University of Colorado
Boulder

ECEN 5033 Concurrent Programming

FInal Project Report

Hardik Senjaliya

1 Algorithm: Concurrent Tree

As recommended, the concurrent tree is implemented using hand-over-hand locking mechanism. To achieve the same each node in the tree has a lock.
Here's the comparison between a standard BST node structure and a concurrent BST using hand-over-hand locking.

- Standard BST Node structure

```
typedef struct Node{
        int value;
        int key;
        struct Node *left;
        struct Node *right;
}bst_node;
```

- Concurrent BST Node structure

```
typedef struct Node{
        int value;
        int key;
        struct Node *left;
        struct Node *right;
        pthread_mutex_t lock;
}bst_node;
```

As you can see above, the difference is a Mutex Lock for each node. The purpose of using a mutex for each node to achieve the required result using hand-over-hand locking. Thus, anyone who wants to modify any field of the node, must acquire a lock first and release it after modifying.

1.2 Insert Operation

The insert operation has the following prototype:

- *void insert_node(int key, int value, bst_node *root);*

As seen above, the operation takes three parameters namely, key, value and a pointer to BST node. The function traverses the BST based on the key value. The operation is performed using hand-over-hand locking wherein the parent node is locked first, followed by appropriate right or left child lock after unlocking parent node. The tree is traversed until the null is found and once found a new node will be created with given key and value. If the node is found with the same value then the function will just ignore the value.

1.3 Search Operation

The search operation has the following prototype:

- *void search_node(int key, int value, bst_node *root);*

As seen above, the operation takes three parameters namely, key, value and a pointer to BST node. The function traverses the BST based on the key value. The operation is performed using hand-over-hand locking wherein the parent node is locked first, followed by appropriate right or left child lock after unlocking parent node. The tree is traversed until the node is found with the same key value and once found, the key and related value will be inserted into the C++ STL map.

1.4 Range Operation

The Range operation has the following prototype:

- *void range_query(int lower_range, int upper_range, bst_node *root);*

As seen above, the operation takes three parameters namely, lower end of the range, upper end of the range and a pointer to BST node. The function traverses the BST based on the key value. The operation is performed using hand-over-hand locking wherein the parent node is locked first, followed by appropriate right or left child lock after unlocking parent node. The tree is traversed until the BST is empty. If the value of the key is less than the current root's value then the function will traverse recursively in the left subtree and if it is greater than the current node value then right subtree will be traversed. And if the value is in the range the key and related value will be inserted into the C++ STL map.

2 Code Organization

2.1 Handling Command-line arguments:

To take inputs from the user *getopt_long()* function is used. Following different arguments are handled.

- The very first argument should always be the name of the file. The file contains all the key values of the nodes to be inserted into the BST.
- ops: is used for selecting which operation to be performed. It can take three different values, *create, search and range* and as the name suggests it performs the creating a BST, searching in the BST and searching nodes between the given range in the BST.
- t: is used for selecting the number of threads to be used for the application
- search: when the operation is search, this argument takes a file as a parameter which has key values to be found in the BST.
- l: when the operation is range, this argument takes lower end of the range as a parameter.
- u: when the operation is range, this argument takes the upper end of the range as a parameter.

2.2 Setting up the workload

As mentioned in the Section 2.1 user will select which operation to be performed. And based on the user selection program will perform next steps as given below.

2.2.1 Insertion Operation

- Irrespective of the *selected operation* the program will perform the operation *CREATE* to create a BST. To create a BST program reads a file which contains all the key values to be inserted into the BST. For a key, related value is the key multiplied by two.
- To divide the work fairly between all the threads, after reading the file, the number of nodes to be inserted into the BST will be divided by the total number of threads. Thus, making sure each thread has quite a fair share of nodes in most of the cases.
- For example, let’s say the total number of nodes to be inserted is 100 and given number of threads is 5 then, each thread will work on inserting 20 numbers. Thread 1 will start from index 1 to 20, thread 2 from 21 to 40 and so on.
- Then required number of threads will be spawned and insertion operation will be performed as explained in the *Section 1.3*

2.2.2 Search Operation

- For the search operation, again each thread will be given equal number of nodes to be searched in the BST by dividing the total number of nodes to be searched. The total number of nodes to be searched will be obtained from the file given as a command line argument.
- For example, let's say the total number of nodes to be searched is 100 and given number of threads is 5 then, each thread will work on inserting 20 numbers. Thread 1 will start from index 1 to 20, thread 2 from 21 to 40 and so on.
- After this required number of threads will be spawned to perform the operation.
- Also, the *CREATE* and *SEARCH,* operation will be performed concurrently and therefore if a thread tries to find a number before it was inserted, it won't find as the insertion wasn't performed for the particular node.

