Hindawi
Complexity
Volume 2020, Article ID 8883945, 7 pages
https://doi.org/10.1155/2020/8883945

Research Article

A New Method to Construct the KD Tree Based on Presorted Results

Yu Cao, Huizan Wang, Wenjing Zhao, Boheng Duan, and Xiaojiang Zhang

College of Meteorology and Oceanography, National University of Defense Technology, Changsha, Hunan Province, China

Correspondence should be addressed to Huizan Wang; wanghuizan@126.com

Received 25 September 2020; Revised 1 December 2020; Accepted 10 December 2020; Published 23 December 2020

Academic Editor: Shi Cheng

Searching is one of the most fundamental operations in many complex systems. However, the complexity of the search process would increase dramatically in high-dimensional space. K-dimensional (KD) tree, as a classical data structure, has been widely used in high-dimensional vital data search. However, at present, common methods proposed for KD tree construction are either unstable or time-consuming. This paper proposed a new algorithm to construct a balanced KD tree based on presorted results. Compared with previous similar method, the new algorithm could reduce the complexity of the construction process (excluding the presorting process) from O (KN$\log_2$N) level to O (N$\log_2$N) level, where K is the number of dimensions and N is the number of data. In addition, with the help of presorted results, the performance of the new method is no longer subject to the initial conditions, which expands the application scope of KD tree.

1. Introduction

How to search quickly is one of the most fundamental problems in many research fields [1–3], such as grid remapping, pattern recognition, and ray tracing. However, the complexity of the search problem would increase dramatically in high-dimensional space. Take the basic problem of finding the nearest point for a target point in a high-dimensional space as an example. The most intuitive way to do this is to compute the distance from all the other points, and then pick out the point with the shortest distance. However, the higher the dimension is, the more expensive it is to calculate the distance. Calculating the distances between the target point and all the other points would soon become unacceptable in high-dimensional cases. At this point, an efficient algorithm to avoid unnecessary calculation is particularly important.

KD tree [4] is a classical data structure that stores K-dimensional points for quick retrieval by the form of a binary tree. Different from a standard binary tree, each level of the KD tree can be divided by different dimensions. Due to its good performance in solving multidimensional searching problems, it has been widely used in multidimensional critical data search (e.g., regional search and nearest neighbor search [5, 6]).

There are two vital operations in building a KD tree: one is choosing the dimension to be divided, and the other is selecting the exact splitting point. In terms of choosing the splitting dimension, the dimension with the largest variance or the widest dispersion is generally recommended [7], because such choices can divide the search space more evenly. Nevertheless, for the sake of simplicity, it is acceptable to divide the dimensions in a circular fashion in many cases, especially when the points are evenly distributed. As for selecting the splitting point, due to fact that no effective rebalancing techniques are available for KD tree reconstruction at present [8], the median point is usually suggested to be chosen so as to build a balanced tree directly [9].

A variety of methods, aiming at picking the median point out quickly and accurately, have been developed at present. Quicksort [10] is one of the most popular sorting algorithms, which could find the median in O (N$\log_2$N) time at best. In addition, an improved method (i.e., quick select algorithm [11]) could even reduce the time to O (N) level, which is regarded as one of the fastest methods. Unfortunately, the performance of both methods might degrade to O (N^2) level at worst. Though some of other methods (e.g., merge sort [12]) may guarantee to obtain the median in O ($N\log_2 N$) time, their performance is still slightly disappointing.

As we all know, sorting algorithms have been relatively mature at present. Therefore, it should be an intuitive idea to build a KD tree based on presorted results. However, few of previous literature are concerned about this idea. To the best of the authors' knowledge, Brown [13] is the first one who proposed a method to build a KD tree based on presorted results in O ($KN\log_2 N$) at worst. Cao [14] further improved the performance of this method by replacing superkeys with keys during the presorting and construction. However, both methods have to maintain K index arrays all the time during the construction, which degrades the complexity of the construction process from O ($N\log_2 N$) to O ($KN\log_2 N$). In addition, the selection of the splitting dimension is restricted for the sake of arrays reusing.

This paper proposed a new method to construct the KD tree based on presorted results, which can not only build a KD tree in O ($N\log_2 N$) time under any conditions, but also arbitrarily choose the splitting dimension. Detailed description of the new method is given in Section 2. Experiments' validation and analysis are shown in Section 3. Finally, the conclusion is drawn in Section 4.

