

Assignment 2
Brian Grenier
1545276
CMPUT 481

March 24, 2021

Summary

Due to the limitation in available hardware from Cybera RAC, it was impossible to scale up the size of the input data enough to feel confident about what the test results implied about the quality of my implementation. As a result, I instead chose to pivot my observations to what impact running the program as a distributed application across many machines, vs. running it on my local machine using purely interprocess communication.

At $p = 8$ and $arrsize \geq 64mil$ the single machine benchmarks show an improvement in speedup of roughly 36%. Looking at figures 7, 9 and 16, 18 the reduction in phase 3 runtime is quite apparent, however, again do to the limitations in hardware, phase two runtime never made any meaningful contributions to the total runtime, and have been omitted from the phase-by-phase analysis. It should be noted that the details about the quality of the hardware at my disposal from Cybera RAC does represent an uncontrolled variable in this experiment, and therefore, it is unknown how the differences in hardware between my local machine, and the Cybera RAC VMs effect the speedup results.

Implementation Details

My implementation was done in C++ (compiled using `mpicxx -Wall -std=c++17 -Ofast`), but must rely heavily on C-style arrays due to the fact that array buffers are the only form of message that the MPICH library understands.

I aimed to use the collective operations in MPI as much as possible, only using `MPI_Gather` and `MPI_Bcast` until phase 3, where `MPI_Isend` and `MPI_Irecv` are used. I made this decision because my algorithm for dividing up the local partitions into its p components generates a separate vector for each of these components. Therefore, using something like an `MPI_Scatterv` operation would require modifications to the algorithm to keep everything in a contiguous array, allowing plenty of problems due to off-by-one errors. Since Phase 3 follows the pattern of "everyone sends everyone else a message", having a simple for loop where every node makes a non blocking send and receive to the i th process, and then simply waits to receive all its messages maps very well to the logic of what needs to be done.

Experimentation Using Random Data

Experiment Setup

Due to the limited resources offered from Cybera RAC creating some strange results at higher array lengths, I also mirrored the test suite on my local machine to compare the results. The baseline for both tests was a single machine sorting the array using `std::sort()`. The tests were run using 2, 4, 6 and 8 cores, on array sizes of 100k, 32mil, 64mil and 96mil. The array sizes I used are much smaller than they were in assignment 1, as I was limited by the fact that each Cybera RAC VM only has 2GB of ram, and therefore, going much past 100mil elements caused massive slowdown, and at times cause the entire program to crash.

Verifying Correctness

During the initial implementation I relied heavily on 36 integer dataset from the PSRS paper, ensuring the results matched at every step of the algorithm. In order to ensure correctness on larger datasets, once the algorithm has completed, I first assert that the array is in fact sorted,

then I sort the original unsorted array using `std::sort()`, and assert that the two arrays are identical.

Results

The timings of each phase follows the guidelines from the PSRS paper, with phase one starting after the data has been distributed to all processors, and phase 4 ending after each processor has finished sorting their local partitions, not including any time it would take to join the partitions back together to the master process.

Cybera RAC Performance

My hardware configuration for Cybera RAC was 4 VMs, each with 2 VCPUs, and 2GB of ram. Due to the nature of this configuration, I had 2 different hostfiles, the first used for core counts 2 and 4, where each node in use would receive exactly one job, and then another for core counts 6 and 8, where each node in use would receive exactly 2 jobs. However, this configuration severely limited the size of the inputs that I could test, as at $arrsize = 96mil$, and $p = 2$, a slowdown of roughly 34% was observed. My hypothesis to explain this anomaly is that near the end of the algorithm, the memory pressure on the master node is too great to maintain reasonable performance.

Since each element in the array is of type `long int`, and therefore 8 bytes in size, and $96000000 * 8 = .768GB$, approaching half of total memory available to the system has been depleted, the master node then will receive the copy of its local partition of the local data, which will be roughly $N/2$ elements, and then another $N/2$ elements during the message passing section in phase 3. Between all of the various message passing and data copying, the master node uses roughly $(N * 2) * 8$ bytes of memory by the completion of the sort, largely due to the fact that the original array of unsorted data must be preserved in order to verify the correctness of the PSRS sort. This issue can be observed in Figure 8, where the time taken for phase 4 at $p = 2$ spikes significantly.

