TOWARDS AN EFFICIENT PARALLEL BINARY SEARCH TREE USING LOCK-FREE INSERTION

A.M. Dogar and M.A. Khan

*COMSATS Institute of Information Technology, Sahiwal, Pakistan.
Bahauddin Zakariya University, Multan, Pakistan.
Corresponding author's email: manan@ciitsahiwal.edu.pk

ABSTRACT: Binary Search Tree (BST) was widely used in a large number of applications in order to search data in an efficient manner. On the modern multi-core systems, the implementation of parallel Binary Search Tree (BST) was unable to achieve maximum performance due to a high cost of locking mechanism, which was inevitable since the deployment of multiple parallel threads require locks to be implemented. This paper proposed a parallel lock-free BST which allowed for parallel insertion of data. Our proposed approach used atomic instructions like Compare, Swap, Fetch and Add to implement mutual exclusion and lock avoidance. The proposed implementation outperformed the sequential and the existing lock-based parallel binary search tree implementation. The proposed implementation of the parallel BST was evaluated on different platforms like Intel Xeon and Intel Core i5 processor based systems. The proposed approach achieved up to 12% performance improvement over the parallel lock-based implementation.

Keywords: Atomic Instructions, Binary Search Tree, Code Optimization, Lock-free, Tree Insertion.

(Received 16-02-2017 Accepted 21-12-2017)

INTRODUCTION

Binary Search Tree (BST) is a core data structure which is used for the management of ordered data and its manipulation operations (Cormen and Russel, 2009). It uses decrease and conquer technique for efficient searching because the complexity of search operation of BSTs is *log(n)* (Adamchik, 2016; Cormen, 2009; Furajh, 2000) . That's why it is widely used in DNA and Protein sequence analysis in bioinformatics where the searching of data plays a pivotal role.

To achieve the maximum performance on multi-core processors, parallel algorithms have become inevitable to make each core execute different instructions simultaneously. There are, however, complexities using a shared memory implementation, as race for the possession of resources leads to deadlocks and starvation of resources. To avoid these issues, parallel algorithms with *obstruction-free*, *lock-free* and *wait-free* properties are being designed and implemented (Bahra, 2013).

Parallel BSTs are implemented on both distributed and shared memory architectures. The initial concurrent implementation of BSTs for shared memory architectures using multiple threads was made in 1995 (Solworth, 1995). Similarly, a parallel implementation of a BST (Feng, 2011) enhances search efficiency for DNA sequences. The insertion operation of the parallel BST incorporates locks which ultimately becomes a bottleneck on modern multi-core systems.

In contrast to these approaches, an efficient implementation of insertion operation for the parallel BSTs using threads is proposed while avoiding the above-mentioned locking mechanism. This is accomplished by incorporating the *Compare and Swap* (CAS) and *Fetch and Add* (FAA) atomic instructions which can perform two or more operations simultaneously without using any locking mechanism.

In general, a BST contains a single root node, which contains links to left and right sub-trees. After having created the root node, a lock based parallel implementation invokes multiple threads for the creation of sub-trees. Due to multiple threads, the lock-based implementation uses a locking mechanism to ensure mutual exclusion while inserting data in sub-trees (Arbel, 2014). Implementation approach for this paper avoids locking mechanism for insertion that ultimately reduces the cost of computation. Although the lock-based algorithms are more efficient as compared to sequential algorithms, but the locking mechanism is very time consuming. A lock free BST creation is actually a refined form of its lock-based counterpart which avoids locking mechanism by using atomic instructions.

In parallel algorithms, an algorithm is *obstruction-free* algorithm if no process is suspended or blocked due to any type of obstruction. An algorithm is *wait-free* if a process can perform finite number of operations before its completion, whereas, an algorithm is called *lock-free* if there will be no starvation of the resources (Obstruction Freedom).

The hardware architecture of new computers supports atomic instructions which are used to implement lock free algorithms. The atomic instructions execute more than one basic operation (read, write, compare and swap) as a unit while maintaining the mutual exclusion and controlling the interrupt mechanism (Scogland, 2015).

