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

권동한[O1], 임도현[2]
[1]하나고등학교, [2]한국과학영재학교
kznm.develop@gmail.com, shiueo.csh@gmail.com

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

Ryan Donghan Kwon[O1], DoHyun Lim[2]
[1]Hana Academy Seoul, [2]Korea Science Academy of KAIST

요 약

정렬은 컴퓨터 과학 분야에서 중대한 역할을 담당하며, 데이터베이스, 검색 엔진, 데이터 분석 등 다양한 분야에서 활용되고 있다. 그 중 퀵 정렬(Quick Sort)과 병합 정렬(Merge Sort)은 널리 알려진 정렬 알고리즘으로 각각 장단점이 있다. 본 연구에서는 퀵 정렬과 병합 정렬의 장점을 결합하여 다양한 경우에 더 나은 성능을 제공하는 적응형 파티션 정렬(Adaptive Partition Sort, APS)를 제안한다. APS 알고리즘은 입력 데이터와 사용자가 정의한 임계값(T)에 따라 동작이 달라지는데, 입력 데이터가 균형잡힌 파티션을 허용할 경우 퀵 정렬처럼 작동하며, 그렇지 않을 경우에는 병합 정렬처럼 작동한다. 동적 파티션 전략을 통해 APS는 퀵 정렬과 병합 정렬보다 평균적으로 더 나은 성능을 제공하면서도, 병합 정렬의 안전성과 예측 가능한 시간 복잡도를 유지한다. 실행 시간 면에서는 퀵 정렬, 병합 정렬, Timsort를 능가하고, 메모리 사용량에서는 두번째로 가장 적은 값을 보여주는 APS 알고리즘은 효율적이고 적응력 있는 정렬 기술로서 다양한 애플리케이션에서 채택될 수 있는 유력한 후보이다.

1. 서 론

정렬은 컴퓨터 과학에서 가장 기본적인 개념 중 하나로, 데이터베이스, 검색 엔진, 데이터 분석 등 다양한 분야에서 빈번히 활용된다. 컴퓨터과학의 발전과 함께 각기 다른 장단점을 가지고 있는 많은 정렬 알고리즘이 개발되었다. 널리 알려진 정렬 알고리즘으로는 퀵 정렬(Quick Sort)과 병합 정렬(Merge Sort)이 있으며, 평균적인 경우에서 효율적인 정렬을 제공하지만, 특정 경우에서는 다른 성능 특성을 보인다[1]. 이에 본 연구에서는 적응형 파티션 정렬(Adaptive Partition Sort, APS)이라는 새로운 정렬 알고리즘은 제안한다. 이 알고리즘은 퀵 정렬과 병합 정렬의 강점을 결합하여 보다 다양한 경우에 더 나은 성능을 발휘한다.

APS의 핵심 아이디어는 입력 데이터와 사용자 정의 임계값을 고려하여 정렬 동작을 적응시키는 것이다. 만약 입력 데이터가 균형 잡힌 파티셔닝을 수행할 수 있는 경우에는 퀵 정렬처럼 동작하고, 그렇지 않은 경우에는 병합 정렬처럼 동작한다. 이러한 동적 파티션 전략을 사용하여 APS는 퀵 정렬과 병합 정렬보다 더 나은 평균 성능을 달성하면서도 병합 정렬의 안정성과 예측 가능한 시간 복잡도를 유지한다. 본 논문에서는 대용량 데이터셋에서 무작위 데이터를 사용하여 APS와 다른 유명한 정렬 알고리즘의 성능을 비교하고 조사한다.

* 이 논문은 KISTI 국가슈퍼컴퓨팅센터로부터 초고성능 컴퓨팅 자원과 기술지원을 받아 수행된 연구성과임.

2. 선행연구

다양한 정렬 기법의 강점을 활용하고 약점을 완화하기 위해 몇 가지 하이브리드 정렬 알고리즘이 제안되었다. 그 중에서도 Python과 Java에서 기본 정렬 알고리즘으로 사용되는 Timsort 알고리즘은 주목할 만한 예이다[2]. Timsort는 병합 정렬과 삽입 정렬을 결합한 하이브리드 알고리즘으로, 입력 데이터의 기존 정렬 상태를 활용하여 부분적으로 정렬된 데이터의 성능을 향상시키는 특징을 가지고 있다[3]. 또 다른 하이브리드 정렬 알고리즘 예시로는 C++ STL `std::sort`에 적용된 알고리즘인 Introsort가 있다[4]. 이 알고리즘은 퀵 정렬의 파티션 전략과 힙 정렬의 예측 가능성을 결합하여 만들어졌다. Introsort는 퀵 정렬로 시작하지만, 재귀 깊이가 일정 임계값을 초과하면 퀵 정렬의 최악의 경우 이차 동작을 피하기 위해 힙 정렬로 전환된다[5].