2.2.3 Range Operation

- For the range operation, same as above, in most cases, each thread will be given equal length of range to find the nodes in between.
- For example, if the given range is 1 to 10 and given number of threads is 2 then, thread one will try to find all the nodes between 1 & 5 and thread 2 will try to find all the nodes between 5 & 10.
- Also, the *CREATE* and *RANGE,* operation will be performed concurrently and therefore if a thread tries to find a number before it was inserted, it won't find as the insertion wasn't performed for the particular node.

3 Compilation & Execution

- **Makefile**: as the name suggests it is a Makefile to compile and build the application. Use **make** on the terminal to do the same.
- **main.cpp**: this is the source file for the lab. It has the main() function and all other required functions to fulfill the requirements.
- **random.sh**: this file is a bash script used for generating a test file with random number of integers.
- **log.h**: a simple logging system for different log levels. Disabled by default to save time used by printfs.

3.2 Compilation

- After downloading the submitted .zip file, extract and go into the directory Mysort
- Use the **make** command to build and create an executable named **main.**

3.3 Execution

- ./main --help prints the instruction to execute the program
- ./main *file_name.txt* -t 4 --ops *operation* --search *file_name.txt -l 10 -u 20*

As described in the *Section 2.1, following are the valid execution instructions*

- *./main create_nodes.txt -t 4 --ops create*
- *./main create_nodes.txt -t 4 --ops search --search search_file.txt*
- *./main create_nodes.txt -t 4 --ops range -l 10 -u 30*

4 Performance Testing

Following are the various test cases performed on the machine specification as given below.

- CPU(s): 4
- Thread(s) per core: 2
- Core(s) per socket: 2
- Socket(s): 1
- Processor: Intel Core i5-7200U @ 2.50GHz

Each operation is tested with with 8 threads and 4 threads with different number of nodes.

4.1 Insertion Operation

		Mutex Lock		Reader-Writer Lock	
Number of Threads	Number of Nodes	Time Taken(s)	Time Taken(ns)	Time Taken(s)	Time Taken(ns)
8	10	0.001147	1118629	0.001517	1517147
8	1K	0.001544	1543984	0.001385	1385123
8	10K	0.001316	1316064	0.000432	431693
8	100K	0.000583	583078	0.000606	606395
8	1M	0.000414	413891	0.000210	209835
8	10M	0.000195	195187	0.000247	246683
8	100M	0.000605	604647	0.000242	241702

