Supplementary Material for Lazy-Merge: A Novel Implementation for Indexed Parallel K-way In-place Merging

Ahmad Salah, Kenli Li*, Member, IEEE, and Keqin Li, Fellow, IEEE

1 An Application for Lazy-Merge Non-standard Merging Format

In this section, we discuss why Lazy-Merge, a non-standard merging format, is advantageous, in terms of a better run-time, over the standard format. We support this advantage using experimental results obtained from applying our standard to a real-time application, and later, we present its applicability to a general case.

We considered merging e-mail lists to show that Lazy-Merge has an advantage over the standard merging format. In the merging of e-mail lists, there is a set of e-mail lists, and we assume that each e-mail appears in only one e-mail list. Each e-mail list contains a set of fields for each record, i.e., first name, last name, and e-mail address. Merging e-mail lists is a common task in which copies of e-mail lists are merged into a single list with respect to a certain field; we assumed the e-mail address field is the one to consider for the merging process. The original e-mail lists are kept untouched for further use, whereas copies of the lists are involved in the merging task. Thus, the final merged list is ordered by e-mail address. k e-mail lists merging can be realized as k-way merging, where each e-mail list is a segment. In some cases, there is a need to analyze the merged list contents to know the share of each list in a certain portion of the list. For example, if sending a letter succeeded only for the first 10% of the final merged e-mail list, then the user might need to know how many e-mail addresses from each e-mail list received this letter. This can be viewed as the share of each segment for a certain portion of the final merged list.

- *Ahmad Salah, Kenli Li, and Keqin Li are with College of Information Science and Engineering, Hunan University, Changsha, Hunan, China, and the National Supercomputing Center in Changsha, Hunan, China. E-mail: {ahmad, lkl}@hnu.edu.cn*
- *Ahmad Salah is also with Computer Science Department, College of Computers and Informatics, Zagazig University, Zagazig, Egypt.*
- *Keqin Li is also with Department of Computer Science, State University of New York, New Paltz, New York 12561, USA. E-mail: lik@newpaltz.edu*
- *Corresponding author: Kenli Li*

Lazy-Merge, a non-standard format, stores an extra piece of information compared with the standard format; Lazy-Merge stores the share of each segment to its partition. For example, if the first partition represents the first 10% of the total merged list, then each segment share of the first 10% from the final merged list is known in constant time. Using the standard merging format to calculate the segment shares, we need to search the original e-mail lists for each element in the first 10% elements of the final merged list. For Lazy-Merge, if the portion to be considered is the first 12% of the merged list and the partition size is 10%, then only 2% should be searched for their segment shares. These shares will be added to the first partition segment shares. For the standard merging format, the same situation results in searching for each element of the first 12% elements in the merged list.

Generally, Lazy-Merge has a better execution time when a copy of a set of segments are merged, and the share of each segment is queried for a certain portion of this merged list. Thus, for the standard merging format, determining the segment shares of certain successive m elements out of n merged elements requires a time complexity of $O\left(\left(\log n\right) m\right)$ because each of the m elements should be binary searched over the entire segment, where the total size of the entire segment is n. For the same task using Lazy-Merge, the time complexity is $O\left(\left(\log n\right)\left(m-r\right)\right)$, where $r \leq m \leq n$, and $r = p \cdot s$, where p is the number of partitions in m, and s is the partition size.

A motivational example is presented in Figure 1. In Figure 1, Lazy-Merge's format contains two partitions; each partition has a different color and has four elements. To answer the question of how many e-mail addresses from each list are in the first 4^{th} e-mail addresses of the final merged list in Figure 1, using the standard format, we need to perform four searching operations on the three lists. However, using Lazy-Merge's format, the answer is of a constant time because the first 4^{th} e-mail addresses are included in the first partition. In the first partition, each sub-segment presents its corresponding segment's/list's share. If we want to know the list shares

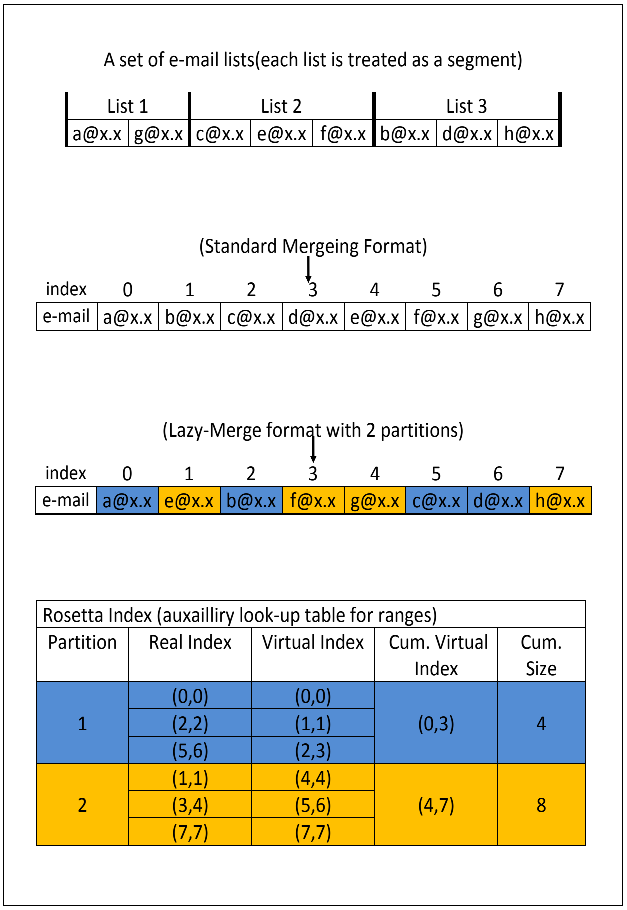

Fig. 1: A motivational example to utilize Lazy-Merge's format for better execution time

for the first 6^{th} e-mail addresses of the merged list, using Lazy-Merge's format, we need to perform only two binary search operations for the 5^{th} and 6^{th} e-mail addresses. However, using the standard format, we need to perform six binary search operations.