- ***__sync_val_compare_and_swap***

The instruction *__sync_val_compare_swap* (David, 2013) executes two operations, compare and swap, in a single throw. It's algorithm takes the new input value, compares it with existing value, and if it matches then there was no change, otherwise it swaps the existing value with the new value (Molka, 2014).

- ***__sync_fetch_and_add***

The instruction *__sync_fetch_and_add*, is used to access the operand (data element to be added), and then to add the fetched value into the existing value.

(Howley *et al*, 2012) suggests a concurrent BST algorithm that builds the tree using single-word reads, writes, and compare-and-swap. The algorithms given by (Feng, 2011; Bronson, 2010; Bender, 2005) for BSTs also supports parallel processing. These algorithms are much faster than previous non-parallel BSTs. Parallel operations are done by constructing parallel sub-trees which were managed by "mutexes". A shared memory based asynchronous system for different tree operations using single-word compare-and-swap operations are suggested by (Brown, 2014). This linear implementation makes the tree operations like delete, insert and update to work on different portions of the tree. An optimized approach using the concept of transactional memory for efficient Adelson-Velskii and Landis (AVL) tree operations is given by (Bahra, 2013). The AVL trees are a refined form of BSTs with the characteristic of height balancing. Another approach proposed by (Kung *et al,* 1980) supports multiple parallel processes which can execute the operations like search, insert, delete and rotation on a tree. Using lazy splaying, an implementation of parallel search trees is suggested by (Afek, 2016). The suggested approach makes changes in those nodes which are most commonly accessed without creating any type of bottleneck at the root level. Similarly, another approach of concurrent BST suggested by (Arbel, 2014) uses Read-Copy-Update (RCU) based synchronization mechanism. The approach however requires fine-grain locks in order to synchronize concurrent updates.

In a study (Natarajan, 2014) introduces a lock-free algorithm, which works by marking *edges* instead of nodes. As compared to other lock-free algorithms, their modification approach for a BST operates on a small portion of the tree at an instance. Consequently, the suggested approach is shown to work with reduced number of conflicts.

A lock-free algorithm for parallel operations on a BST using asynchronous shared memory is proposed by (Ramachandran, 2015). Their algorithm combines the features of two different approaches for read and write dominated workloads. Similarly, an algorithm which changes its contention according to read-write load is proposed by (Chatterjee, 2014), which uses single-word CAS operation. In case of read-heavy, concurrent Remove operations are avoided during traversal, and adapted to interval contention. For the write-heavy situations, the algorithm allows for the concurrent Remove.

In contrast to these approaches, the suggested approach makes use of hardware primitives which supports atomicity of multiple operations. By incorporating the architecture level instructions, the approach results in efficient insertion of data while avoiding any complex operations that could otherwise degrade the performance.

MATERIAL AND METHODS

The proposed algorithm created a BST which was used to match the substrings in a large string. For creating the tree, the approach given by (Feng, 2011) was adapted to enhance its performance through lock-free insertion. Initially, the input data was divided into parts (substrings of fixed length) equal to the number of threads created for parallel processing. Each thread then scanned its allocated portion of input data to generate a sub-tree. The generated sub-tree could be traversed in parallel to match for similarity of another input substring.

```
1.  TreeST   -- Structure for tree
2.     struct TreeST *left // Left sub-tree
3.     struct TreeST *right // Right sub-tree
4.     char key[len] // len represents length of substring key
5.     int count  // To count repetitions of the substring
6.  NodesLink   -- Structure for node link
7.     struct NodesLinkST *next
8.     struct TreeST *node
9.  TreeST Tree -- Declaration of Tree
10. Tree *root = NULL -- Declaration and initialization of root
11. MAIN Function
```

```
12.     Read entire input file in string str and divide it into
        chunks according to THREADCOUNT
13.     //creating the number of threads defined by user
14.     For i = 0 to THREADCOUNT-1
15.         Create and execute thread to work on chunk i
16.     End For
17. End MAIN
```