Table 1: Comparison for Insertion operation with 8 threads

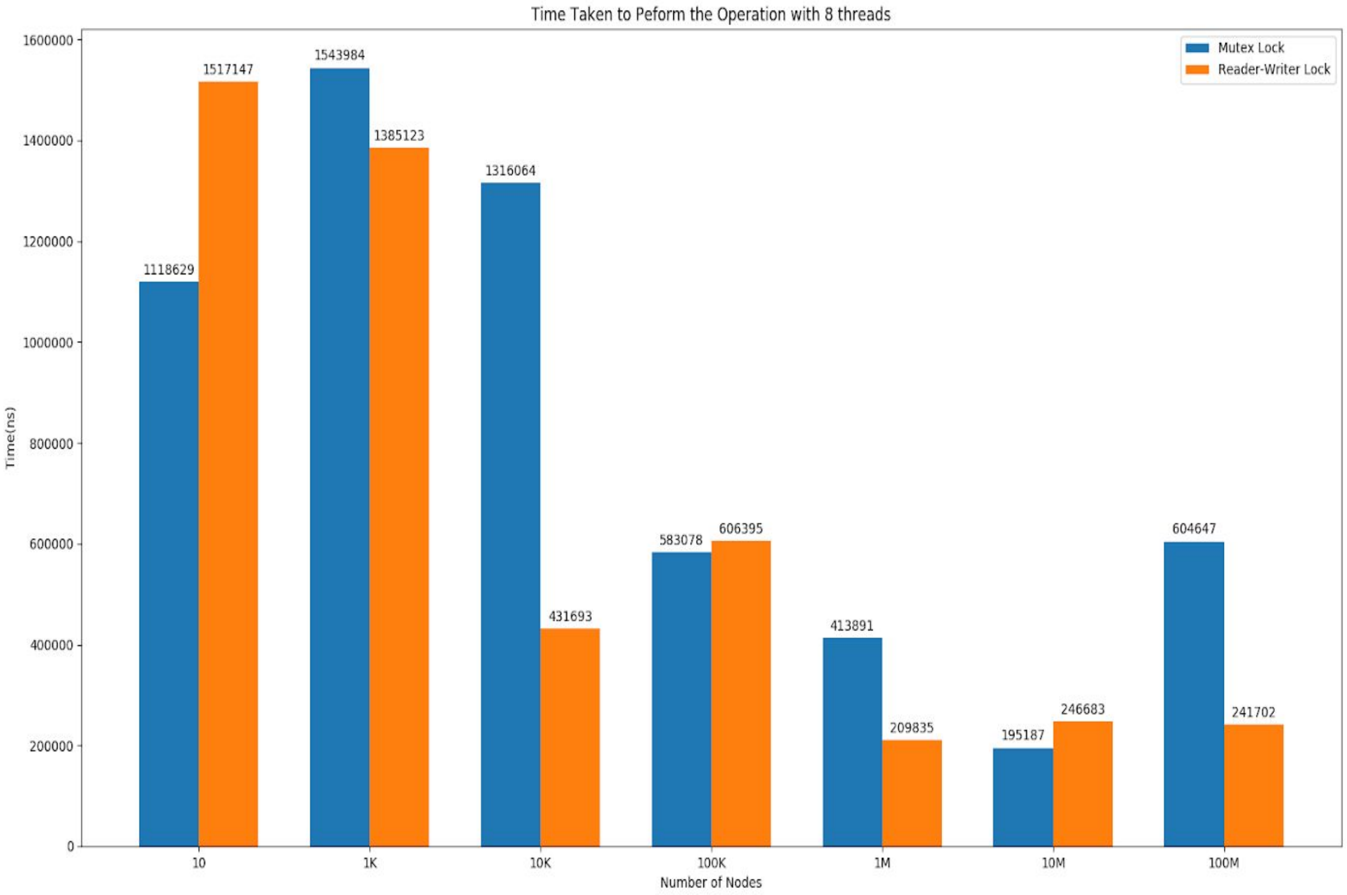

Figure 1: Comparison for Insertion operation with 8 threads

		Mutex Lock		Reader-Writer Lock	
Number of Threads	Number of Nodes	Tame Taken(s)	Time Taken(ns)	Time Taken(s)	Time Taken(ns)
4	10	0.000435	435232	0.000748	747660
4	1K	0.000510	509735	0.000444	443668
4	10K	0.000065	64572	0.000655	654808
4	100K	0.000068	67857	0.000112	111756
4	1M	0.000096	95589	0.000074	74053
4	10M	0.000075	74858	0.000076	75624
4	100M	0.000169	168861	0.000075	75366

