Parallelize DSW Algorithm

Lijuan Yin, Wenhui Li, and Dong Zhang

Abstract—In this project, we propose a parallelized DSW algorithm, useful to periodically rebalance an arbitrary unbalanced BST. The proposed algorithm splits the tree into several sub-trees and concurrently do “Tree-to-Vine” to these sub-trees using multiple threads, and finally connects them into a complete vine; It also splits the vine into several semi-equal sized subvines and do “Vine-to-Tree” to these subvines using multiple threads, and combines them into a complete balanced tree. Our extensive experiments including BST generation, “Tree-to-Vine”, and “Vine-to-Tree” parallelization comparisons show that this parallelized DSW algorithm performs much better compared to its counterpart sequential DSW algorithm.

Keywords—DSW, parallel, BST, Tree-to-Vine, Vine-to-Tree

1 INTRODUCTION

Day-Stout-Warren (DSW) algorithm is a simple algorithm designed by Quentin F. Stout and Bette Warren in their 1986 paper [1], based on work done by Colin Day in 1976 ([2]). This algorithm takes an arbitrary binary search tree and rebalances it to a balanced tree. The runtime is linear to the number of nodes and only a constant amount of space is required. In the original paper, this algorithm was only implemented by linear programming. The goal of our method is to utilize modern multithreading techniques to parallelize this method and thus to gain performance benefits.

A binary search tree (BST) is an efficient and widely used structure to maintain ordered data. The major advantage of binary search tree over other data structures is that the related operations of insertion, deletion, and searching can be very efficient. The expected runtime to such operations is $\Theta(\log(n))$. Unfortunately, this happens only when the tree is balanced.

The worst case runtime is $O(n)$ when the tree degenerates to a list.

To avoid the worst-case scenario, the original DSW algorithm was proposed to periodically rebalance the entire tree into an equivalent tree of optimal shape. Their algorithm proceeds in two phases: The binary tree is first transformed into a vine in which each parent node has only a right child and the nodes are in sorted order; The vine is then transformed into a balanced tree. In our method, each of these two phases is parallelized separately.

The paper is organized as follows. Section 2 reviews the original DSW algorithm. Section 3 covers related work, covering the evolution of DSW algorithm, concurrent methods for trees, and related balance technique in trees. Section 4 describes our methods for parallelizing DSW algorithm, including the framework, parallelization for “Tree-to-Vine” and “Vine-to-Tree” stages. Section 5 shows our extensive experiments and promising results. In Section 6, we do some further discussion for the methods and results. Section 7 draws a conclusion for our project and points out the future work directions.

2 THE ORIGINAL DSW ALGORITHM

As we have introduced in Section 1, the original DSW algorithm proceeds in two phases. A procedure “Tree-to-Vine” reconfigures the initial tree into an sorted vine, and also returns the number of nodes of the vine. Then the procedure “Vine-to-Tree” uses the vine and size information to create a balanced tree.

2.1 Tree-to-Vine

This algorithm proceeds top-down through the tree, a pointer $\text{vine}_\text{tail}$ points to the tail of
the portion known to be the initial segment of the vine, and a pointer remainder points to the root of the portion which may need additional work. If remainder points to a node with no left child, then that node can be added to the tail of the vine. Or if the remainder points to a node with a left child then a rotation is performed, as illustrated in Fig. 1.

### 2.2 Vine-to-Tree

The $k$th step of this phase is illustrated in Fig. 2. Each triangle represents a complete binary tree with $2^k - 1$ nodes, and each of the $2^j - 1$ circles represents a spine node. Each white triangle is reattached to the right side of the black spine node above and the resulting tree is attached to the left side of the white spine node below. The result is an ordered tree with $2^j - 1$ spine nodes and $2^j - 1$ complete subtrees of $2^{k+1} - 1$ nodes each. This operation is called a compression. The compression is performed repetitively until an ordered complete binary tree is produced.

