CMSE 822 Homework 6

Mengzhi Chen and Tong Li

In this report, we analyze the performance of our implementation of parallel bucket sort algorithm using MPI. In part 1, we compare the performances of single-source and all-source bucket sort programs. By measuring the execution time of different, we conclude that the all-source version performs better because of load balance. In part 2, in order to remedy load imbalance caused by non-uniform distribution, we select pivots based on sampling. Then the gains and losses of pivot selection are discussed.

Part 1

- Modules GNU/4.9 and OpenMPI/1.10.0 are used.
- All data are average of four repetitions.
- Version 1 (v1) is the single-source bucket sort program and version 2 (v2) is the all-source one.

a)

We test our programs on HPCC, starting from single socket (14 cores) and using up to five nodes (140 cores). Also, the problem sizes N are 100 million (10^8), 500 million (5×10^8) and 2 billion (2×10^9). The total execution times, efficiencies and speedups are shown in Fig 1.

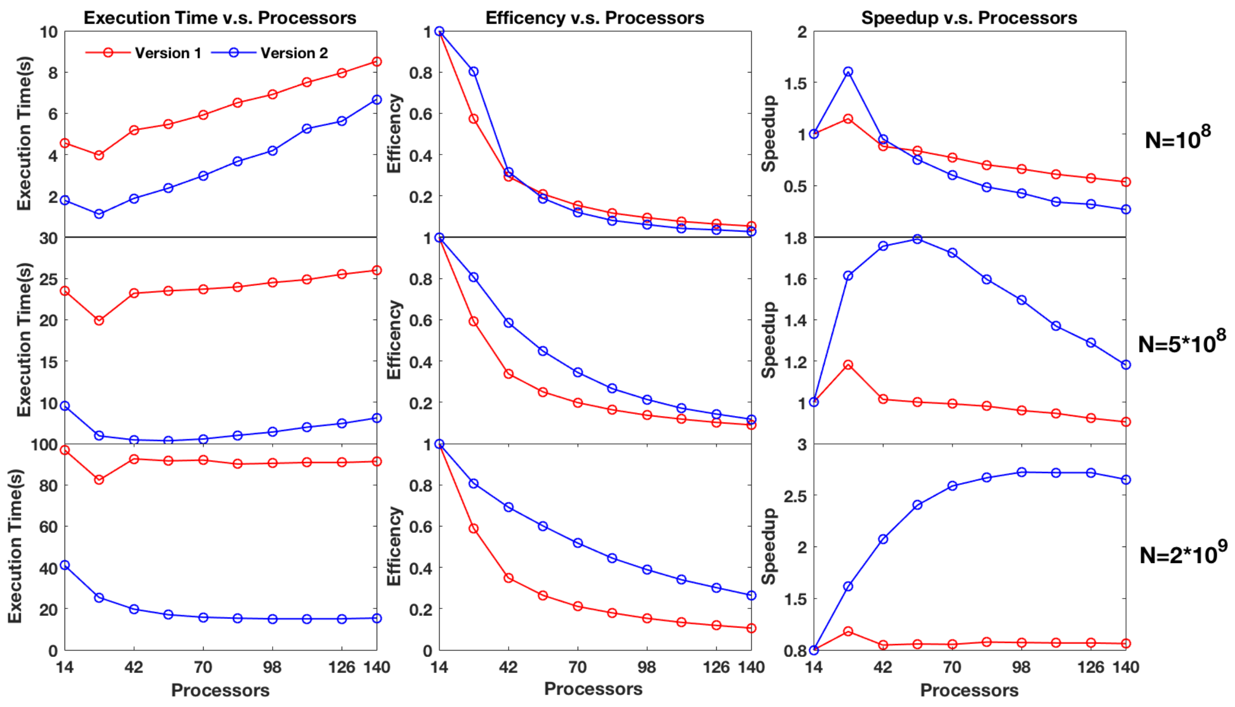

Fig 1

Both two versions do not scale well or even perform worse with increasing process number (p). We believe that it is the result of competition between increasing communications and decreasing local calculations. Moreover, we see that as N grows, v2 performs better than v1. This can be understood as in v1 root process bears almost all preparation load which increases as N goes up. However, in v2 loads are balanced among all processes.

