Peter Lesslie

2208496

Abstract

Three sorting methods are compared: InsertionSort, MergeSort, and ColumnSort. ColumnSort is run using InsertionSort and MergeSort as auxiliary sort methods. The complexity of ColumnSort is experimentally determined. ColumnSort is identified as a reasonable distributed sort method.

Procedure

Testing was done using the std::chrono::high_resolution_clock library. Data was generated by the standard cstdlib library rand() function. In the GNU implementation, which is used by the cycle2 servers, RAND_MAX is well over 2,000,000, so data could be generated correctly for the 1,000,000 element test (on some implementations RAND_MAX is only (2^15)-1). Though rand() is known to be prone to cycles in the lower bits of the numbers generated, the true randomness of the data generated was not particularly important for this test, so an auxiliary library was not warranted. The timer was started inside the function call (so as to not count the time to push arguments on to the stack and jump to the function location). The timer was stopped after the array was fully sorted—in MergeSort and ColumnSort, the timer was stopped before deleting the extra memory allocated during the sort. Each sort method was tested 10 times for each number of elements (36,000, 72,000, 144,000, 288,000). ColumnSort was tested with 2, 3, 5, and 10 rows (except for 1,000,000, which was not tested with 3 because 1,000,000 mod 3 != 0).

Verification of Correctness

The correctness of the algorithms was checked by using the sort method in question on a large (>100,000 elements) array and comparing the output to the output from the std::sort method. The likelihood of sorting n elements by chance is 1/n! (if no duplicates), so it is very unlikely that with random inputs the sort method would be able to sort the data correctly and the method not be correct. The test drivers were also checked with AddressSanitizer, MemorySanitizer, and Valgrind to check for going beyond array bounds. The test programs can be found in the subdirectories (InsertionSort/, MergeSort/, or ColumnSort/) in the file *driver.cpp*. Macros, rather than passing a functor, choose the auxiliary function used to save the extra function calls needed to wrap the function in a struct.

Results

Table 1: Insertion Sort Timing Results

Insertion Sort				
	Number of Items (times in seconds)			
Run #	36000	72000	144000	288000
1	0.25146	0.893166	3.356716	14.3385
2	0.255461	0.880578	3.55282	14.3428
3	0.254405	0.882456	3.56978	14.3086
4	0.254131	0.88058	3.54875	14.7288

5	0.254713	0.880205	3.56983	14.6215
6	0.254039	0.878086	3.69697	14.3621
7	0.255697	0.89537	3.5563	14.3528
8	0.268409	0.878888	3.5837	14.3515
9	0.269252	0.879173	3.56641	14.7115
10	0.24348	0.879288	3.57361	14.4798
Average	0.256105	0.882779	3.557489	14.45979

Table 2: Merge Sort Timing Test Results

Merge Sort				
	Number of Items (times in seconds)			
Run #	36000	72000	144000	288000
1	0.004545	0.009579	0.020248	0.042717
2	0.004473	0.009475	0.020045	0.042202
3	0.004426	0.009401	0.019938	0.042176
4	0.004423	0.009409	0.019959	0.042174
5	0.004429	0.009416	0.019959	0.042188
6	0.004422	0.009429	0.019958	0.042137
7	0.004426	0.009407	0.019918	0.042084
8	0.004426	0.009427	0.019954	0.042167
9	0.004431	0.009415	0.019957	0.042143
10	0.004434	0.009448	0.019949	0.042051
Average	0.004444	0.009441	0.019989	0.042204

Column Sort Using Insertion Sort, with 36,000 elements				
	S value (times in seconds)			
Run #	2	3	4	5
1	0.186494	0.136888	0.0777014	0.0423064
2	0.0187155	0.120242	0.078038	0.0421646
3	0.186581	0.119234	0.0782669	0.0422655
4	0.186608	0.136256	0.0777616	0.0421205
5	0.186668	0.140418	0.0781041	0.0422175
6	0.186652	0.143694	0.0780384	0.0422777
7	0.187384	0.143342	0.078074	0.0423863
8	0.186532	0.143136	0.0779711	0.0424645
9	0.186474	0.141181	0.0778924	0.0421255
10	0.186077	0.139688	0.0779244	0.0422437
Average	0.16981855	0.1364079	0.07797723	0.04225722

