병렬화된 Partial Rebuild를 사용하는 균형이진탐색트리

김태우○
전산학부, 한국과학기술원
travis1829@kaist.ac.kr

Balanced Binary Search Trees with Parallelized Partial Rebuild

Tae Woo Kim○
School of Computing, KAIST

요 약

Partial rebuild를 이용한 tree rebalancing은 tree rotation을 이용한 방식보다 구현이 쉽고 직관적이지만, 이를 활용하는 트리들은 이보다 더 유명하고 준수한 성능으로 알려진 AVL tree, Red Black tree 등에 비해 자주 사용되지 못했다. 이에 따라, 이 논문에서는 기존의 partial rebuild의 코드를 조금만 바꾸는 간단한 방법으로 partial rebuild를 병렬화해 수행시간을 크게 줄일 수 있음을 보인다. 또, 이를 통해 partial rebuild를 사용하는 Weight Balanced tree를 단조증가 수열에서의 성능을 개선하여 이전에는 AVL tree보다 느렸던 경우가 이후에는 빨라지도록 할 수 있음을 보이고자 한다. 특히, 병렬화로 균형이진탐색트리의 성능을 개선한 다른 방법들은 매우 복잡하고, 또, 여러 개의 operation을 동시에 수행할 때의 성능만을 개선한다는 단점이 있었지만, 이 논문에서 소개하는 방법은 위 단점이 없다.

1. 서론

균형이진탐색트리는 매우 기본적이고 중요한 자료구조이다. 일반적인 이진탐색트리는 트리의 높이가 $O(n)$이 될 수 있지만, 균형이진탐색트리는 지속적인 rebalancing을 통해 트리의 높이가 항상 $O(\log n)$이도록 하며, 이를 통해 탐색, 삽입, 삭제 등을 $O(\log n)$ 또는 amortized $O(\log n)$만에 수행할 수 있도록 한다. 잘 알려진 균형이진탐색트리로는 AVL Tree[1], Red Black Tree[2, 5] 등이 있으며, 이 트리들은 tree rotation을 이용하여 트리를 rebalance한다.

그러나, 위와 같이 tree rotation을 써서 rebalancing을 하는 균형이진탐색트리들은 구현이 복잡하고 실수하기 쉽다고 알려져 있다. Munro et al.[6]에 의하면, 이러한 rebalancing을 구현하려면 트리의 모든 경우를 숙지해둔 후, rebalancing을 할 때는 트리의 현재 경우를 파악해 그 경우에 해당하는 복잡한 rotation을 쓰도록 해야 하는데, 이것이 평범한 프로그래머에게는 실수하기 쉽고 구현이 복잡하다고 비판했다.

이와 달리, partial rebuild(1)을 이용한 rebalancing은 일반화된 알고리즘을 사용하므로 트리의 모든 경우의 수를 세심히 따져보지 않아도 되며, 트리 중위 탐색(inorder traversal)이나 분할 정복(divide and conquer)과 같은 일반적인 알고리즘으로 쉽고 직관적으로 구현할 수 있다. (3장 참고) 이를 활용한 트리의 예로는 Scapegoat tree[4]나 변형된 Weight Balanced tree[7] 등이 있지만, 상대적으로 더 유명하고 높은 성능으로 알려진 AVL tree나 Red Black tree 등에 비해, 위 트리들은 비교적 자주 사용되지 못했다.

이에 따라, 이 논문에서는 간단한 병렬화로 partial rebuild의 수행시간을 크게 줄일 수 있음을 보이고, 이를 통해 partial rebuild를 사용하는 Weight Balanced tree (줄여서 WB tree)를 단조증가 수열에서의 성능을 향상해, AVL tree보다 느렸던 경우가 병렬화 후에는 더 빨라지도록 할 수 있음을 보인다. 이외에도, 추가적인 활용법을 간략히 소개한다.