At lower array sizes, the results are much more like what is to be expected (see Figure 9), with a gradual and consistent decrease in phase 1 time, however, I expected to see a more significant improvement in phase 4 as core count increases. In the case of phase 3, it appears that sending multiple smaller messages, is more performant than fewer, larger messages, as is evident by the significant improve in phase 3 between $p = 2$ and $p = 4$. As well as despite the fact that more messages must be sent with $p = 6$ and $p = 8$, many of these messages will be send to another process on the same node, rather than over the network.

Unfortunately due to the limitations in terms of compute resources, an array size large enough to drown out the overhead of using a distributed system is not possible, and the most significant speedup observed is still less than $p/2$.

Times (In Seconds)					
	1 core	2 cores	4 cores	6 cores	8 cores
1000000	0.0972	0.0969	0.0562	0.0515	0.045
32000000	3.6674	3.4467	1.66	1.2528	1.0021
64000000	7.833	7.0901	3.4076	2.5856	2.0775
96000000	11.8871	17.887	5.326	3.8818	3.1673

Figure 1

Speedups				
	2 cores	4 cores	6 cores	8 cores
1000000	1.0034	1.7299	1.8888	2.1606
32000000	1.064	2.2093	2.9274	3.6595
64000000	1.1048	2.2987	3.0295	3.7704
96000000	0.6646	2.2319	3.0623	3.7531