Column Sort Using Insertion Sort, with 72,000 elements

	S value (times in seconds)			
Run #	2	3	4	5
1	0.658813	0.482638	0.315224	0.165236
2	0.656147	0.484362	0.3109	0.164766
3	0.657033	0.485742	0.311987	0.165324
4	0.656318	0.483526	0.311176	0.165402
5	0.659014	0.483743	0.311577	0.165298
6	0.676081	0.48296	0.311975	0.167186
7	0.792932	0.541157	0.31094	0.168566
8	0.658524	0.57888	0.31186	0.168113
9	0.658815	0.48413	0.313236	0.168169
10	0.658978	0.483547	0.311284	0.169137
Average	0.673266	0.4990685	0.3120159	0.1667197

Column Sort Using Insertion Sort, with 144,000 elements

	S value (times in seconds)			
Run #	2	3	5	10
1	2.6484	1.95446	1.26029	0.662646
2	2.6791	1.97208	1.27091	0.672642
3	2.69333	1.94883	1.25774	0.661181
4	2.63736	1.9526	1.25956	0.660387
5	2.69937	1.96406	1.26314	0.660619
6	2.6455	1.9504	1.2711	0.659131
7	2.6795	1.95323	1.26637	0.660223
8	2.64645	1.96198	1.25975	0.660485
9	2.65915	1.95179	1.25812	0.660306
10	2.64794	1.95393	1.26192	0.66266
Average	2.66361	1.956336	1.26289	0.662028

Column Sort Using Insertion Sort, with 288,000 elements

	S value (times in seconds)			
Run #	2	3	5	10
1	10.6938	7.91307	5.07644	2.67783
2	10.8073	7.92573	5.07167	2.6626
3	10.9084	7.89987	5.07711	2.67226
4	10.7565	7.91756	5.07931	2.67154
5	10.8039	7.93367	5.0811	2.65994
6	10.9004	7.8812	5.08397	2.67675
7	10.9399	7.89681	5.07279	2.66092

8	10.8979	7.8656	5.07121	2.82573
9	10.9007	7.87547	5.08461	2.66179
10	10.8123	7.88778	5.08405	2.71427
Average	10.84211	7.899676	5.078226	2.688363

Column Sort Using Insertion Sort, with 1,000,000 elements

	S value (times in seconds)		
Run #	**2**	**5**	**10**
1	133.53	62.0504	32.6755
2	132.666	620215	32.4595
3	133.047	62.4833	32.5279
4	134.144	62.5771	32.4943
5	132.532	62.2667	32.4931
6	133.17	62.2797	32.668
7	131.381	62.3853	32.9073
8	131.03	62.2093	32.855
9	129.951	63.5028	32.5651
10	132.864	62.8162	32.7675
Average	132.4315	62077.75708	32.64132

Column Sort Using Merge Sort, with 36,000 elements

	S value (times in seconds)			
Run #	**2**	**3**	**5**	**10**
1	0.00993096	0.0112376	0.0112338	0.0093549
2	0.0127444	0.0093043	0.010039	0.0119541
3	0.00953909	0.0121445	0.0093154	0.0094426
4	0.0123306	0.0093075	0.0117109	0.0111637
5	0.00986946	0.0126004	0.0092717	0.0100937
6	0.0116479	0.0093089	0.0120265	0.0096508
7	0.0102986	0.0127358	0.0092932	0.0111384
8	0.0108703	0.0093134	0.0125625	0.0092477
9	0.0107887	0.0115169	0.0092927	0.0119347
10	0.00981142	0.0099446	0.0121999	0.0092483
Average	0.010783143	0.0107414	0.0106946	0.0103229