2. Algorithm

2.1. Basic Idea. Brown's method would prepare an ordered index array for each partition. Each partition can be implemented quickly according to the index array it depends on. Taking the splitting order adopted by Brown as an example (i.e., split by x, y and z in turn for 3D data), assuming that the index array is arranged from small to large, elements less than the median belong to the left subtree, while elements greater than the median belong the right subtree (this rule will be adopted through this paper). The first partition is based on the x index array. The middle element of the x index array is just the median point. Elements in the upper part of the index array belong to the left subtree, while elements in the lower part of the array belong to the right subtree. The overhead of selecting the median point is only O (1). In order to prepare for the second partition, it needs to form a new y index array after O (N) comparisons. In theory, with the new y index array, the second partition has been able to be carried out successfully. However, z index array is also updated by Brown in the first partition. In fact, this is accomplished for the preparation of the third partition. If z index array is not updated during the first partition, the program will not be able to determine which subtree the element in z index array belongs to in the second partition, which would lead to a failure of forming the new z index array in the third partition. However, such operations lead to the degradation of the complexity of KD tree construction from O ($N\log_2 N$) to O ($KN\log_2 N$).

Therefore, in order to build a KD tree in O ($N\log_2 N$) time, only the index array useful for next partition can be maintained during the construction. To achieve this goal, we need to ensure that each element clearly knows which subtree it belongs to in each partition. Hence, we designed three additional integer arrays to record the state of corresponding elements. They are BN array, SS array, and CUR array. The size of each array is N. BN[i] is used to record the starting position of the subtree to which element i belongs. SS[j] is used to record the number of the remaining elements in the subtree of the element whose starting position is j. CUR[k] is used to record the number of elements that have been arranged in the subtree of the element whose starting position is k. If CUR[BN[i]] is less than half of SS[BN[i]], it means that element i should belong to the left subtree of the current subtree. Similarly, if CUR[BN[i]] is larger than half of SS[BN[i]], it means that element i should belong to the right subtree of the current subtree. If CUR[BN[i]] equals half of SS[BN[i]], it means that element i is the median of the current subtree. Then, BN[i] should point to the middle of the domain where the subtree occupies (supposing it points to M), and SS[M] should be set to 0. When CUR[BN[i]] equals SS[BN[i]], it means that all the elements in the current subtree have been processed, and corresponding elements in SS and CUR arrays should be reset so as to prepare for the next split. More specifically, supposing the starting position of the left subtree and the right subtree of the current subtree is L and R respectively, then SS[L] and SS[R] should be set to half of SS[BN[i]], while CUR[L] and CUR[R] should be set to 0.

Let us take building a KD tree for seven 2D points as an example. The coordinate information of these seven points is shown on the left side of Figure 1 under "Tuples." Index arrays presorted by values in the x and y coordinates are listed from small to large and shown under "Presorted Results." The initialization of BN, SS, and CUR arrays is shown under "Initial." BN is initialized to 0, indicating that the data set of the current subtree starts from position 0. SS is initialized to 7, and it means that there are 7 elements in current subtree, and CUR is initialized to 0, indicating that none of the elements in the current subtree have been arranged. It should be noted that, for SS array and CUR array, only SS[0] and CUR[0] are meaningful, and assignments for other elements have no influence on the final result.

After the completion of these preparations, the construction of the KD tree formally begins. For the sake of simplicity, subtrees in each level are divided by x and y sequentially and cyclically. Then, the first partition is accomplished by processing the elements in x index array in turn from top to bottom. Elements 2, 4, and 3 are less than the median, so they belong to the left subtree and should be left in the upper part of the domain occupied by the current subtree (i.e., their corresponding value in the BN array should be 0). Element 0 is the median, so it should be located in the middle of the current subtree (i.e., BN[0] should be 3). Moreover, as this point will no longer belong to any subtree after being an intermediate node of the KD tree, SS[3] should be set to 0. Elements 5, 6, and 1 are greater than the median, so they belong to the right subtree and should be left in the lower part of the domain occupied by the current subtree (i.e., their corresponding value in the BN array should be 4). In the process of traversing the subtree starting from 0, the

