Array sort: an adaptive sorting algorithm on multi-thread

Xin Huang1, Zhijing Liu1, Jinyang Li1
1School of Computer Science and Technology, XiDian University, TaiBei South Road No.2, Xian, People’s Republic of China
E-mail: huangxinxd@gmail.com

Abstract: Sorting is the most fundamental operation in database system. There are many classical sorting algorithms and among them the most commonly-used sorting algorithm in modern database system is merge sort. Merge sort is an efficient, general-purpose, comparison-based sorting algorithm. As merge sort is based on a divide and conquer model, it has found wide use in parallel computing algorithms. In this study, the authors present an adaptive sorting algorithm based on merge sort which is called array sort. Array sort not only takes advantages of merge sort, but also simplify the merge step by using a tag array. The proposed implementation consists of three phases: tag array creation phase, tag array split phase and list merge phase. Instead of just evaluating array sort in a single thread environment, the authors also run their experiments on multi-thread to test how array sort performs.

1 Introduction

With the development of modern processor architecture, multi-core processors come out. Since each core may contain multiple threads, how to make full use of these computing resources remains to be a challenge. Among all the sorting algorithms, merge sort has the potential to utilize parallel resources well due to the exploit of the divide-and-conquer method [1]. Therefore, we propose a merge sort based sorting algorithm to promote the performance of sorting algorithms. By prepossessing the initial list when creating the tag array and exploiting the tag array to merging the list, array sort performs more stable and efficient than merge sort. Array sort has a best-case time complexity of $O(n)$ because it uses a tag array, and if the tag array is empty, the list is already sorted and the algorithm stops after $n$ iterations. In this case, array sort's $O(n)$ beats merge sort's $O(n\log n)$. Array sort's average-case and worst-case time complexity are $O(n\log n)$, which are the same to merge sort, but in actual experimental testing, array sort outperforms merge sort. For space, array sort will initialise a tag array of about $\frac{n}{2}$ elements to merge the sublist, finally merge sort merges all the sublists together to get the sorted list.

The original version of merge sort first divides the unsorted initial list into half continuously until each sublist containing only one element, and then merge sort compares the value of two elements to merge the sublist; finally merge sort merges all the sublists together to get the sorted list. In this paper, we analyse how the tag array influence the efficiency of array sort and evaluate the performance of array sort algorithms on multi-thread. The contribution of this paper is as follows:

- Introducing a tag array to simplify merge sort algorithm when merging sublists.
- Prepossessing the initial list when creating the tag array to facilitate the following merging step.
- Experiments run on multi-thread have proved that the parallel model of array sort performs more efficiently.

The rest of the paper is organised as follows. Section 2 presents related work. Section 3 gives an overall description of array sort. Section 4 introduces how a tag array affects the performance of array sort by simplifying the list's structure. Section 5 shows the method that we used to split the tag array. Section 6 describes the process of merging sublists. Section 7 discusses the parallel scheme we used on multi-thread. Section 8 presents the experimental results. Section 9 concludes.

2 Related work

Sorting [2] is a common operation in computer science, there are many operations based on sorting in computer applications. With the development of computer science, plenty of classic sorting algorithms come into being. Quick sort is the most studied by people at first. Developed by Tony Hoare in 1959 and published in 1961, it is still a commonly used algorithm for sorting. However, quick sort cannot be efficient in various input lists, as it performs poorly when the initial list is orderly or reverse.

It is Furtak et al. [3] that first exploited Single Instruction Multiple Data (SIMD) instructions for sorting arrays in main memory based on CPU. Their algorithm improves the last few steps of quicksort and the performance is more efficient. Then Intel research lab proposes a sorting algorithm [4] based on bitonic merge network, which makes use of the register shuffle instructions during the merging phase. Soon after, AA-sort [5] shows a multi-core SIMD algorithm based on comb sort and merge sort, which utilises odd–even merge network during the merging phase. Then our lab puts forward an efficient sorting algorithm [6] which mixed uses both odd–even merge network and bitonic merge network during the merging phase and achieves good results.

With the upgrading of hardware device, graphics processing units (GPUs) appear. Thus, sorting algorithm based on GPUs comes into being, which takes the advantages of modern processor architecture, as it exploits SIMD instruction sets based on GPUs. Sorting algorithms based on GPUs usually rely on sorting networks, in which merge sort plays an important role. There are two basic algorithms of sorting networks: bitonic merge sort and odd–even merge sort. The bitonic merge sort proposed by Batcher [7] uses SIMD instructions to fulfil a predefined order sorting. Then, GPU ballotsSort [8], based on adaptive bitonic sort [9], deals with data as 2D arrays or textures for data parallelism, it takes advantage of overlapping pointers and fragment processor memory accesses to reduce memory latency. Soon after, GPUBiSort [10] is proposed, which makes use of bitonic trees to pre-process the data and reduces the total number of comparisons.