Figure 2

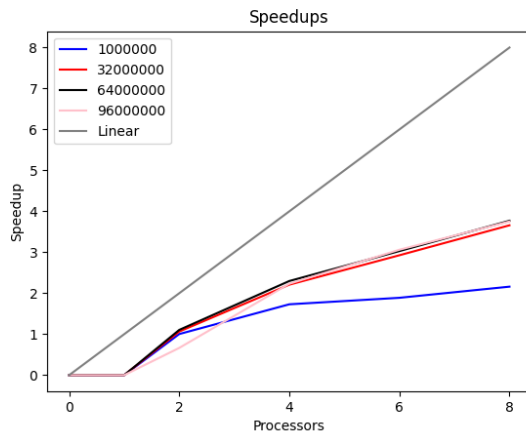


Figure 3

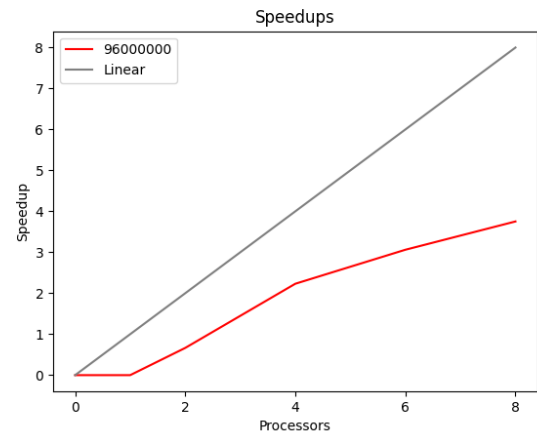


Figure 4

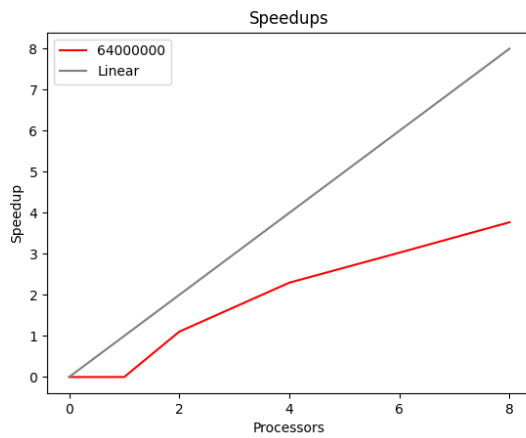


Figure 5

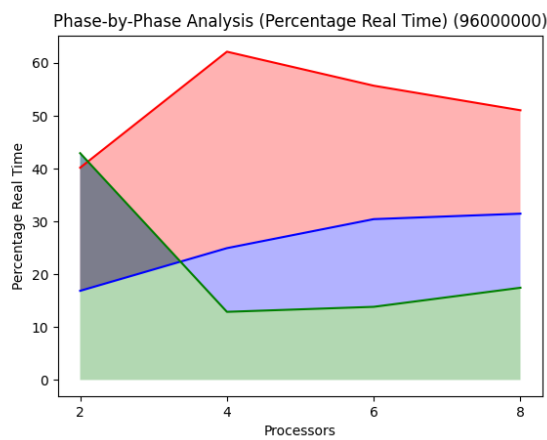


Figure 6: Notice the irregular behaviour of Phase 4

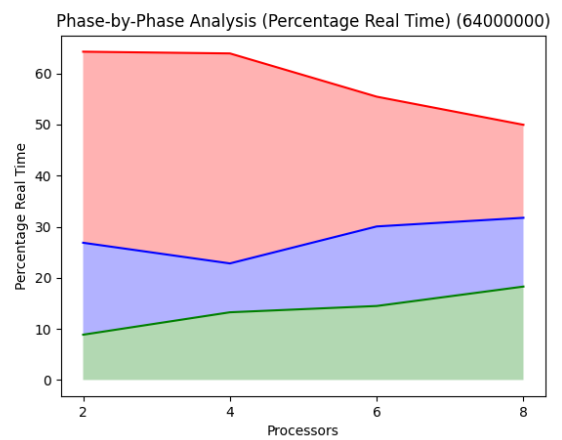


Figure 7

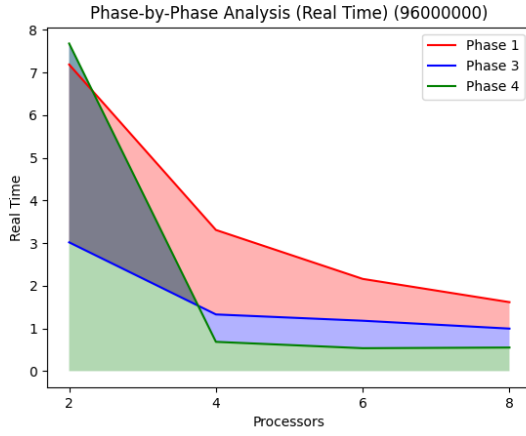


Figure 8: Notice the irregular behaviour of Phase 4

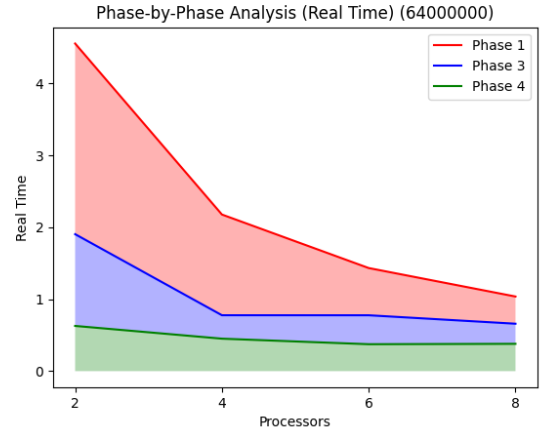


Figure 9

Local Machine Performance

My hardware configuration for the local machine test is an AMD Ryzen 7 3700x, with 32 GB of memory. As is evident in figures 10, and 11, removing the memory constraints, and using purely interprocess communication has improved performance considerably, resulting in a 36% improvement in speedup at $arrsize = 96000000$ and $p = 8$, versus the distributed version. The time savings from using interprocess communication is quite evident in figure 15 and 17m making up only 10-20% of the total runtime, vs. 30-35% in the distributed version.