Table 2: Comparison for Insertion operation with 4 threads

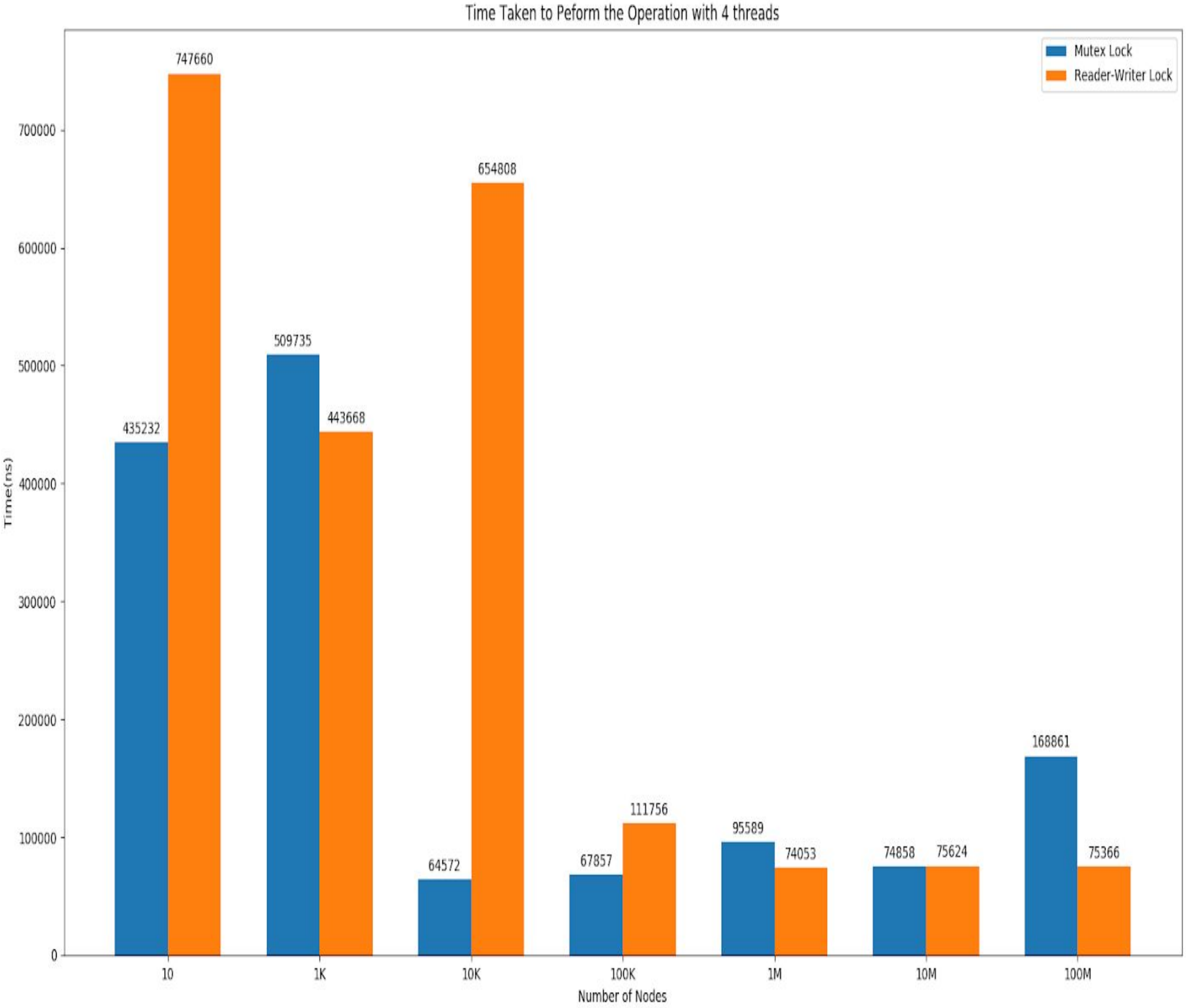

Figure 2: Comparison for Insertion operation with 4 threads

4.2 Search Operation

			Mutex Lock		Reader Writer Lock	
Number of Threads	Number of Nodes	Number of Nodes to be searched	Time Taken(s)	Time Taken(ns)	Time Taken(s)	Time Taken(ns)
8	10	10	0.000343	343030	0.001359	1358932
8	1K	500	0.000188	187842	0.001523	1528544
8	10K	5K	0.000169	169237	0.000829	829050
8	100K	50K	0.000229	228523	0.000330	330305
8	1M	100K	0.000145	145403	0.000195	195119
8	10M	1M	0.000276	275576	0.000207	207065
8	100M	10M	0.000196	195837	0.000152	152958