Figure-1: Core structures and the *MAIN* function used for parallel BST

```
1. THREADCODE Function
2.      //Let TID be the thread ID
3.      For each substring str of size s in a chunk
4.          Create node n by allocating memory
5.          n->left = NULL
6.          n->right = NULL
7.          strcpy (n->key, str)
8.          CALL INSERT (root, s, TID)
9.      End For
10. End THREADCODE
```

Figure-2: *THREADCODE* function for creating new nodes

```
1. INSERT Function (Tree * n, int Len, int TID)
2.      //n represents the node, Len represents the length of the string n->key
3.      //TID represents the Thread ID,  str represents the string to be inserted
4.      // Let root be pointer to the first node, and let flag represent the result of comparison
5.      Tree  *tr, *m
6.      tr = root
7.      m=NULL
8.      While (true)
9.              If ( tr == NULL) Then
10.                     If  (m == NULL) Then // Root node
11.                             If  (__sync_val_compare_and_swap (&root, 0, n)) Then
12.                                     Add link to node n for thread TID
13.                                     return
14.                             Else
15.                                     tr = root
16.                             End If
17.                     Else If (flag < 0) Then // Left Child
18.                              If (__sync_val_compare_and_swap(&m->left, 0, n)) Then
19.                                     Add link to node n for thread TID
20.                                     return
21.                             Else
22.                                     __sync_val_compare_and_swap(&tr, tr, m->left)
23.                             End If
24.                     Else // Right Child
25.                             If (__sync_val_compare_and_swap(&m->right, 0, n)) Then
26.                                     Add link to node n for thread TID
27.                                     return
28.                             Else
29.                                     __sync_val_compare_and_swap(&tr, tr, m->right)
30.                             End If
31.                     End If
32.             End If
33.             m = tr
34.             flag = memcmp( n->key, tr->key, Len )
35.             If ( flag == 0 ) Then
```

```
36.                     __sync_fetch_and_add(&(tr->count), 1);
37.                     return
38.             End If
39.             // Now move tree pointer to left or right
40.             If ( flag < 0 ) Then
41.                     tr = tr->left
42.             Else
43.                     tr = tr->right
44.             End If
45.     End While
46. End INSERT
```

Figure-3: Pseudo-code of the *INSERT* function used to insert data in the tree.

The basic structures and the *MAIN* functions used for the creation of BSTs are given in Fig-1. From lines 1 to 5, a *TreeST* data structure is defined, which included left & right child, key and a count variable for every node. The *key* is the substring saved in every node and the *count* variable is used to count the total occurrences of same substring in a string. The *NodesLink* structure from lines 6 to 8 are used to traverse the tree by using pointers to the next node. At line number 9, *Tree* is declared as the object of *TreeST* structure and at line number 10, the *root* of the tree was declared. The pseudo code for the *MAIN* function was given in lines 11 to 17. The *MAIN* function was used to divide data into chunks and invoke the threads to process those chunks. At line 12, the entire input data (file) was read and then divided into chunks. From lines 14 to 16, the threads were created and set to work on chunks as formed in the previous step.

The pseudo code for the *THREADCODE* function is given in Fig-2. Subsequent for the creation of the root node by *MAIN* function, the lines 3 to 6 of the *THREADCODE* function created new nodes. At line 7, the input substring was copied into the newly created node. At line 8, the *INSERT* function was called to place newly created node in the tree.

The pseudo code for the *INSERT* function is given in Fig-3. The function being called by threads simultaneously, was used to perform comparisons and insert nodes in the tree. The parameters passed to *INSERT* function included the pointer to the newly created node, length of the key of the newly created node and the Thread Id which was going to perform insertion. The pointers *tr* and *m* were initialized to values *root* and *NULL*, respectively, in lines 5 to 7. For traversing the tree, a loop was set to start at line 8. At line 9, the *INSERT* function checked whether the tree was empty and subsequently set the root to the newly created node in lines 10 to 13. If the tree was not empty, the code then added links to left child using lines 17 to 23 or links to right child using lines 24 to 31. The left or the right child was decided through comparison operation which was performed at line 34. If the key of the newly created node matched with that of the current node, the occurrence was incremented through lines 35 to 38. If the key value of the newly created node was less than that of the current node, the pointer was moved to, otherwise the pointer was moved to right through lines 40 to 44. The pointer and the flag values were subsequently used in next iteration of the loop during traversal of the tree. Since the algorithm was specially designed for parallel insertions, at start it checks which thread was going to insert new node and in which particular portion of memory. Each thread called *INSERT* function without acquiring any locks, whose implementation was made to work through atomic instructions.