### 3 Related Work

Adelson-Velskii and Landis first proposed the AVL tree [3] to dynamically balance binary search trees. Ever since then, a plethora of dynamical methods have been proposed. In 1976, A.Colin Day presented an algorithm [2] to accomplish the balancing of a binary search tree with free overhead of the AVL tree. He converted a binary search tree into a degenerate tree (a linked list through the right child pointers with null pointers for all the left child pointers), and compressed it into a balanced tree. After that, Quentin F. Stout and Bette L. Warren made a further improvement on this algorithm [1], noting that the first phase of Days original algorithm could be replaced with an alternative method very similar to the second phase of Days algorithm. To compare its different version, Timothy J. Rolfe revisited them all and presents the corresponding codes [4] acceptable for modern machines but in a sequential manner, and compared them in terms of efficiency and capacity.

As the proliferation of multiprocessor computers, data structures which support several processes are in great need. Dictionary data structures like skip-lists and AVL trees have been well studied in a concurrent manner. Carla [5] developed two solutions for concurrent search and insertion in AVL trees. One is intended to allow several reads to share nodes with a writer process, the other is based on the first one and combines parallelization techniques to implement concurrency for writes. Joaquim et al [6] got parallel dictionaries on AVL trees for insertions and deletions, by taking advantage of virtual plane waves which allows it to develop an EREW dictionary for $k$ keys with $k$ processors and time complexity $O(\log n + \log k)$. Muralidhar Medidi and Narasingh Deo also presented such a parallel dictionary on AVL trees [7] by developing an explicit processor scheduling to avoid simultaneous reads while performing $k$ searches. They also reached the same time complexity as above. Fabio [8] considered this AVL-trees based dictionary in a message-passing
distributed environment, in which a key can be operated with $O(lg^2 N)$ messages in terms of insertion, deletion, and search. Bronson et al. [9] introduced a concurrent relaxed AVL tree which is based on optimization techniques that combines software transactional memory and specific knowledge from the algorithm. Their experimental results showed the new method beat a concurrent skip list by over 30% in terms of throughput. JUAN [10] provided concurrent red black tree that uses optimized concurrency techniques and new balancing operations to scale well and support contention and claimed the proposed method was able to boost up to 14% than the best-known dictionary solutions.

The above literature shows how to achieve concurrent operations including insertion, deletion, and search, on either AVL trees or red-black trees. Also a multitude of concurrent cases have been studied in B-tree in database applications [11], [12], [13].

Both AVL trees and red-black trees are self-balancing trees, which means the tree rebalance takes place after each or certain operation. This is time-consuming if half of the tree needs to be updated. However, DSW executes its rebalance periodically and its cost can be amortized over many operations. This implies that this algorithm will “turn off” the rebalance for a short period of time in order to speed up the request processing. A plethora of literature investigates this attribute from 1978, when L. J. Guibas first mentioned this in their paper [14]. Then this technique spread over the tree family, including AVL trees [3], [15], B trees [16], [17], (a,b) trees [18], 2-3 trees [19], and red-black trees [14], [20], [21]. As most literature points out, how often should trees be rebalanced depends heavily on applications, therefore we just leave this part alone but focus on the parallelization of the DSW algorithm.

Furthermore, the emerging massively parallel Graphics Processing Units (GPU) offer an opportunity to accelerate the process. Many researches have parallelized tree searching or traversal algorithms on GPUs such as the k-D tree traversal algorithm [22], decision trees and forests [23], B+ tree search [24], and even creation and rotations for balanced trees [25]. To best our knowledge, no DSW algorithm has been analyzed and implemented in GPU environment.

4 Our Method

4.1 Some Analysis

Before implementing DSW algorithm, we need a program that can generate balanced/unbalanced binary trees which have certain number of nodes and can perform regular operations on nodes.

The first phase of DSW algorithm is to transform the binary tree into a vine. Its a top-down approach that gradually adds the remainder points to the vine_tail directly or by rotating. This process runs in $O(n)$ time. One possible way to parallelize it is to transform parts of the binary tree separately if this proves to be effective. We call this approach static multi-thread processing method. However, this approach includes a static distribution of the nodes to the threads, and not all the trees are the same, so its hard to design such distribution of the nodes. Therefore, we also proposed a dynamic multi-thread processing approach, in which we distribute the nodes to the threads dynamically: after one thread has finished an operation of one node, we assign it to another node.