Table 3: Comparison for Search operation with 8 threads

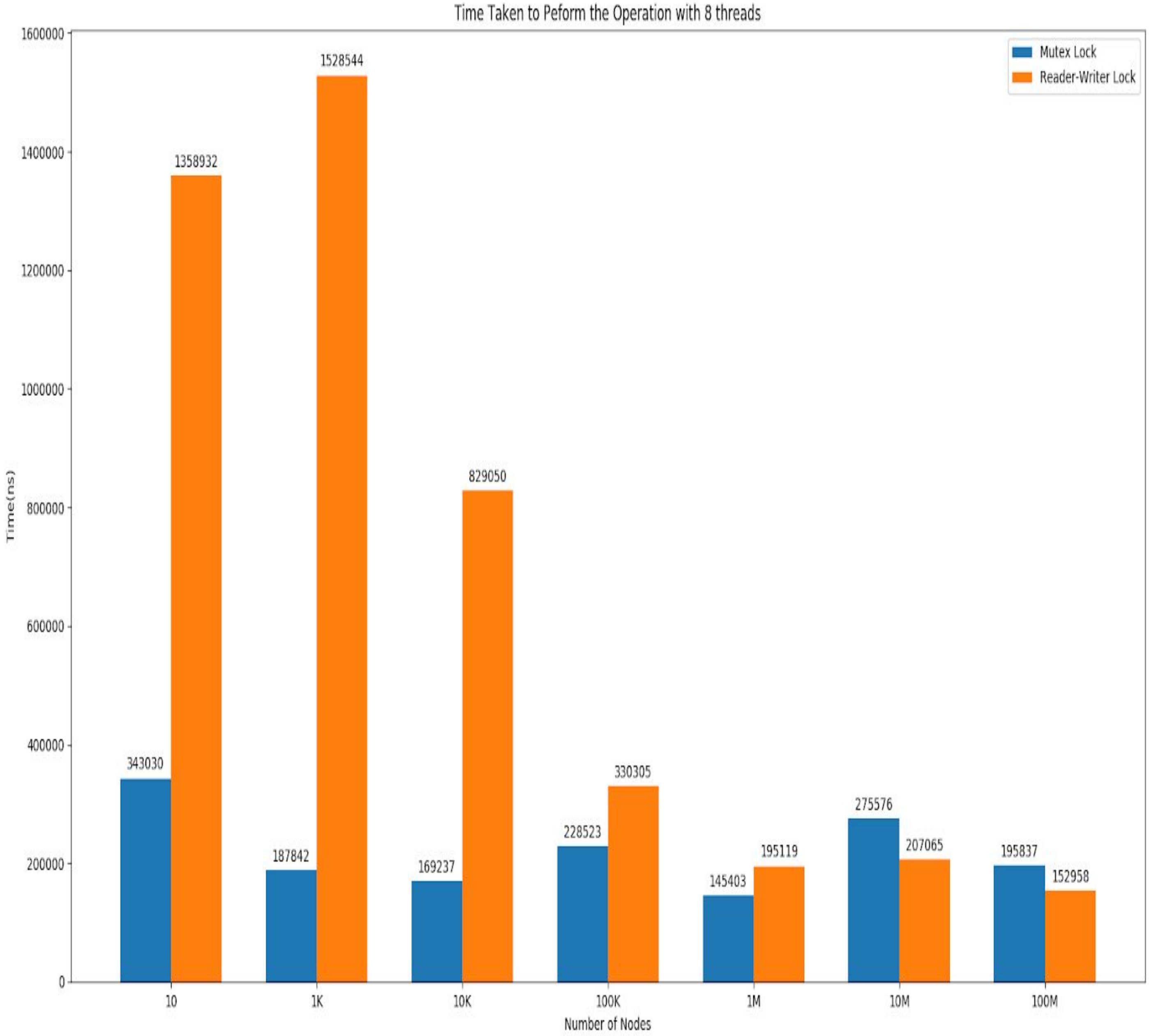

Figure 3: Comparison for Search operation with 8 threads

			Mutex Lock		Reader Writer Lock	
Number of Threads	Number of Nodes	Number of Nodes to be searched	Time Taken(s)	Time Taken(ns)	Time Taken(s)	Time Taken(ns)
4	10	10	0.000343	343030	0.000416	415935
4	1K	500	0.000069	69375	0.000064	63523
4	10K	5K	0.000075	75433	0.000089	89179
4	100K	50K	0.000084	84452	0.000083	83086
4	1M	100K	0.000091	90937	0.000071	70801
4	10M	1M	0.000074	73930	0.000078	77810
4	100M	10M	0.000073	73067	0.000075	74848