1.1 Expiremental Results

In these experiments, we evaluated the running time for counting each e-mail list share for a certain portion of the final merged list. We fixed the total length of the merged lists to have 2^{17} e-mail addresses and the number of lists segments to be 128. We retrieved the successive elements in the second and third quarters; this equals half of the elements of all the merged e-mail addresses. The standard merging format has a single retrieval time for the aforementioned portion. In this experiment, the average retrieval time for the standard merging format is 3.34 seconds.

Lazy-Merge uses partitions to merge e-mail lists segments. The number of partitions affects the retrieval time. Thus, we varied the number of partitions from 2 to 128 as shown in Table 1. As observed in Table 1 for two partitions, Lazy-Merge takes 3.37 seconds to retrieve the

TABLE 1: Retrieving elements of the second and third quarters of merged list of size 2^{17} for Lazy-Merge format

No. of partitions	time in seconds
2	3.37E+0
4	1.50E-5
8	2.41E-5
16	4.91E-5
32	1.01E-4
64	1.93E-4
128	3.79E-4

e-mail addresses of the second and third quarters. This longer running time, compared with the other values, is obtained because the second and third quarters, the searched elements, have zero complete partitions. The searched quarters are partially included in the first and second partitions. The first partition includes the first and second quarters, and the second partition includes the third and fourth quarters. Thus, each element of the retrieved quarters should be used to determine its original e-mail list/segment. On the other hand, when Lazy-Merge has four partitions, the second quarter of the merged list equals the second partition, and the third quarter of the merged list equals the third partition. Thus, counting the segment shares of the second and third quarters is performed in constant time because the segment shares for each partition are already known. The same explanation holds for the remaining values of Table 1.

In Table 1, the retrieved elements are in the range 25% to 75% of the final merged list. Thus, starting from 4 partitions, this range includes only complete partitions, and no partial partition is included, as discussed above. In the second experiment, we retrieved the elements in the range 21% to 71% of the final merged list. The standard format retrieval time in this experiment is 3.34 seconds, which is the same as the previous experiment because in both experiments, we retrieved 50% of the final merged list. Intuitively, for the standard merging format, the range of starting and ending indexes has no effect on the retrieval time. Table 2 shows a faster retrieval time using the Lazy-Merge format compared with the standard merging format. We selected the range 21% to 71% of the final merged list to guarantee that this retrieved range include some complete and some partial partitions as we vary the number of partitions. For those partially-included partitions, their elements should be searched one at a time. In Table 2, the retrieval time decreases as the number of partitions increases. This behavior is linked to the increasing partition number and decreasing partition size. As a result, the number of partitions completely included in the query range increases. Thus, the retrieval time decreases in the Lazy-Merge format. These results outline the superiority of the Lazy-Merge format compared with the standard format for this kind of application.

TABLE 3: Merging Algorithms' Execution Times

	1×10^7		2×10^7		3×10^7	
	Split	Shuffle	Split	Shuffle	Split	Shuffle
Random	1.95	1.55	3.90	3.08	5.85	4.90
Fully interlaced	4.38	1.46	9.11	2.95	13.76	4.58
10%-interlaced	1.58	4.43	3.18	9.11	4.81	SO
20%-interlaced	1.86	SO	3.79	SO	5.77	SO
30%-interlaced	2.40	SO	4.91	SO	7.41	SO
40%-interlaced	2.66	SO	5.46	SO	8.39	SO
50%-interlaced	2.94	SO	6.04	SO	9.21	SO

TABLE 2: Retrieving elements in the range of 21% to 71% of the merged list of size 2^{17} for Lazy-Merge format

No. of partitions	time in seconds
2	3.37
4	1.63
8	0.83
16	0.42
32	0.22
64	0.12
128	0.07

2 Comparison of Sequential In-place Merging Algorithms

In this section, we perform a comparison between *SplitMerge* and *ShuffleMerge* algorithms. This comparison shows the behavior of the two algorithms under different list sizes and contents.

For smaller list sizes, the test includes three different contents. The first case has segments of random elements, the second has segments of fully interlaced elements, and the last one has partially interlaced elements. Two segments are interlaced if the final merged list contains a range in which an element from one list is followed by an element from the other list repeatedly. For example, segments $t = [1, 3, 5]$ and $q = [2, 4, 6]$ are considered fully interlaced segments, $t = [1, 3, 5, 7]$ and $q = [4, 6, 8, 10]$ are partially interlaced segments by 50% of the elements, and $t = [1, 2, 3, 5]$ and $q = [11, 13, 14, 12, 15]$ are non-interlaced segments. The input lists include only 2 segments.

The following tests consist of randomly generated 2-way merging problems of different sizes. The first list size is 1×10^7, and each subsequent list increases by 1×10^7. This test includes three varied sized input lists. Each of these lists includes two equal-sized and ordered segments, which are the input to the 2-way in-place merging algorithm.

There are three different tests:

1) Random test: Each segment contains randomly generated ordered elements.
2) Fully interlaced test: The first segment contains the even numbers from 0 to $size - 2$, and the second segment contains the odd numbers from 1 to $size - 1$, where $size$ is the list size.
3) Partially interlaced test: For each list, we considered five cases: the second segment interlaced with the first segment by 10%, 20%, 30%, 40% and 50% of the elements. Thus, we have 5 different problems for each list.

Table 3 shows the execution times in seconds for the three tests. The first row presents the input list size. In general, *SplitMerge* outperforms the *ShuffleMerge* algorithm. Each reported execution time is the average of executing the algorithms three times.