E-First on 29th March 2019
Accepted on 7th February 2019
Revised 20th January 2019

eISSN 2051-3305
doi: 10.1049/joe.2018.5154
www.ietdl.org

This is an open access article published by the IET under the Creative Commons Attribution License (http://creativecommons.org/licenses/by/3.0/)
Since there is no optimisation algorithm for the original merge sort, we consider improvements from these perspectives: how to reduce the number of comparisons and how to make full use of existing ordered sublists in the dataset. Since the array has the characteristic of easy to locate data, which is beneficial to split and merge data. Thus, we propose an adaptive sorting algorithm based on merge sort, which introduces tag arrays to simplify the merge phase. Our algorithm also provides multi-threaded version to promote the efficiency.

3 Array sort

Array sort is stable, adaptive, divide-and-conquer sorting algorithm. It is based on merge sort, but is more selective on what it merges. Merge sort splits the unsorted initial list into half continuously until each sublist containing only one element, and then swaps the elements if needed, finally merges all the sublists together to get the sorted list. Array sort introduces a tag array to split the initial list into sublists of two or three elements. According to the tag array, array sort determines which part of the list to merge. The complete array sort consists of the following steps:

Step 1: putting the initial list into index array.
Step 2: traversing and pre-processing the index array to create tag array.
Step 3: splitting the list according to the tag array.
Step 4: merging the sublists to get the result list.

We will detail each part of array sort in later chapters.

4 Tag array creation

The most important part in array sort is tag array, which helps us simplify the merge phase of sorting. The creation of tag array accompanies with several optimisations. First, array sort puts the initial list into the index array, which is used to pre-process the initial list and different from the tag array. When the initial list fills with the index array, the pointer points to the last elements. Thus, array sort traverses the index array from the last index to the first index. While traversing, array sort compares the value of two adjacent elements to find sublists in ascending order. If array \( x > array[x−1] > array[x−2] \), then we can know that the sublist is probably a part of an ascending sublist and go on traversing to next element to find the whole ascending sublist. If array \( x < array[x−1] < array[x−2] \), it is clear that the sublist is a length 3 descending list. Thus, we swap array [x] and array [x−2] to make the sublist in ascending order and continue traversing to next element. Only when array \( x < array[x−1] \) but array \( x−1 > array[x−2] \), we add x to the tag array. For example, the initial list is shown as in Fig. 1, the index array can be easily obtained. After pre-processing, we just swap the array [6] and array [8] and get two ascending sublist: array [1] to array [4] and array [6] to array [8].

After pre-processing the index array, array sort continues to create tag array. Since index array already has several ascending sublists after pre-processing, we just need to record the beginning array number of each ascending sublists. These numbers make up the tag array which will be used as a basis to split the list in merging phase. As we can see in Fig. 2, the arrows point to the start index of each ascending sublists and the tag array is used to record this information. Since array sort traverses the index array from the last index to the first index, the tag array records elements in reverse order to avoid the data movement. From the tag array, we can know that five ascending sublists will continue to be merged in the following merging phase.

5 Tag array splitting

Different from merge sort, array sort splits the tag array instead of the actual array. It is obvious that tag array is much smaller than the actual array. Thus, splitting the tag array is much easier compared with the actual array. The smaller the array, the less splitting and merging, thus, the array sort is more efficient.

The tag array will be split into half until the length of the array on one side is 2 to 3. The letters m and n are used to stand for the lower boundary and the upper boundary of tag array. Fig. 3 details how we split the tag array. As can be seen from the figure, the tag array has a length of 10 and the initial values of m and n are 0 and 9. After the first splitting, we get two arrays length of 5 and the values of m and n have also changed. According to the new values of m and n, we continue splitting the array into small pieces. Until the array is split into length of 2 or 3, we stop splitting and start to merge. Otherwise, we will proceed to the next split.

6 Merging sublists

The algorithm will merge sublists according to the tag array. For the tag array length of 3, we preferentially merge the first two arrays. Take Fig. 3 as an example, for array [0] to array [2], we merge array [0] and array [1] in priority and then merge array [3] with new array formed by array [0] and array [1]. From the point of view of the index array, array [20] and array [19] will be merged first and then array [15] to array [18] shall be merged with the new array. For the tag array length of 2, we just merge two arrays without any special operations. As the merge algorithm goes on, the tag array also changes. Fig. 4 describes the process of change. After each merging, we only need to remove the merged array element in the tag array. The algorithm will end until only the starting element 0 is left in the tag array. The specific merging algorithm will be described in subsequent paragraphs.