Table 4: Comparison for Search operation with 4 threads

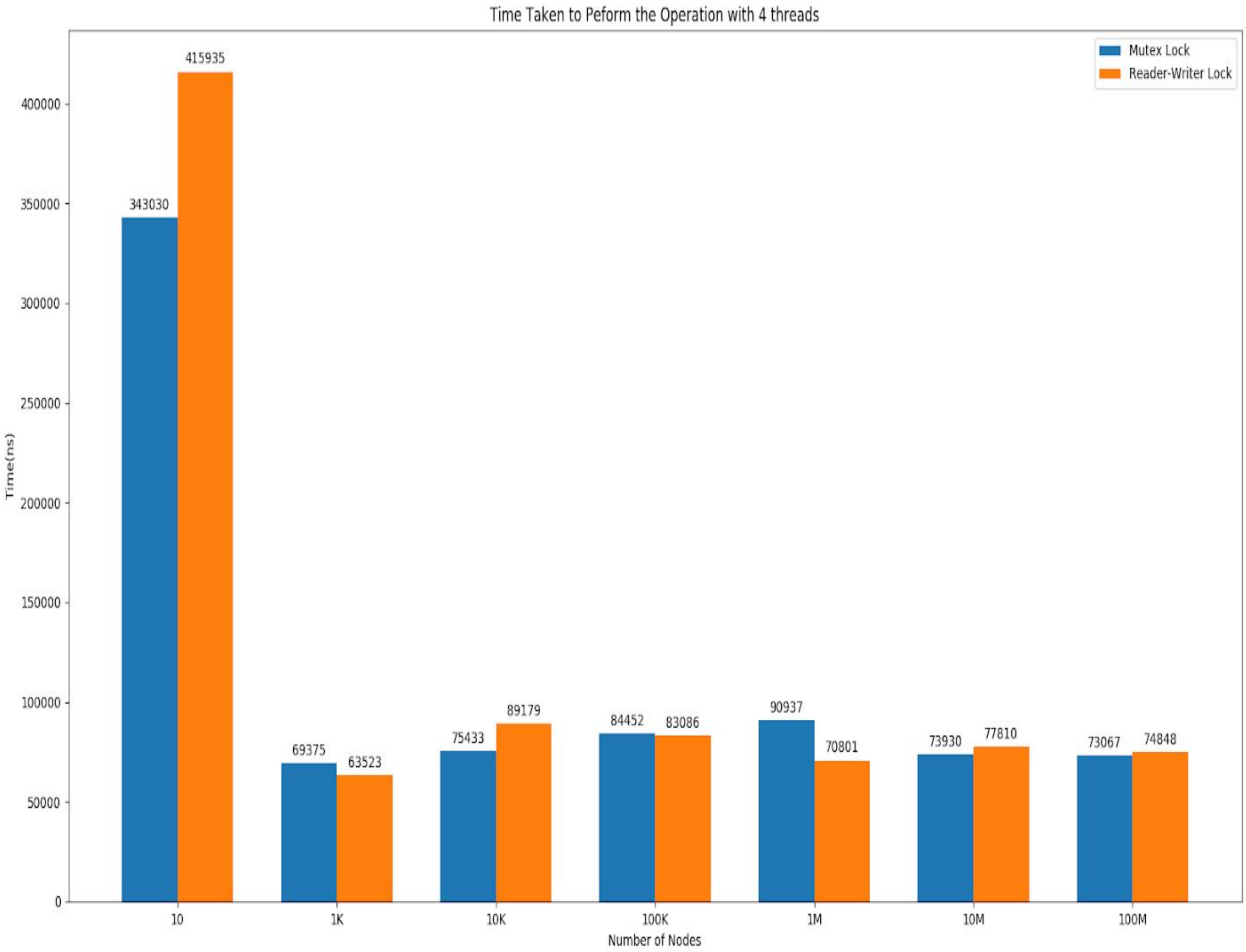

Figure 4: Comparison for Search operation with 4 threads

4.3 Range Operation

Number of Threads	Number of Nodes	Range	Time Taken(s)	Time Taken(ns)
8	10	1-5	0.001369	1368891
8	1K	50-100	0.000545	544546
8	10K	1000-5000	0.000333	333290
8	100K	50K-90K	0.000216	216004

