Effective implementation of parallelization of concurrent data structures and algorithms

By Nishit Prasad
Masters in Computer Science
University of Central Florida
Orlando, Florida
nshtpd@knights.ucf.edu

By Divyasree Sadhukhan
Masters in Computer Engineering
University of Central Florida
Orlando, Florida
divyasree@knights.ucf.edu

***Abstract*—Implementation of concurrent data structures, understanding their performance benefits and their fairness limitation is the aim of our project. This paper presents performance analysis in the form of graphs and compares between the various methods employed.**

I. INTRODUCTION

The increasing computation power in modern computers in the form of several cores per processor and more processors, makes it necessary to rethink or to redesign sequential algorithms and data structures. An obvious approach would be to use parallelism. In this paper, we try implementing parallel programming on the concurrent data structures such as stacks and linked lists. We implement the lock based and lock free approaches to analyze the performance benefits of both types of data structures and the methods we have performed.

II. CONCEPT

Mutual exclusion is the traditional way to implement shared data structures in distributed systems. Locks are used in mutual exclusion to allow only one process to access the same part of shared memory at a time. However, the lock-based approach for implementing shared data structures cannot really take advantage of all processes. Only one operation at a time can enter the critical section, the code to access and modify the shared data structure that is protected by a lock, and all the other processes attempting to access the shared data structure simultaneously must wait. Thus, any delay of a process in the critical section can cause performance problems. Possible sources of delay include process pre- emptions, page faults, remote memory accesses, and cache misses. Lock-based algorithms in general, is that the scheduler must guarantee that threads are preempted infrequently (or not at all) while holding the locks. Otherwise, other threads accessing the same locks will be delayed, and performance will suffer. This dependency on the capriciousness of the scheduler is particularly problematic in hard real-time systems where one requires a guarantee on how long method calls will take to complete. We can eliminate this dependency by designing a lock-free implementation.

III. BACKGROUND

The data structures we are using in this project are stack and linked list.

STACKS

Stack or LIFO (last in, first out) data structure is an abstract data type that serves as a collection of elements, with two principal operations: push, which adds an element to the collection, and pop, which removes the last element that was added.

So, the operation to be described in brief are:

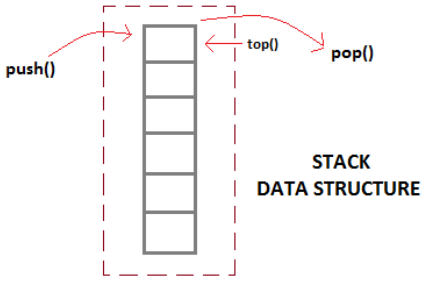
push(o): insert o on top of stack

pop(): remove object from top of stack top( ): look at object on top of stack (but don't remove)

size(): number of objects on the stack

isEmpty(): (size = = 0)?

This is the basic implementation of a stack. An element is inserted into the stack with the help of a push() operation. The topmost element in the stack is removed with the help of the pop() operation. top() is used to check the topmost element of the stack. The removal although is done by the pop() method. size() gives the number of objects in the stack and isEmpty() is used to determine whether the stack is empty or not. If the stack is empty then no elements can be popped off the stack but elements can be added to the stack further forming a new set of objects altogether.



Now, a stack can be implemented in two ways:

1. It can be array-based

2. It can be linked-list based.

Since we plan to implement the concurrency using a linked-list- based stack, below is given algorithm for linked-list-based stack implementation.

For inserting an element, the pseudo code is as follows:

Push(element):

- Create a new node.
- Get the previous top element and assign it to the next element of the newly created node.
- Assign the new node to top.