The *ShuffleMerge* can not merge the partially interlaced lists due to the Stack Overflow error, we denoted this error by SO in Table 3. The reason for the Stack Overflow error is massive recursive calls. *ShuffleMerge* uses recursive calls to merge the elements of any disordered sub-segment; discorded sub-segment is defined as a set of contiguous elements where the leftmost element is larger than the rightmost element.

For fully interlaced elements, the i^{th} elements of the two input segments are less than $(i+1)^{th}$ elements of the same segment. Thus, the fully interlaced lists have no recursive calls, because each two i^{th} elements of the two segments are merged, and they are locaated pervious to the merged $(i+1)^{th}$ elements. In contrast, merging the partially interlaced lists result in many disordered sub-segment which is handled by recursive calls. For *SplitMerge*, the contents of the lists does not affect the depth of the recursive stack. For example, the deepest recursive function stack for a list with size 1×10^7 and 10% interlaced elements is 50,008 for *ShuffleMerge*, and 21 for *SplitMerge*.

To test the two algorithms under larger list sizes, we utilized a test includes 10 lists. The smallest list size is 1×10^7, and the largest list size is 1×10^8; the increasing step is 1×10^7. We utilized two kinds of tests, random and fully interlaced tests. Table 4 shows that *ShuffleMerge* outperforms *SplitMerge* in the fully interlaced test. For the random test, *SplitMerge* starts to slightly outperforms *ShuffleMerge* for the last three lists. This behavior indicates that the *SplitMerge*'s running time increases in smaller rates than *ShuffleMerge* as the lists size increases. We did not include the partially interlaced test because *ShuffleMerge* has the Stack Overflow problem, which is indicated in the previous test.

3 The Thorough Comparison of Lazy-Merge and GL-Merge

The basic idea of the Lazy-Merge and the GL-Merge is to divide the two input segments into independent smaller

TABLE 4: Merging Algorithms' Execution Times in Seconds

	Shuffle		Split	
Size	Rand.	Interlaced	Rand.	Interlaced
1×10^7	1.55	1.46	1.95	4.38
2×10^7	3.08	2.95	3.90	9.11
3×10^7	4.90	4.58	5.85	13.76
4×10^7	6.61	6.37	7.99	18.81
5×10^7	7.70	7.37	9.99	23.58
6×10^7	9.29	9.12	11.72	28.49
7×10^7	12.04	11.05	13.65	33.99
8×10^7	16.78	16.12	15.62	39.02
9×10^7	19.43	18.55	17.60	43.74
1×10^8	20.54	20.10	19.56	48.82

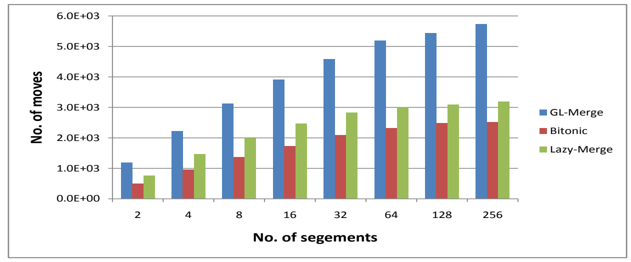

Fig. 2: Number of moves for different number of segments.

sub-segments merging tasks. The GL-Merge performs a series of steps to make the correct sub-segments contiguous, which is called sub-segments re-arrangement, and then starts the independent local merging tasks in parallel. But these re-arrangement steps have their own costs in terms of increasing the number of moves, and consequentially the execution times, as illustrated in Tables 5 and 6.

In Table 5, we can notice that when the number of threads is one, the number of moves of GL-Merge's re-arrangement steps is almost the half of the number of moves required to merge the independent re-arranged sub-segments. The number of moves of GL-Merge's re-arrangement steps increases as the number of threads increase; this is linked to the overhead of re-arranging extra sub-segments, as the number of sub-segments increases as the number of threads increases. For example, if we use 2 threads, then we have to rearrange the sub-segments to have two independent merging tasks, meanwhile if we use 128 threads, then we have to have 128 independent merging tasks. The numbers of moves of GL-Merge's re-arrangement steps and merging are almost equalized at certain number of cores, depending on the list size. Afterward, the GL-Merge's re-arrangement steps number of moves becomes more than the merging number of moves.

Table 6 lists the corresponding execution times of the number of moves listed in Table 5. We should notice that the GL-Merge's re-arrangement steps utilize a massive number of synchronization, $O(n/p)$, as mentioned in the paper. On the other hand, the independent merging tasks have no synchronization at all. That justifies the small execution times of the merging step in comparison to the GL-Merge's re-arrangement steps. The sequential GL-Merge, which uses 1 thread, includes re-arrangement steps and merging steps to have two independent merging tasks, but the entire steps are executed sequentially.

In contrast, the Lazy-Merge algorithm divides the input segments into sub-segments through partitioning and then starts merging without sub-segments re-arrangement. This replaces the overhead of sub-segments re-arrangement with the partitioning time. The partitioning step only finds the elements of each partition in each input segment; these elements are expressed as a sub-segments. Thus, the partitioning steps deliver each partition a set of sub-segment's boundaries. As explained, the partitioning step includes no extra moves. Thus, herein, we include only the execution times of the partitioning step and the parallel and independent merging step in Table 7. Apparently, when the number of the threads is one, no partitioning is required, because the entire segments are merged instead of the sub-segments. Thus, the partitioning times in Table 7 are denoted as Not Applicable, N/A, when the number of threads/partitions is one. For Lazy-Merge, using 1 thread means performing a sequential binary merging tree, and partitioning is not included.

4 The Number of Moves for Different Number of Segments and Threads

To practically validate the fact increasing the number of segments increases the total number of moves, we run the three algorithms using 1 thread for a list of size 256. Figure 2 shows the total numbers of moves to merge this list with the fixed size, and with different number of segments. In Figure 2, The last set of columns presents the extreme case; when each segment has only one element, 256 segments. Because the segment number does not represent the order of the elements; then, we still need to order the segments, the elements, by merging.