Experimental Setup: The proposed algorithm was implemented on the architectures having the Intel Xeon E-5520 (with 8 Cores), and the Intel Core i5 processors (with 4 Cores). For evaluation of the algorithms, the string based searching was performed in Protein sequence data files. The Protein sequences consist of specific characters (20 Amino Acid characters). These amino acids or base elements repeat in a specific pattern to form a complete Protein sequence. As sample data, Protein sequences were downloaded from the National Center for Biotechnology Information website (NCBI, 2016). The input data file contained 196101 protein sequences having a total of 66293940 amino acid characters.

The experimentation was performed to evaluate sequential, lock-based parallel reported by (Feng, 2011) and our lock-free parallel implementation to insert substrings of size 2, 4 and 8 characters in the BST, using 2, 4 and 8 (POSIX) threads reported by (Barney, 2016). The experimental setup and other configurations are given in Table 1.

Table-1: Experimental Setup Details

Processor	Compiler and Operating System
Intel Xeon-E5520, Cache 256 MB (With 8 Cores), 8 GB RAM	GCC v 4.1, Fedora Core 10
Intel Core i5 2.2 GHz, Cache 3 MB (With 4 Cores), 4GB RAM	GCC v 4.1, Ubuntu 12

RESULTS AND DISCUSSION

The performance of the sequential, lock-based parallel, and the lock-free parallel implementations with input string size of 2 characters on the Intel Xeon and Intel Core i5 based systems are given in Table-2. As shown in the results, our lock-free algorithm outperforms the lock-based parallel and sequential implementations in all configurations, using 2, 4, and 8 threads.

Table-2: Results on Intel Xeon & Core i5 with input string size of 2 characters

Algorithm	Execution Time in Seconds (2 Threads)		Execution Time in Seconds (4 Threads)		Execution Time in Seconds (8 Threads)	
	Intel Xeon	Intel Core i5	Intel Xeon	Intel Core i5	Intel Xeon	Intel Core i5
Sequential	10904	11700	10904	11700	10904	11700
Lock-based Parallel	8105	8700	8290	6312	8508	5524
Lock-free Parallel	7600	7820	7705	5000	7880	4670