	Tuples (x, y)	Presorted results		Initial			After first split			After second split		After third split		
		x	y	BN	SS	CUR	BN	SS	CUR	BN	SS	BN	SS	F
0	(6, 1)	2	0	0	7	0	3	3	0	3	1	3	0	2
1	(9, 4)	4	2	0	7	0	4	7	0	5	0	5	0	3
2	(1, 2)	3	6	0	7	0	0	7	0	0	1	0	0	4
3	(3, 5)	0	1	0	7	0	0	0	0	1	0	1	0	0
4	(2, 8)	5	3	0	7	0	0	3	0	2	1	2	0	6
5	(7, 9)	6	4	0	7	0	4	7	0	6	0	6	0	1
6	(8, 3)	1	5	0	7	0	4	7	0	4	1	4	0	5

Figure 1: An example to construct the KD tree based on the new method.

value of CUR[0] increases continuously, which is used to determine whether the current element belongs to the left subtree or the right subtree. After all the elements in the current subtree have been visited, CUR[0] is reset to 0. At the same time, the number of elements in the left and right subtrees will also be reset; that is, SS[0] and SS[4] will be set to 3. After the first partition, the values of these three arrays are shown in Figure 1 under "After First Split." It can be seen that the CUR array is an auxiliary array, which only affects the current partition. After the current partition is completed, it will return to 0 again. Therefore, its state will no longer be displayed in the subsequent partition process. The second partition will be accomplished according to the y index array. The first element in y index array is 0, and the beginning position of the current element (i.e., BN[0]) is 3. SS[3] is zero, which means that the current element does not need to be processed. The second element is 2, the starting position of element 2 (i.e., BN[2]) is 0, and the number of elements in the current subtree is 3. Therefore, element 2 belongs to the left subtree of the current subtree, and BN[2] should be 0. The third element is 6, starting at 4 and belonging to the left subtree of the current subtree, so BN[6] is 4. The fourth element is 1, and the starting position is 4, which should be the median of the current subtree. Therefore, BN[4] is set to 5, and SS[5] is set to 0. The remaining elements can be processed similarly. The results are shown in Figure 1 under "After Second Split." When all the elements are picked out as a node in the KD tree, all the elements in the SS array will be zero, as shown under "After Third Split" in Figure 1. Then, the final index array (i.e., F array in the right side of Figure 1) used to construct the KD tree can be easily obtained from the BN array (it can be done by placing i in the position of BN[i] of F array). It can be seen from the process descripted above that when building a KD tree, the new method does not manage to prepare a complete index array for each partition, but only maintains the set of data to be partitioned next time. This idea enables the new method no longer need to update K index arrays every time. Thus, it reduces the time cost.

It should be noted that we have just obtained the final index array for construction, not the final KD tree. Previous algorithms use recursive method to build KD tree, so the child node can easily link back to the parent node. However, the previous process no longer uses recursion, so it is impossible to transfer the information of the child node to the parent node, unless more space is used to store the information of the parent node. Fortunately, after obtaining the final index array F, we can visit the index array again according to the same rules and order recursively as before to build the KD tree. The difference is that the current recursive traversal process no longer needs to find the median point but only needs to get the middle point from the array range occupied by the subtree. It should be noted that if the selection of the split dimension does not follow some fixed rules, additional arrays are needed to record the selection results of the split dimension before building the KD tree recursively at last.

2.2. Detailed Description. The detailed description of building a KD tree based on the new method is shown in Figure 2. The first step is presorting and initialization, after that, choosing the splitting dimension and processing BN, SS, and CUR arrays repeatedly, until each element in the SS array is zero. Subsequently, the final index array F is constructed according to the BN array (in fact, the construction of F array can also be accomplished during the partition without the help of BN array). Finally, the KD tree is exactly constructed recursively according to the F array. The main process of dealing with BN, SS, and CUR arrays is shown in the large box. Elements in the D index array (i.e., the splitting dimension) will be visited in turn from the beginning to the end. Suppose that the current element is tmpi. If the current element has already been selected (i.e., tmpsize is 0), it no longer needs to be processed. Otherwise, it is whether it belongs to the left subtree or the right subtree or is the median will be judged according to CUR[tmpBN]. If it belongs to the left subtree or the right subtree, BN[tmpi] will point to the starting position of the corresponding left or right subtree. Otherwise, BN[tmpi] will point to the middle of the segment occupied by the current subtree, and its corresponding value in the SS array will be set to zero. When all the elements in the current subtree have been processed, the corresponding elements in SS and CUR arrays will be reset as described in the foregoing subsection.