Similarly, we used a list of 256 elements divided into two segments with varied number of threads/partitions to track the change in the total number of moves as the number of threads/partitions varies. We excluded the bitonic merge from this experiment, because we showed that increasing the number of threads has no effect on the number of moves for the bitonic merge. For Lazy-Merge, the number of threads equals the number of partitions. We varied the number of threads from 2 to 256 threads, in a step of multiplying by two. Figure 3 shows a decreasing number of moves as the number of threads/partitions increases. Lazy-Merge's number of moves equals zero when the number of the partitions equals 256, because we have 256 partitions. Each partition has one element; thus, no data movement is required

TABLE 5: The detailed # of moves for the GL-Merge algorithm using a list of size 2^{17} with varied # of segments and threads

	Seg.	Threads							
		1	2	4	8	16	32	64	128
Re-arrangement steps	2	262296	262051	278451	285066	288426	289868	285368	280125
	4	524009	524484	556556	569908	577578	577312	568336	558467
	8	785939	785765	835095	855177	865257	866049	851816	833467
	16	1048088	1047646	1112557	1141044	1154516	1153992	1139185	1106121
	32	1309462	1309828	1391361	1426521	1442622	1439123	1418773	1382561
	64	1571721	1571294	1669211	1711896	1728589	1726911	1694350	1661385
	128	1833929	1833128	1946072	1997281	2017107	2013744	1978604	1922605
Merging	2	491787	490534	424159	343001	271338	196278	118194	45960
	4	980575	979844	842086	683800	535655	384827	221082	82167
	8	1470960	1468400	1260360	1022290	802319	571245	314069	116793
	16	1956620	1959050	1677360	1377440	1070070	744346	426291	134694
	32	2434700	2439760	2090710	1705460	1332370	913932	492141	164231
	64	2918860	2915430	2494110	2038130	1569840	1082390	565628	185316
	128	3383350	3382100	2895910	2351320	1823250	1241760	631472	174152

TABLE 6: The detailed execution times for the GL-Merge algorithm using a list of size 2^{17} with varied # of segments and threads

	Seg.	Threads							
		1	2	4	8	16	32	64	128
Re-arrangement steps	2	4.20082	4.91249	5.32203	3.01683	1.79967	3.27	3.31521	2.8945
	4	6.72141	6.26772	7.22957	4.47931	2.51571	6.08506	6.00539	5.51756
	8	7.5181	6.74095	7.83568	4.6688	3.00261	8.30078	8.84059	7.38648
	16	6.99998	7.35459	8.21013	4.87807	4.6679	10.4961	11.6374	11.8204
	32	6.72221	8.58589	8.66431	5.22026	4.79726	13.7319	14.0462	13.0874
	64	6.73216	9.00018	8.45555	5.10194	4.15231	15.1776	16.828	16.8955
	128	8.29334	8.25366	8.70909	5.38584	4.56368	17.4021	18.1878	21.4714
Merging	2	0.015965	0.0159869	0.0139761	0.0116019	0.00960207	0.00772905	0.00756502	0.00649405
	4	0.045315	0.0324259	0.0282631	0.0236449	0.0198591	0.0268109	0.0168288	0.0212479
	8	0.0794172	0.0495369	0.0435638	0.03706	0.0322161	0.0408468	0.0356798	0.0437551
	16	0.138774	0.06879	0.0610678	0.0539856	0.0490179	0.0597777	0.0798814	0.0923746
	32	0.222887	0.0898805	0.0815363	0.0751507	0.0731456	0.11358	0.120752	0.157171
	64	0.386073	0.116134	0.109431	0.107623	0.111247	0.161132	0.194456	0.295337
	128	0.715808	0.15293	0.148364	0.15509	0.179821	0.244591	0.306948	0.530356

TABLE 7: The detailed execution times for the Lazy-Merge algorithm using a list of size 2^{17} with varied # of segments and threads

	Seg.	Threads (Partitions)							
		1	2	4	8	16	32	64	128
Partitioning	2	N/A	5.71012E-4	6.07967E-4	8.37088E-4	0.00428391	0.00279093	0.00745511	0.011143
	4	N/A	5.7292E-4	6.47068E-4	7.16925E-4	0.00189996	0.00331998	0.00626206	0.00917602
	8	N/A	6.02961E-4	6.83069E-4	8.23021E-4	0.00295496	0.00312304	0.00639892	0.0101981
	16	N/A	6.46114E-4	8.39949E-4	0.00116801	0.00270295	0.00328207	0.00825	0.013571
	32	N/A	0.00116801	0.00181389	0.00335503	0.00492597	0.00487399	0.011029	0.0167949
	64	N/A	0.00232792	0.00401998	0.00415492	0.00826192	0.0116849	0.014621	0.022568
	128	N/A	0.00688696	0.010922	0.016428	0.0113029	0.016248	0.0271809	0.04706
Merging	2	0.0235901	0.020659	0.011682	0.00520301	0.00497103	0.00369191	0.00322008	0.00428605
	4	0.037282	0.0395501	0.0176101	0.00747705	0.00880909	0.00709701	0.00362301	0.0041399
	8	0.064183	0.0620639	0.029789	0.015058	0.01299	0.00684309	0.00880504	0.0082829
	16	0.0991631	0.0840099	0.0394461	0.019644	0.014147	0.01068	0.00998807	0.00893092
	32	0.23418	0.126	0.0628099	0.0350969	0.02474	0.020402	0.0139351	0.011059
	64	0.501214	0.188002	0.0998361	0.059267	0.0367539	0.02755	0.021539	0.018539
	128	1.35768	0.296346	0.174599	0.110669	0.069736	0.051661	0.037102	0.0354331

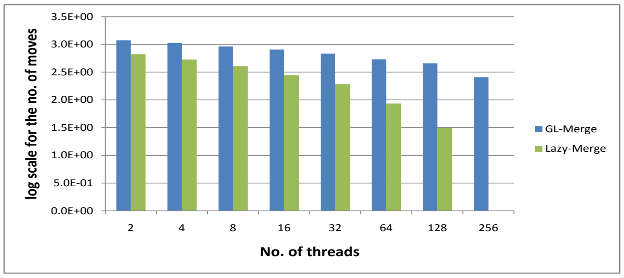

Fig. 3: Number of moves for different number of threads(partitions).

since the partitions array maintains the order. In other words, to access element i, one can access partition i directly.