특히, 이 논문에서 소개하는 방법은 병렬화로 균형이진탐색트리의 성능을 개선한 다른 방법들과 달리, thread의 기본적인 활용법만으로 구현할 수 있으며 일반적인 partial rebuild 알고리즘을 매우 조금만 바꾸면 구현할 수 있다. 기존에는 복잡하게 lock을 배치하고서 race condition이 일어나지 않도록 조심스럽게 설계하거나, 또는, lock-free 형태로 훨씬 더 복잡한 방법을 사용했으며, 특히, 이렇게 하더라도 동시에 여러 개의 operation을 수행하는 경우에만 성능을 향상할 수 있었다. 이와 달리, 이 논문에서 소개하는 방법은 성능증가 폭은 더 작고 제한적이지만, 훨씬 간단하게 구현할 수 있다.

(1) unbalanced subtree를 perfectly balanced subtree로 바꾸는 알고리즘

2. 수행시간의 이론적 분석

먼저, 균형이진탐색트리에서의 삽입/삭제 operation의 수행시간을 tree rotation을 사용하는 트리와 partial rebuild를 사용하는 트리로 나누어 이론적으로 분석해보고자 한다. 이를 위해, 전자의 예로는 AVL tree, 후자의 예로는 WB tree를 이용하겠지만, 아래의 분석은 일반적으로 다른 트리들에도 적용된다.

먼저, 삽입/삭제 operation을 할 때, AVL tree와 WB tree는 노드를 삽입/삭제할 위치를 먼저 찾은 후, 이곳에 노드를 삽입/삭제한다. 이후, AVL tree는 만약 왼쪽 subtree와 오른쪽 subtree의 높이 차가 2인 곳이 있다면 이를 (single/double) tree rotation으로 rebalance하고, 필요하다면 최대 $O(\log n)$번 반복한다. 이와 달리, WB tree는 상수 $0 < \alpha < 1/2$에 대해, 어느 subtree의 weight(2)에 α를 곱한 값보다도 왼쪽/오른쪽 subtree의 weight이 작다면, 이 subtree를 rebuild한다. 이러한 subtree가 여러 개 있었다면, 제일 큰 subtree만 rebuild한다.

그러므로, 삽입/삭제 operation의 수행시간은 트리를 탐색하는 시간과 rebalance하는 시간의 합으로 표현할 수 있다. 이때, 트리의 높이를 t_h, tree rotation의 수행시간을 t_r, 그리고 partial rebuild의 수행시간을 t_p라 하면, AVL tree의 삽입/삭제 수행시간 t_{avl}과 WB tree의 삽입/삭제 수행시간 t_{wb}는 대략 다음과 같은 Amdahl model로 표현할 수 있다. (a는 비례상수)

$$t_{avl} \le a t_h + t_h t_r$$

$$t_{wb} = \begin{cases} a t_h & (rebuild\text{이 없는 경우}) \\ a t_h + t_p & (rebuild\text{이 있는 경우}) \end{cases}$$

(2) subtree의 크기 + 1

AVL tree의 경우, 하나의 삽입/삭제 후 t_h개 이하의 tree rotation이 일어난다. 이와 달리, WB tree의 경우, 여러 삽입/삭제 operation을 계속 수행하면서 subtree의 균형이 깨질 때까지 기다린 후, 한 번에 그 subtree를 rebuild하므로, t_{wb}는 rebuild를 하는 경우와 하지 않는 경우 2가지로 나뉘며, t_p는 rebuild를 수행하는 subtree의 크기에 비례해 달라진다.

