CRA-W DREU Final Paper

MARISSA MOCENIGO
Bryn Mawr College

MENTOR: KUNAL AGRAWAL
Washington University in St. Louis

Summer 2010

Abstract

The order maintenance algorithm employs a linked list of nodes with ordered numerical keys and an implementation for insert, delete, and order functions. In some instances where the list has no space for a new node, an insertion will require the list to be reordered. In a serialized implementation, the reordering does not tax the efficiency of the overall algorithm as the other function calls run much faster and is thus faster on average; however, when the algorithm is written in parallel, this single function call can significantly slow down the algorithm's performance. We aimed to reduce the time needed to reorder the list and thus speed up the algorithm in parallel. This was achieved by implementing overlaying skip lists and by adding additional sublists. This project is significant to research with parallelized amortized algorithms. Amortized algorithms have proven difficult to effectively mutithread as the slower functions will delay all subsequent function calls, as is the case with the order maintenance algorithm; therefore, this algorithm provides some insight into the efficiency of amortized algorithms in parallel.

Introduction

Order maintenance algorithms seek to organize a list with insert, delete, and order functions. Dietz and Sleator collaborated on creating an algorithm to speed up the renumbering of a list of nodes. Their order maintenance data structure uses a circularly linked list to store the nodes and implements these three functions.[3]

Insert(x,y)
This is the most complex of the algorithm's functions. The new node, y, is to be inserted after the existing node, x. If the difference between the numerical key of x and its successive node is greater than 1, then y will be placed evenly between the two. Otherwise, the list will be renumbered.

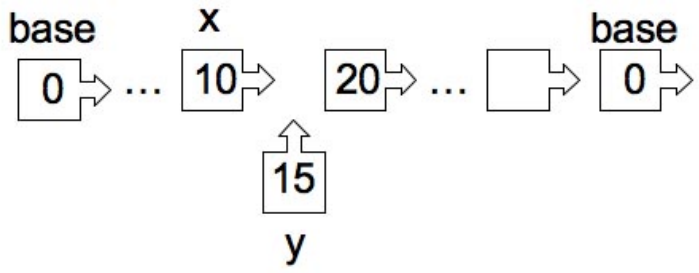

Figure 1: An Example of an Insertion (Without Reordering)

Delete(x)
The node is simply removed. Later implementations might include renumbering when a node is removed, but the original data structure does not specify this.

Order(x,y)
Based on the numerical keys of the nodes, this function will return true if x precedes y and false if x succeeds y or either node does not exist.

Each node is assigned an integer key, and the entire list is organized numerically. Starting from the node x, the function searches for the next node where the difference in keys divided by the number of nodes is less than 0.1. This value is arbitrary; the smaller the number the larger the gaps between nodes and the less reordering will be needed. If no node is found, the program will determine the smallest key value that satisfies this requirement and renumbers the entire list. The insertions can be run in parallel as long as no insertion is attempting to modify the same part of the list. If this occurs, one process must wait for the other to complete.

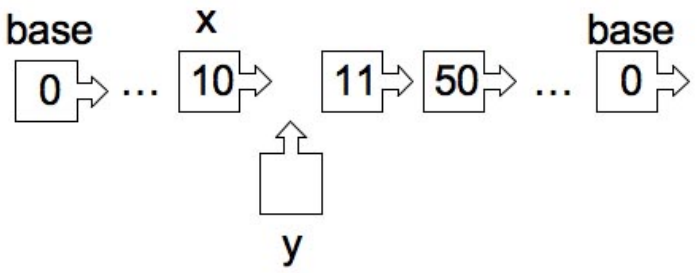

Figure 2a: An Example of an Insertion where Reordering is Needed

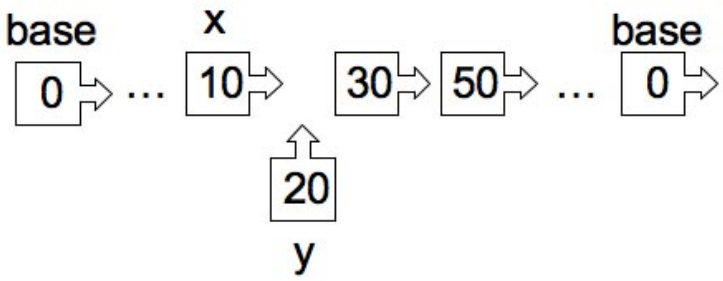

Figure 2b: Example of Insertion after List is Reordered

The insertion function of the order maintenance algorithm as defined by Dietz and Sleator runs in O(log n) time.[3] Amortized analysis takes the average time required to perform a sequence of operations[2] and thus the longer reordering insertions are averaged with the much quicker and more frequent insertions; however, this is not the case in parallel. When an amortized algorithm is modified to run in parallel, the longer function may be called simultaneous to the short functions, and thus the smaller functions will not be able to access the block of nodes being modified. Consequently, we endeavor to make the reordering faster or less frequent.