The process of building a KD tree recursively according to the final index array F is shown in Figure 3. The first step is to obtain the splitting dimension. The splitting dimension

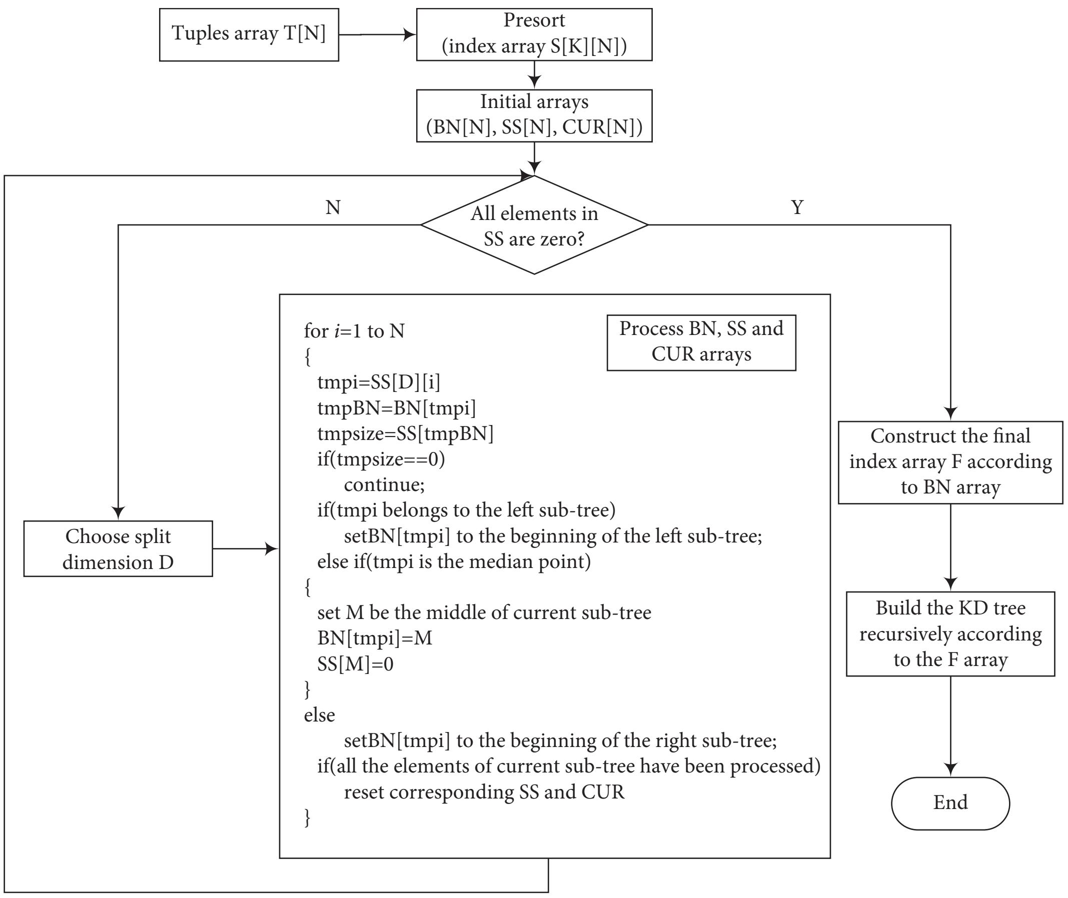
Tuples array T[N]
Presort (index array S[K][N])
Initial arrays (BN[N], SS[N], CUR[N])
All elements in SS are zero?
N
Y
Process BN, SS and CUR arrays
for i=1 to N
{
tmpi=SS[D][i]
tmpBN=BN[tmpi]
tmpsize=SS[tmpBN]
if(tmpsize==0)
continue;
if(tmpi belongs to the left sub-tree)
setBN[tmpi] to the beginning of the left sub-tree;
else if(tmpi is the median point)
{
set M be the middle of current sub-tree
BN[tmpi]=M
SS[M]=0
}
else
setBN[tmpi] to the beginning of the right sub-tree;
if(all the elements of current sub-tree have been processed)
reset corresponding SS and CUR
}
Choose split dimension D
Construct the final index array F according to BN array
Build the KD tree recursively according to the F array
End

FIGURE 2: Building a KD tree based on the new method.

can be obtained either by following the same rules or from the record of previous partition results. The next step is to select the median point. In fact, the middle element of the segment occupied by current subtree is just what we want. Subsequently, the starting position and the number of elements for the left and right subtrees will be calculated, respectively. Finally, the left and right subtrees will be constructed recursively until the number of elements in the current subtree is zero.