The second phase of DSW is to transform the “vine” into a route balanced tree by compressing. An ordered vine with $2m - 1$ nodes needs to be compressed for $m - 1$ times to produce an ordered complete binary tree. It is obvious that we could compress concurrently in each step of compression. For this phase, we also have similar approaches as mentioned above. However, in this case, we believe it is not necessary to design dynamic threads to process the nodes, since the graph is a “vine” and we know the structure of “vine”.

Sometimes a tree is implemented as an array of records. In order to get a sorted array, we could sort the vine in a top-down manner. Again, we could sort concurrently, like the merge sort. The data structure would affect the efficiency of the algorithm, so we also want to know which data structures are better for the penalization of DSW algorithm.

The expected results would be a paralleled program that handle binary tree which is no-
nticeable faster than the version without parallelization. We want to demonstrate the results by testing for different sizes and structures for input trees and also different parallelization methods.

4.2 Parallelization of “Tree-to-Vine”

In this section, we are going to introduce three methods we tried to convert tree to vine. **Static Multithreading Method**, we pre-assign parts of the BST to each of the threads to process. This is the method we finally used and it is the only one which performs better than original DSW algorithm; We would like to introduce another two methods we tried for parallelization of “Tree-to-Vine” algorithm: **Semi-Dynamic Multithreading Method**, the threads are created and initialized once as we traverse the tree; **Dynamic Multithreading Method**, where we assign a new thread each time we do a rotation.

4.2.1 Static Multithreading Method

For the static multithreading method, each of the threads is pre-assigned one part of the BST before any of the threads start to convert the tree to vine. During this assignment, the tree is traversed using breath first searching (BFS) method. Currently, we only deal with $2^k$ threads. As shown in the Fig. 3, in the simplest case, if we only have one thread to assign, then the blue root node at level 0 and all of its children nodes will be assigned to that thread to convert; if we have two threads, the two green nodes at level 1 and their children will be assigned respectively to each of the two threads; if we have four threads, the four red nodes at level 2 will be assigned respectively and so on.

However, for an unbalanced tree, it is not always guaranteed that there are $2^d$ of nodes at level $d$ of the tree. For instance, as shown in the Fig. 3, at level 3, there are only 7 nodes instead of $2^3 = 8$ nodes. In this case, we will assign these seven purple nodes to each of seven threads and a null node to the remaining thread.

After each of the threads finished converting their part of the tree to a vine, we basically have one small tree and the leaves of this tree are vines we just converted to. Now it is very straightforward to convert them into one final vine.

4.2.2 Semi-dynamic Multithreading Method

![Fig. 4. Semi-dynamic multithreading method, assign new threads.](image)

In our second method, the threads are assigned more robustly. The first thread starts right away from the root node and the remaining threads are assigned during the traversing. The basic idea is whenever a parent node has a left child, it will continue to convert its right child tree while assigning the left child three to a new thread.

For example, suppose there are 3 threads to be assigned, blue, red and green threads. As shown in the Fig. 4, at first the blue thread starts to convert the tree from the root. It finds that the root has a left child, therefore the new red thread starts to convert the left child tree and the blue thread continues to convert the
right child tree. Then later, the blue thread finds another left child tree and assigns it to the remaining green thread.

Fig. 5. Semi-dynamic multithreading method, combine the vines.

This assigning procedure could either stop when there are no more threads to assign, or we could recycle threads. That is, once a thread finishes converting its assigned part of tree, it waits in a queue until there is a new left child tree to convert. In this mode, the procedure stops when there is no more nodes left to convert. We tested both of these two modes and found that the recycling mode performed equally if not worse than the non-recycling mode. The reason is, at least for the random trees we created, to assign some small parts of the tree to new threads is more time consuming than to let threads always finish their work. In the experiment section of our paper, we only show our results of non-recycling mode. While we assign nodes to threads, we keep records in a record list for each thread we assign. Whenever a thread finishes converting nodes to a vine, the vines head and tail are kept in its record. In this way, after all the threads finish converting, we can combine all the vines very quickly using the heads and tails in our record list. Fig. 5 shows how we will combine the vines for the tree in Fig. 4. You may notice that there are two vines, thus two records, for the blue thread. When assigning the new green thread, we found that the blue thread had move down from the threads first converting node, so we kept two records for the blue threads.