The method that we used to merge the sublists is quite common. Since all the sublists are ascending lists, we just compare the first element of each sublist and return the smaller one back to the result set. The detailed process is shown in Fig. 5. First of all, we put the first sublist into the temporary array. Then, we compare the first element of the temporary array and the second sublist and the smaller one will be put back to the result array. With the algorithm going on, the elements of the temporary array become fewer and fewer. Finally, the algorithm completes when the temporary array is empty. A new ascending sublist is generated which will be merged with other sublists. As all the sublists are merged, the entire sorted array we obtained is the result list.

Compared with merge sort, array sort reorganises the structure of the initial list through simple pre-processing and introduces tag
array to splitting and merging the list. These optimisations significantly reduce the sublists need to merge and make array sort have a better performance. For various different input lists, array sort shows good adaptability. It is a stable algorithm and its efficiency does not change dramatically due to the input.

7 Multi-threaded version

The original version of merge sort performs well on multi-thread due to its divide-and-conquer method. While merging sublists on multi-thread, each thread operates in a separate space and has no influence on other thread's operations. Thus, there are no read–write conflicts which will affect the efficiency of parallel computing. In parallel, we do not have to consider the lock and unlock scheme to resolve the possible read–write conflicts. As a result, merge sort on multi-thread can easily speed up the efficiency of algorithm.

Unlike merge sort, array sort pre-processes the list before merging the list. Since pre-processing is a global operation of the entire list, multi-threaded computing may require data exchange between threads which will significantly reduce the efficiency of parallel computing. Moreover, different threads operate on the same list will cause read–write conflicts. Handling these read–write conflicts will further reduce the efficiency of parallel computing. All in all, pre-processing is not suitable for parallel computing. In other words, parallel computing does not improve its efficiency.

Thus, there remains to be two schemes to fulfil array sort on multi-thread. One is multi-threading the merge phase instead of the whole algorithm. The other is splitting the initial list and delivering the sublists to different threads for parallel computing, merging the result list of each thread to obtain the final sorted list. Fig. 6 describes the process of merging sublists on multi-thread. As we can see from the figure, four threads merge different parts of tag array and there is no data exchange between each thread. Whenever a thread finishes merging, it will continue merging the subsequent part of the tag array until the entire tag array has been merged. After the first round of merging, a new tag array is generated. According to the new tag array, four threads continue the second round of parallel merging. Only when the element 0 is left in the tag array, the algorithm is finished. Since this type of parallelism is similar to the merge sort and the whole algorithm is not processed in parallel, we propose another more suitable parallel scheme.

Learning from the method of divide-and-conquer, array sort splits the initial list evenly and delivers the sublists to different threads. It should be noted that we are only logically splitting the array and specifying the parallel threads to handle specific parts of the array. Thus, split operation does not consume any time and the number of parts while splitting should be equal to the number of parallel threads. As we illustrate in Fig. 7, the initial list is divided into four parts and each part is sorted by different threads in parallel. Thus, we will obtain four sorted sublists after array sort. The final sorted list is the result of sublists’ merging operation. The benefit of this scheme is that it takes advantage of divide-and-conquer methods and solves the problems that may arise in parallel computing. Thus, the scheme allocates separate spaces to each thread based on sublists. Each thread operates on its own space and has no data exchange with each other. Potential read–write conflicts will not occur either. Compared with the previous scheme, it exploits the feature that merge sort is a divide-and-conquer algorithm and solves the problem that pre-processing is inefficient in parallel computing. Since pre-processing is an important part of our algorithm, we prefer to choose the latter one as our parallel scheme.

8 Experimental results

As we have introduced array sort in detail above, we will compare the experimental results in different environments to further illustrate the advantages of array sort. First of all, we compare the execution efficiency of merge sort and array sort in single-threaded environment. Among them, the application of a variety of different input data makes the comparison more objectively. Then, we compare merge sort with array sort on multi-thread to verify the performance of array sort in multi-threaded environment. Finally, we compare the performance of array sort in single-threaded and multi-threaded environments to make sure the adaptability of array sort for multithreading.

All our experiments are evaluated on a dual-core four-threaded Intel core i5 processor running at 2.7 GHz. Each core has a 32 kB L1 data cache and a 256 kB L2 data cache. The two cores use a shared 3 M L3 cache with a cache line size of 64 bytes. The operating system is Mac OS High Sierra and the size of main memory is 8 GB. All programs are written in C++11 and compiled using g++ 4.9.4 with optimisation level O3.