이때, $t_h = O(\log n)$이고, $t_r = O(1)$이므로 $t_{avl} = O(\log n)$이다. 이에 반해, t_p는 최악의 경우 $O(n)$이므로 $t_{wb} = O(n)$이지만, 분할상환분석(amortized analysis)을 이용하면 t_{wb}는 amortized $O(\log n)$임이 알려져있다.[7]

여기서 주목할 점은, 트리를 탐색하는 것이나 tree rotation을 이용해 rebalance를 하는 것은 단계적으로 이루어지므로 at_h, $t_h t_r$와 같은 부분은 병렬화할 수 없는 반면, t_p는 3장에서 볼 수 있듯 병렬화할 수 있다는 것이다. 쓰레드가 k개인 CPU를 사용하면, 이론적으로 t_{wb}를 다음 수준으로 줄일 수 있다.

$$t_{wb} = \begin{cases} at_h & (rebuild\text{이 없는 경우}) \\ at_h + \frac{t_p}{k} & (rebuild\text{이 있는 경우}) \end{cases}$$

이때, 4장에서 볼 수 있듯 at_h보다 t_p가 평균적으로 더 크므로, 병렬화를 활용하면 rebuild가 포함된 각각의 삽입/삭제 operation의 수행시간을 크게 줄일 수 있다.

이를 통해, AVL tree와 달리 WB tree는 rebalancing을 병렬화해 전체적인 성능을 향상할 수 있으며, 특히, 추후 4장에서 볼 수 있듯, 병렬화 전에는 WB tree가 AVL tree보다 느렸던 경우가 병렬화 후에는 더 빠른 경우도 있다.

3. 알고리즘

partial rebuild 알고리즘은 일반적으로 다음과 같이 간단히 구현할 수 있다. 먼저, 1) 트리 중위 탐색을 이용해 subtree의 모든 노드를 key가 증가하는 순서대로 배열에 복사한 후, 2) 분할 정복을 이용해 배열로부터 perfectly balanced tree를 만들면 된다. 의사 코드로 표현하자면 다음과 같으며, 과정 1)에 해당하는 것이 GetCopy, 2)에 해당하는 것이 BuildTree이다.

```
1  GetCopy(t, arr, index)
2    if t.left ≠ null then
3      GetCopy(t.left, arr, index)
4      index ← index + t.left.size
5    arr[index] ← t
6    if t.right ≠ null then
7      GetCopy(t.right, arr, index + 1)
8
9  BuildTree(arr, s, f)
10   if s > f then
11     return null
12   m ← ⌊(s + f)/2⌋
13   arr[m].left ← BuildTree(arr, s, m − 1)
14   arr[m].right ← BuildTree(arr, m + 1, f)
15   arr[m].size ← f − s + 1
16   return arr[m]
```

이때, 위 코드를 조금만 변경하여 병렬화시킬 수 있다. 위 GetCopy와 BuildTree는 재귀함수이며 내부에서 자기 자신을 2번 재귀호출하는데, 각각의 재귀호출을 서로 다른 thread가 맡아서 동시에 수행하도록 할 수 있다. 다음은 병렬화를 활용하도록 수정한 의사 코드의 한 예이며, GetCopy와 BuildTree는 그대로 둔 채 GetCopyP와 BuildTreeP라는 함수를 추가했다.

```
1  GetCopy(t, arr, index)
2    // 생략
3
4  BuildTree(arr, s, f)
5    // 생략
6
7  GetCopyP(t, arr, index, depth)
8    if t.size < 6000 or depth ≥ 3 then
9      GetCopy(t, arr, index)
10     return
11   if t.left ≠ null then
12     handler ← ThreadSpawn(GetCopyP, t.left, arr, index, depth + 1)
13     index ← index + t.left.size
14   arr[index] ← t
15   if t.right ≠ null then
16     GetCopyP(t.right, arr, index + 1, depth + 1)
17   handler.join()
18
19 BuildTreeP(arr, s, f, depth)
20   if f − s + 1 < 6000 or depth ≥ 3 then
21     return BuildTree(arr, s, f)
22   if s > f then
23     return null
24   m ← ⌊(s + f)/2⌋
25   handler ← ThreadSpawn(BuildTreeP, arr, s, m − 1, depth + 1)
26   arr[m].right ← BuildTreeP(arr, m + 1, f, depth + 1)
27   arr[m].left ← handler.getReturnValue()
28   arr[m].size ← f − s + 1
29   return arr[m]
```

GetCopyP 및 BuildTreeP가 GetCopy 및 BuildTree와 비교하여 크게 달라진 부분을 파란색으로 표시하였다.