4.2.3 Dynamic Multithreading Method

For this dynamic multithreading method, in general, we do a depth-first search (DFS) for the graph and assign a thread to each of the unlocked nodes and do rotation. Finally, we process the tree using standard “Tree-to-Vine” algorithm in order to get the final vine for the input tree. We will explain this method in more details below.

Fig. 6. Dynamic multithreading method, DFS for the input unbalanced tree.

This method has a DFS framework, and a node-lock based dynamic DSW algorithm. For the input tree, we do a DFS, as show in Fig. 6 to it and simultaneously we assign a thread to the node to do the rotation as required by the standard DSW algorithm. If we examine the rotation procedure of DSW algorithm, we will see that only two nodes need to be locked (shown in Fig. 7). Therefore, when traversing the tree by DFS, for each node, we check itself and its left child. If the two nodes are not locked, we assign a thread for them and do the rotation; otherwise, we push the nodes into a stack which we will process later. If the number of threads exceeds our maximum, we just stop.
the DFS and wait for the number to decrease. After the DFS, we process the items in the stack recursively. At last, we apply a standard DSW algorithm to our output to get the final vine.

4.3 Parallelization of “Vine-to-Tree”

Our parallelization algorithm for “Vine-to-Tree” consists of three steps: 1) Identify the keynodes and their levels. 2) Split the vine into a number of subvines and do “Vine-to-Tree” to each in order to get a subtree for each of them. 3) Assemble the keynodes and subtrees to form the final balanced tree.

For the first step, the proposed algorithm finds $T-1$ nodes from the input vine as the keynodes (in which $T$ is the number of threads we want to use, it could be 2, 4, 8, ...). It finds the keynodes this way: we want to find $2^i$ keynodes from the $i$th level. For the first level, the node in the middle of the vine is selected to be the only keynode in this level. For the $i$th level, the node in each of the subvines are selected to be the keynodes in this level. As illustrated in Fig. 8, we want to use 4 threads for this example, so we need to find 3 keynodes. As described above, the first level keynode is the 8th node and the second level keynodes are the 4th and 12th node in the vine. In order to get the length of the original vine and get the desired keynodes, traversal of the vine is needed. However, it only requires one traversal of the vine, because the length of the vine could be obtained from the “Tree-to-Vine” step above. Therefore, the length of the vine is used to compute the locations of the keynodes.

For the second step, since the keynodes are already found, they are used to split the vine into several subvines. It is done by removing the pointers between the keynodes and their adjacent nodes. For $T-1$ keynodes, $T$ subvines could be found and each of them is used to get a subtree. As illustrated in Fig. 9, in this example, 4 threads are used for computing. After this step, there are 3 keynodes separated and also 4 subvines illustrated in different colors. For each of the subvines, the original DSW “Vine-to-Tree” algorithm is applied and formed a subtree for each subvine. An example is also shown in Fig. 9.

For the third step, the keynode and subtree results could be assembled into a full balanced tree as the final results. The keynodes are as-

![Fig. 8. Illustration for identifying the locations and levels of the keynodes.](image)

![Fig. 9. Illustration for splitting the vine and results of subtrees.](image)

![Fig. 10. Illustration for assembling keynodes and subtrees.](image)
sembled by relating their levels: for the first level there is only one node; for the second level, there are two nodes, so they are attached to the left and right of the first level node respectively according to their values; for the third level, there are four nodes, and so on. For the subtrees, when splitting the subvines in step 2, each subvine is assigned to a keynode as its left/right subtree, and in this step the subtrees are attached to their corresponding parent keynodes. As illustrated in Fig. 10, in this example, we first assemble the 3 keynodes by their levels, and then attach the 4 subtree results to their parent keynodes.

5 Experiments

5.1 The Efficiency of the Proposed Method

We did several experiments on a variety of trees with huge number of nodes to demonstrate the efficiency of the proposed method.