2.3. Complexity Analysis. From the foregoing discussion, it is known that the whole process of building a KD tree is comprised of two parts. The first part is to form the final index array for construction, while the second part is to exactly build the KD tree according to the final index array. In the first process, one of the presorted index arrays will be traversed during each partition. Although each element might be accessed in a different order, the final result is that all the elements will be processed once. Therefore, the time complexity of each partition is O (N). There are O ($\log_2 N$) times of partition in need, so the time complexity for the first process is O ($N\log_2 N$) in total. It is worth noting that although each partition will access some selected nodes, the overhead is limited, and the impact on the final execution time is negligible. In the second process, each selection of the KD node is accomplished in O (1) time. Therefore, it takes O (N) time to build the KD tree. Adding up the complexity of these two parts, the final time complexity of the new method is O ($N \log_2 N + N$). When N is large, the total time complexity can be considered as O ($N\log_2 N$). It is notable that the time complexity of the new method is independent of the initial order of original elements. It means that, even under the worst condition, the time complexity of the new method is still O ($N\log_2 N$) level, which is just what we expected.

3. Experiments

The new method is accomplished in C language. As the presorting process is not the focus of this paper, for the sake of simplicity, the function “qsort”, which is provided by the C standard library, is used for presorting. There are two data sets used for testing. One owns 2^{24} 6-dimensional real elements, which are randomly generated between 0 and 100 with 6 valid decimal places. The other owns 2^{17} 6-dimensional real elements with the same range. The main difference between these two data sets is that the elements in the latter data set are arranged from large to small in each dimension.

Figure 4 shows the construction time (seconds) for $2^{18} \leq N \leq 2^{24}$ 4-dimensional randomly generated real elements based on the new method and the improved method (more details about the improved method can be seen in [14]). Since the execution time of these two methods is

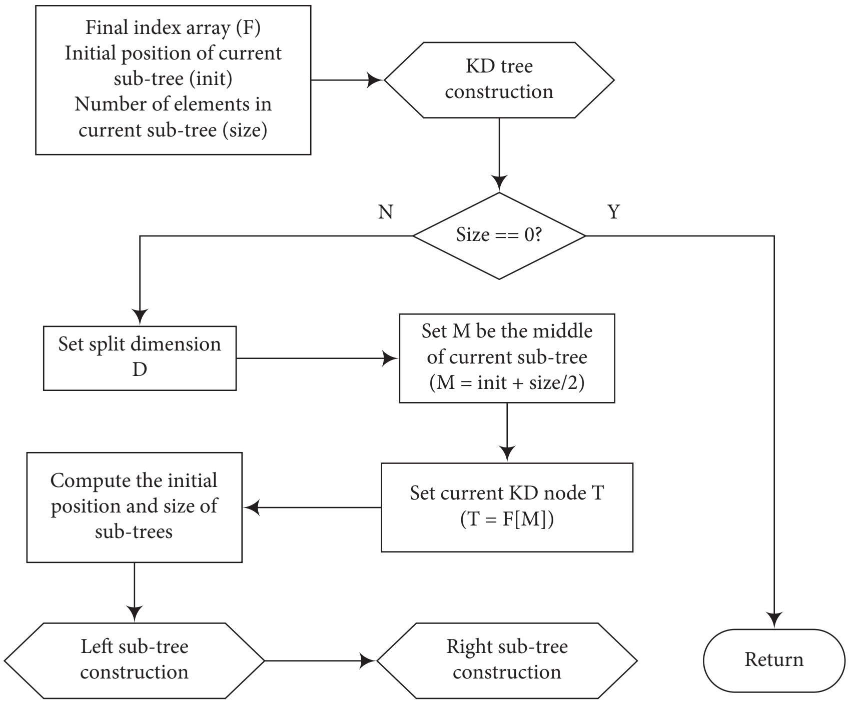
Final index array (F)
Initial position of current sub-tree (init)
Number of elements in current sub-tree (size)
KD tree construction
N
Y
Size == 0?
Set split dimension D
Set M be the middle of current sub-tree (M = init + size/2)
Compute the initial position and size of sub-trees
Set current KD node T (T = F[M])
Left sub-tree construction
Right sub-tree construction
Return