For removing an element, the pseudo code is as follows:

Pop(element):

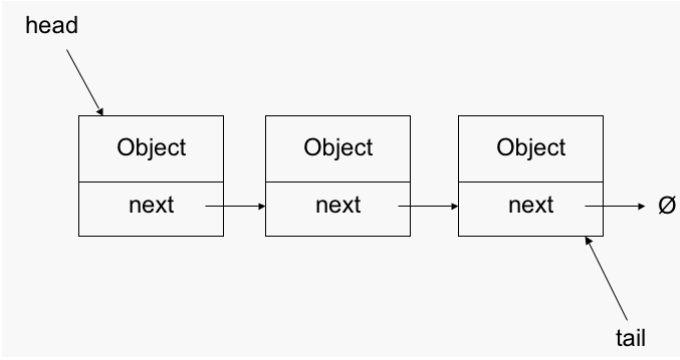
- Assign the topmost element to a temporary variable
- Assign the next element of the top as the new top element
- If you want to display the popped off element, return it. The element is removed.

A code snippet:

```
public void add (int element) {
  Node n = new Node(element);
 n.next = top;
 top = n;
}
public node remove () {
 Node temp = top;
top = top.next;
return temp;
}
```

LINKED-LISTS

Linked List is a representation of a group of nodes which together represent a sequence. Under the simplest form, each node is composed of data and a reference (in other words, a link to the next node in the sequence).



The entry point into any linked list is the head of the list. Head of the list is not a node but a reference to the first node in the list. In other words, head can be considered a lvalue. In an empty list the value of head is equal to null. We also know that any linked list ends with null pointer (unless it is a circular linked list where last node is connected to the head node).

The only way to navigate a list is by using a reference to the next node from the current node. Often referred as "memory leaks", if a reference to a node is lost, then the entire list from that point on may not be accessible.

BASIC OPERATIONS

Some of the basic operations that can be performed on a linked list are

1. Traversing a linked list.

2. Appending a new node (to the end) of the list

3. Prepending a new node (to the beginning) of the list

4. Inserting a new node to a specific position on the list

5. Deleting a node from the list

6. Updating the data of a linked list node

Traversing a Linked List:

A linked list is a linear data structure that needs to be traversed starting from the head node until the end of the list. Unlike arrays, where random access is possible, linked list requires access to its nodes through sequential traversal. Traversing a linked list is important in many applications. For example, we may want to print a list or search for a specific node in the list. Or we may want to perform an advanced operation on the list as we traverse the list. The algorithm for traversing a list is fairly trivial. a. Start with the head of the list. Access the content of the head node if it is not null. b. Then go to the next node (if exists) and access the node information c. Continue until no more nodes (that is, you have reached the last node)

Appending a new Node to the end of the list

Sometimes we need to append (or insert to end) new nodes to the end of a list. Since we only have information

about the head of the list, we need to traverse the list until we find the last node. Then we insert new node to the end of the list. In this case, it is important to consider special cases such as list being empty.

Let us consider a method append that inserts nodes to the end of the list.

```
public void append(Node N ) {
Node ptr, prev; ptr = prev = head;
while (ptr != null)
ptr = ptr.next; // assume next is a public field in the node class
prev.next = N;
N.next = null;
}
```

Prepending a new Node to the beginning of the list

Let us try to prepend a new node to the beginning of a list or in other words insert a node to the beginning of the list. Prepending a node to a list is easy since there is no need to traverse the list and find the place to insert into the list. If list is empty, we make new node the head of the list. Otherwise, we connect new node to the current head of the list and make new node the head of the list.

```
public void prepend(Node N ) {
  if (head == null)
    head = N;
     else {
       N.next = head;
       head = N;
}
```

Inserting a new Node to an arbitrary position in the list Suppose we need to insert (to some specific location) a new node to a list. It is important that we traverse the list to find the place to insert the node. In the traversal process, we must maintain a reference to previous and next nodes of the list. Then we will insert a new node between previous and next.

```
public void insertInOrder(Node N ){
Node prev, ptr;
prev = ptr = head;
while (ptr != null && N.compareTo(ptr) > 0) {
prev = ptr;
ptr = ptr.next;
}
prev.next = N;
N.next = ptr;
}
```

Deleting a Node from the list To delete a specific node from the list (if the node exists). It is important that we traverse the list to find the node to delete. In the traversal process, we must maintain a reference to previous node of the list. Unlike in other programming languages (such as C) we don't need to worry about deallocating memory associated with the deleted node, since Java garbage collector takes care of this.