Adding Sublists

The serialized algorithm is very fast, and the only delay in parallel occurs with the longer renumbering function. The first approach we took to speed up the renumbering was to add sublists to the existing nodes. Each node has a sublist of a set size. The insertion method works similarly, but if renumbering is needed, the entire shorter list is renumbered. When the sublist nears capacity, it is split and a new node is added to the top list. While it is still possible to add nodes solely to the top list, most insertions are done within the sublists. As the sublists are small, renumbering the list is always relatively quick.

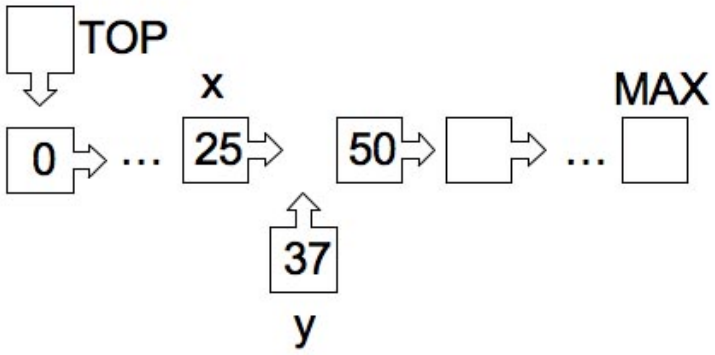

Figure 3: Sublists in the Order Maintenance Data Structure

Skip Lists

To speed up renumbering in the top list, skip lists were implemented over the existing linked list. This effectively allows us to parallelize the renumbering process. A skip list works similarly to a balanced binary tree.[2] The skip list created several layers on top of the original list with each layer containing a fraction of the nodes in the original list. The probability that a node in the previous list will appear in the next level is 50%. Consequently, the top-most list contains only a few nodes. This makes searching through the list much faster. The search function takes the value of the key and will start the search at the top of the list. It will search until it finds a node with a greater key value and return the previous node. It will continue from this node at the next level of the list until the equivalent value is found. Using the skip lists, we can estimate the number of elements in the list and approximately split the list. Then, each side of the list is renumbered.

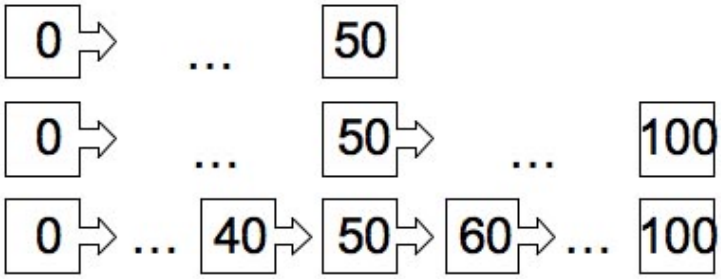

Figure 4: A Skip List

Future Work

This project has a lot of potential for future research with the efficiency of amortized time algorithms. Further adjustments can be made to the order maintenance data structure; for example, helper locks may be implemented to aid in renumbering. Helper locks benefit programs with a large critical section by reassigning blocked processors to the slower task.[1] If a short insertion process

attempted to acquire a node and found it locked, then the processor would be reassigned to help the processor that is renumbering the desired node. Consequently, this will make the long process faster and will not hinder the shorter processes once they are able to run.

Additionally, the methods used in this project can be applied to other amortized time algorithms. All amortized time algorithms face the same efficiency issues as the order maintenance algorithm. If this algorithm can run more efficiently in parallel, other amortized time algorithms can reasonably be modified for a parallel platform.

Conclusion

An efficient implementation of the order maintenance data structure supports insert, order, and delete operations in O(1) time amortized. Order queries compare the value of the node's labels. The insert function assigns a new node, y, a value larger than that of x and smaller than the subsequent values in the list. In most instances, this function is very fast; however, if no available integer label exists, some elements of the list will need to be renumbered to accommodate the insertion.

This project implemented the order maintenance data structure in parallel using Cilk. Inserts and order queries can run in parallel as long as they do not attempt to access the same parts of the list. When renumbering occurs, a large section of the list is locked down, and other inserts will be unable to run. In order to reduce the time during which the processes are blocked, we used overlaying skip lists to parallelize the renumbering operation. Amoritzed data structures present a challenge to parallelize as the longer functions tend to block out a number of faster processes; therefore, this algorithm provides some insight into the efficiency of amortized algorithms in parallel.

References

[1] Kunal Agrawal, Charles E. Leiserson, and Jim Sukha. Helper Locks for Fork-Join Parallel Programming. 2010.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, third edition, 2009.

[3] Daniel D. Sleator and Paul F. Dietz. Two Algorithms for Maintaining Order in a List. *Annual ACM Symposium on Theory of Computing*, 1988.