GetCopyP의 경우, 첫 번째 달라진 점은 8~10번 줄로, 여기서 GetCopyP는 subtree의 크기가 너무 작거나 재귀호출의 깊이가 클 경우 바로 GetCopy로 넘어간다. 이러면 subtree의 크기가 작을 때도 thread를 만들거나 thread를 과도하게 만드는 걸 막게 되며, 이를 통해 thread overhead로 인해 오히려 성능이 저하되는 걸 막을 수 있다. 다만, 위의 6000과 3이란 수는 임의의 적절한 값으로, 환경에 따라 적절한 값은 다를 수 있다.

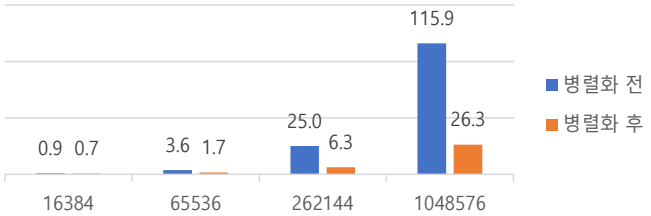
GetCopyP의 두 번째 달라진 점은 12번과 17번 줄이다. 12번 줄에서 볼 수 있듯, GetCopyP는 GetCopy와 달리, 직접 재귀호출을 수행하지 않고 새로운 thread를 만들어 이 thread가 재귀호출을 수행하게 한다. 이후, 16번 줄에 있는 재귀호출은 직접 수행한 후, 17번 줄에서 thread와 join한다.

BuildTreeP의 경우도 달라진 점들은 GetCopyP와 비슷하다. 더 자세히는 C++ 구현 예시[9]에서 확인할 수 있다.

4. 실험 결과

병렬화 전과 후의 수행시간을 8 thread 3.4GHz CPU 컴퓨터에서 실험했다. 모든 실험은 10회 반복 후 평균값을 사용했다.

4.1 Partial rebuild의 수행시간 분석



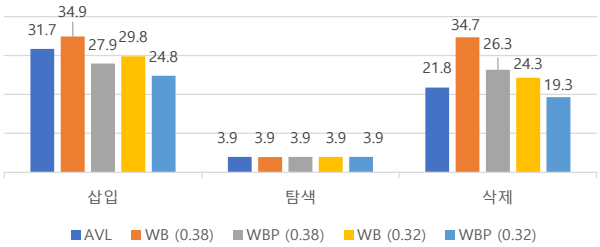
[그림1] subtree의 크기에 따른 partial rebuild의 수행시간 (ms)

[그림1]에서 볼 수 있듯, subtree의 크기가 작을 때는 병렬화의 효과가 작았지만, 크기가 커질수록 효과도 점점 커져,

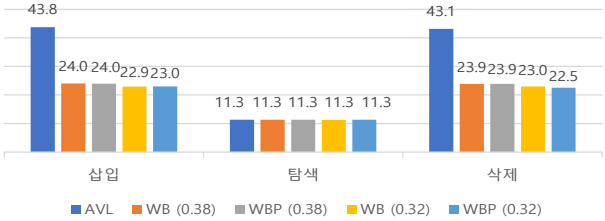
subtree의 크기가 2^{18}(=262,144)일 때는 75%, 크기가 2^{20}(=1,048,576)일 때는 77%만큼 수행시간이 감소했다.