b)

To see why v2 performs better than v1, we measure the execution time of different phases. The whole algorithm is partitioned into five phases: array generation, bin determination, array distribution, local sort and result gather. We show the situation with N = 2 billion in Table 1, which is characteristic enough for our analysis.

The major differences between v1 and v2 locate in the first three phases. V1 spends much more time than v2 on these parts. Unfortunately, these phases contribute significantly to total execution time, especially with large process number. In v1 root process bears a heavy load, which seriously hurts the performance. However, v2's processes share almost the same load, which raises its efficiency. The two versions show similar performance in local sort phase, because each process works independently without communication in this phase. V1 performs better than v2 in gather phase because less communication occurs in v1's gather part, but this difference is too small. Therefore, v2 has a better load balance and is much faster than v1.

Table 1 (N = 2 billion)

	Generate time (s)		Bin time (s)		Distribute time (s)		Local sort time (s)		Gather time (s)	
#processes	v1	v2	v1	v2	v1	v2	v1	v2	v1	v2
14	42.07	2.98	13.14	1.00	6.84	1.02	31.20	28.68	3.71	7.35
28	42.10	1.53	13.71	0.67	7.08	1.05	15.68	14.54	3.73	7.61
42	41.73	1.00	29.63	0.82	7.53	1.44	10.24	9.47	3.46	7.04
56	42.41	0.82	30.29	0.64	8.05	1.93	7.55	6.71	3.32	6.96
70	42.85	0.60	31.27	0.50	8.59	2.63	6.02	5.34	3.26	6.78
84	41.75	0.50	31.33	0.42	9.10	3.29	4.88	4.41	3.20	6.76
98	41.99	0.43	31.29	0.36	9.64	3.84	4.15	3.62	3.19	6.82
112	41.86	0.38	31.97	0.31	10.15	4.49	3.63	3.21	3.17	6.71
126	41.76	0.34	31.76	0.29	10.74	5.10	3.22	2.68	3.17	6.68
140	42.02	0.30	31.91	0.25	11.42	5.79	2.85	2.49	3.15	6.63

Part 2

- Modules GNU/4.9 and OpenMPI/1.10.0 are used.
- All data are average of four repetitions.

1.

Version 2 (v2) is the all-source bucket sort program using random numbers with a uniform distribution, while v2_sqaure uses the square of each number from a uniform distribution.

The two scenarios' total execution times (with N = 2 billion) are shown in table 2. We see that v2_sqaure takes almost three times longer time than v2 with either p. Thus, date distribution does have impact on performance. Non-uniform distribution causes load imbalance which hurts the performance.

Table 2 (N = 2 billion)

	Uniform Distribution (v2)			Squared Distribution (v2_sqaure)		
#processes	Min time (s)	Max time (s)	Ave time (s)	Min time (s)	Max time (s)	Ave time (s)
14	40.90	41.33	41.03	125.59	127.01	126.29
28	25.29	25.49	25.41	88.40	88.58	88.52
42	19.61	20.12	19.77	73.11	73.35	73.21
56	16.93	17.33	17.07	62.88	63.21	63.01
70	15.74	16.02	15.85	57.15	58.35	57.70
84	15.27	15.53	15.37	53.64	53.88	53.74
98	14.98	15.13	15.07	49.73	50.48	49.97
112	14.98	15.16	15.10	47.76	48.26	47.91
126	15.07	15.13	15.10	45.30	45.54	45.40
140	15.37	15.59	15.47	44.62	44.75	44.69

3.