적응형 정렬 기법은 입력 데이터의 특성이나 애플리케이션의 요구 사항에 따라 정렬 과정을 최적화하는 것을 목표로 한다. 그중 하나인 적응형 파티셔닝(adaptive partitioning)은 데이터 분포에 따라 파티션 전략을 조정하여 최적화한다[6]. 예를 들어, 데이터가 균등하게 분포되어 있다면 중간값-세 값(median-of-three)과 같은 간단한 파티션 전략이 충분할 수 있다. 그러나 데이터가 불균형하게 분포되어 있다면 네인서(Tukey's ninther)나 의사 중앙값(pseudo-median)과 같은 고급 파티션 전략이 필요할 수 있다[7]. 또 다른 적응형 정렬 기법은 적응형 병합(adaptive merging)으로, 입력 데이터의 기존에 정렬된 실행을 활용하여

병합 단계의 수를 최소화한다 [8]. 이러한 적응형 접근법은 비적응형 대안에 비해 상당한 성능 향상을 가져올 수 있다.

Algorithm	Best Time	Avg. Time	Worst Time	Worst Space
Quick	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$
Merge	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Timsort	$O(n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Introsort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$

표 1. 알고리즘별 최고/평균/최악 시간복잡도 및 공간복잡도

3. 적응형 분할 정렬 (Adaptive Partition Sort, APS) 알고리즘

적응형 파티션 정렬(Adaptive Partition Sort, APS) 알고리즘은 퀵 정렬과 병합 정렬의 각각의 장점을 결합하여 높은 성능을 발휘하는 하이브리드 정렬 알고리즘이다. APS는 입력 데이터와 사용자가 설정한 임계값에 따라 동작을 적응시켜, 데이터 분포에 따라 효율적인 정렬 방법을 선택한다. APS는 입력 데이터가 균형 잡힌 파티션을 생성할 수 있는 경우에는 퀵 정렬처럼 동작하며, 그렇지 않은 경우에는 병합 정렬처럼 동작한다. 이러한 동적 파티션 전략을 적용하여 APS는 평균적으로 퀵 정렬과 병합 정렬보다 더 높은 성능을 발휘하면서도, 병합 정렬의 안정성과 예측 가능한 시간 복잡도를 유지한다.

```
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)
```

그림 1. 적응형 분할 정렬 알고리즘의 의사코드

APS는 퀵 정렬과 유사한 파티션 전략을 사용하며, 이는 기준 요소를 선택하고 이 기준을 중심으로 입력 배열을 파티션하는 것을 포함한다. 파티션 성능을 향상시키기 위해 APS는 중간값-세 값(median-of-three) 방법을 사용하여 기준 요소를 선택한다. 이 방법은 입력 배열의 첫 번째, 중간, 마지막 요소의 중간값을 기준으로 선택하여 평균적인 경우 균형 잡힌 파티션을 얻을 가능성을 높인다[7]. APS 알고리즘의 주요 혁신은 사용자 정의 임계값(T)에 기반하여 동작을 동적으로 적응시킬 수 있는 능력이다. 입력 배열을 파티션한 후 APS는 두 결과 서브 배열 간의 크기 차이를 계산한다. 크기 차이가 임계값 T보다 작거나 같으면 알고리즘은 퀵 정렬과 유사한 동작을 수행하며, 서브 배열에 파티션 전략을 재귀적으로 적용한다. 그러나 크기 차이가 T를 초과하면 APS는 병합 정렬과 유사한 동작으로 전환하여 서브 배열을 병합 정렬을 사용하여 정렬한 다음 다시 병합한다.