5 Execution Times of Parallel In-place Merging Algorithms

To fully understand the behavior of the Lazy-Merge, Bitonic merge, and GL-Merge, we listed here in Tables 8 to 12 the execution times for the entire lists of dataset-1 and dataset-2.

TABLE 8: Lazy-Mege dataset-1 execution times

Size	Seg.	Threads(Partitions)							
		1	2	4	8	16	32	64	128
2^13	2	0.00173	0.00268	0.00463	0.00268	0.01934	0.02721	0.01773	0.03903
	4	0.02402	0.00743	0.00527	0.01207	0.00760	0.00729	0.00829	0.02147
	8	0.03781	0.01623	0.00888	0.00648	0.00792	0.00907	0.01174	0.04911
	16	0.07532	0.02540	0.01790	0.01306	0.01156	0.01092	0.01259	0.02284
	32	0.13592	0.04309	0.02779	0.02195	0.01860	0.01662	0.01890	0.04419
	64	0.22780	0.06138	0.04604	0.03135	0.02608	0.02392	0.02408	0.02705
	128	0.37026	0.08419	0.06190	0.04665	0.03828	0.03646	0.03619	0.04753
2^14	2	0.00885	0.00612	0.00290	0.00252	0.00643	0.00527	0.01104	0.02049
	4	0.01687	0.01265	0.01010	0.00475	0.00812	0.00667	0.01294	0.02035
	8	0.03238	0.01668	0.00949	0.00796	0.01041	0.00674	0.01299	0.02225
	16	0.06396	0.03128	0.01900	0.01464	0.01321	0.01115	0.01231	0.02120
	32	0.17287	0.05256	0.03405	0.02463	0.02428	0.02030	0.02043	0.03378
	64	0.33030	0.08548	0.05617	0.04135	0.03675	0.03300	0.03413	0.04028
	128	0.61137	0.12873	0.08593	0.06394	0.05052	0.04646	0.04469	0.06115
2^15	2	0.01071	0.00867	0.00687	0.00428	0.00992	0.00610	0.01204	0.01865
	4	0.02507	0.01496	0.01191	0.00573	0.00704	0.00760	0.01334	0.01711
	8	0.05133	0.02870	0.01510	0.01101	0.01010	0.01040	0.01389	0.01852
	16	0.07495	0.04269	0.02190	0.01602	0.01468	0.01411	0.01397	0.02406
	32	0.19041	0.06679	0.04249	0.03016	0.02506	0.02107	0.02349	0.03163
	64	0.41009	0.11247	0.07296	0.05082	0.04217	0.03861	0.04046	0.04267
	128	0.91262	0.18113	0.11740	0.08345	0.06408	0.05822	0.06085	0.07111
2^16	2	0.01605	0.01401	0.01052	0.00644	0.00759	0.00725	0.01119	0.02244
	4	0.02773	0.02410	0.01291	0.00650	0.01026	0.00821	0.01195	0.01778
	8	0.04147	0.03349	0.02081	0.01293	0.01053	0.01015	0.01627	0.01961
	16	0.08684	0.05360	0.03228	0.01588	0.01961	0.01424	0.02222	0.02232
	32	0.20228	0.08707	0.04773	0.03212	0.04918	0.02060	0.02550	0.03233
	64	0.46281	0.14859	0.09080	0.06097	0.04667	0.03938	0.04401	0.04247
	128	1.19868	0.24312	0.14909	0.10508	0.07502	0.06277	0.06608	0.07645
2^17	2	0.02367	0.02092	0.01186	0.00676	0.00900	0.00783	0.01153	0.02073
	4	0.04173	0.03900	0.01815	0.01917	0.01318	0.01064	0.01210	0.02041
	8	0.05660	0.05983	0.03083	0.01596	0.01615	0.01033	0.01585	0.02325
	16	0.08937	0.08150	0.03956	0.02275	0.02149	0.01682	0.01873	0.02732
	32	0.21175	0.12230	0.06430	0.04116	0.02876	0.02169	0.02334	0.03153
	64	0.48623	0.19461	0.10169	0.06585	0.05007	0.04105	0.04158	0.04440
	128	1.33534	0.30817	0.17867	0.11754	0.08395	0.06840	0.07442	0.08176