Figure 3: Building a KD tree recursively according to the final index array.

supposed to be proportional to N $\log_2 N$, the x-axis expands by a factor of $\log_2 N$, so that the result can be fitted to two straight lines. The construction time based on the new method is marked by circles and fitted as the solid line, while construction time based on the improved method is marked by diamonds and fitted as the dashed line. It can be seen clearly from the picture that the execution time of these two methods increases almost linearly with $N\log_2 N$ indeed, which demonstrates the correctness of our analysis about the complexity of the new method. What is more, the performance of the new method is always better than the improved method in the current case, which is exactly what we expected. In addition, the overhead of the construction grows a little bit faster when the number of elements is too large, and this might be due to the increase of Cache failure caused by large scale data.

In order to further verify the performance of the new method in different dimensions, we carefully compare the new method with the improved method and the quick select method (more details about the process of the quick select method can be found in [15]). Since the construction time of all the three methods tends to grow linearly with the increase of the dimension, the results for these three methods are also fitted to straight lines (see Figure 5). It is pleasing to find that the new method has almost equivalent performance with the quick select method. Though the execution time based on the new method is still slightly larger than that based on the quick select method, considering that the quick select method is the best at handling random data, the new approach has performed well enough. In addition, the improved method performs equivalently in our 3D case and better in our 2D case, and this is because the improved method does not maintain unnecessary index arrays when

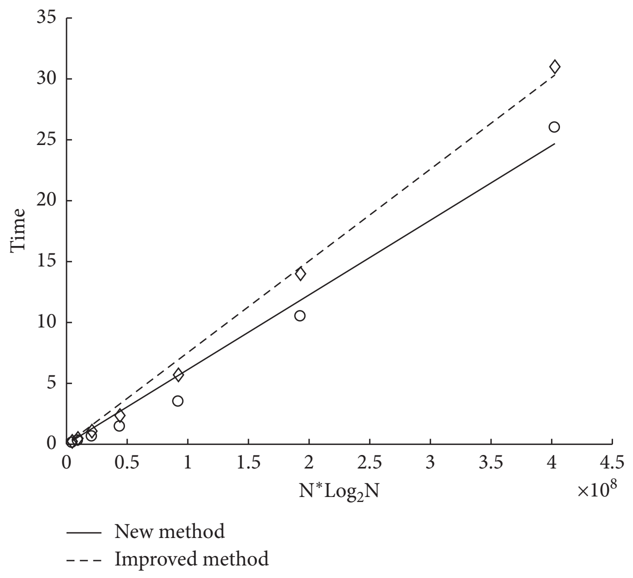

Figure 4: Construction time (seconds) for $2^{18} \leq N \leq 2^{24}$ 4-dimensional randomly generated real elements.

processing 2D data. Meanwhile, the operation adopted by the improved method is simpler. The disadvantage of maintaining all index arrays in the improved method is only shown when dealing with high-dimensional problems.

The quick select method adopted in this paper always chooses the first element as the pivot element to partition the remaining subarray. Therefore, its performance in each recursion would degrade to O (N^2) level when the arrays are arranged from large to small. Figure 6 shows the construction time (seconds) for 2^{17} real elements (the elements