Table 5: Comparison for Range operation with 8 threads

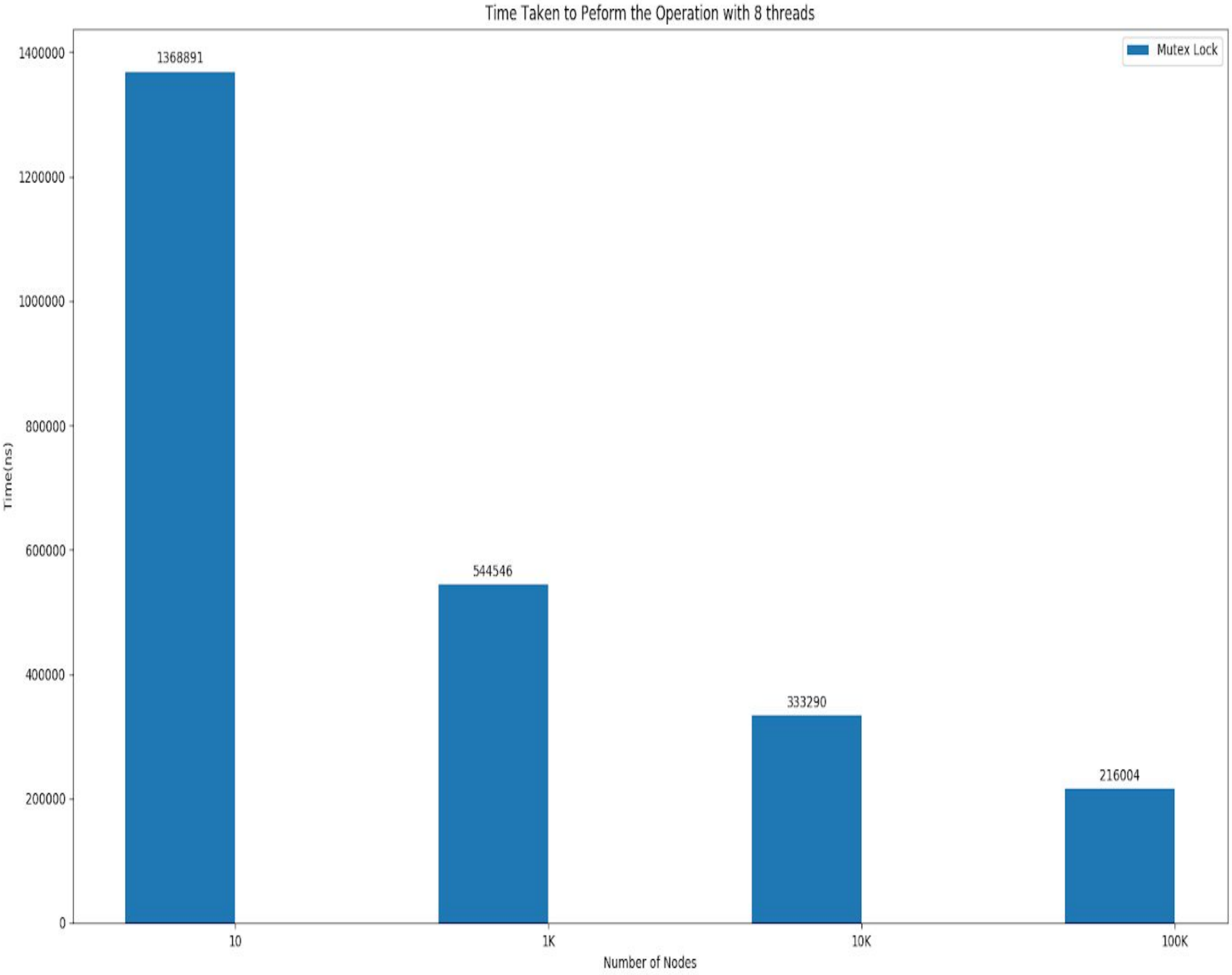

Figure 5: Comparison for Range operation with 8 threads

Number of Threads	Number of Nodes	Range	Time Taken(s)	Time Taken(ns)
4	10	1-5	0.000070	69527
4	1K	50-100	0.000072	71774
4	10K	1000-5000	0.000091	90789
4	100K	50K-90K	0.000077	76836