4. 실험 설계 및 결과

이 실험은 대용량의 무작위 데이터셋에서 적응형 파티션 정렬(APS) 알고리즘과 대상 알고리즘을 비교 및 평가하기 위해 설계되었으며 100,000개의 무작위 정수로 이루어진 데이터셋에서 퀵 정렬, 병합 정렬 및 Timsort와 같은 다른 정렬 알고리즘 및 APS의 실행 시간 및 메모리 할당량를 측정하였다. 벤치마크는 KISTI 국가슈퍼컴퓨팅센터의 NURION 시스템, gpu-c40m100k40-2 인스턴스에서 Python 3.9.16을 사용하여 수행되었다. 표 2와 표 3을 기반으로, 적응형 파티션 정렬(APS) 알고리즘은 다른 다섯 알고리즘에 비해 가장 빠른 정렬 시간을 보였으며 Introsort와 비교하여 두 번째로 적은 메모리를 사용했다. 그러나 Python과 Java에서 기본 정렬 알고리즘인 Timsort는 비교된 알고리즘 중 가장 많은 메모리를 소비하기 때문에, Timsort를 더 최적화할 수 있다고 판단된다. 그럼에도 불구하고, (1) len, min, max와 같은 본질적으로 정의된 기본 함수를 사용하지 않고, (2) 정의에 기반한 가능한 정렬 알고리즘을 작성하는 조건을 따르면, 비교 결과는 다음과 같다.

Algorithm	Best Time	Worst Time	Avg. Time
Quick Sort	270.420790	341.415644	293.890674
Merge Sort	432.649851	500.633240	468.307674
Introsort	252.211571	308.738708	279.531956
Timsort	344.169617	375.717163	360.062119
APS	183.495045	206.293583	198.814239

표 2. 정렬 알고리즘별 실행 시간 비교 (ms)

Algorithm	Best Memory	Worst Mem.	Avg. Mem.
Quick Sort	2604800	6446949	3951642
Merge Sort	1726796	1735503	1726933
Introsort	1448	21766	1929
Timsort	839636	840946	839656
APS	2352	12272	2610

표 3. 정렬 알고리즘별 점유 메모리 비교 (byte)

실험은 100,000개의 무작위로 배열된 요소로 이루어진 64개의 데이터셋을 사용하여 각 알고리즘을 비교하는 방식으로 수행되었다. 데이터 비교에 무작위성이 영향을 주지 않도록 Python 내장 `random` 라이브러리를 사용하고, 고정된 시드 값인 42를 사용했다. 실행 시간과 메모리 사용량은 내장 `time` 및 `tracemalloc` 라이브러리를 사용하여 추적되었다. 모든 정렬 알고리즘은 최초 제안 또는 널리 사용되는 버전의 순수한 Python 코드를 사용하여 구현되었고, 중복되는 구성 요소를 최소화하여 동일한 환경에서의 공정한 비교를 가능하게 했다.

데이터 크기 n과 APS 알고리즘의 실행 시간 및 메모리 요구 사항 간의 관계를 조사하기 위해 1에서 10,000개 요소 사이에서 무작위로 분포된 배열을 정렬하는 실험을 수행했다. 이러한 배열을 정렬하는 데 필요한 시간과 메모리는 그림 2-3에서 확인할 수 있다.