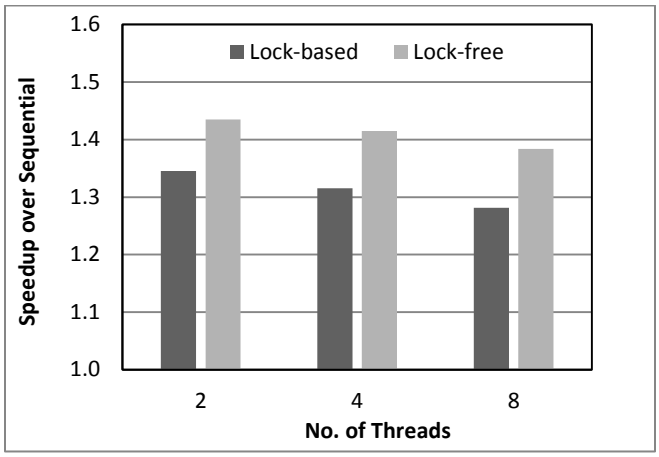

Fig. A

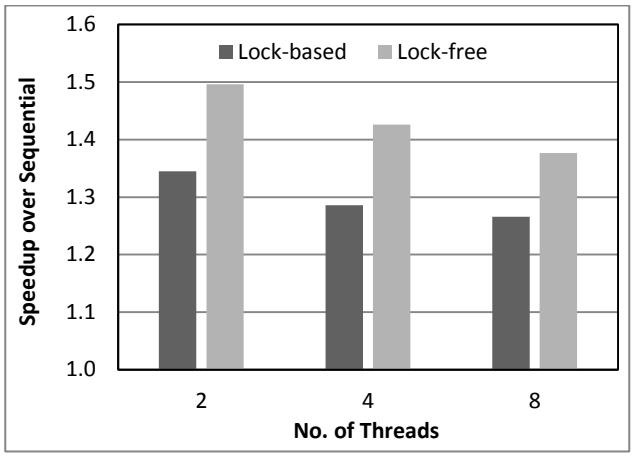

Fig. B

Figure-A: Performance analysis of lock-based and lock-free algorithms on Intel Xeon based system with input size of 2 characters, Fig. B: Performance analysis of lock-based and lock-free algorithms on Intel Core i5 based system with input size of 2 characters

On the Intel Xeon based system, the speedup results of the lock-based and the suggested lock-free implementations over the sequential implementation are given in Fig-4. The lock-free algorithm outperformed the lock-based algorithm which also exploited parallelism. As shown in the figure, the speedup obtained by the lock-free algorithm ranged from 1.37 to 1.44. On average, the lock-free algorithm had speedup of 1.41, whereas the lock-based algorithm had average speedup of 1.31. Consequently, our proposed lock-free implementation performed 10% better than the lock-based parallel implementation. Similarly, for the Intel Core i5 based system, the speedup results of the lock-based and the suggested lock-free implementations over the sequential implementation are given in Fig. 5. As shown in the figure, the speedup obtained by the lock-free algorithm ranged from 1.36 to 1.50. On an average, the lock-free algorithm has speedup of 1.43, whereas the lock-based algorithm had average speedup of 1.30. Consequently, the lock-free implementation performed 13% better than the lock-based parallel implementation.

With input string size of 4 characters, the performance of the sequential, lock-based parallel, and lock-free parallel implementations on the Intel Xeon and Intel Core i5 based systems is given in Table-3. As shown in the results, the lock-free algorithm outperforms the lock-based parallel and sequential implementations in all configurations, using 2, 4, and 8 threads.

On the Intel Xeon based system, the speedup results of the lock-based and the suggested lock-free

implementations over the sequential implementation. While using the input string of 4 characters are given in Fig. 6. As shown in the figure, the speedup obtained by the lock-free algorithm ranged from 1.27 to 1.36. On an average, the lock-free algorithm had speedup of 1.30, whereas the lock-based algorithm had average speedup of 1.19. Consequently, the lock-free implementation performed 11% better than the lock-based parallel implementation. Similarly, on the Intel Core i5 based system, the speedup results of the lock-based and the lock-free implementations over the sequential implementation are given in Fig. 7. As shown in the figure, the speedup obtained by the lock-free algorithm ranged from 1.3 to 1.38. On an average, the lock-free algorithm has speedup of 1.34, whereas the lock-based algorithm had average speedup of 1.22. Consequently, the lock-free implementation performed 12% better than the lock-based parallel implementation.

Table-3: Results on Intel Xeon and Core i5 with input string size of 4 characters

Algorithm	Execution Time in Seconds (2 Threads)		Execution Time in Seconds (4 Threads)		Execution Time in Seconds (8 Threads)	
	Intel Xeon	Intel Core i5	Intel Xeon	Intel Core i5	Intel Xeon	Intel Core i5
Sequential	12370	13890	12370	13890	12370	13890
Lock-based Parallel	10100	11200	10459	11310	10560	11512
Lock-free Parallel	9100	10010	9670	10400	9725	10650