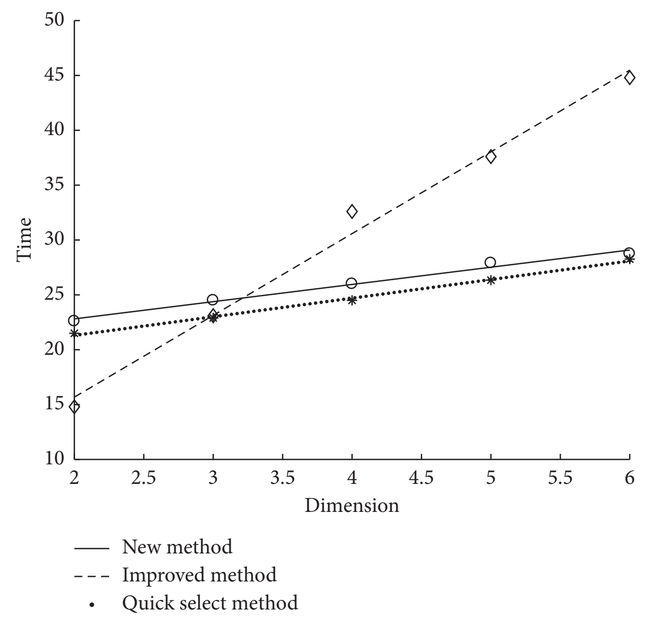

Figure 5: Construction time (seconds) for 2^{24} randomly generated real elements in different dimensions.

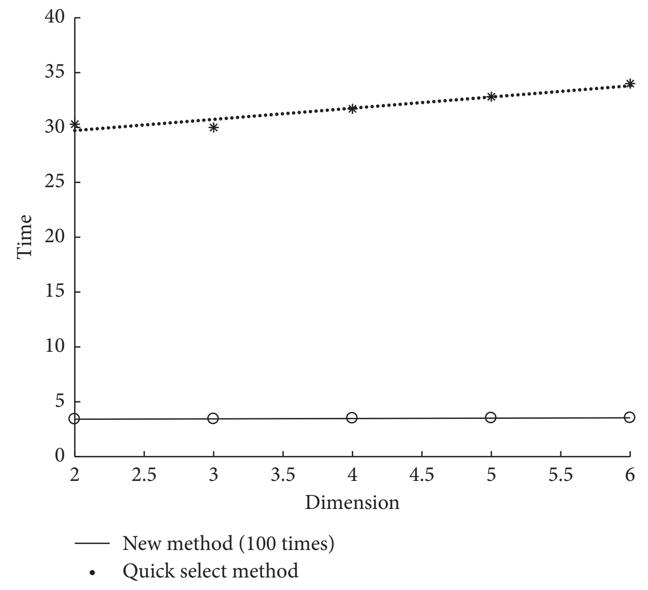

Figure 6: Construction time (seconds) for 2^{17} real elements (the elements are arranged from large to small in each dimension) in different dimensions. The execution time based on the new method is magnified by 100 times.

are arranged from large to small in each dimension) in different dimensions. As the construction time based on the new method is approaching zero, the result of the new method is magnified by 100 times in this figure. Comparing with Figure 5, it can be seen that the execution time based on the quick select method in Figure 6 even exceeds the time cost of processing 2^{24} elements in Figure 5. In contrast, the performance of the new method is quite stable, which performs far better than the quick select method. It illustrates that the performance of the new method is not affected by the initial conditions of the data.

4. Conclusions

In this paper, we proposed a new method that guarantees to construct the KD tree in O ($N\log_2 N$) time (excluding the overhead of presorting process), with the help of three additional integer arrays. Compared with previous methods, the new method has almost equivalent performance with that based on the quick select method for random data and performs much better under extreme conditions. Though the improved method [14] performs better in two dimensional cases, the benefits of the new method soon become apparent in high-dimensional cases.

The new method is suitable for complex systems that need multidimensional queries of massive data. For example, in a cloud storage system based on key-value pair model, multidimensional queries often require a complete scan of the entire data set, which is very inefficient. With the help of KD tree, the efficiency of the query would be improved greatly [16], and our method will improve the efficiency of building the KD tree significantly.

In order to further reduce the execution time of KD tree construction, parallelism is essential [17–20]. Construction method based on recursion usually carries out the construction of the left and right subtrees in parallel. However, the main part of the new method no longer employs the recursive scheme. Therefore, discovering the characteristics of the data structure and the process of the new method so as to develop new parallel algorithms is the focus of our future research.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

Our previous work about the improved method has been published in “2020 IEEE 11th International Conference on Software Engineering and Service Science (ICSESS)” [14]. In this paper, the authors proposed a new method to construct the KD tree without continuously updating the presorted index arrays, which performs much better than the improved method in high-dimensional cases. The authors would like to thank the funding from the National Key R&D Program of China (2018YFC1406202) and the National Natural Science Foundation of China (41830964).

