Parallel Sorting Using Message Passing Model

HingOn Miu (hmiu) An Wu (anwu)
Carnegie Mellon University

Steps of Optimization:

1. We have a discussion about whether the samples should be generated sequentially or in parallel. If the samples are generated sequentially, all other processes would need to send over their local data arrays to root process. If the samples are generated in parallel, then all other processes have to send over their local sample arrays to root process. Since a local data array is much larger than a local sample array when the data size is large, the communication cost of the sequential method is obviously bigger than the parallel method. Moreover, the sequential method puts more workload on the root process and so it makes the other processes hang (waiting to receive pivots) for a longer time than the parallel method. Hence, the sequential method has lower resource utilization than the parallel method. So, we decide that the samples should be generated in parallel.

2. We tried to use for loops to receive and send data across processes. However, these receive and send are blocking calls and so these sequential for loops are slow. Therefore, we decide to use gather and broadcast to pass local samples and pivots across processes, and it is proven to be much faster.

3. After bucketing the local data, each process should send each bucketed data to each responsible process. We tried the naive solution that uses a for loop and sends bucketed data to each process sequentially. Expectedly, this method ran pretty slow. Then, we decide to use alltoall and alltoallv instead of the sequential send and receive method. First, each process uses alltoall to inform all other processes how many data it is going to send to them. Next, since each process knows how many data all other processes is going to send to it, each process uses alltoallv to receive data from all other processes. This smarter method turns out to run much faster.

4. Our sorting function ran well on smaller data set (eg. 1M), but it got a segmentation fault when it ran on bigger data set (eg. 100M). We later figured out that the system ran out of memory because of the large static 2D array declared locally. This 2D static array was used to bucket the data to prepare to send to each processes. For a large data size N, each process need to declare a static 2D array of size N * P. If there are 100 processors and 1 billion floats to sort, then each process would require 1 billion * 100 * sizeof(float) = 400 billions bytes. This is very memory inefficient. Hence, we decide to use an array of std::vector's instead to save memory space, and so each process would only require to allocate memory for N / P floats on average for bucketing the local data.

5. Finally, we conclude that it makes no sense to go through all the processes communication codes when there is only 1 processor in the whole system because they essentially make no changes. So, we add a simple if-statement at the top such that our function simply sorts the data array when the processor count is 1.

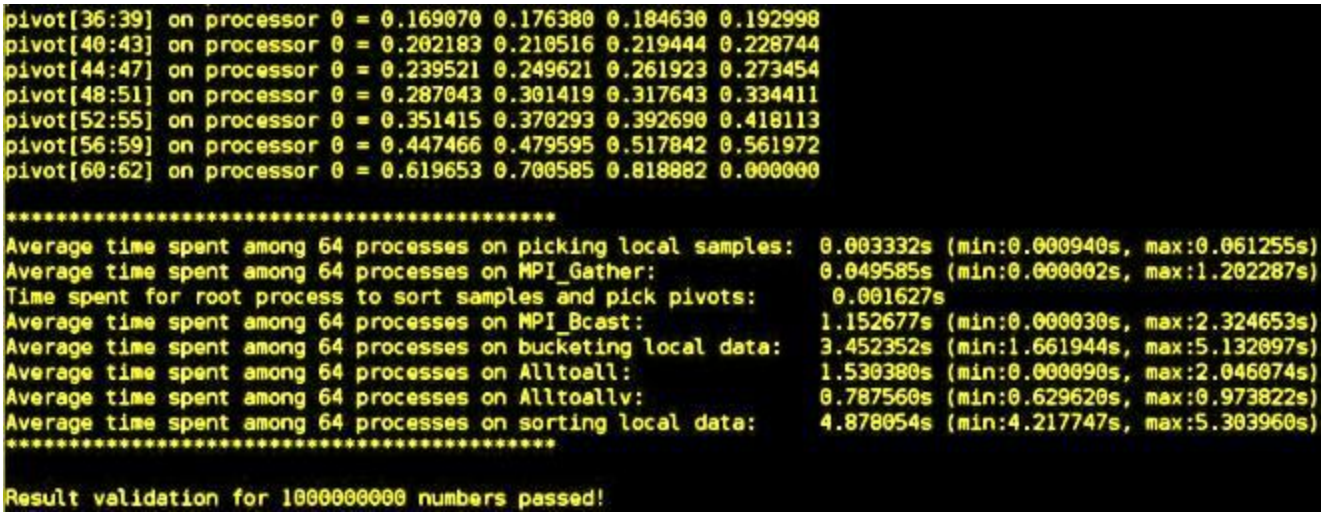
Questions:

A. In the 1 million element array case, speedup for large P may be quite poor. Please describe why the shape of the speedup graph looks the way it does.