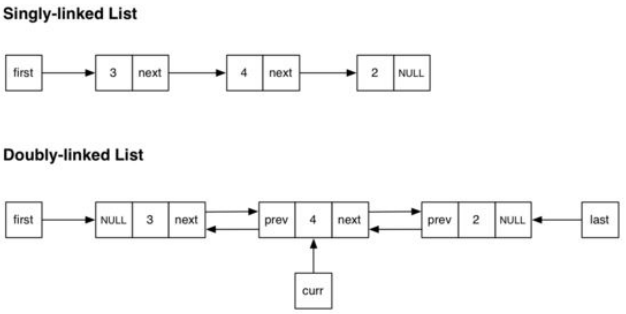
Let us take a look at the code to delete a node.

```
public void delete(Node N ) {
Node prev, ptr;
while (ptr != null && ptr.compareTo(N) != 0) {
prev = ptr;
ptr = ptr.next;
}
if (ptr == null)
return;
prev.next = ptr.next;
```

But singly linked-list at times turn inefficient for removing from the tail.

In contrast, a doubly linked list is a linked data structure that consists of a set of sequentially linked records called nodes. Each node contains two fields, called links, that are references to the previous and to the next node in the sequence of nodes.

The two node links allow traversal of the list in either direction. While adding or removing a node in a doubly linked list requires changing more links than the same operations on a singly linked list, the operations are simpler and potentially more efficient (for nodes other than first nodes) because there is no need to keep track of the previous node during traversal or no need to traverse the list to find the previous node, so that its link can be modified.



Doubly Linked list implementation:

public class MyDeque implements Deque

```
{
    DLNode header, trailer;
    int size;
public MyDeque() {
    header = new DLNode();
    trailer = new DLNode();
    header.setNext(trailer);
    trailer.setPrev(header);
    size = 0;
}
```

IV. INTRODUCING LOCK-BASED AND LOCK-FREE STACKS

In this section we will talk about the lock based and lock free implementations of the stack data structure. We have implemented it in Java.

We use 5 threads in these implementations. Among these 5 threads, Thread1, Thread2 and Thread3 are in charge of push operations whereas Thread4 and Thread5 are in charge of pop operations. They have an atomic integer set which counts the operations the thread involves. It counts the number of push operations and the number of pop operations. And as well by the getCount methods we get the count for each of the threads and their corresponding push and pop operations.

Firstly lets have a look at the lock based stacks. We have a new Operation class here which extends Thread which is the class giving access to the threads. This contains the run method, where we actually define the operations to be performed by the threads and decrements their respective counters saying that their operations are decrementing considering their limit. We next directly go to the push and pop methods.

In the push method, since this is not fine grained locking we only apply one thread throughout. So in the beginning of the push operation, we lock it. Next we set the top as the current and create a new node and set the current as the top's next field. Next it increments its counter after getting done with the operations just to keep a track of the number of push operations it has performed.

```
void push(T item){
    lock.lock();
    try {
      Node curr = top;
      top = new Node(item);
      top.next = curr;
      /*System.out.println("Item Pushed " + top.item.getValue());*/
      if(Thread.currentThread().getId()==9){
        count_Push1++;
      }
      else if(Thread.currentThread().getId()==10){
        count_Push2++;
      }
      else{
        count_Push3++;
      }
    } finally {
    lock.unlock();
```

In the pop method, we again lock it initially and check if the top is null. Next we assign the top's next as the top which means we are removing the node from the stack. So, if it matches with the thread ID, we again increment the counter just to keep a track of the number of pop operations the thread has performed. At the end of the whole of the operations, we unlock it.

```
void pop(){
  lock.lock();
    try {
      if (top==null){
      /*System.out.println("Stack is empty");*/
      } else {
      /*System.out.println("Item Popped " + top.item.getValue());*/
        top = top.next;
        if(Thread.currentThread().getId()==12){
          count_Pop4++;
        }else{
          count_Pop5++;
        }
      }
    }finally {
      lock.unlock();
    }
}
```

