적응적 분할 정렬(Adaptive Partition Sort): 퀵 정렬과 병합 정렬의 결합을 통한 효율적 정렬 알고리즘 개발

권동한[O1]
[1]하나고등학교
kznm.develop@gmail.com

Adaptive Partition Sort: Developing an Efficient Sorting Algorithm by Combining the Strengths of Quick Sort and Merge Sort

Ryan Donghan Kwon[O1]
[1]Hana Academy Seoul

요 약

This paper presents an investigation into the performance of the Adaptive Partition Sort (APS) algorithm, a hybrid sorting technique that combines the strengths of Quick Sort and Merge Sort. The primary focus of this study is to evaluate the performance of APS against other well-known sorting algorithms such as Quick Sort, Merge Sort, Timsort, and Introsort when handling large datasets with randomized data. A comprehensive experimental setup is described, which involves generating random datasets and measuring the execution time of each algorithm. Although the paper provides insights into the performance of APS with various threshold values, it does not delve into determining which partitioning strategy leads to the optimal solution. The results of this study contribute to a better understanding of the efficiency and adaptability of the APS algorithm in the context of sorting large, random datasets.

I. Introduction

Sorting is a fundamental operation in computer science, with numerous applications in diverse fields, such as databases, search engines, and data analysis. Over the years, several sorting algorithms have been developed, each with its own strengths and weaknesses. Two of the most widely used sorting algorithms are Quick Sort and Merge Sort, both of which offer efficient sorting in the average case, but they exhibit different performance characteristics in specific scenarios [1]. In this paper, we propose a novel sorting algorithm called Adaptive Partition Sort (APS), which combines the strengths of Quick Sort and Merge Sort to achieve better performance in a wide range of scenarios.

The main idea behind APS is to adapt its behavior based on the input data and a user-defined threshold value. When the input data allows for a balanced partition, the algorithm acts like Quick Sort; otherwise, it behaves like Merge Sort. This dynamic partitioning strategy enables APS to achieve better average-case performance than both Quick Sort and Merge Sort while maintaining the stability and predictable time complexity of Merge Sort. This paper investigates the performance of APS by comparing it against other well-known sorting algorithms on large datasets with randomized data.

* This work was supported by the KISTI National Supercomputing Center with supercomputing resources including technical support.

II. Related Work

Several hybrid sorting algorithms have been proposed to leverage the strengths of different sorting techniques while mitigating their weaknesses. One notable example is the Timsort algorithm, used as the default sorting algorithm in Python and Java [2]. Timsort is a hybrid of Merge Sort and Insertion Sort that takes advantage of existing sorted runs in the input data, improving the performance on partially sorted data [3]. Another example is Introsort [4], which combines the partitioning strategy of Quick Sort with the predictability of Heap Sort. Introsort starts as a Quick Sort, but if the recursion depth exceeds a certain threshold, it switches to Heap Sort to avoid the worst-case quadratic behavior of Quick Sort [5].

III. Adaptive Partition Sort Algorithm

The Adaptive Partition Sort (APS) algorithm is a hybrid sorting technique that combines the strengths of Quick Sort and Merge Sort. The key idea behind APS is to adapt its behavior based on the input data and a user-defined threshold value. When the input data allows for a balanced partition, the algorithm acts like Quick Sort; otherwise, it behaves like Merge Sort. This dynamic partitioning strategy enables APS to achieve better average-case performance than both Quick Sort and Merge Sort while maintaining the stability and predictable time complexity of Merge Sort.