4.2 삽입/탐색/삭제 operation의 수행시간 분석

다음으로, 삽입/탐색/삭제 operation의 수행시간을 1) AVL tree, 2) WB tree ($\alpha=0.38$), 3) 병렬화한 WB tree ($\alpha=0.38$), 4) WB tree ($\alpha=0.32$) 5) 병렬화한 WB tree ($\alpha=0.32$) (줄여서 WBP tree)로 나눠 분석했다. WB tree는 α값에 따라 성능이 변하므로, 두 가지의 α값에 대해 실험했다. 실험은 단조증가 수열 또는 무작위 수열 두 가지 경우에 대해 10M개의 삽입/탐색/삭제 operation을 수행한 후, 총 걸린 시간을 측정했다.



[그림2] 단조증가 수열, 삽입/탐색/삭제 수행시간 (s)



[그림3] 무작위 수열, 삽입/탐색/삭제 수행시간 (s)

탐색은 수행시간이 항상 같았고, 탐색보다 삽입/삭제의 수행시간이 크다는 점에서 알 수 있듯, 일반적으로 트리는 rebalancing에 많은 시간을 사용한다.

4.2.1 WB tree를 병렬화하기 전과 후

먼저, 단조증가 수열의 경우, WB tree를 병렬화하기 전과 후의 차이가 있었다. $\alpha=0.38$일 때, 삽입은 21%만큼, 삭제는 26%만큼 수행시간이 감소했고, $\alpha=0.32$의 경우도 비슷했다.

무작위 수열의 경우, 성능변화가 거의 없었다. 이는, 무작위 수열의 경우에는 subtree를 rebuild하는 경우가 적고 하더라도 크기가 작은 경우가 많은데, 이런 경우 3장에서 봤듯, GetCopyP/BuildTreeP가 바로 GetCopy/BuildTree로 넘어가도록 해 성능 저하가 일어나지 않도록 했기 때문이다.

4.2.2 AVL tree와 WB tree의 비교

단조증가 수열의 경우, AVL tree와 WB tree ($\alpha=0.38$)을 비교해보면 병렬화전에는 삽입 operation의 수행시간이 WB tree에서 더 컸다. α가 클수록 rebuild가 잦으므로, α를 감소시키면 WB tree의 삽입 성능이 빨라질 수 있지만, 이 경우 트리의 높이가 커지게 된다. 이와 달리, 병렬화를 활용할 시 트리의 높이를 희생하지 않고도 삽입 성능을 향상할 수 있었으며, 결과적으로 AVL tree보다 빨라지게 할 수 있었다.

비슷하게, AVL tree와 WB tree ($\alpha=0.32$)를 비교하면, 병렬화전에는 삭제의 수행시간이 WB tree에서 더 컸지만, 병렬화 후에는 더 작았다.

무작위 수열의 경우, AVL tree가 항상 제일 느렸다.

4.2.3 결론

단조증가 수열의 경우에는 병렬화를 사용해 WB tree의 삽입/삭제 성능을 향상할 수 있었으며, AVL tree보다 빨라지도록 할 수 있었다. 무작위 수열의 경우에는 병렬화 전과 후가 거의 같았지만, 둘 다 AVL tree보다 빨랐다. 즉, 병렬화를 활용함으로써, $\alpha=0.32$의 경우처럼 WB tree가 항상 AVL tree보다 삽입/삭제의 평균적인 성능이 빠르도록 할 수 있었으며, 그러면서도 탐색은 성능이 비슷하였다.

5. 균형이진탐색트리에 병렬화를 활용한 다른 방법들과의 비교

멀티쓰레드를 활용하여 균형이진탐색트리의 성능을 개선하기 위해, 이전의 연구들은 하나의 트리를 여러 개의 thread가 동시에 읽고 수정하도록 하였다. 이를 위해 lock을 쓰거나[3] lock-free parallelism 등을 이용했지만[8], 위 방법들은 race condition, ABA problem 등을 고려하여 매우 복잡하게 설계되어 구현이 매우 어렵고, 또, 여러 개의 operation을 동시에 하는 경우의 성능만을 개선한다.