5.1.1 Experiments for Random Generated Trees

We use random generated trees to test the efficiency of the proposed method. In each of the experiments, we randomly generate a tree with a fixed number of nodes, and use the tree as an input for all algorithms (single-thread DSW and multi-thread DSW). A straightforward comparison is shown in Table 1 and Fig. 11. “Tree Size” is the number of nodes for the tree, and “Overall” is the total runtime of “Tree-to-Vine” and “Vine-to-Tree”. Column “S” shows results for single-thread DSW, and columns “2T”, “4T”, “8T”, “16T” and “32T” show results for 2-thread, 4-thread, 8-thread, 16-thread and 32-thread DSW respectively. We observe that, generally our methods outperform the single-thread DSW, and when we increase the number of nodes, the efficiency of our proposed method is much clearer.

We also show detailed results for our multi-thread implementation in Table 2. Similar to Table 1, but we show how much time does every thread take averagely for both “Tree-to-Vine” and “Vine-to-Tree” algorithms; also since “Vine-to-Tree” algorithm consists a vine traversal which could not be paralleled, we show the runtime separately. We observe that, for the same input tree, the runtime for each thread takes less and less time when we increase the number of threads, but the traversal takes almost same time for each experiments.

5.1.2 Experiments for Balanced Trees

In this section, we generate balanced trees to test the efficiency of the proposed method. In each of the experiments, we generate a balanced tree with a fixed number of nodes, and use the tree as an input for all algorithms

<table>
<thead>
<tr>
<th>Tree Size</th>
<th>S</th>
<th>2T</th>
<th>4T</th>
<th>8T</th>
<th>16T</th>
<th>32T</th>
</tr>
</thead>
<tbody>
<tr>
<td>10^6</td>
<td>0.042</td>
<td>0.051</td>
<td>0.053</td>
<td>0.049</td>
<td>0.044</td>
<td>0.043</td>
</tr>
<tr>
<td>10^7</td>
<td>0.435</td>
<td>0.369</td>
<td>0.310</td>
<td>0.293</td>
<td>0.265</td>
<td>0.286</td>
</tr>
<tr>
<td>10^8</td>
<td>5.05</td>
<td>3.77</td>
<td>2.98</td>
<td>2.18</td>
<td>2.45</td>
<td>2.18</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Overall</th>
<th>S</th>
<th>2T</th>
<th>4T</th>
<th>8T</th>
<th>16T</th>
<th>32T</th>
</tr>
</thead>
<tbody>
<tr>
<td>10^6</td>
<td>0.010</td>
<td>0.013</td>
<td>0.013</td>
<td>0.014</td>
<td>0.011</td>
<td>0.012</td>
</tr>
<tr>
<td>10^7</td>
<td>0.075</td>
<td>0.057</td>
<td>0.050</td>
<td>0.052</td>
<td>0.050</td>
<td>0.050</td>
</tr>
<tr>
<td>10^8</td>
<td>1.38</td>
<td>0.84</td>
<td>0.73</td>
<td>0.48</td>
<td>0.58</td>
<td>0.48</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TREE</th>
<th>2T</th>
<th>4T</th>
<th>8T</th>
<th>16T</th>
<th>32T</th>
</tr>
</thead>
<tbody>
<tr>
<td>10^6</td>
<td>0.025</td>
<td>0.023</td>
<td>0.024</td>
<td>0.021</td>
<td>0.018</td>
</tr>
<tr>
<td>10^7</td>
<td>0.374</td>
<td>0.298</td>
<td>0.245</td>
<td>0.225</td>
<td>0.200</td>
</tr>
<tr>
<td>10^8</td>
<td>3.66</td>
<td>2.91</td>
<td>2.24</td>
<td>1.69</td>
<td>1.83</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Overall</th>
<th>S</th>
<th>2T</th>
<th>4T</th>
<th>8T</th>
<th>16T</th>
<th>32T</th>
</tr>
</thead>
<tbody>
<tr>
<td>10^6</td>
<td>0.015</td>
<td>0.011</td>
<td>0.006</td>
<td>0.003</td>
<td>0.001</td>
<td></td>
</tr>
<tr>
<td>10^7</td>
<td>0.239</td>
<td>0.177</td>
<td>0.135</td>
<td>0.069</td>
<td>0.036</td>
<td></td>
</tr>
<tr>
<td>10^8</td>
<td>2.34</td>
<td>1.71</td>
<td>1.16</td>
<td>1.10</td>
<td>1.01</td>
<td></td>
</tr>
</tbody>
</table>

