A-Sort: A Sorting Algorithm

Anonymous author

August 2023

Abstract

Here is where I would say what is in this document.

1 INTRODUCTION

Sorting plays a critical role in many applications, such as searching for information, event-driven simulation, or commercial computing. In many applications, such as the Student Information Management System[citation needed], there is a need to arrange the items inside the application in some order. It makes it easy to find items and modify information about items, as disorganized items need to be checked on an item-by-item basis. For example, students are arranged in a list according to student ID from smallest to largest. When trying to find a student with student ID x, you can check each item from start to finish to see if it is the target student. If the student ID of the current one is greater than x, you know that the target student does not exist in the list. Therefore, finding an item in an ordered list can be expected to reduce the number of checks by half.

Given a set of n items, each item can be characterized by a number key, and sorting requires that all items are arranged in a list from front to back with subscripts 1 to n, where for any two items the item with the smaller subscript has the larger key value. In the first example above, we can consider the student ID as the key and then sort the students by ID from smallest to largest. The meaning of key varies under different problems. Note that there are also similar problems to sorting. Some people study XXX, but XXX[citation needed]. Others have considered XXX, but XXX[citation needed].

Motivated by the above limitations, we propose the first sorting algorithm called First Sort (A-Sort), which sorts unordered items based on comparisons and swaps. Specifically, we designed an algorithm that incorporates two operations: 1) using comparisons

to find the smaller value, and 2) using swaps to adjust item positions. The first one performs a comparison to find the smallest value in the unsorted items one by one and then moves the item from the unsorted items list to the sorted items list. The key values of the elements moved to the sorted items list are ordered, and the number of unsorted items continues to decrease until finally all items are ordered. Second, we achieve sorting without extra space by swapping two items in the original list so that there are sorted items at the front of the list and unsorted items at the back of the list. In addition, we reduce the time required for comparison by comparing them in parallel at the same time. In terms of efficiency, the time complexity of A-Sort is $O(n^2)$, reducing the comparison time by 1/3. Thanks to in-place sorting, the space complexity of A-Sort is $O(n)$. The experiments also show that on the XXX dataset, A-Sort can sort XXX items in about XXX time, where parallel comparison reduces the time by $XX\%$.

We summarize our contribution as follows.

- To the best of our knowledge, we are the first to formulate the sorting problem and propose the first sorting algorithm, A-Sort, which can have a time complexity of $O(n^2)$.
- We proposed speedup techniques such as parallel optimization techniques. They take full advantage of the hardware features.
- We demonstrate the superiority of A-Sort by real data. It can sort a set of disordered elements in a certain order with $O(n^2)$ time complexity and $O(n)$ space complexity.

The remainder of the paper is organized as follows. Section 2 defines the problem. Section 3 presents our sorting algorithm. Section 4 gives the implementation details of parallel optimization. Section 5 shows the experimental results. Section 6 reviews related work and Section 7 concludes our paper.

2 PRELIMINARIES

2.1 Problem Definitions

Definition 1 (Item List)**.** *A collection of items $L = \{a_1, a_2, ..., a_n\}$ is represented by a list. Each item $a_i \in L$ is associated with a unique weight key$(a_i) \in \mathbb{R}^+$.*

Example 1. *Figure 1 shows an example list with 6 items. Each column is an item, the subscript in the first row is its index indicating where this item is in the list, the*

second row is its key, and the third row is a value with no specific meaning. For example, $key(a_2) = 7$ *and* $key(a_5) = 2$.

item	a_1	a_2	a_3	a_4	a_5	a_6
key	3	7	4	6	2	5
value	8	6	3	5	7	9

Figure 1: An example item list

Definition 2 (Sorting)**.** *Given a list of items L, the sorting returns a list of items* L' *that satisfies* $\forall a_i, a_j \in L', i < j \Rightarrow key(a_i) < key(a_j)$.

Example 2. *For example 1, the answer of the sorting is the list* $L' = \{a_5, a_1, a_3, a_6, a_4, a_2\}$ *with the keys arranged as* $\{2, 3, 4, 5, 6, 7\}$ *and values arranged as* $\{7, 8, 3, 9, 5, 6\}$.

3 A-SORT

We'll start with an overview of A-Sort sorting and cover each step in detail in later sections.

3.1 Overview

Algorithm 1 gives an overview of the A-Sort sorting algorithm. In lines 4-9, we find the item with the smallest key from the unordered collection of items. In lines 3-12, we sort all the items by continuously removing the item with the smallest key from the set of unordered items and adding it to an ordered list in order. Intuitively, to order a number of numbers from smallest to largest, we can put the smallest value in the first position, then the second smallest value in the second position, and so on until all the numbers are in its place. Inspired by this idea, we designed the A-Sort algorithm.