APS employs a partitioning strategy similar to Quick Sort, which involves selecting a pivot element and partitioning the input array around this pivot. To improve the partitioning performance, APS uses the median-of-three method to choose the pivot element. This method selects the median of the first, middle, and last elements of the input array as the pivot, which increases the likelihood of obtaining balanced partitions in the average case [6]. The main innovation of the APS algorithm is its ability to dynamically adapt its behavior based on a user-defined threshold value (T). After partitioning the input array, APS calculates the size difference between the two resulting subarrays. If the size difference is less than or equal to the threshold value T, the algorithm proceeds with the Quick Sort-like behavior, recursively applying the partitioning strategy to the subarrays. However, if the size difference exceeds T, APS switches to Merge Sort-like behavior by sorting the subarrays using Merge Sort and then merging them back together.

```
1   function adaptive_partition_sort(A, T, left, right):
2       if left < right:
3           pivot_index = median_of_three(A, left, right)
4           partition_index =
~               partition(A, left, right, pivot_index)
5
6       size_difference = abs((partition_index - 1)
~           - left - (right - partition_index))
7
8       if size_difference <= T:
9           adaptive_partition_sort(
~               A, T, left, partition_index - 1)
10          adaptive_partition_sort(
~               A, T, partition_index, right)
11      else:
12          merge_sort(A, left, partition_index - 1)
13          merge_sort(A, partition_index, right)
14          merge(A, left, partition_index - 1, right)
```

Figure 1. Pseudocode of APS Algorithm

IV. Experimental Results

The experimental setup was designed to evaluate the performance of the Adaptive Partition Sort (APS) algorithm compared to Quick Sort, Merge Sort, and Timsort on a large dataset containing random data. The execution time of the sorting algorithms was measured on a dataset of 100,000 random integers. The benchmark was run on the KISTI National Supercomputing Center's NEURON system, gpu-c40m100k40-2 instance, using Python 3.9.16. As shown in Table 1, APS demonstrated competitive performance compared to Quick Sort and especially Merge Sort, but was slower than Timsort, which is currently known as the optimal algorithm.

Furthermore, additional experiments were conducted to determine the optimal threshold value for the APS algorithm when given a data set of 100,000 samples. Testing various threshold values and averaging the execution times revealed that a threshold value of less than 2,000 provides the best performance. However, since the results were not consistent in all cases, it appears that the characteristics of the data set can have a significant impact on the optimal threshold value.

Algorithm (ms)	Best Time	Worst Time	Average Time
Quick Sort	267.763138	327.25215	296.163488
Merge Sort	426.17178	583.059311	452.732164
Timsort	17.953634	20.164251	19.432977
APS	280.852556	372.13707	319.631994

Table 1. Compare execution speeds by algorithm

V. Conclusion

In this paper, we have introduced the Adaptive Partition Sort (APS) algorithm, a novel hybrid sorting technique that combines the strengths of Quick Sort and Merge Sort. By adapting its behavior based on the input data and a user-defined threshold value, APS achieves better average-case performance than both Quick Sort and Merge Sort while maintaining the stability and predictable time complexity of Merge Sort. Experimental results on large datasets with randomized data demonstrate the competitive performance of APS compared to other well-known sorting algorithms, such as Quick Sort and Merge Sort. Although APS showed slower speed compared to Timsort, it can be considered advantageous as a hybrid sorting algorithm that combines the predictability of time and space complexity of Quick Sort and Merge Sort, respectively.

Future work could focus on further refining the adaptive partitioning strategy to identify the optimal threshold value for different types of input data, as well as exploring other partitioning strategies that could lead to even better performance. Additionally, it would be interesting to investigate the performance of APS in parallel and distributed computing environments, as well as its applicability to real-world problems in domains such as databases, search engines, and data analysis.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to Algorithms," MIT Press, 2009.

[2] T. Peters, "Timsort," accessed April 2022, https://svn.python.org/projects/python/trunk/Objects/listsort.txt.

[3] N. Auger, V. Jugè, C. Nicaud, and C. Pivoteau, "On the worst-case complexity of TimSort," arXiv preprint arXiv:1805.08612.

[4] D. R. Musser, "Introspective sorting and selection algorithms," Software: Practice and Experience, vol. 27, no. 8, pp. 983-993, 1997.

[5] M. A. Weiss, "Data Structures and Algorithm Analysis in C++, 4th ed.," Pearson, 2013.

[6] R. Sedgewick, "Algorithms in C++, parts 1-4: fundamentals, data structure, sorting, searching, 3rd ed.," Pearson Education, 1998.