Column Sort Using Merge Sort, with 72,000 elements

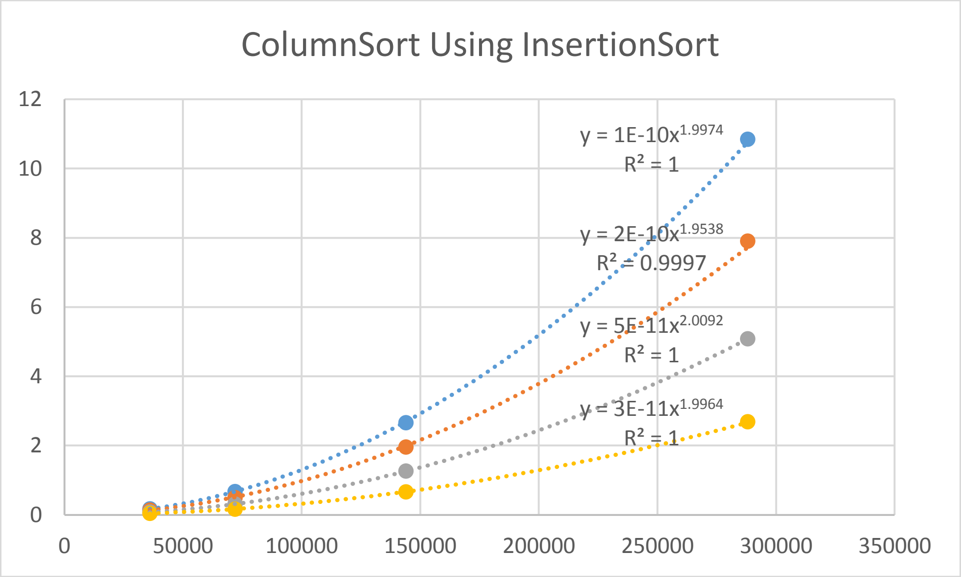
	S value (times in seconds)			
Run #	**2**	**3**	**5**	**10**
1	0.0224565	0.0195993	0.0194996	0.0194219

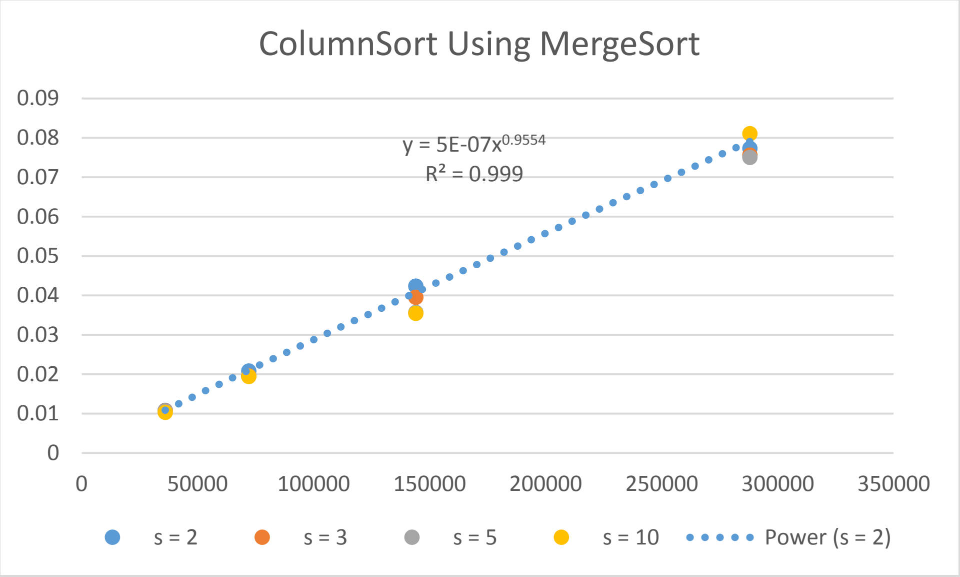
2	0.0223678	0.019621	0.0194982	0.0194377
3	0.0223808	0.0195892	0.0194944	0.0194296
4	0.0200901	0.0196135	0.0195116	0.0194464
5	0.0200801	0.019606	0.0194992	0.0194445
6	0.0200926	0.0196008	0.0195037	0.0194304
7	0.0200876	0.0195924	0.0194968	0.0194376
8	0.0200982	0.0195955	0.0194909	0.0194228
9	0.0200741	0.0195858	0.0194938	0.0194355
10	0.0200919	0.0196011	0.0194848	0.0194269
Average	0.020782	0.01960046	0.0194973	0.0194333