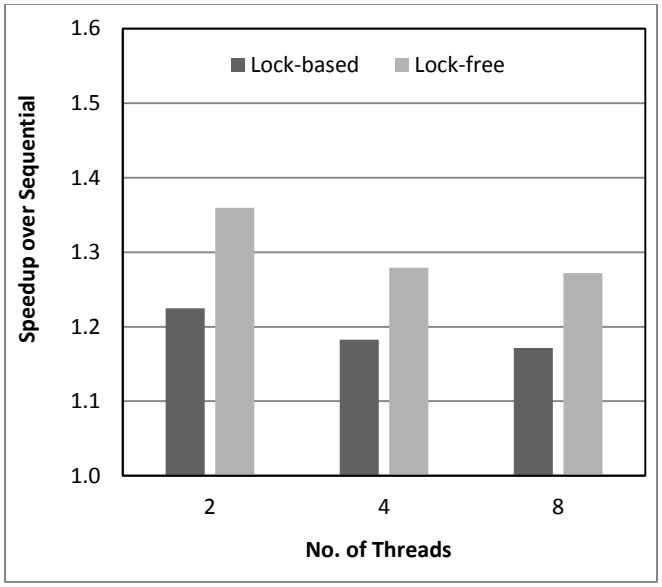

Fig. A

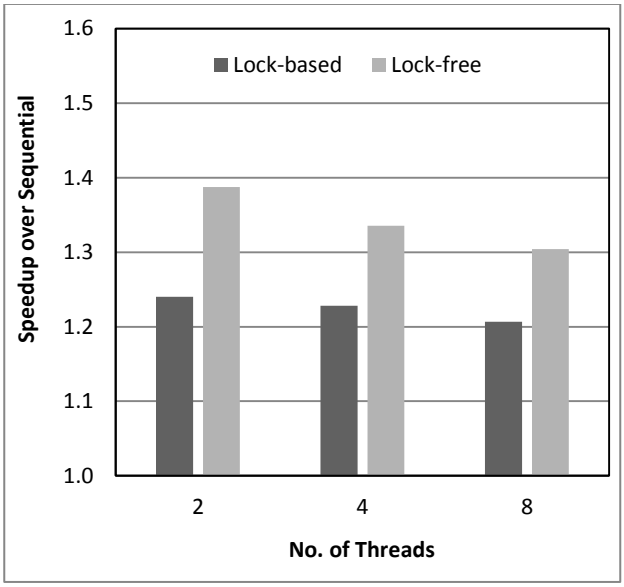

Fig. B

Fig. A: Performance analysis of lock-based and lock-free algorithms on Intel Xeon based system with input size of 4 characters, Fig. B: Performance analysis of lock-based and lock-free algorithms on Intel Core i5 based system with input size of 4 characters

Table-4: Results on Intel Xeon & Core i5 with input string size of 8 characters

Algorithm	Execution Time in Seconds (2 Threads)		Execution Time in Seconds (4 Threads)		Execution Time in Seconds (8 Threads)	
	Intel Xeon	Intel Core i5	Intel Xeon	Intel Core i5	Intel Xeon	Intel Core i5
Sequential	13670	15100	13670	15100	13670	15100
Lock-based Parallel	11010	11600	11300	11703	11590	11935
Lock-free Parallel	10100	10400	10330	10545	10720	10890