References

[1] W.-F. Hou, D.-W. Li, C. Xu et al., “An advanced k nearest neighbor classification algorithm based on kd-tree,” in *Proceedings of the 2018 IEEE International Conference of Safety Produce Informatization (IICSPI)*, pp. 902–905, Chongqing, China, December 2018.

[2] B. Choi, R. Komuravelli, V. Lu et al., “Parallel SAH k-D tree construction,” in *Proceedings of the Conference on High Performance Graphics (HPG ’10)*, pp. 77–86, Saarbrucken, Germany, June 2010.

[3] M. Shevtsov, A. Soupikov, and A. Kapustin, “Highly parallel fast KD-tree construction for interactive ray tracing of dynamic scenes,” *Computer Graphics Forum*, vol. 26, no. 3, pp. 395–404, 2007.

[4] J. L. Bentley, “Multidimensional binary search trees used for associative searching,” *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[5] B. Xiao and G. Biros, “Parallel algorithms for nearest neighbor search problems in high dimensions,” *SIAM Journal on Scientific Computing*, vol. 38, no. 5, pp. 667–699, 2016.

[6] Md. M. A. Patwary, N. R. Satish, N. Sundaram et al., “PANDA: Extreme scale parallel K-nearest neighbor on distributed architectures,” in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 494–503, Chicago, IL, USA, May 2016.

[7] J. H. Friedman, J. L. Bentley, and R. A. Finkel, “An algorithm for finding best matches in logarithmic expected time,” *ACM Transactions on Mathematical Software*, vol. 3, no. 3, pp. 209–226, 1977.

[8] G. Adelson-velskii and E. Landis, “An algorithm for the organization of information,” *Proceedings of the USSR Academy of Sciences*, vol. 146, pp. 263–266, 1962.

[9] P. K. Agarwal, K. Fox, K. Munagala et al., “Parallel algorithms for constructing range and nearest-neighbor searching data structures,” in *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium*, San Francisco, CA, USA, June 2016.

[10] C. A. R. Hoare, “Quicksort,” *The Computer Journal*, vol. 5, no. 1, pp. 10–16, 1962.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest et al., *Introduction to Algorithms*, The MIT Press, Cambridge, MA, USA, 3rd edition, 2009.

[12] J. Zhang, Y.-P. Gao, Y.-S. He et al., “Algorithm improvement of two-way merge sort based on OpenMP,” *Applied Mechanics and Materials*, vol. 3682, no. 1403, pp. 24–29, 2015.

[13] R. A. Brown, “Building a Balanced k-d Tree in O(kn log n) Time,” *Journal of Computer Graphics Techniques*, vol. 4, pp. 50–68, 2015.

[14] Y. Cao, X.-J. Zhang, B.-H. Duan et al., “An improved method to build the KD tree based on presorted results,” “, in *Proceedings of the 11th International Conference on Software Engineering and Service Science*, pp. 71–75, Beijing, China, October 2020.

[15] Y. Cao, B. Wang, W.-J. Zhao et al., “Research on searching algorithms for unstructured grid remapping based on kd tree,” in *Proceedings of the 3rd International Conference on Computer and Communication Engineering Technology*, pp. 29–33, Beijing, China, August 2020.

[16] J. He, Y. Wu, F. Yang et al., “Multi-dimensional cloud index based on KD-tree and R-tree,” *Journal of Computer Applications*, vol. 34, no. 11, pp. 3218–3221, 2014.

[17] G. D. Fatta and D. Pettinger, “Dynamic load balancing in parallel kd-tree K-means,” in *Proceedings of the 2010 IEEE 10th International Conference on Computer and Information Technology*, pp. 2478–2485, Bradford, UK, June 2010.

[18] L. Hu, S. Nooshabadi, and M. Ahmadi, “Massively parallel KD-tree construction and nearest neighbor search algorithms,” in *Proceedings of the 2015 IEEE International Symposium on Circuits and Systems (ISCAS 2015)*, vol. 5, pp. 2752–2755, Lisbon, Portugal, May 2015.

[19] D. Wehr and R. Radkowski, “Parallel kd-tree construction on the GPU with an adaptive split and sort strategy,” *International Journal of Parallel Programming*, vol. 46, no. 6, pp. 1139–1156, 2018.

[20] M. Cheng, “Parallel SAH based KD tree construction algorithm,” *Advanced Materials Research*, vol. 433-440, pp. 3543–3547, 2012.