COL818 : Principles of Multiprocessing Systems

Name: Vatsal Jingar
Entry Number: 2020CS50449

Date: 7 May 2024
Assignment: 3

Lock-Free Dynamically Resizable Arrays

1 Algorithm and Pseudo Code

In this paper, authors have given an algorithm for lock free implementation of concurrent vector data structure of C++.

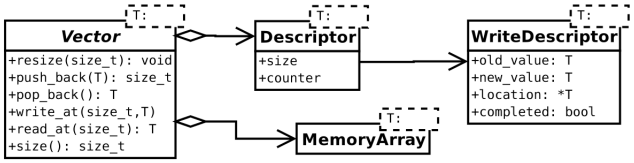


Fig. 1. Lock-free Vector. T denotes a data structure parameterized on T.

Figure 1: Lock Free Vector parameterized by T

The above diagram shows the structure of lock free vector. Each vector object contains the location of a descriptor and the actual data. Descriptor is key to achieve lock free property of vector with minimal ABA problem persistent. The paper utilize two level array to avoid synchronization hazards. Vector provides six methods to be called by user.

- push_back : Increases size of vector by 1, potentially allocating more space.
- pop_back : Decreases size of vector by 1
- read : Reads the data at give location
- write : writes the give data at given location
- reserve : Provides the extra space to the vector.
- size : Returns the size of vector.

Each vector points to array of buckets. Each bucket would be either empty or utilized by the vector for storing the values. As the size of vector grows, it keeps allocating arrays for new buckets. Each bucket is represented by a bucket index.
When the size of vector grows the new elements goes to different bucket.

1.1 Notion of Helping

The problem of making lock free vector class involves a fundamental problem of making push_back linearizable. Push_back involves two operations which should take effect without any other thread changing the descriptor object in between. Push_back needs to make changes in descriptior object because it changes the size of vector and can potentially allocate more space for storing elements and similarly pop_back should also follow these constraints. To solve this problem, the paper proposes an idea that if the first threads sleeps after CAS to modify the descriptor then the other thread will first complete the pending operation.

1.2 AllocBucket

This method allocates new space with size double of previous bucket size. And then tries to atomically assign data[bucket] := mem if it is NULL. Here mem is new memory allocated. We check NULL, which signifies if anyother person has allocated the space for data[bucket]. If CAS is not successful this implies some one else had allocated the space. So we delete our own space.

Algorithm 6 At $vector, i$

$pos \leftarrow i + \texttt{First\_Bucket\_Size}$
$hibit \leftarrow HighestBit(pos)$
$idx \leftarrow pos \ xor \ 2^{hibit}$
return $\&vector.memory[hibit - HighestBit(\texttt{First\_Bucket\_Size})][idx]$

Figure 2: AllocBucket

1.3 Push_Back

Each push_back operation first check whether new memory should be allocated or not. After which it creates a new descriptor with pending operation and swaps it with current descriptor atomically. It keeps trying till it succeeds. Finally it completes its operation (if it is not done yet by any other).

Algorithm 1 pushback $vector, elem$

repeat
  $descriptor_{current} \leftarrow vector.descriptor$
  $CompleteWrite(vector, descriptor_{current}.pending)$
  $bucket \leftarrow HighestBit(descriptor_{current}.size + \texttt{First\_Bucket\_Size}) - HighestBit(\texttt{First\_Bucket\_Size})$
  if vector.memory[bucket] = NULL **then**
    $AllocBucket(vector, bucket)$
  end if
  $writeop \leftarrow new \ WriteDesc(At(descriptor_{current}.size)\hat{}, elem, descriptor_{current}.size)$
  $descriptor_{next} \leftarrow new \ Descriptor(descriptor_{current}.size + 1, writeop)$
until $CAS(\&vector.descriptor, descriptor_{current}, descriptor_{next})$
$CompleteWrite(vector, descriptor_{next}.pending)$

Figure 3: Push_back

1.4 Pop_Back

Pop_back is similar to push_back with only one difference that the corresponding operation would be null. Also, it does not check whether memory should be allocated or de-allocated.