TABLE 9: Lazy-Mege dataset-2 execution times

Size	Seg.	Threads(Partitions)							
		1	2	4	8	16	32	64	128
	2	0.00173	0.00268	0.00463	0.00268	0.01934	0.02721	0.01773	0.03903
	4	0.231301	0.251	0.115	0.054	0.031	0.020	0.019	0.017499
	8	0.273406	0.379	0.171	0.083	0.052	0.038	0.034	0.031
2ˆ20	16	0.310612	0.514	0.225	0.103	0.070	0.053	0.041	0.033
	32	0.455035	0.652	0.303	0.136	0.080	0.070	0.054	0.070
	64	0.694905	0.841	0.377	0.273	0.114	0.081	0.070	0.061
	128	1.5936	1.052	0.492	0.235	0.134	0.122	0.118	0.114
	2	0.290331	0.263	0.121	0.061	0.043	0.032	0.036	0.035
	4	0.452292	0.501	0.232	0.114	0.069	0.056	0.046	0.045
	8	0.523433	0.758	0.339	0.148	0.092	0.075	0.061	0.051
2ˆ21	16	0.563906	1.003	0.452	0.203	0.093	0.083	0.141	0.050
	32	0.952583	1.268	0.564	0.254	0.124	0.108	0.090	0.067
	64	0.9747	1.571	0.708	0.308	0.202	0.131	0.105	0.079
	128	1.8809	1.916	0.881	0.397	0.225	0.185	0.157	0.139
	2	0.576097	0.709	0.243	0.115	0.083	0.059	0.051	0.045
	4	0.862919	0.990	0.445	0.200	0.102	0.081	0.075	0.056
	8	1.03319	1.488	0.661	0.284	0.152	0.116	0.093	0.063
2ˆ22	16	1.10895	1.986	1.091	0.378	0.229	0.145	0.117	0.080
	32	1.24266	2.485	1.100	0.474	0.211	0.181	0.142	0.098
	64	1.52409	3.067	1.345	0.575	0.250	0.236	0.172	0.130
	128	2.40463	3.640	1.614	0.707	0.344	0.305	0.231	0.186
	2	1.14544	1.013	0.466	0.222	0.138	0.104	0.081	0.067
	4	1.7367	1.990	0.881	0.385	0.189	0.178	0.124	0.090
	8	2.02693	2.960	1.311	0.562	0.263	0.215	0.161	0.122
2ˆ23	16	2.17695	3.949	1.755	0.764	0.326	0.281	0.196	0.131
	32	2.29904	4.950	2.166	0.920	0.418	0.348	0.251	0.158
	64	2.59058	6.001	2.641	1.130	0.496	0.409	0.316	0.205
	128	3.50215	7.125	3.121	1.350	0.586	0.513	0.388	0.274
	2	2.30956	2.029	0.908	0.411	0.230	0.178	0.142	0.110
	4	3.49231	3.924	1.766	0.756	0.444	0.297	0.227	0.147
	8	4.04084	5.935	2.628	1.118	0.521	0.447	0.311	0.211
2ˆ24	16	4.34247	7.902	3.465	1.470	0.648	0.542	0.384	0.244
	32	4.51227	9.848	4.333	1.824	0.779	0.665	0.467	0.282
	64	4.79873	11.937	5.205	2.205	0.923	0.784	0.564	0.331
	128	5.73426	13.952	6.148	2.592	1.105	0.954	0.685	0.432
	2	4.62103	3.990	1.813	0.811	0.404	0.342	0.263	0.196
	4	6.91763	7.924	3.498	1.528	0.711	0.589	0.452	0.295
	8	8.09311	11.804	5.242	2.219	0.967	0.832	0.592	0.367
2ˆ25	16	8.67834	15.667	6.925	2.916	1.256	1.036	0.765	0.451
	32	9.21115	19.739	8.663	3.624	1.519	1.341	0.901	0.579
	64	9.34976	23.658	10.395	4.417	1.859	1.498	1.098	0.624
	128	10.2511	27.979	12.173	5.116	2.154	1.795	1.287	0.752
	2	9.29309	8.064607	3.644211	1.629595	0.795218	0.679	0.573028	0.392922
	4	13.9213	15.91247	7.077147	3.071405	1.392382	1.090109	0.827013	0.540885
	8	16.1947	23.62802	10.38358	4.424005	1.995061	1.580916	1.13425	0.70331
2ˆ26	16	17.4497	31.43381	13.76856	5.872153	2.529965	2.121168	1.47453	0.85689
	32	18.079	39.29527	17.33116	7.291348	3.085752	2.505566	1.796528	1.085969
	64	18.9419	47.07522	20.65363	8.719883	3.603543	3.058697	2.103843	1.219487
	128	19.9409	55.78	24.24843	10.20485	4.193857	3.4925	2.447138	1.427551