Now lets have a look at the lock free stack implementations. The lock free stack is a linked list whose top field points to the first node or null if the stack is empty. We have a lot of classes here. The first one we would like to explain is the Backoff class. So with the help of this class we actually ask the thread to back off for some random amount of time. So, we initialize two variables as the minimum and maximum delays and another limit and a random value. So we have this delay assigned as the random limit where we get a random number and in the next step we ask the limit to choose between the twice the limit and the max delay and It backs off for that amount of time. The thread actually sleeps for this delay period of time before accessing the stack again.

Next we have the LockFreeExchanger class. A LockFreeExchanger<T> object permits two threads to exchange values of type T. If one thread calls the object's exchange() method with its argument and another thread calls the same object's exchange() method with its argument , then first's call will return the second's value and vice versa. On a high level, the exchanger works by having the first thread arrive to write its value, and spin until a second arrives. The second then detects that the first is waiting, reads its value, and signals the exchange. They each have now read the other's value, and can return. The first thread's call may timeout if the second does not show up, allowing it to proceed and leave the exchanger, if it is unable to exchange a value within a reasonable duration. This has three states as we see, EMPTY, WAITING and BUSY. If the state is EMPTY, then the thread tries to place its item in the slot and set the state to WAITING using a compareAndSet() .If it fails, then some other thread succeeds and it retries. If it was successful, then its item is in the slot and the state is WAITING, so it spins, waiting for another thread to complete the exchange. If another thread shows up, it will take the item in the slot, replace it with its own, and set the state to BUSY, indicating to the waiting thread that the exchange is complete. The waiting thread will consume the item and reset the state to 0. Resetting to empty() can be done using a simple write because the waiting thread is the only one that can change the state from BUSY to EMPTY . If no other thread shows up, the waiting thread needs to reset the state of the slot to EMPTY. This change requires a compareAndSet() because other threads might be attempting to exchange by setting the state from WAITING to BUSY. If the call is successful, it raises a timeout exception. If, however, the call fails, some exchanging thread must have shown up, so the waiting thread completes the exchange. If the state is WAITING, then some thread is waiting and the slot contains its item. The thread takes the item, and tries to replace it with its own by changing the state from WAITING to BUSY using a compareAndSet(). It may fail if another thread succeeds, or the other thread resets the state to EMPTY following a timeout. If so, the thread must retry. If it does succeed changing the state to BUSY, then it can return the item. If the state is BUSY then two other threads are currently using the slot for an exchange and the thread must retry.
Next is the EliminationArray class where it calls the exchanger to traverse and keep on exchanging values for the number of threads who are experiencing the backoff.
Next we come to the push and pop methods. A pop() call that tries to remove an item from an empty stack throws an exception. A push() method creates a new node, and then calls tryPush() to try to swing the top reference from the current top-of-stack to its successor. If tryPush() succeeds, push() returns, and if not, the tryPush() attempt is repeated after backing off. The pop() method calls tryPop(), which uses compareAndSet() to try to remove the first node from the stack. If it succeeds, it returns the node, otherwise it returns null. This implementation is lock-free because a thread fails to complete a push()or pop() method call only if there were infinitely many successful calls that modified the top of the stack. The linearization point of both the push() and the pop() methods is the successful compareAndSet(), or the throwing of the exception in case of a pop() on an empty stack.

The pop and the push follow the below

```
protected Node tryPop() throws EmptyException {
  Node oldTop = top.get();
  if (oldTop == null) {
    throw new EmptyException();
  }
  Node newTop = oldTop.next;
  if (top.compareAndSet(oldTop, newTop)) {
    return oldTop;
  } else {
    return null;
  }
}
public T pop() throws EmptyException {
  while (true) {
    Node returnNode = tryPop();
    if (returnNode != null) {
      return returnNode.value;
    } else {
      backoff.backoff();
    }
  }
```

The Push algorithm:

```
protected boolean tryPush(Node node){
  Node oldTop = top.get();
  node.next = oldTop;
  return(top.compareAndSet(oldTop, node));
}
public void push(T value) {
  Node node = new Node(value);
  while (true) {
    if (tryPush(node)) {
      return;
    } else {
      backoff.backoff();
    }
  }
```

This does not face ABA problem because Java's garbage collector takes care of it. It says that a node cannot be reused by any other thread as long it is accessible by its present thread.