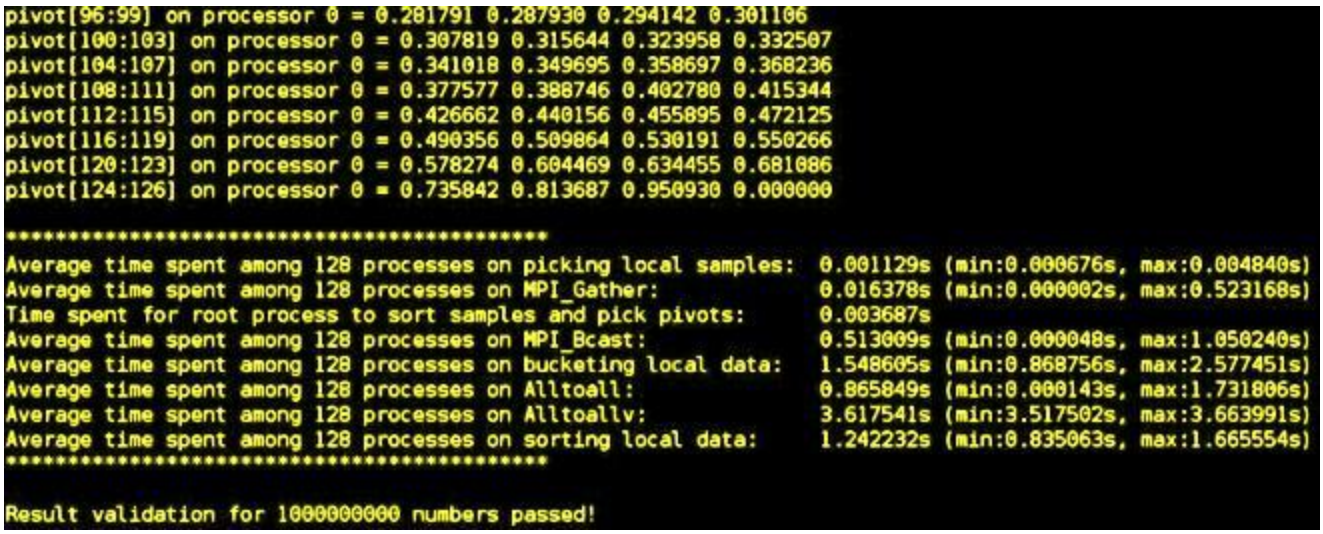
   For a large P like 64, each process averagely gets 1,000,000 / 64 ≈ 15600 local elements. For a larger P like 128, each process averagely gets 1,000,000 / 128 ≈ 7800 local elements. We observe that our sorting function takes more time on P = 128 than P = 64, and the only noticeable difference between them is the average number of local input elements. Therefore, we know the decrease of local input elements results in the increase in runtime for large P. An explanation for this behavior is that the increased communication cost among processors is more significant than the reduced cost in sorting the local data array, which puts a limit on increasing processors/decreasing local problem size.

B. What are reasons why speedup is not perfect for large P even when the array size is large. Is it workload imbalance? Is it communication costs? Make measurements of your program (time spent in each phase, time spent communicating, waiting for a barrier etc.) that allow you to support your answer with real data. (Much like we did in Assignment 1)

Timing of N = 1B and P = 64:



Timing of N = 1B and P = 128:



Communication cost:

When the number of processors goes from 64 to 128, there is a huge increase (3.61754 / 0.78756 ≈ 4.6 times) in average time spent of passing the bucketed data around all processors through Alltoallv. So, the communication cost increases significantly.