TABLE 10: Bitonic merge dataset-1 execution times

Size	Seg.	Threads)							
		1	2	4	8	16	32	64	128
2^13	2	0.003026	0.003119	0.002509	0.004491	0.021386	0.019271	0.020524	0.028936
	4	0.005935	0.009721	0.004968	0.002958	0.062351	0.012063	0.02488	0.05392
	8	0.008415	0.010944	0.006012	0.004551	0.043651	0.039813	0.058277	0.112794
	16	0.010891	0.008224	0.009501	0.006637	0.065063	0.07249	0.10603	0.247666
	32	0.013183	0.014727	0.006621	0.005763	0.526707	0.09941	0.204597	0.396252
	64	0.01979	0.011591	0.011989	0.010067	0.930199	0.554165	0.266472	0.61227
	128	0.017345	0.017063	0.012889	0.011046	0.131489	0.455939	0.680843	1.27835
2^14	2	0.006396	0.009873	0.005173	0.003106	0.023398	0.006665	0.009846	0.018717
	4	0.012391	0.013718	0.010052	0.004417	0.007866	0.014502	0.026504	0.047061
	8	0.018	0.016591	0.009896	0.008698	0.150749	0.02411	0.048824	0.081676
	16	0.023449	0.016016	0.014148	0.011237	0.250146	0.058017	0.113703	0.193689
	32	0.028095	0.019337	0.014291	0.010798	0.059588	0.121163	0.212152	0.440568
	64	0.032809	0.031469	0.01726	0.014156	1.02937	0.154788	0.60708	1.40351
	128	0.03705	0.030589	0.017888	0.017812	0.191754	0.378606	0.692479	1.25286
2^15	2	0.013688	0.012983	0.010211	0.004424	0.004188	0.010253	0.015402	0.020068
	4	0.026458	0.021324	0.011346	0.010065	0.077322	0.023287	0.03008	0.048515
	8	0.038405	0.025972	0.016524	0.011431	0.022648	0.044598	0.069564	0.114303
	16	0.0496	0.041392	0.020925	0.012818	0.012308	0.061465	0.119558	0.206535
	32	0.060067	0.043191	0.023621	0.018589	0.16041	0.129389	0.233279	0.391162
	64	0.069846	0.047758	0.030965	0.026271	0.03788	0.235406	0.421247	0.73166
	128	0.079019	0.050877	0.03553	0.025585	0.036811	0.455289	0.761092	1.35702
2^16	2	0.028999	0.021317	0.013747	0.010202	0.014952	0.013249	0.017034	0.023021
	4	0.056268	0.03855	0.019046	0.012678	0.044537	0.034048	0.035173	0.054782
	8	0.08184	0.05104	0.031262	0.021595	0.016846	0.065513	0.077749	0.113349
	16	0.106377	0.067875	0.040203	0.030332	0.0194	0.352458	0.133729	0.466724
	32	0.128257	0.074081	0.047833	0.040265	0.528754	0.15875	0.254369	0.382972
	64	0.150211	0.090155	0.055298	0.036464	0.448115	0.418235	0.467293	0.70762
	128	0.16946	0.09941	0.066378	0.048791	0.089417	0.500808	0.840808	1.29519
2^17	2	0.061428	0.041	0.023375	0.01505	0.016404	0.021973	0.022106	0.023701
	4	0.119388	0.077006	0.041097	0.030098	0.040749	0.048274	0.048182	0.061482
	8	0.174318	0.104946	0.058407	0.038348	0.029709	0.078133	0.088474	0.130991
	16	0.225643	0.130169	0.079374	0.052237	0.047527	0.114968	0.167395	0.243745
	32	0.274941	0.158488	0.095956	0.062726	0.115968	0.192337	0.281449	0.462084
	64	0.321073	0.184362	0.106973	0.074048	0.160839	0.343249	0.506209	0.879565
	128	0.360311	0.208943	0.126713	0.085043	0.169821	0.577214	0.909951	1.63609

TABLE 11: Bitonic merge dataset-2 execution times

Size	Seg.	Threads							
		1	2	4	8	16	32	64	128
	2	0.530379	0.289815	0.146971	0.0808611	0.0527129	0.050909	0.0588059	0.0603189
	4	1.02951	0.554924	0.286694	0.150069	0.119057	0.110455	0.117873	0.143954
	8	1.50511	0.825544	0.418468	0.230514	0.112545	0.18067	0.20654	0.27376
2^20	16	1.9502	1.05654	0.556505	0.278181	0.147663	0.2823	0.342982	0.4728
	32	2.37064	1.27426	0.660433	0.342218	0.216223	0.417626	0.580195	0.80421
	64	2.76814	1.49416	0.769618	0.395358	0.230574	0.60105	0.933845	1.46111
	128	3.14366	1.70042	0.871982	0.458781	0.255045	0.893153	1.70759	2.58598
	2	1.15381	0.603792	0.304433	0.170095	0.0910089	0.0942941	0.098412	0.107145
	4	2.18719	1.17025	0.603255	0.306635	0.173421	0.186729	0.199561	0.221292
	8	3.1899	1.71528	0.86673	0.448596	0.231169	0.314047	0.330841	0.386573
2^21	16	4.15617	2.22094	1.13973	0.577289	0.298726	0.454959	0.486711	0.616549
	32	5.0254	2.70289	1.38443	0.716032	0.374323	0.661594	0.777752	1.04483
	64	5.88323	3.16268	1.61913	0.8217	0.554818	0.939112	1.22862	1.64547
	128	6.68731	3.58625	1.82695	0.950184	0.779068	1.36943	2.03626	3.63264
	2	2.38025	1.26801	0.646844	0.342684	0.181121	0.182265	0.230462	0.193364
	4	4.59936	2.4872	1.26849	0.645177	0.344675	0.571556	0.371668	0.390661
	8	6.7948	3.62067	1.82728	0.94369	0.512787	0.558698	0.581171	0.625736
2^22	16	8.76227	4.6981	2.36938	1.21369	0.644627	0.779506	0.823294	0.927595
	32	10.6585	5.72882	2.91076	1.47908	0.796092	1.08838	1.17911	1.4109
	64	12.4717	6.65782	3.39648	1.74106	0.926141	1.49168	1.75508	2.25288
	128	14.2355	7.562	3.85073	1.98998	1.05188	2.15824	2.61576	3.48325
	2	4.98569	2.66911	1.36081	0.697759	0.374413	0.426991	0.380582	0.37384
	4	9.81977	5.20972	2.63331	1.35812	0.693739	0.740388	0.732336	0.754243
	8	14.268	7.60978	3.85321	1.94817	1.01829	1.08575	1.09483	1.15234
2^23	16	18.3728	9.889	5.02586	2.56305	1.35648	1.47371	1.514	1.61212
	32	22.6014	11.9952	6.14353	3.13807	1.64302	1.95669	2.02646	2.23454
	64	26.2131	14.1119	7.16279	3.73471	2.19366	2.52488	2.6904	3.25362
	128	29.8904	15.9749	8.13616	4.14404	2.14276	3.36863	3.85911	4.94073
	2	10.3457	5.57976	2.86842	1.4576	0.781857	0.871748	0.807713	0.764204
	4	20.3801	10.8867	5.58817	2.84928	1.4658	1.59912	1.54078	1.49633
	8	30.1627	16.1116	8.17715	4.18914	2.14912	2.46351	2.27403	2.26676
2^24	16	39.2578	20.8601	10.6201	5.40377	2.77843	3.13641	3.0211	3.04683
	32	47.4148	25.4232	12.8714	6.57466	3.37483	3.91321	3.86888	4.00653
	64	55.7521	29.4641	15.0988	7.70675	3.96902	4.78684	4.86655	5.34117
	128	63.0387	33.689	17.1417	8.76231	4.50987	6.09361	6.34223	7.34898
	2	21.8325	11.6461	5.96139	3.03649	1.58052	1.71782	1.65808	1.59025
	4	43.0069	22.7023	11.6318	5.95512	3.08726	3.40704	3.24569	3.08817
	8	62.5671	33.6928	17.1653	8.78648	4.62458	4.96235	4.68549	4.56001
2^25	16	81.5473	43.581	22.2394	11.3716	5.88324	6.35786	6.26329	6.06883
	32	98.8698	53.256	27.0484	13.7618	7.13093	7.75572	7.63245	7.67448
	64	116.686	61.9364	31.6271	16.1574	8.32849	9.35978	9.23292	9.52069
	128	132.668	70.7723	36.1321	18.4582	9.54111	11.0335	11.2624	12.1516
	2	45.5754	24.3838	12.4156	6.31149	3.23358	3.4642	3.29328	3.22585
	4	88.4876	47.7154	24.3177	12.3868	6.40725	6.80664	6.57861	6.35742
	8	129.901	69.9481	35.6011	18.1892	9.48298	10.1185	9.7781	9.37132
2^26	16	170.838	90.6113	46.5311	23.7862	12.3843	13.2334	12.7904	12.3207
	32	208.154	110.735	56.8244	28.9646	15.1	16.0723	15.6094	15.3171
	64	245.15	130.345	66.2734	33.7909	18.8942	20.0685	19.3006	18.772
	128	280.191	147.902	75.5019	38.5034	20.0534	22.1714	21.8548	22.2225