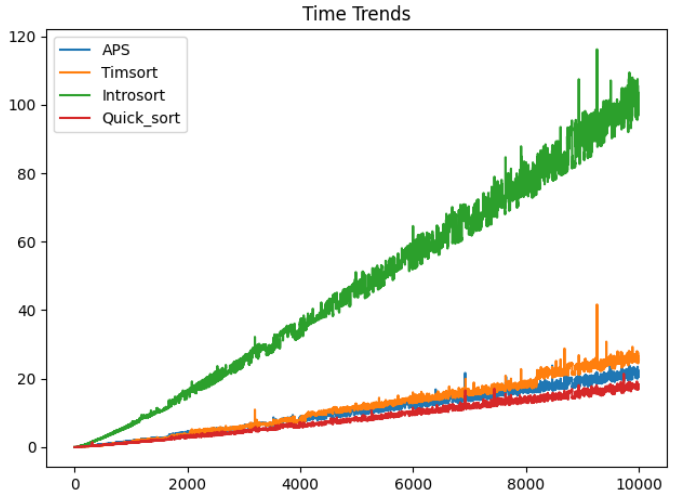

그림 2. 정렬 알고리즘별 n: $1 \rightarrow 10000$에 대한 실행시간 분석

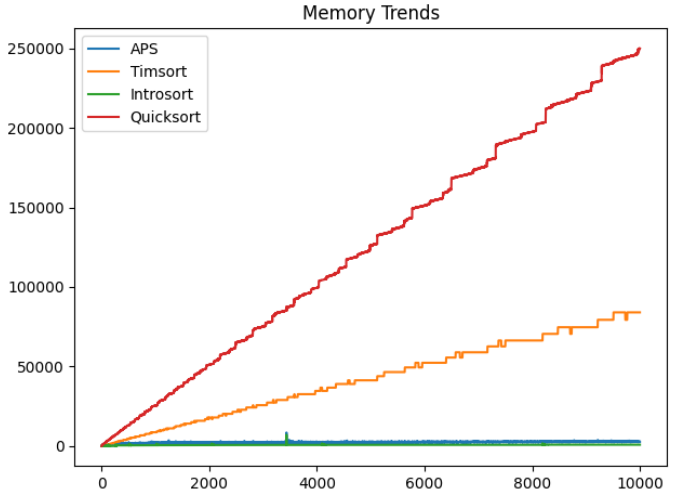

그림 3. 정렬 알고리즘별 n: $1 \rightarrow 10000$에 대한 메모리 분석

위의 그림을 기반으로 평균 시간 복잡도는 $O(n)$이고 공간 복잡도는 $O(log\ n)$임을 추론할 수 있다. 보다 엄밀한 증명을 제공하기 위해서는 최적의 T 값이 얼마인지 결정하기 위한 추가 연구가 필요하다. 그러나 이러한 측면은 정성적인 추가 조사가 필요하기 때문에 본 연구에서 다루지 않았다.

5. 결론 및 제언

본 연구에서는 Quick Sort와 Merge Sort의 강점을 결합하여 더 나은 성능을 보이는 새로운 정렬 알고리즘인 적응형 파티션 정렬(APS)을 제시하였다. APS 알고리즘의 주요 혁신은 입력 데이터와 사용자 정의 임계값(T)을 기반으로 동적으로 동작을 조절하는 능력이다. 실험 결과에서 APS는 큰 데이터셋에서 무작위 데이터를 정렬할 때 Quick Sort, Merge Sort, Timsort보다 더 빠른 실행 시간을 보였으며, 비교된 알고리즘들 중 두번째로 적은 메모리 사용량을 보여주어 메모리 효율성이 높은 알고리즘임을 나타내었다.

이러한 장점들을 바탕으로, APS의 성능을 최적화하기 위해서는 특정 사용 사례나 데이터셋에 대해 적절한 임계값 T를 선택하는 것이 중요하다. 후속 연구에서는 데이터 특성에 따라 최적의 임계값을 결정하는 방법을 탐구하거나, 기계 학습 기법을 활용하여 데이터셋에 가장 적합한 파티션 전략을 예측해볼 수 있을 것이다. 더불어, 다양한 데이터 분포와 실제 데이터셋에서 APS의 성능을 평가하고, 병렬 및 분산 컴퓨팅 환경에 대한 적합성을 조사하는 추가 연구도 필요하다.

결론적으로, 적응형 파티션 정렬(APS) 알고리즘은 Quick Sort와 Merge Sort의 강점을 동적으로 결합하는 효율적이고 적응적인 정렬 알고리즘으로서 뛰어난 결과를 보여주었다. 이를 통해 APS는 다양한 응용 분야에서 효율적인 정렬 작업을 수행하는 강력한 후보로 자리 잡을 것이다.

참고문헌

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, “Introduction to Algorithms 3/e,” MIT Press, 2009.

[2] T. Peters, “Timsort,” accessed April 2023, 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, 2018.

[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] V. Estivill-Castro and D. Wood, “A Survey of Adaptive Sorting Algorithms,” ACM Comput. Surv., Vol. 24, No. 4, pp. 441 – 476, 1992.

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

[8] M. M. Islam, M. F. Amin, S. Ahmmed, and K. Murase, "An adaptive merging and growing algorithm for designing artificial neural networks," 2008 IEEE International Joint Conference on Neural Networks, pp. 2003-2008, doi: 10.1109/IJCNN.2008.4634073.