V. INTRODUCING LOCK-BASED AND LOCK-FREE LINKED-LISTS

For the lock-based linked lists, we have used fine-grained locking algorithm and used Java.

We again use 5 threads in these implementations. Among these 5 threads, Thread1, Thread2 and Thread3 are in charge of push operations whereas Thread4 and Thread5 are in charge of pop operations. They have an atomic integer set which counts the operations the thread involves. It counts the number of push operations and the number of pop operations. And as well by the getCount methods we get the count for each of the threads and their corresponding push and pop operations.

In fine-grained locking algorithm we donot use a single lock. Instead we try to synchronize every access to an object and we split the object into independently synchronized components. By this we ensure that method calls interfere only when trying to access the same component at the same time. So we implement the fine grained locking in the following ways. We lock individual nodes and do not lock the list as a whole. So, when a thread traverses the list, we lock each node by the lock and unlock methods. The lock is also acquired in a hand over hand locking manner. This is done because, while one thread decides to lock one node to refer to a certain node, another thread might remove it. For example, if there are two nodes a and b, where a is the predecessor and b is the current node say, it is not safe to unlock a before unlocking b because anther head might remove b from the list in the interval between unlocking a and locking b. And another point to be remembered is that, to guarantee progress, it is important that all methods acquire locks in the same order, starting at the head and following the next references towards the tail. So, it needs to start from the very beginning, that is from the head node.

In our implementation, we have two methods:

- insert(T item) : This adds *item* which is of type T, returns true if and only if the value is already there.
- remove(T item) : This removes the *item* from the linked list.

Now, let us have a look at the insert method of our implementation. Initially, we have taken the hashcode of the *item* which gives us the key of the item. Next we lock the head node and assign the head to the predecessor. The reason we do that is because we need to change the head to the new value that we are trying to insert. Our next step would be, to assign the predecessor's next value to the current and we lock the current. These are the times when we actually implement the fine grained locking where we lock each node. Next we traverse through the linked list by comparing the key of the current node just assigned to the key of the new element to be inserted. Then we unlock the predecessor. After unlocking we have the node free to be assigned the current value. Then we assign the current's next value to the current and lock it. This is where the actual insert operation happens. If the keys we compared are equal, this means that the value exists already in the linked list and it returns false because there is no more value to be inserted to the list. If it does not match, the next operation is the insert operation where a new node is created and item is inserted into the correct position desired. If the node is correctly inserted into the right place desired, it returns true. Next is unlocks the current and hence the predecessor, whose values changed with the traversal and insertion of the nodes.

The Insert method algorithm we followed:

```
public boolean add(T item) {
 int key = item.hashCode();
 while (true) {
    Node pred = head;
    Node curr = head.next;
    while (curr.key < key) {
     pred = curr; curr = curr.next;
    }
    pred.lock();
    try {
      curr.lock();
      try {
        if (validate(pred, curr)) {
          if (curr.key == key) {
            return false;
          } else {
            Node node = new Node(ite
            node.next = curr;
            pred.next = node;
            return true;
          }
        }
      } finally {
        curr.unlock();
      }
    } finally {
      pred.unlock();
    }
  }
```