TABLE 12: GL-Merge dataset-1 execution times

Size	Seg.	Threads							
		1	2	4	8	16	32	64	128
2^13	2	0.0116479	0.023541	0.017709	0.017446	0.895488	0.20291	0.163682	0.224494
	4	0.0285029	0.0486791	0.0339499	0.0463872	0.447414	0.360091	0.40701	0.460085
	8	0.052285	0.059947	0.048625	0.0456669	0.0759649	0.520523	0.612043	1.22744
	16	0.0825241	0.0759611	0.0631709	0.0553598	0.383106	1.49558	0.722774	0.989299
	32	0.131654	0.100581	0.084775	0.079036	0.532404	0.848781	0.992609	1.60817
	64	0.178745	0.115139	0.105167	0.105633	0.883399	1.13928	1.47087	2.49524
	128	0.227794	0.137308	0.131136	0.136863	0.835084	1.57041	2.32384	3.16011
2^14	2	0.0678711	0.0625069	0.0437679	0.0315399	0.557248	0.353857	0.375603	0.336956
	4	0.0925739	0.123754	0.0881791	0.0629611	0.281276	0.631735	0.697454	0.694736
	8	0.120564	0.159482	0.115188	0.114036	0.922094	0.909339	1.09182	1.16544
	16	0.146531	0.187845	0.146362	0.123602	0.301242	1.26815	1.29451	1.62787
	32	0.231791	0.225364	0.171522	0.155619	0.430792	1.59894	1.78336	2.34696
	64	0.322872	0.279249	0.205701	0.197386	0.440347	1.97403	2.38364	3.55752
	128	0.429894	0.306991	0.264874	0.242735	1.48167	2.58049	3.27182	5.49015
2^15	2	0.215268	0.25394	0.150071	0.095324	0.096662	0.653382	0.68402	0.587151
	4	0.343804	0.313266	0.258704	0.165916	0.407001	1.21791	1.23755	1.29258
	8	0.402808	0.369857	0.350981	0.260443	0.385174	1.71791	1.91056	1.92488
	16	0.454102	0.522634	0.356999	0.261687	0.394964	2.42146	2.40717	2.85744
	32	0.478917	0.647129	0.427664	0.363969	4.20934	3.5704	3.61953	4.69114
	64	0.649176	0.716007	0.495788	0.399583	5.37294	5.62134	4.1525	7.44274
	128	0.842961	0.659578	0.547299	0.435283	2.78745	6.03055	5.39717	7.47952
2^16	2	0.853915	1.10566	1.0034	0.652439	0.809943	1.99576	1.43109	1.30309
	4	1.2069	1.04231	1.3196	0.814233	0.575781	2.5022	2.49704	2.36701
	8	1.58309	1.84311	1.50349	0.930292	1.20365	4.37739	3.7481	3.64966
	16	1.65045	1.87609	1.56287	1.02404	1.9493	4.76394	5.26946	5.40294
	32	1.8884	2.15	1.85351	1.20107	1.02514	6.07527	6.27328	7.1342
	64	1.77461	2.11004	1.84567	1.1952	2.11504	7.19184	7.82537	8.60078
	128	2.20972	2.24928	1.96266	1.37303	1.40513	8.75955	9.64546	11.3362
2^17	2	4.86082	4.43371	5.22183	3.10343	1.77739	3.46354	3.19592	3.0359
	4	6.21939	6.17272	7.25232	4.36912	2.79991	5.97192	5.87838	5.30618
	8	7.65141	8.02727	7.79437	4.76759	3.13657	8.33931	8.50056	8.13996
	16	7.26409	8.06726	8.32731	5.21259	4.98227	10.3355	10.6589	10.3065
	32	8.12765	8.39038	8.29709	5.01117	4.3283	12.8568	13.0687	13.6048
	64	8.23442	8.26955	8.5726	5.22282	4.3323	14.7554	15.59	16.4759
	128	8.05692	9.46181	8.73264	5.40208	5.52466	21.9058	21.3764	21.3636