TABLE 1
Runtime Comparisons of the Proposed Method with Original DSW for Random Trees. All the results are in seconds. “S” for Single-thread DSW, “2T” for 2-thread DSW, and so on.

TABLE 2
Detailed Runtime for the Proposed Method for Random Trees. All results are in seconds. “TTV” for “Tree-to-Vine” and “VTT” for “Vine-to-Tree”; “2T” for 2-thread DSW, “4T” for 4-thread DSW, and so on.
Fig. 11. ‘Runtime Comparisons of the Proposed Method with Original DSW for Random Trees. All the results are in seconds. “S” for Single-thread DSW, “2T” for 2-thread DSW, and so on.’

(single-thread DSW and multi-thread DSW). A comparison is shown in Table 3 and Fig. 12. “Tree Size” is the number of nodes for the tree, and “Overall” is the total runtime of “Tree-to-Vine” and “Vine-to-Tree”. Column “S” shows results for single-thread DSW, and columns “2T”, “4T”, “8T”, “16T” and “32T” show results for 2-thread, 4-thread, 8-thread, 16-thread and 32-thread DSW respectively. We observe that, comparing to the random-tree results, our method perform better in “Tree-to-Vine” step. This is reasonable, because we could assign each thread about the same number of nodes to process, but for the random trees, the load for each thread is not well balanced. However, for the “Vine-to-Tree” step, for both balanced and random trees, the performances are similar, because the vine we process is similar.

<table>
<thead>
<tr>
<th>Tree Size</th>
<th>Overall</th>
<th>S</th>
<th>2T</th>
<th>4T</th>
<th>8T</th>
<th>16T</th>
<th>32T</th>
</tr>
</thead>
<tbody>
<tr>
<td>2^20 - 1</td>
<td>0.069</td>
<td>0.090</td>
<td>0.095</td>
<td>0.082</td>
<td>0.064</td>
<td>0.117</td>
<td></td>
</tr>
<tr>
<td>2^24 - 1</td>
<td>0.233</td>
<td>0.306</td>
<td>0.258</td>
<td>0.276</td>
<td>0.235</td>
<td>0.234</td>
<td></td>
</tr>
<tr>
<td>2^28 - 1</td>
<td>5.60</td>
<td>3.47</td>
<td>3.94</td>
<td>2.71</td>
<td>3.10</td>
<td>2.84</td>
<td></td>
</tr>
</tbody>
</table>

**Table 3**

Runtime Comparisons of the Proposed Method with Original DSW for Balanced Trees. All the results are in seconds. “S” for Single-thread DSW, “2T” for 2-thread DSW, and so on.

<table>
<thead>
<tr>
<th>Tree Size</th>
<th>TTV (Each Thread)</th>
<th>Overall</th>
</tr>
</thead>
<tbody>
<tr>
<td>2^20 - 1</td>
<td>0.012</td>
<td>0.022</td>
</tr>
<tr>
<td>2^24 - 1</td>
<td>0.048</td>
<td>0.048</td>
</tr>
<tr>
<td>2^28 - 1</td>
<td>1.03</td>
<td>0.55</td>
</tr>
</tbody>
</table>

**Table 4**

Detailed Runtime for the Proposed Method for Balanced Trees. All results are in seconds. “TTV” for “Tree-to-Vine” and “VTT” for “Vine-to-Tree”; “2T” for 2-thread DSW, “4T” for 4-thread DSW, and so on.