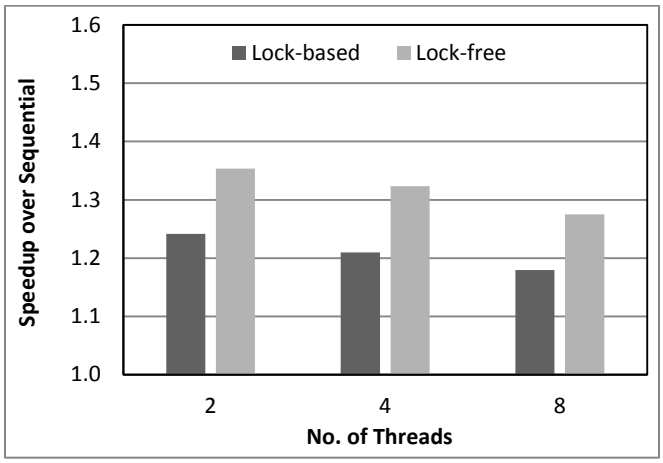

Fig. A

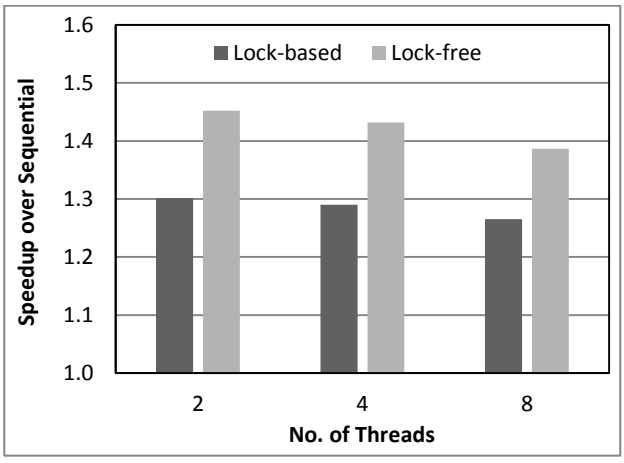

Fig. A

Figure-A: Performance analysis of lock-based and lock-free algorithms on Intel Xeon based system with input size of 8 characters, Fig. B: Performance analysis of lock-based and lock-free algorithms on Intel Core i5 based system with input size of 8 characters

The execution performance of the sequential, lock-based parallel, and the lock-free parallel implementations with input string size of 8 characters on the Intel Xeon based system is given in Table-4. Similar to the results for strings of sizes 2 and 4, the lock-free algorithm continued to outperform the lock-based parallel and sequential implementations in all configurations using 2, 4, and 8 threads.

The speedup results of the lock-based and the lock-free implementations over the sequential implementation using the Intel Xeon based system are given in Fig. 8. As shown in the figure, the speedup obtained by the lock-free algorithm ranged from 1.27 to 1.35. On average, the lock-free algorithm has speedup of 1.32, whereas the lock-based algorithm had an average speedup of 1.21. Consequently, the proposed lock-free implementation performed 11% better than the lock-based parallel implementation. Similarly, on the Intel Core i5 based system, the speedup results of the lock-based and the lock-free implementations over the sequential implementation are given in Fig. 9. As shown in the figure, the speedup obtained by the lock-free algorithm ranged from 1.38 to 1.45. On an average, the lock-free algorithm had speedup of 1.42, whereas the lock-based algorithm had average speedup of 1.29. Consequently, the lock-free implementation performed 13% better than lock-based parallel implementation.

It was evident from the results that the lock-free implementation outperformed the parallel lock-based and sequential implementations. Overall, using 2 characters as input, on the Intel Xeon and the Intel Core i5 based systems, the average speedup attained by the lock-free parallel implementations was 1.42. This was better than the lock-based system which was able to attain overall average speedup of 1.31. Consequently, there was 11% improvement in execution speed obtained by the proposed lock-free algorithm. In the second scenario with 4 characters as input on the Intel Xeon and the Intel Core i5 based systems, the average speedup attained by the lock-free parallel implementations was 1.32. This was better than the lock-based system which was able to attain overall average speedup of 1.21. Consequently, there was again 11% improvement in execution speed obtained by the proposed lock-free algorithm. Similarly, using 8 characters as input on the Intel Xeon and the Intel Core i5 based systems, the average speedup attained by the lock-free parallel implementations was 1.37, which was better than the lock-based implementation. Overall, there was 12% performance gain in execution speed obtained by the proposed lock-free algorithm. The consistent performance improvement over the lock-based parallel implementation showed the significance of the lock-free approach suggested in the present algorithm.

Conclusion: The proposed lock-free implementation invokes multiple threads with each thread operating on a chunk of input data. The results show that the proposed lock-free implementation outperforms lock-based parallel implementations. In future, the proposed approach will be enhanced for distributed memory systems having heterogeneous architectures.

REFERENCES

Adamchik, V. (2016). Binary Trees, Available from: https://www.cs.cmu.edu/ ~adamchik/15-121/lectures/Trees/trees.html.

Afek, Y., B. Korenfeld, and A. Morrwason (2016). Concurrent search tree by lazy splaying, Available

from:http://www.cs.tau.ac.il/~afek/LazySplaying.pdf.

Arbel, M., and H. Attiya (2014). Concurrent updates with RCU: search tree as an example, In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, Pages 196-205, ACM New York, U.S.A.

Bahra, A. (2013). *Non-blocking algorithms and scalable multicore programming*, Communications of the ACM, Volume 56, issue 7, 50-61, ACM, New York, U.S.A.