Next we will have a look at the remove method of our implementation. In this method we only desire to remove the desired item from the list. So we initially take the predecessor and the current nodes and initialize them to null. Next we take the hashcode and that is our key to be compared. So our next step is to lock the head node. While we do so the next step of ours is to assign the head value to the predecessor and assign the predecessor's next value to the current. Next is to lock this new current field. The next step is same as the insert method. We

need to traverse the linked list for the correct node by comparing its key. So try and traverse till its key is greater than the current's key, once found we unlock the predecessor and mark the current node to the predecessor. Next obvious step is marking the current's next node as the current and locking it, because it is this node, whose value will be changed that is whose value will be removed. If the key is found finally and it is the same , we remove the node by, setting the current's next as the predecessor's next and thus returning true by showing that the node is removed from the linked list. Next we unlock the current node and the predecessor node.
The remove method algorithm we followed is as follows:

```
public boolean remove(T item) {
  int key = item.hashCode();
  while (true) {
   Node pred = head;
   Node curr = head.next;
   while (curr.key < key) {
    pred = curr; curr = curr.next;
    }
    pred.lock();
    try {
      curr.lock();
      try {
        if (validate(pred, curr)) {
          if (curr.key != key) {
            return false;
          } else {
            curr.marked = true;
            pred.next = curr.next;
            return true;
          }
        }
      } finally {
        curr.unlock();
      }
    } finally {
      pred.unlock();
    }
```

Next the lock-free linked list's implementation comes into play. In this implementation we do not use the locks. We eliminate the use of locks as a whole. We use compareandset to change the field references and refer them to the new nodes we wish to update to. But there is risk in even this procedure. Some other threads might change the values of the nodes while the other threads tend to change or remove them. So we treat the node's next and marked fields as a single atomic unit: any attempt to update the next field when the marked field is true will fail.

An AtomicMarkableReference<T> is an object from the java.util.concurrent.atomic package that encapsulates both a reference to an object of type T and a Boolean mark. These fields can be updated atomically.
We also have another additional method here, the find method. The find() method takes a head node and a key a, and traverses the list, seeking to set predecessor to the node with the largest key less than a, and current to the node with the least key greater than or equal to a. As thread A traverses the list, each time it advances current, it checks whether that node is marked. If so, it calls compareAndSet() to attempt to physically remove the node by setting predecessor's next field to current's next field. This call tests both the field's reference and Boolean mark values, and fails if either value has changed. A concurrent thread could change the mark value by logically removing predecessor, or it could change the reference value by physically removing current node. If the call fails, the previous thead restarts the traversal from the head of the list; otherwise the traversal continues.
Next we will have a look at the insert method which inserts the item. Suppose a thread calls the insert function. It uses find() to locate its predecessor and current fields. If current's key is equal to a's, the call returns false. Otherwise, insert() initializes a new node to hold the value, and sets a to refer to current node. It then calls compareAndSet() to set predecessor to that node. Because the compareAndSet() tests both the mark and the reference, it succeeds only if predecessor is unmarked and refers to current. If the compareAndSet() is successful, the method returns true, and otherwise it starts over.

```
public boolean add(T item) {
   int key = item.hashCode();
   while (true) {
    Window window = find(head, key);
    Node pred = window.pred, curr = window.curr;
    if (curr.key == key) {
        return false;
      } else {
        Node node = new Node(item);
        node.next = new AtomicMarkableReference(curr, false);
        if (pred.next.compareAndSet(curr, node, false, false))
          return true;
        }
      }
    }
```

Next is the remove() method. When one thread calls remove() to remove an item, it uses find() to locate the predecessor and current. If current's key fails to match a's, the call returns false. This call succeeds only if no other thread has set the mark first. If it succeeds, the call

returns true. A single attempt is made to physically remove the node, but there is no need to try again because the node will be removed by the next thread to traverse that region of the list.

```
public boolean remove(T item) {
  int key = item.hashCode();
  boolean snip;
  while (true) {
   Window window = find(head, key);
   Node pred = window.pred, curr = window.curr;
   if (curr.key != key) {
     return false;
   } else {
     Node succ = curr.next.getReference();
     snip = curr.next.attemptMark(succ, true);
     if (!snip)
       continue;
     pred.next.compareAndSet(curr, succ, false, false);
     return true;
   }
  }
```