Times (In Seconds)					
	1 core	2 cores	4 cores	6 cores	8 cores
1000000	0.0536	0.0338	0.0216	0.0163	0.013
32000000	2.0976	1.2725	0.6785	0.4997	0.442
64000000	4.353	2.6109	1.3882	1.0099	0.8536
96000000	6.67	4.0302	2.1491	1.5916	1.3073

Figure 10

Speedups				
	2 cores	4 cores	6 cores	8 cores
1000000	1.5832	2.4762	3.2949	4.1108
32000000	1.6485	3.0917	4.1979	4.7461
64000000	1.6673	3.1357	4.3104	5.0997
96000000	1.655	3.1036	4.1906	5.1022

Figure 11

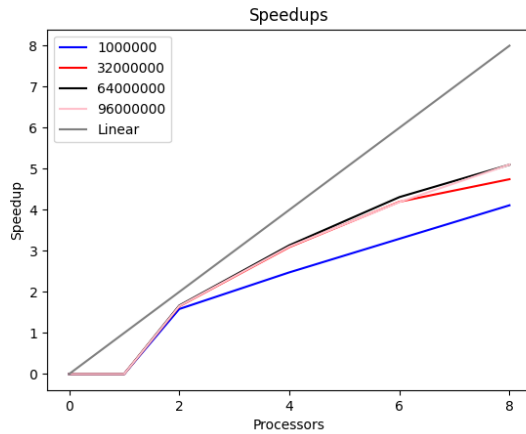


Figure 12

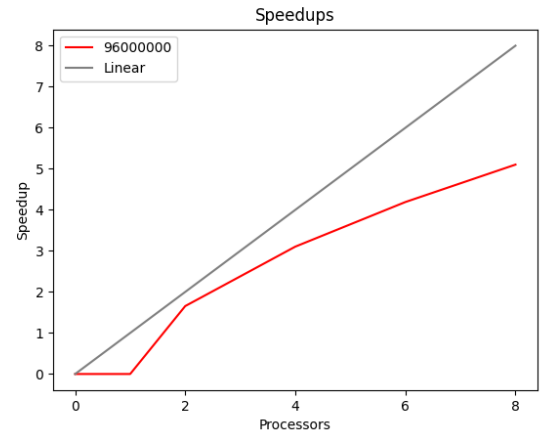


Figure 13

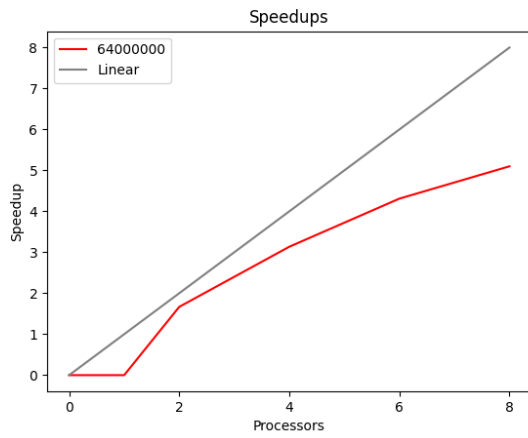


Figure 14

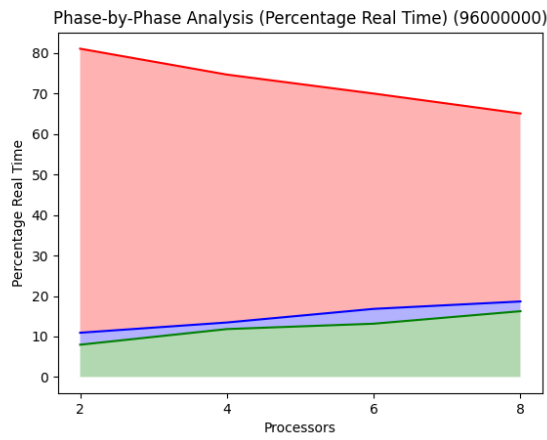


Figure 15

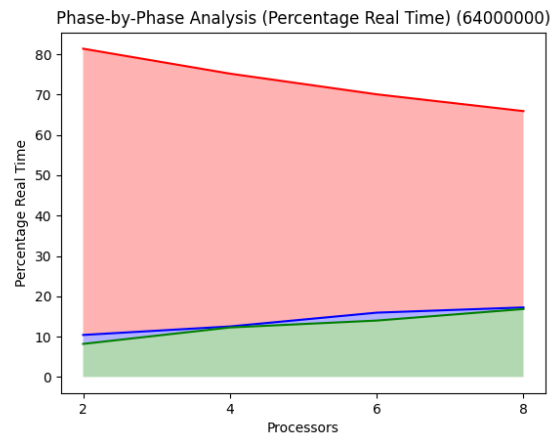


Figure 16

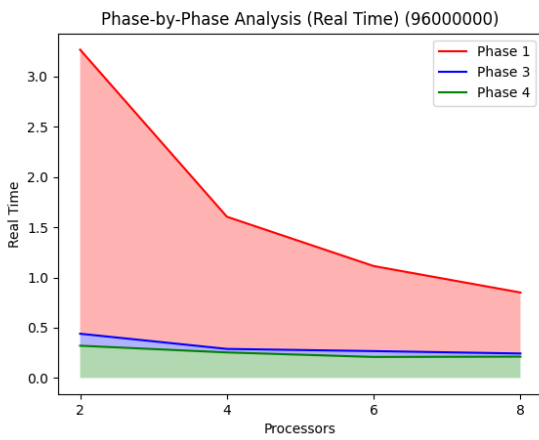


Figure 17

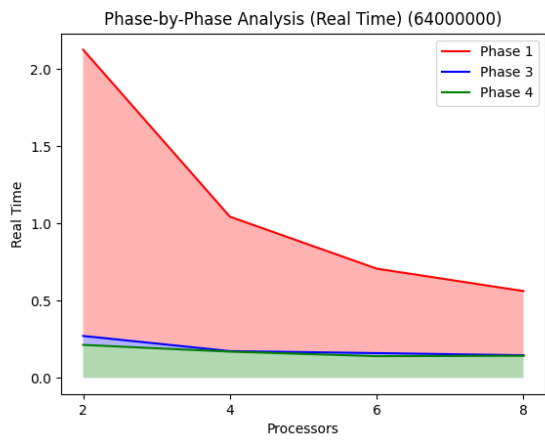


Figure 18

Conclusion

Weakness in Implementation

In phase 3, I chose to use a 2d `std::vector`, and expanded the partitions using the `std::vector.push_back()` operator, which means at larger input sizes, these vectors likely must resize multiple times each, damaging runtime. However, using a fixed size array would require estimating the size that each partition should be, which could prove to be dangerous, so I opted to use `std::vector` in the name of safety, at the expense of runtime, and likely memory usage (since `std` containers tend to overallocate memory in order to avoid resizing too often).

Lessons/Conclusions

While the data does clearly demonstrate the performance price we must pay in order to utilize message passing over a network, I fear the awkward hardware configuration I used, specifically at $p = 6, 8$, where each VM receives 2 jobs, meaning that not every message will be sent over the network, might have muddied my results somewhat. I could have avoided that issue had I used a configuration of 8 single core VMs, however, in that case each VM would only get 1GB of ram each, which would have worsened the memory issues I already experienced, especially on the master node.

While MPI is an excellent, and nearly unbelievably easy tool to use, in writing PSRS as a distributed memory program (vs. the shared memory implementation from A1), I often felt as though my hands were tied in terms of making optimizations. Using a shared memory approach, you are able to improve granularity through the intelligent (and cautious) use of shared data structures, especially in phase 3. However, using a distributed memory approach, if one process needs information about another process, you have no choice but to pay the message passing overhead, and otherwise rely on the library to make any possible optimizations through using aggregate operations such as `MPI_Scatter`, and `MPI_Gather`. Leaving the only improvements we can make as programmers is to improve the time complexity of our algorithms, and in the case of PSRS, the work done by each process is relatively simple, and does not leave much room for meaningful optimizations.