3.2 Find the Smallest Item

To find the minimum item in a set, we need to first assume an item with an infinite key, and then compare each item in the set to the item with the largest current key. If the key value of this item is greater than the key value of the item with the largest current

Algorithm 1 A-Sort Overview

Input: *A collection of items* L

Output: *A list of items* L' *that satisfies* $\forall a_i, a_j \in L', i < j \Rightarrow key(a_i) < key(a_j)$

```
1:  P ← {}
2:  Q ← L
3:  while Q ≠ ∅ do
4:      t ← empty item with key = ∞
5:      for each a ∈ Q do
6:          if key(a) < key(t) then
7:              t ← a
8:          end if
9:      end for          ▷ Find the item with the smallest key in the unordered collection
10:     P ← {P, t}
11:     Q ← Q − t
12: end while
13: L' ← P
```

key, then this item will become the new item with the largest key. Here, we use a loop to enumerate all the elements in the collection and use if statements for comparison and judgment.

Example 3. *For sample 1, we can try to find the smallest element by following these steps. First as in line 4 of Algorithm 1, we start by letting the key value of t be infinity. Then as in line 5, we enumerate from the first to the last element. The first element compared to t is* a_1*, and there is no doubt that the key value of* a_1 *is smaller, so t is updated to* a_1*. The second element compared to t is* a_2*, and* $key(a_1) = 3 < key(a_2) = 7$*, so t is still* a_1*. Next,* a_3 *and* a_4 *have key values less than* a_1*'s key value 3 so t remains as* a_1*. t is updated to* a_5*, because* $key(a_5) = 2 < key(t) = key(a_1) = 3$*. Similarly,* a_6 *is unable to update t.*

3.3 Adjust Item Positions

After finding the item with the smallest key in the unordered set, we need to move it from the unordered set to the ordered list. In practice, we store n items in an array of size n and treat the front of the array as an ordered sequence and the back of the array as an unordered set. As shown in Algorithm 2, the ordered sequence is empty at the beginning, and all are unordered sets. After the first for loop, we find the item with the smallest key, so the ordered sequence should now be of size 1 and located at the position where the array subscript is 1. At this point, we can exchange the item with the smallest key with

the item whose subscript is 1 in the array. After this the items with subscript 1 form an ordered list and the items with subscripts 2 to n form an unordered set. Similarly, the item obtained in the second loop will be exchanged with the item with subscript 2.

Algorithm 2 Adjust Item Positions

Input: *Array A with the first k small elements in order, subscript j of the k + 1th small element.*

Output: *Array A with the first k + 1 small elements in order*

1: $tmp \leftarrow a_k$
2: $a_k \leftarrow a_j$
3: $a_j \leftarrow tmp$

Example 4. *After finding the minimum item in example 3, we need to move the minimum item with subscript 5 to the position with subscript 1. If an item is represented by a key-value pair* $(key, value)$*, the initial condition can be represented as* $\{(3,8),(7,6),(4,3),(6,5),(2,7),(5,9)\}$. *The situation after the first move can be expressed as* $\{(2,7),(7,6),(4,3),(6,5),(3,8),(5,9)\}$. *After this, we need to find out the smallest item from the set with subscripts 2 to 6 using the minimum item finding algorithm, which is the second smallest item* $(3,8)$. *Then, we need to exchange* $(3,8)$ *with* $(7,6)$ *which has subscript 2. After the swap, the situation becomes* $\{(2,7),(3,8),(4,3),(6,5),(7,6),(5,9)\}$.

3.4 Complexity

As shown in Algorithm 1, A-Sort contains two loops nested within each other. When the outer loop is executed for the first time, the inner loop will enumerate all n elements. When the outer loop executes for the second time, there are only $n-1$ elements left in the unordered set, so the inner loop executes $n-1$ times. Each time the outer loop executes once the unordered set is reduced by one element. Therefore, the loop executes n times when the set is empty. The total number of executions is $n+(n-1)+(n-2)+...+1 = (n+1)*n/2$. That is, the time complexity of A-Sort is $O(n^2)$.

In practice, we use arrays to store unordered sets and ordered sequences, and only use a constant number of variables. Therefore, the space complexity of A-Sort is $O(n)$.