Barney, B. (2016). POSIX threads programming. Available from: https://computing.llnl.gov/tutorials/pthreads/.

Bronson, N. G., J. Casper, H. Chafi, and K. Olukotun (2010). A practical concurrent binary search tree. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Pages 257–268, ACM, New York, U.S.A.

Bender, M. A., J. T. Fineman, S. Gilbert, and B. C. Kuszmaul (2005). Concurrent cache-oblivious B-trees. In *Proceedings of the 17th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, Pages 228–237, ACM, New York, U.S.A.

Brown, T., Trevor, F. Ellen, and E. Ruppert (2014). 'A general technique for non-blocking trees'. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '14)*, 329-342, ACM New York, U.S.A.

Chatterjee, B. , N. Nguyen, and P. Tsigas (2014). Efficient lock-free binary search trees, *Proceedings of the ACM symposium on Principles of distributed computing*, Paris, France.

Cormen, T. H., C. E. Lewaserson, R. L. Rivest, and C. Stein (2009). *Introduction to algorithms*, 3rd Ed., The MIT Press, Cambridge, U.S.A.

David, T., R. Guerraoui, and V. Trigonakis (2013). Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. In ACM Symp. on Op. Sys. Prin., SOSP '13, pages 33–48.

Feng, J., D. Q. Naiman, and B. Cooper (2011). *A parallelized binary search tree*. JITSE, Vol. I, Issue 1.

Furajh, I., S. Aluru, S. Goil, and S. Ranka (2000). *Parallel construction of multidimensional binary search trees*, IEEE Transactions on Parallel and Distributed Systems, 11(2):136-148.

Herlihy, M., V. Luchangco, and M. Moir (2003). Obstruction-free synchronization: double-ended queues as an example. In *Proceedings of the 23rd IEEE ICDCS*, pages 522–529, IEEE Computer Society, Washington, U.S.A.

Herlihy, M., Wait-free synchronization (1991). *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1): 124–149, ACM, U.S.A.

Howley, S. V., and J. Jones (2012). A non-blocking internal binary search tree, In *Proceedings of the 24th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'12)*, 161-171, ACM, New York, U.S.A.

Kung, H. T., and P. L. Lehman (1980). *Concurrent manipulation of binary search trees*, ACM Transactions on Database Systems, Vol. 5, No. 3, September 1980, Pg. 354-382, ACM New York, U.S.A.

Mitchell, M., J. Oldham, and A. Samuel (2001). *Advanced Linux programming*. 1st Ed., New Riders Publishing, Indianapolis, U.S.A.

Molka, D., D. Hackenberg, and R. Schone (2014). Main Memory and Cache Performance of Intel Sandy Bridge and AMD Bulldozer. In Work. on Mem. Syst. Perf. And Corr., MSPC '14, pages 4:1–4:10.

Natarajan, A., and N. Mittal (2104). Fast concurrent lock-free binary search trees, *Proceedings of the 19th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, Orlando, Florida, U.S.A.

NCBI (2016). *Protein sequences*, National Center for Biotechnology Information, Available from: http://www.ncbi.nlm.nih.gov/.

Obstruction Freedom (2016), Available at http://www.cs.yale.edu/homes/aspnes/pinewiki/ObstructionFreedom.html

Ramachandran, A., and N. Mittal (2015), *A fast lock-free internal binary search tree*, Proceedings of the 2015 International Conference on Distributed Computing and Networking Article No. 37, Goa, India

Russell, S. J., and P. Norvig (2009). *Artificial Intelligence, A Modern Approach*, 3rd Ed., Prentice Hall, Inc., U.S.A.

Scogland, T. R. W., and W. Feng (2015). Design and Evaluation of Scalable Concurrent Queues for Many-Core Architectures. In ACM/SPEC International Conference on Performance Engineering (ICPE).

Solworth, J. A., and B. Reagan (1994). Arbitrary order operations on trees. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing, 21-36*, Springer-Verlag, London, U.K.

Solworth, J. A., and B. Reagan (1995). Parallelizing tree algorithms: Overhead vs. Parallelism. In *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing, 440-454*, Springer-Verlag, London, U.K.