### 5.2 Comparison between Different “Tree-to-Vine” Methods

Since it took us a lot of time to find the proposed well-performing method, and during the investigation, we have devised some other methods for “Tree-to-Vine” step of DSW which do not work as well as the proposed method, but we still want to show the results here in Table 5.
Fig. 12. 'Runtime Comparisons of the Proposed Method with Original DSW for Balanced Trees. All the results are in seconds. “S” for Single-thread DSW, “2T” for 2-thread DSW, and so on.'

<table>
<thead>
<tr>
<th>Tree Size</th>
<th>S</th>
<th>SN(2)</th>
<th>SN(4)</th>
<th>D(2)</th>
<th>D(5)</th>
<th>D(10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^4$</td>
<td>0.000</td>
<td>0.002</td>
<td>0.004</td>
<td>4.809</td>
<td>4.625</td>
<td>5.688</td>
</tr>
<tr>
<td>$10^5$</td>
<td>0.002</td>
<td>0.008</td>
<td>0.019</td>
<td>124.4</td>
<td>123.3</td>
<td>115.7</td>
</tr>
<tr>
<td>$10^6$</td>
<td>0.002</td>
<td>0.006</td>
<td>0.014</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

TABLE 5
Comparison between several “Tree-to-Vine” methods we used to propose. “SN” is the Semi-Dynamic method and “D” is the dynamic method.

6 Discussion
When it comes to parallelize DSW algorithm, it is true that the devil is in the detail. A test benchmark containing a variety of testing cases is expected, how to find a balanced point for choosing the number of threads for every environment requires to be unique but manageable. We would like to discuss some problems in our implementation and highlight the right way to realize the algorithm.

It is quite simple to build any arbitrary BST tree in a sequential manner which is recursively calling the function itself until it finds its right location in the tree. When the number of the tree grow to hundreds of millions, it takes a plethora of time such as tens or even hundreds of seconds to build this tree, which serves as an input for our experiment. At least two methods are popular to accelerate this. One approach is to create a dedicated file for each test case, which seems to be acceptable when the node number is small, but prohibitive to access if the node number is large. The other method is to build this tree with the help of multiple cores and multiple threads. However, it turns out to be more time-consuming. The main reason is that the overhead introduced by the multiple threads takes time and also the recursive function is not suitable for parallelism if it has a larger depth. This can be tackled by using iterative function.

At the first sight, for the “Tree-to-Vine” algorithm, the dynamic processing method should work better since it is very flexible for distributing the nodes to the threads. However, in practice, creating threads and allocating resources for them also takes some time. Therefore, as we can observe from the experimental results, the dynamic processing method does not work better than other paralleling approaches.

In the implementation, we tried to use both “PThread” and “Window Multi-thread API (WinAPI)”, and we found that on Windows platform, “WinAPI” performs better. The reason is that, “WinAPI” has more efficient interface with Windows platform and it could allocate the sources for creating the threads, thus takes less time.

Generally, after investigating and implementing our methods, we found that the overhead of creating threads should be considered seriously for multi-core programming. Sometimes, it takes much longer time than we expected.

7 Conclusion and Future Work
The proposed parallelization of DSW algorithm runs significantly faster than single-thread DSW, both for the “Tree-to-Vine” and “Vine-to-Tree” step. On a 8-core computer, the runtime does not drop a lot by using more than 8 threads. The runtime of each thread
keeps decreasing when we increase the number of threads, however, the total runtime of all threads does not decrease if using more than 8 threads. Our “Tree-to-Vine” method works better on balanced trees, however, “Vine-to-Tree” method works similar well for both balanced and random trees.

One potential area for improvement over the implementation presented here would be to use of lock-free and wait-free synchronization techniques [26], [27], [28], [29]. The use and performance of the presented algorithm in a task-parallel environment [30] will also be explore in future work.

There are also some considerations about future research work that the mechanism to trigger the rebalance operation is more than welcome for its application, which may involve with the recording of its fundamental operations such as insertion or deletion. Another interesting research direction is to make this algorithm scalable and distributable for large-scale distributed datacenters or clusters, which aims at providing faster and cost-effective solutions.

ACKNOWLEDGMENT

The authors would like to thank Professor Damian Dechev for the great lectures he gave and his help to our projects.

REFERENCES