4 PARALLEL OPTIMIZATION

As we know, modern computers have gone from single-core to multi-core, which means that computers have gone from being able to execute only one instruction at a time to being able to execute multiple instructions at a time. This provides us with hardware support to increase computational efficiency and reduce computational time. The inner loop in Algorithm 1 accomplishes the task of finding the minimum value among multiple elements, which involves multiple comparison instructions. On a single-core computer, we can only compare one element at a time with the current minimum value, which takes n units of time. Whereas on a dual-core computer, we can compare every 4 elements as a group as shown in Figure 2. In the first unit of time, the 4 elements are divided into 2 groups of 2 elements and compared separately to obtain the smaller value. In the second unit of time, the 2 smaller values obtained in the previous comparison are compared to obtain the smallest value of the 4 elements. Of these 4 elements, 1 is the current minimum value and 3 are the elements in the unordered set to be compared. In other words, comparing 3 elements takes only 2 units of time, which is 2/3 of the original, saving 1/3 of the operations.

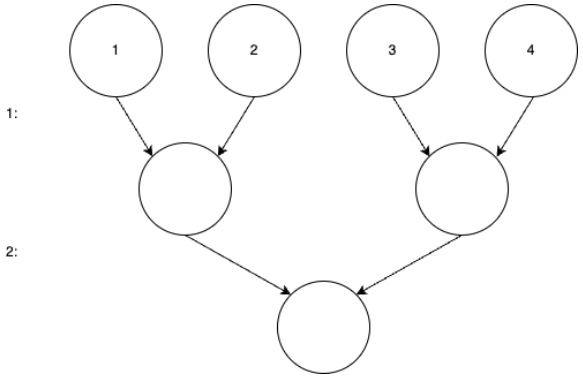

Figure 2: Parallelism on dual-core computers

Example 5. *Suppose now that there are four numbers to be compared, where* $t = 4, a2 = 5, a3 = 2, a4 = 3$. *The ordinary version entails a three-stage comparison, where first the comparison of* t *and* a_2 *results in* t *being smaller. Then the second stage* t *is compared with* a_3 *to obtain that* a_3 *is smaller. Finally,* a_3 *and* a_4 *are compared to conclude that* a_3 *is smaller. The first stage of the parallel version performs* t *and* a_2 *comparisons, and* a_3 *and* a_4 *comparisons, respectively, and concludes that* t *and* a_3 *are the smaller values. The second stage compares* t *and* a_3 *and concludes that* a_3 *is the smallest value.*

5 EXPERIMENTS

5.1 Experimental Setup

In our experiments, all algorithms are implemented in C++ and compiled with O3 optimization by the GNU C++ compiler. The programs were executed on a machine equipped with two Intel Xeon Gold 5220R 2.2GHz processors and 512GB of RAM with the CentOS 7 Linux distribution installed.

Datasets. Following existing work [citation needed], we used three publicly available collections of items in XXX, including XXX, XXX, and XXX. their details can be found in Table 1. The last column of Table 1 shows the size of the item set, defined as the number of elements in the set and denoted by n.

Comparing Algorithms. Since we are concerned with sorting efficiency, we compare the normal and parallel versions of A-Sort at different set sizes n.

5.2 Experiment Results

Figure 3 shows the sorting times on the three sets with different n. The results of varying n are shown in Figure 3, respectively.

Sorting time for varying n. It can be observed that as we change the set size by increasing n, the sorting time of the algorithm becomes larger on all data sets. The time taken by our A-Sort algorithm to sort increases squarely with the increase in n. Comparing the single-core and multi-core versions of the A-Sort algorithm, we can see that the multi-core version takes $XX\%$ of the time of the single-core version.

5.3 Summary

1. Our proposed A-Sort runs orders of magnitude faster than the best-known XXX solutions.

2. Our A-Sort scales to large collections in terms of time and space efficiency. On a collection of n items, the sort runs in XX or less. The additional space consumption is negligible compared to the space used to store the items.

3. The proposed parallel techniques are all useful for speeding up the processing, using $XX\%$ of the original time.

Figure 3: The sorting times on the three sets with different n

6 RELATED WORK

7 CONCLUSION

This paper presents the problem of sorting. The state-of-the-art algorithm is called XXX, which beats the previous algorithm. However, XXX is XXX. We propose the fastest algorithm to date, called A-Sort, which sorts by constantly finding the minimum value in an unordered set and forming an ordered list in sequence. Our A-Sort runs orders of magnitude faster than XXX and has similar space consumption. Experimental results demonstrate the superiority of the proposed A-Sort in terms of sorting efficiency. For future work, one can consider improving the efficiency of sorting algorithms, such as using the idea of partitioning. Another interesting topic is to explore more applications of sorting. Since the minimum spanning tree problem on graphs can consider the selection of edges according to edge weights from smallest to largest, sorting algorithms may come into play here.