Table 6: Comparison for Search operation with 4 threads

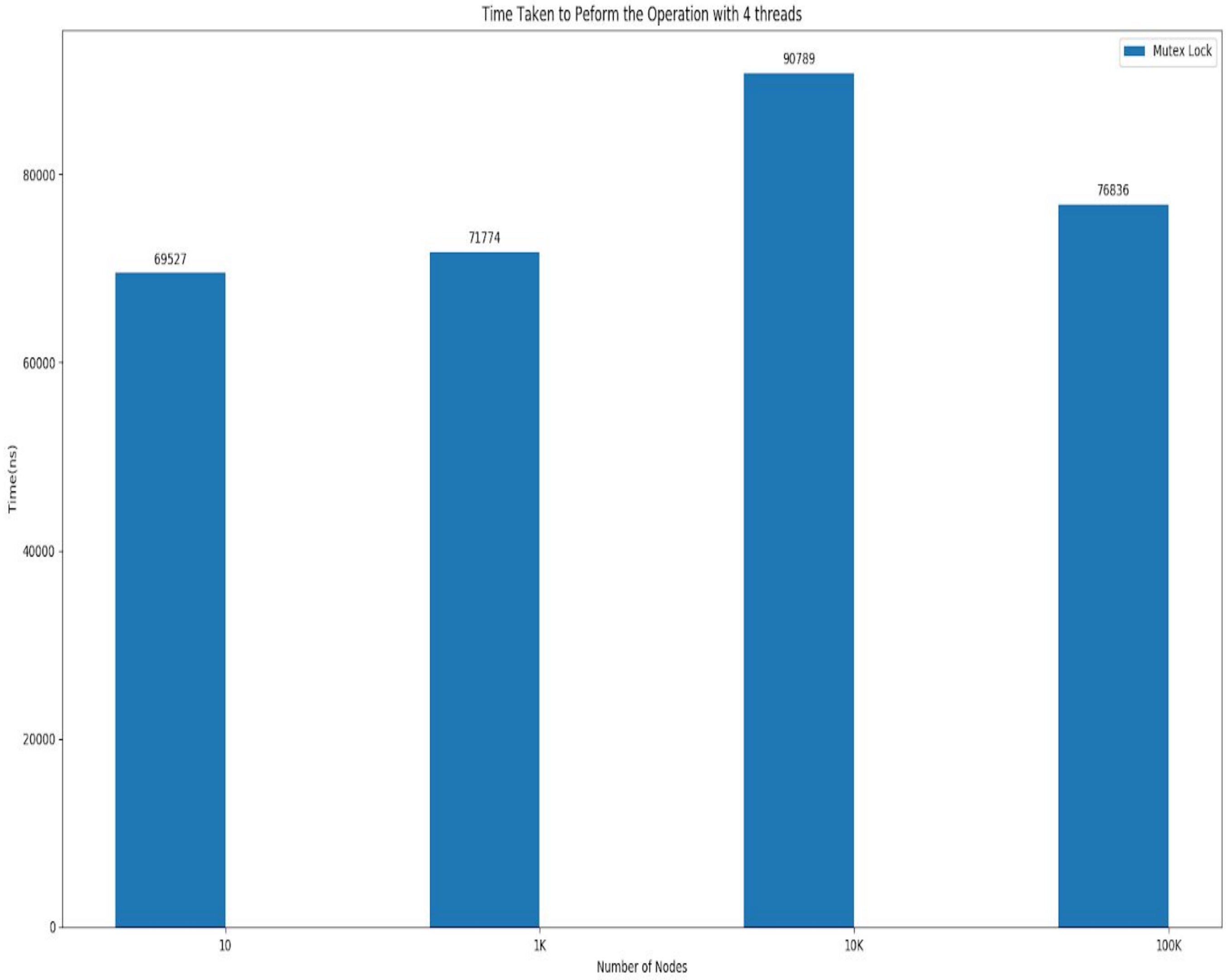

Figure 6: Comparison for Range operation with 4 threads

Note: Range Operation is not working for Reader Writer lock in multithreading environment, although it works for single threaded application.