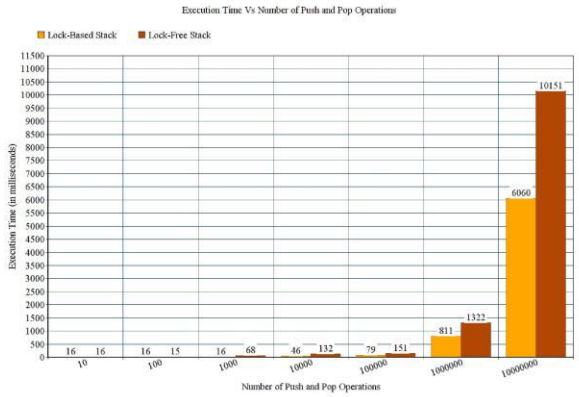
If the attemptMark() call fails, remove() starts over.

The next method we have implemented is the contains() Here it tests if the current node is marked. If so, we must apply curr.next.get(marked) and check that marked[0] is true.

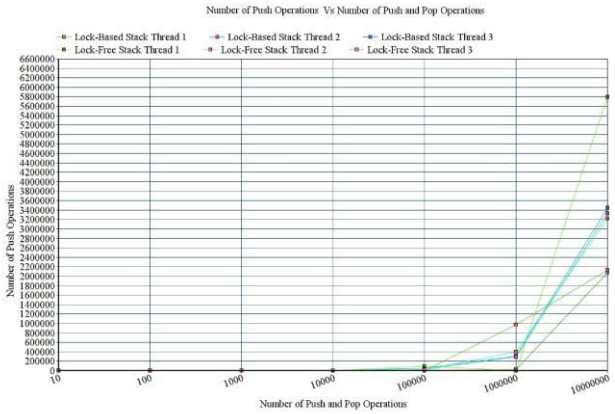
VI. PERFORMANCE ANALYSIS

Our aim in the project was to determine the performance of the data structures based on certain parameters and observe how it changes or affects them.

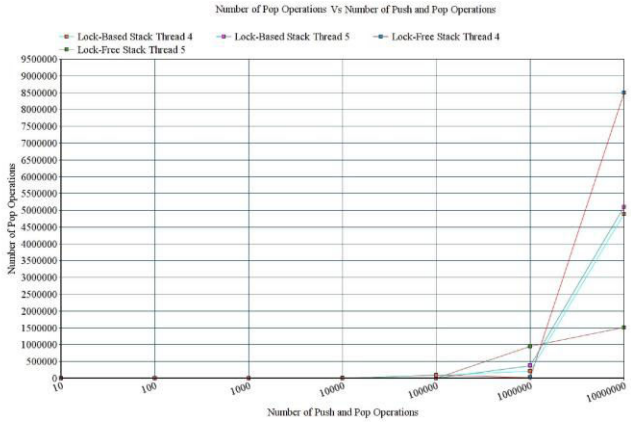
Let's first have a look at the stack operations. In the graph shown below we can see that the we have two parameters, the execution time and the number of operations. The number of operations mainly include the push and the pop operations in each of the implementations. As we can see, that we keep on increasing the number of operations in the order of 10^2*n and increment n each time. So we see, that when 10 operations were assigned, it was low, and it kept on increasing with the increase in the number of operations for a static thread count of 5. So we see the differences between the lock based and the lock free stack operations.



The next graph to be observed is the number of push operations vs the total number of operations again for the two types of implementations: lock based and lock free.



The next graph to be observed is the number of pop operations vs the total number of operations again for the two types of implementations: lock based and lock free.



Lock-free data structures don't necessarily show better performance, despite the apparent benefits. In a linked list, fine grain locks don't necessarily perform better than the mutex lock. In these cases, the overhead of acquiring/releasing locks is too large for small and large processor counts.

VII. ACKNOWLEDGEMENT

I would like to take this opportunity to thank Dr. Damian Dechev, professor in the University of Central Florida, a faculty of the Department of Computer Science, for giving us this opportunity.

VIII. REFERENCES

1. The Art of Multiprocessor Programming by Nir Shavit and Maurice Merlihy.

a.