Our experimental data are pseudo-random numbers generated by Mersenne Twister algorithm increased from 1000 elements to 1,000,000 elements and there are three types of input data: best, average and worst. Among them, best corresponds to the ordered input data, average corresponds to the disordered input data and worst corresponds to the reverse input data. It should be noted that the algorithm for generating pseudo-random numbers does not affect the experimental results. We run each experiment ten times to take the average execution time and each time we use a set of newly generated data to run experiment. Fig. 8 reveals the execution time of merge sort and array sort in single-threaded environment. As the amount of input data increases, the execution time becomes longer and longer and the difference in execution time between algorithms is also growing. Merge sort performs efficiently when the input data is ordered or reserve ordered.
However, the execution time drastically increases when merge sort deals with the disordered input data. This is due to the ordered input data requires less data comparisons and data movements during the merge sort. By contrast, when the input data is disordered or reserve ordered, the execution time of array sort remains stable. This is because pre-processing takes a period of time to traverse the array and it is an inevitable operation in array sort. When the input data is ordered, array sort will get the result after pre-processing without merging, therefore, array sort performs extremely well in this case. As can be seen from the result, in normal cases, array sort is much more efficient than merge sort. Optimisation based on merge sort does increase the efficiency of the algorithm.

In the multi-threaded version of the experiment, array sort remains an efficient performance. Fig. 9 shows that array sort performs better than merge sort on multi-thread. As the method of divide-and-conquer can be easily run in parallel and does not affect the efficiency of parallel computing, thus, the execution efficiency of multi-threaded algorithms is mainly determined by the performance of algorithms on single-thread. With the increase of parallel threads, the differences in efficiency between algorithms become smaller. This is because parallel threads divide the initial data into smaller parts for processing. As we can see from the single-threaded execution results, the smaller the data processed, the smaller the gap between two algorithms. From another point of view, we can find that multithreading does improve the efficiency of array sort but not linearly. Although the method of divide-and-conquer allows each thread to operate independently in parallel, we need to merge the results of each thread before submitting the final result. The more parallel threads, the more time we need to merge the results. This explains why there is less efficiency promotion as parallel threads increase.

9 Conclusion

In this paper, we have proposed an adaptive sorting algorithm which pre-processes the initial data and exploits the generated tag array to simplify the merging step. Our implementation not only involves single-threaded algorithms but also provides multi-threaded versions of all algorithms. By analysing the impact of the number of elements on the execution time of array sort and comparing the experimental results, we can draw the following conclusions: (i) pre-processing and tag array make array sort adapt to various input data and perform more efficient than merge sort. With the increase of data, the efficiency improvement is more obvious; (ii) multi-threaded algorithm does improve the efficiency of the algorithm, but it is not a linear improvement; (iii) the more parallel threads, the less efficient promotion. In summary, array sort is an efficient adaptive sorting algorithm compared with merge sort. However, there remain several techniques that can improve the efficiency of array sort such as parallel mode optimisation; we will study it in our future works. Then we will explore how an adaptive sorting algorithm affects on joining and partitioning operations.
References

[1] Cormen, T.H., Leiserson, C.E., Rivest, R.L.: ‘Introduction to algorithms’ (MIT Press, Cambridge, 2008)

[2] Martin, W.A.: ‘Sorting’, ACM Comput. Surv., 1971, 3, (4), pp. 147–174

[3] Furtak, T., Amaral, J.N., Niewiadomski, R.: ‘Using SIMD registers and instructions to enable instruction-level parallelism in sorting algorithms’. Proc. of the ACM Symp. on Parallelism in Algorithms and Architectures, New York, 2007, pp. 348–357, doi: 10.1145/1248377.1248436

[4] Satish, N., Kim, C., Chhugani, J., et al.: ‘Fast sort on CPUs and GPUs: A case for bandwidth oblivious SIMD sort’. SIGMOD, Indianapolis, USA, 2010

[5] Inoue, H., Mori, T., Komatsu, H., et al.: ‘A high-performance sorting algorithm for multicore single-instruction multiple-data processors’. Softw., Pract. Exp., 2012, 42, pp. 753–777

[6] Huang, X., Liu, Z., Yuan, T.: ‘A high-efficiency sorting algorithm on multicore’. Electrical, Control Engineering and Computer Science: Proc. of the 2015 Int. Conf. on Electrical, Control Engineering and Computer Science (ECECS 2015), Hong Kong, May 2015

[7] Batcher, K.E.: ‘Sorting networks and their applications’. Proc. of the AFIPS Spring Joint Computer Conf. 32 (AFIPS), 1968, pp. 307–314

[8] Govindaraju, N., Gray, J., Kumar, R., et al.: ‘GPUTerasort: high performance graphics co-processor sorting for large database management’. Proc. of the ACM SIGMOD Conf., Chicago, USA, 2006, pp. 325–336

[9] Bilardi, G., Nicolau, A.: ‘Adaptive bitonic sorting: an optimal parallel algorithm for shared-memory machines’. SIAM J. Comput., 1989, 18, (2), pp. 216–228

[10] Greß, A., Zachmann, G.: ‘GPU-ABiSort: optimal parallel sorting on stream architectures’. Proc. of the 20th IEEE Int. Parallel and Distributed Processing Symp., Rhodes, Greece, April 2006, p. 45