Column Sort Using Merge Sort, with 144,000 elements				
	S value (times in seconds)			
Run #	2	3	5	10
1	0.042498	0.0413399	0.0356589	0.0354437
2	0.042356	0.0413642	0.0356524	0.0354782
3	0.0423307	0.0413532	0.035676	0.0354813
4	0.0423445	0.0413501	0.0356636	0.0354266
5	0.0423483	0.0413452	0.035676	0.0354319
6	0.0423358	0.0413616	0.0356622	0.0354253
7	0.0423322	0.0390179	0.0356712	0.0354403
8	0.0423412	0.0359955	0.03568	0.0354722
9	0.0423895	0.0360236	0.0356692	0.0354454
10	0.0423503	0.0360056	0.0356535	0.0354161
Average	0.04236265	0.03951568	0.0356663	0.0354461

Column Sort Using Merge Sort, with 288,000 elements				
	S value (times in seconds)			
Run #	2	3	5	10
1	0.0776899	0.0756462	0.075145	0.0741986
2	0.0772854	0.0756078	0.0751026	0.0742452
3	0.0772866	0.0756072	0.0750872	0.0742548
4	0.0772789	0.0755819	0.0750744	0.0743605
5	0.0772612	0.0756243	0.0750638	0.0841077
6	0.077325	0.0755923	0.0750479	0.0865326
7	0.0773158	0.075654	0.0750376	0.0831869
8	0.0772785	0.0756579	0.0750669	0.0866475
9	0.0773163	0.0756431	0.0750538	0.0866817
10	0.07726	0.0756265	0.075079	0.0867361
Average	0.07732976	0.0756241	0.0750758	0.0810952

Column Sort Using Merge Sort, with 1,000,000 elements			
	S value (times in seconds)		
Run #	2	5	10
1	0.294732	0.283088	0.286591
2	0.287262	0.282923	0.283117
3	0.285631	0.283161	0.28026
4	0.285672	0.282913	0.280485
5	0.286011	0.282993	0.280383
6	0.285619	0.282976	0.280466
7	0.285515	0.282983	0.280356
8	0.285659	0.282977	0.280389
9	0.285567	0.287569	0.280552
10	0.285629	0.289752	0.280443
Average	0.2867297	0.2841335	0.2813042





Conclusions

The fastest run with 1,000,000 was 0.28106 seconds using MergeSort as the auxiliary sort and an S value of 10.

The best S value for all data sets tested was 10, this made the lengths of the columns shorter so the auxiliary sort function was run on smaller inputs. This also points towards the easiest optimization for ColumnSort, which is to use a better sort algorithm since most of the time in the algorithm is spent in the auxiliary sort function. Using the std::sort implementation of quicksort, ColumnSort ran almost 10% faster than with MergeSort. Though MergeSort is the same complexity as quicksort (O{n*logn}), quicksort allows several optimizations to reduce the constants in its complexity expression, and quicksort makes fewer copies than MergeSort. This was the main effect of the extra storage needed by MergeSort: the additional copies slows the algorithm significantly compared to quicksort for this dataset. Additionally, it appears that the complexity of ColumnSort is almost exactly that of the auxiliary function used: for InsertionSort the complexity is O{n^2}, for MergeSort the complexity is O{$n \log n$}. These complexities are gleaned from the graphs shown above. When the input is doubled, the time used by ColumnSort with InsertionSort increases by almost 4 times, whereas ColumnSort with MergeSort increases by a little more than 2 times.

ColumnSort is not stable because when a column is “transposed” then “untransposed” two elements that are the identical can become flipped, and there is no efficient way to make it stable.

ColumnSort though not great for single-processor jobs could be used on out-of-core (e.g. clusters) systems. Each node in a cluster would sort a column, the master node would collect the sorted columns and perform the transposition and un-transpositions.