Workload imbalance:

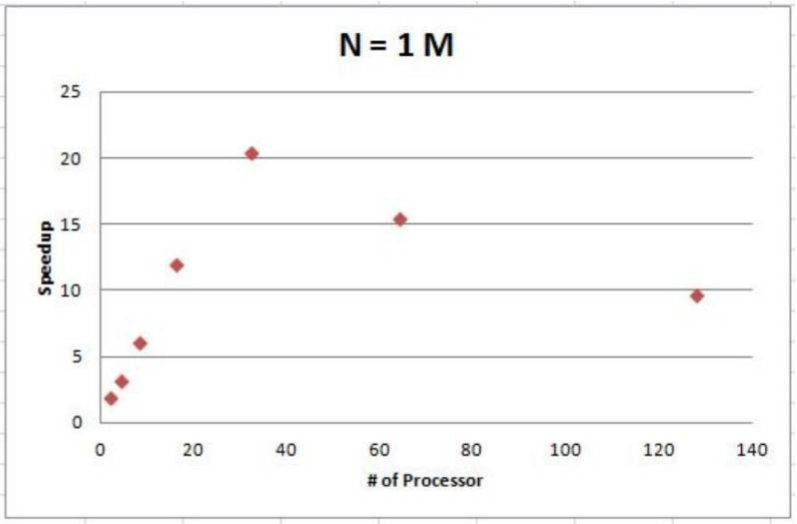
When P = 64, the minimum runtime of local data sort is (4.87805 - 4.21774) / 4.87805 ≈ 13% lower than the average, while the maximum runtime of local data sort is (5.30396 - 4.87805) / 4.87805 ≈ 8% higher than the average. When P = 128, the minimum runtime of local data sort is (1.24223 - 0.83506) / 1.24223 ≈ 33% lower than the average, while the maximum runtime of local data sort is (1.66555 - 1.24223) / 1.24223 ≈ 35% higher than the average. Obviously, the workload imbalance becomes more significant.

Tables & Graphs:

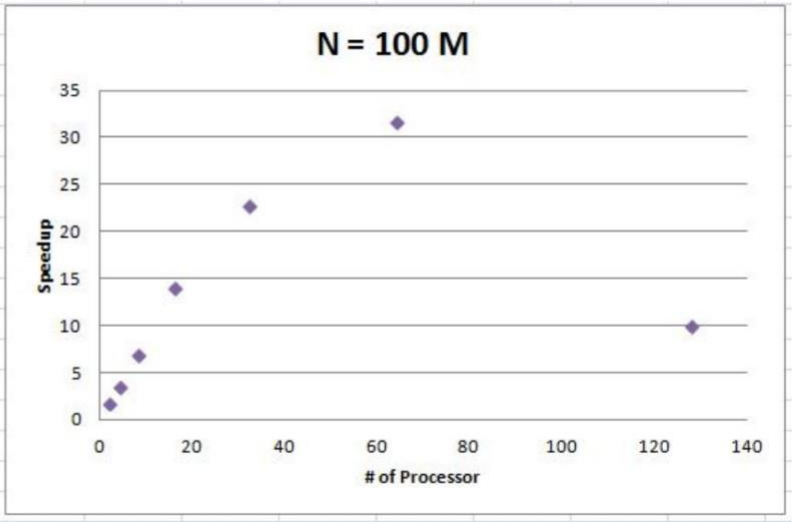
Runtime on Blacklight:

Problem Size	# of Processors						
	2	4	8	16	32	64	128
1 M	0.0681s	0.0395s	0.0211s	0.0107s	0.0063s	0.0083s	0.0132s
100 M	9.1643s	4.6769s	2.3573s	1.1814s	0.7264s	0.5252s	1.6537s
1 B	90.9310s	49.5621s	25.5713s	13.6720s	7.3395s	5.2544s	6.3210s

# of Processor	Speedup	Serial	Parallel
2	1.885462555	0.1284	0.0681
4	3.250632911	0.1284	0.0395
8	6.085308057	0.1284	0.0211
16	12	0.1284	0.0107
32	20.38095238	0.1284	0.0063
64	15.46987952	0.1284	0.0083
128	9.727272727	0.1284	0.0132



# of Processor	Speedup	Serial	Parallel
2	1.814355706	16.6273	9.1643
4	3.555196818	16.6273	4.6769
8	7.053535825	16.6273	2.3573
16	14.07423396	16.6273	1.1814
32	22.89000551	16.6273	0.7264
64	31.65898705	16.6273	0.5252
128	10.05460483	16.6273	1.6537



# of Processor	Speedup	Serial	Parallel
2	1.88418911	171.3312	90.931
4	3.456899526	171.3312	49.5621
8	6.700136481	171.3312	25.5713
16	12.53153891	171.3312	13.672
32	23.34371551	171.3312	7.3395
64	32.60718636	171.3312	5.2544
128	27.10507831	171.3312	6.321