The square of random numbers leads to load imbalance, so selecting pivots by sampling is added to v3 to balance the load among all processes. The total execution times, efficiencies and speedups of v3 are shown in Fig 2.

The tendencies shown in Fig 2 are very similar to those in Fig 1. For N = 2 billion and p = 14, compared with v2_square, the total execution time drops from ~126 s to ~46 s. For other N and p, v3 also performs better. Thus, v3 fixes the load imbalance problem.

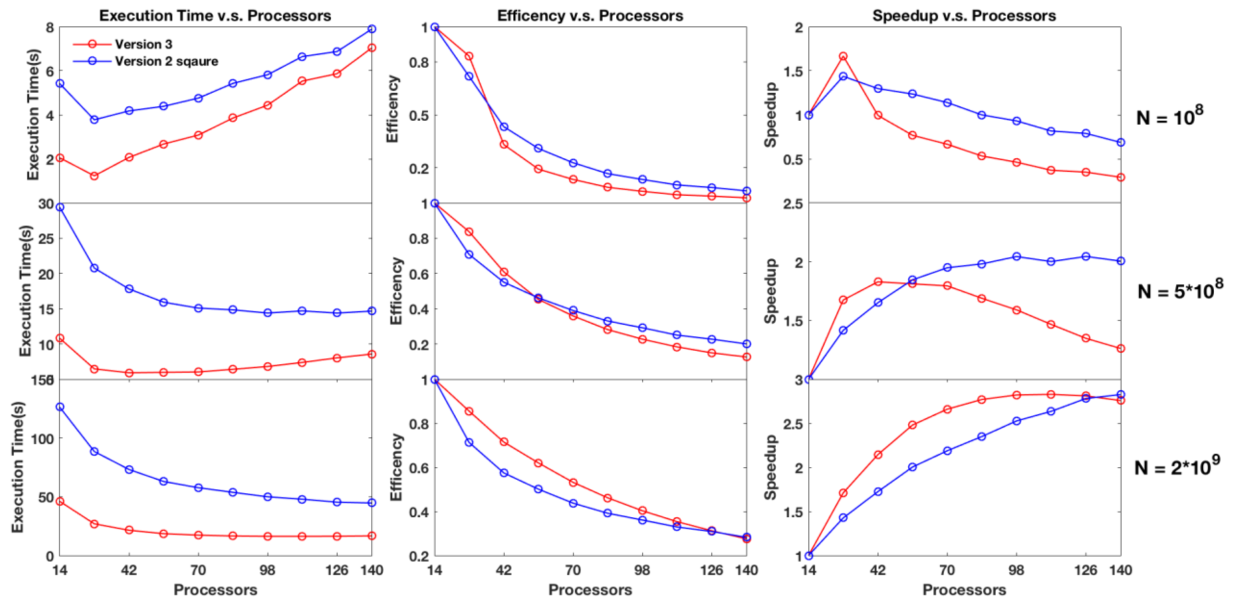

Fig 2

4.

To see how v3 fixes the problem, we measure the execution time of different phases. For clarity, only local sort and pivot selection phases will be discussed below (see Table 3). When p = 14, by consuming ~0.1 s for selecting pivots, we win back ~80 s in local sort phase. For other p, time of pivot selection is shorter than the difference of local sort time between v2_square and v3. Therefore, it is worth selecting pivots.

Table 3 (N = 2 billion)

		Local sort time (s)		Pivots time (s)
#processes	v2_sqaure	v3	Difference	v3
14	113.73	28.68	85.05	0.08
28	78.03	14.54	63.49	0.13
42	63.17	9.47	53.7	0.15
56	53.00	6.71	46.29	0.29
70	47.37	5.34	42.03	0.14
84	42.90	4.41	38.49	0.10
98	38.74	3.62	35.12	0.10
112	36.10	3.21	32.89	0.12
126	33.09	2.68	30.41	0.15
140	31.80	2.49	15.47	0.15