이와 달리, 이 논문에서는 partial rebuild를 병렬화해 트리의 성능을 개선해보았다. 위 방법들처럼 여러 thread가 동시에 트리에 접근하게 해주는 게 아니고, 또, 위 방법들만큼 성능의 증가 폭이 크지 않고 단조증가 수열의 경우에서만 성능 차이가 나타나지만, 위 방법들보다는 훨씬 간단하게 성능을 개선할 수 있었으며 위에서 설명한 단점도 없었다.

6. 추가적인 활용 방안

병렬화된 partial rebuild는 위에서 설명한 방법 외에도 여러 가지로 활용할 수 있다. 예로, 탐색 중심의 워크로드 등에 앞서 이진탐색트리/균형이진탐색트리를 전체적으로 partial rebuild하여 트리의 높이를 줄이고자 할 때, 이를 더 빠르게 수행하기 위해 활용할 수 있다. 또한, tree rotation과 달리, partial rebuild는 균형이진탐색트리 뿐만 아니라 k-d tree 등에서도 사용 가능하다는 큰 장점이 있는데[7], 이때, 비슷한 방법으로 k-d tree의 성능도 개선할 수 있을 것으로 보인다.

7. 결론

Partial rebuild는 구현이 쉽고 직관적이지만, 이를 활용하는 트리는 이보다 더 유명하고 준수한 성능으로 알려진 AVL tree, Red Black tree 등에 비해 자주 사용되지 못했다.

이때, 이 논문에서는 간단한 방법으로 partial rebuild를 병렬화해 수행시간을 크게 줄이고, WB tree와 같은 트리의 성능을 제한적으로 개선해 AVL tree보다 느렸던 경우가 이후에는 빨라지게 할 수 있었다. 특히, 병렬화로 균형이진탐색트리의 성능을 개선한 다른 방법들은 동시에 여러 개의 operation을 할 때의 성능만을 개선하지만, 이 방법은 그렇지 않으며, 또, 기존의 코드를 조금만 바꾸면 되는 등, 훨씬 간단하다. 마지막으로, 병렬화된 partial rebuild를 활용할 방법들을 확인해볼 수 있었다.

참 고 문 헌

[1] G. M. Adel'son-Vel'skii, E. M. Landis, "An algorithm for organization of information," Dokl. Akad. Nauk SSSR, vol. 146, no. 2, pp. 263-266, 1962.
[2] R. Bayer, "Symmetric binary B-trees: Data structure and maintenance algorithms," Acta Informatica, vol. 1, no. 4, pp. 290-306, 1972.
[3] N. G. Bronson, J. Casper, H. Chafi, K. Olukotun, "A practical concurrent binary search tree," ACM SIGPLAN Notices, vol. 45, no. 5, pp. 257-268, 2010.
[4] I. Galperin, R. L. Rivest, "Scapegoat trees," Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms, pp.165-174, 1993.
[5] L. J. Guibas, R. Sedgewick, "A dichromatic framework for balanced trees," Proceedings of the 19th Annual IEEE Symposium on Foundations of Computer Science, pp. 8-21, 1978.
[6] J. I. Munro, T. Papadakis, and R. Sedgewick, "Deterministic Skip Lists," Proceedings of the Third Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 367-375, 1992.
[7] M. H. Overmars, "The Design of Dynamic Data Structures," 1983.
[8] J. J. Tsay, H. C. Li, "Lock-free concurrent tree structures for multiprocessor systems," Proceedings of 1994 International Conference on Parallel and Distributed Systems, pp. 544-549, 1994.
[9] "travis1829/Balanced-BST-with-parallelized-rebuild," 2021. [Online]. Available : https://github.com/travis1829/Balanced-BST-with-parallelized-rebuild. [Accessed Nov. 1, 2021].