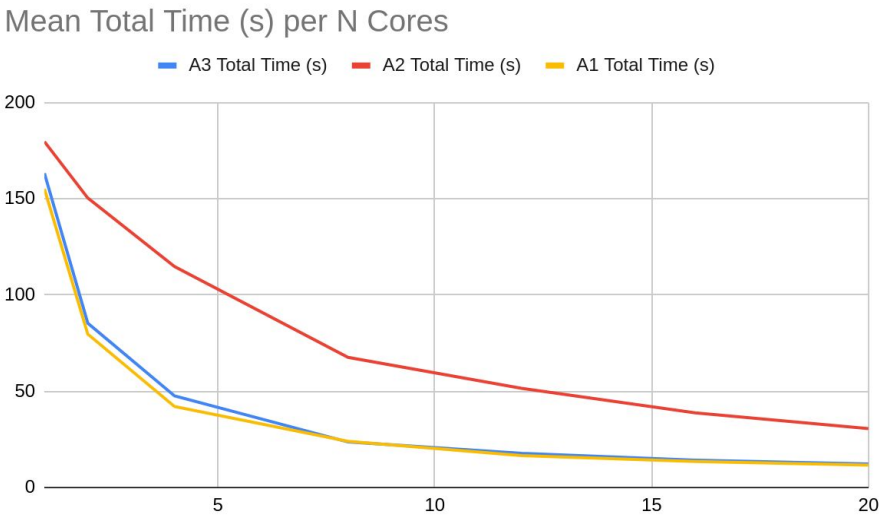


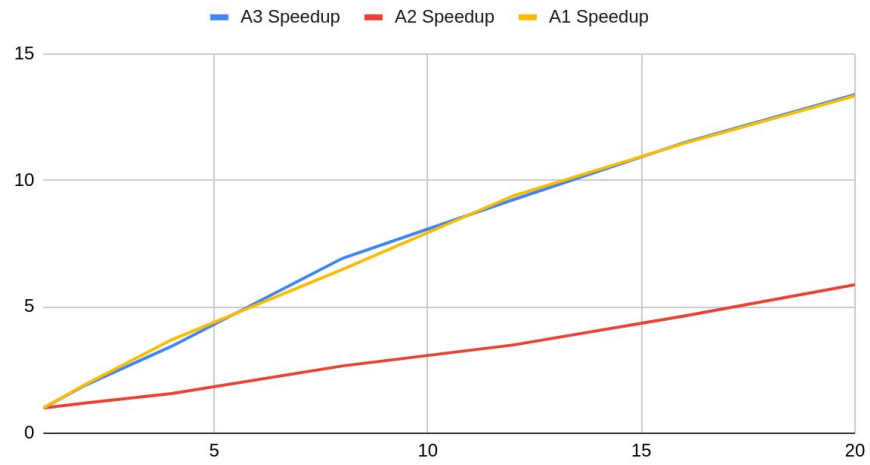
The processing and communications overhead discussed above increases the distribution time for smaller numbers of cores - especially so at p=1. I would argue that p=1 is basically a degenerate case for a distributed-facing sort algorithm, so am unconcerned with that time cost for the most part. Sort time, on the other hand, improves dramatically with the reduction of load imbalance, far outperforming our code from activity 2 for sort time (and the heavily sort-influenced total time). Speedup and parallel efficiency are both far better for a3 than a2.

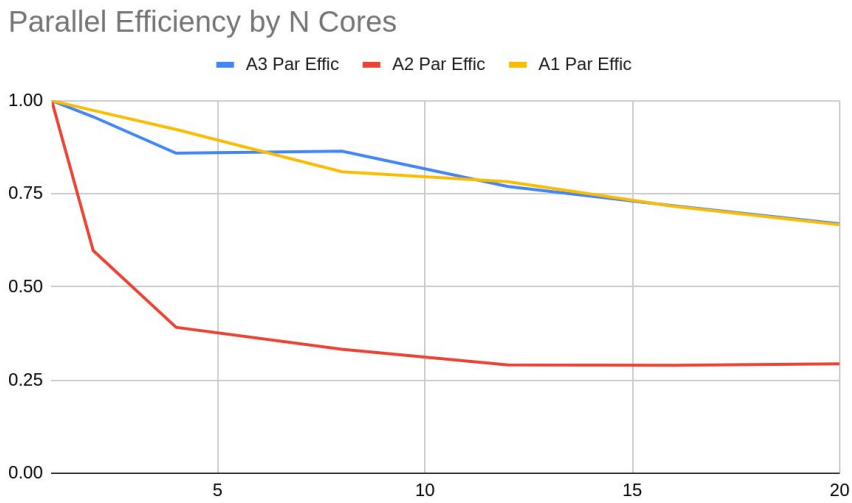
- **Q11: How does the histogram solution compare to the performance achieved in Programming Activity #1?**

  This is the more interesting question, I think. Activity three's algorithm is designed to gracefully handle uneven data distributions, at the cost of some overhead. As discussed above, the computational overhead only impacts our distribution time, and the impact is small enough (given the limited trial we performed) that it has a practical effect only on

runs with p <= 8. Speedup and parallel efficiency curves closely mirror those seen in activity one, and far outperform activity two.







- **Q12: Do you think all distribution sort implementations (e.g., libraries) based on bucketing should use a histogram?**

  No, but only because no solution is universally preferred. Given that the strength of this algorithm lies exactly in its applicability to distributed systems/large-scale data, the small cost (max 1-2 second) to runtimes for small numbers of cores is easily outweighed by the performance improvements gained for non-uniform data distributions. If I had to choose only one, I would certainly choose to use a histogram. Realistically, though, it would be trivial to pass a flag to this method that removed the minimal overhead from the histogram-based bucketing, to allow users with very specific needs to squeeze a little extra performance out of the tool. Would other optimizations be more beneficial? Almost certainly, but the cost in code complexity, api complexity, maintainability, and runtime that would result from adding this logic conditionally would be very minimal.

- **Q13: Can you think of a method that can be used to reduce the overhead of the histogram procedure?**

Sure. Building a histogram from randomly subsampled values could allow us to drop the compute time (especially for large-scale data). This could be implemented as a user-specified behavior, to accommodate different needs pretty simply.

Going in another direction entirely, we could split the procedure into two steps, “training” the bucket start and end values with a histogram, and running the sort. This would sacrifice ergonomics, but would allow users who regularly had to sort similarly-distributed data to “cache” their bucketing scheme and apply it, updating it only as needed.