Algorithm 2 popback *vector*

repeat
  $descriptor_{current} \leftarrow vector.descriptor$
  $CompleteWrite(vector, descriptor_{current}.pending)$
  $elem \leftarrow At(vector, descriptor_{current}.size - 1)$^
  $descriptor_{next} \leftarrow new\ Descriptor(descriptor_{current}.size - 1, NULL)$
until $CAS(\&vector.descriptor, descriptor_{current}, descriptor_{next})$
return *elem*

Figure 4: Pop_back

1.5 At operation

Given an index from which read should happen or write, At fetches the index of bucket in which this element would be present by some binary algebraic manipulation. Finally it returns the memory address of the value that should be present at any given user provided index.

Algorithm 6 At $vector, i$

$pos \leftarrow i +$ `First_Bucket_Size`
$hibit \leftarrow HighestBit(pos)$
$idx \leftarrow pos\ xor\ 2^{hibit}$
return $\&vector.memory[hibit - HighestBit($`First_Bucket_Size`$)][idx]$

Figure 5: At operation

2 Limitations and Implementation

- **Complications in Implementation** : The paper provides the implementation of concurrent vector class for C++. Although, it does not described how it performs CAS operation. CAS can be performed by C++ library ¡atomic¿. But the library does not allow non trivial copyable classes to atomic convertable.

- Therefore, I made my own CAS which typecasts every pointer to long long * and then cas the value of long long. Each descriptor object would be a pointer of type $Descriptor < T >^*$. So the sentinal would be a pointer to descriptor object instead of the object itself. Finally I use asm CAS by converting any pointer to long long *.

- Although this creates the concurrent vector class but the chances of ABA problem to occurr increases. I will explain what ABA problems could occur in these settings.

- Rest of the code goes same as Pseudo code.

- I have implemented the lock free vector in C++ and Java. The purpose of implementing it in Java was to deeply analyse the effect of lock free reference counter. Apart from this, Java includes AtomicReference which converts any class object into atomic object and due to this there no pointers from vector object to descriptor object. Instead the descriptor object lies in vector object.

- **Limitation ABA Problem** : Suppose T0 initiates a push back operation and updates the vector's "Descriptor" object with a write-descriptor indicating that the value of the object at position i is to be changed from A to B. Subsequently, thread T1 interrupts and retrieves the write-descriptor. Later, after T0 resumes and successfully finishes the operation, a third thread T2 can revert the value at position i from B back to A. When T1 resumes execution, its Compare-and-Swap (CAS) operation will succeed, leading to an incorrect execution of the update from A to B.

3 Correctness Proof

Assume that there is an operation O ϵ S. Here S is set of all operations on vector. We assume that this operation O can be executed parallely with other operations. We can observe that reads and write will be wait free. For these two operations, linearization point will occur when both of them will execute. As method only return when it gets executed, therefore these two operations are linearizable. Now, if there is a descriptor modifying operation then it occur in two steps. First it changes the descriptor and then update the data structure contents. Let us define that the time points tcall , and treturn represent the moments of time when Oi has been called and when it returns, respectively. Similarly, time points tdesc and twritedesc denote the instances of time when Oi executes an atomic update to the vector's "Descriptor" variable and when Oi 's write descriptor is completed by Oi itself or another concurrent operation Oc ϵ {Oop1 , Oop2 ..., Oopn }, respectively. If the set Stps = {Ttp1 , Ttp2 ..., Ttpn } represents all instances of time in between tcall and treturn , it is clear that tdesc ϵ Stps and twritedesc ϵ Stps . As mentioned earlier, the pop back operation does not need to utilize a write descriptor. In this case the linearization point of Oi is tdesc . In the case when Oi is a push back, its linearization point is twriteop .

4 Analysis

The benchmarking is done on 11th Gen Intel® Core™ i7-1165G7 @ 2.80GHz on Ubuntu 22.

- We have designed our experiments by generating a workload of various operations (push back, pop back, random access write, and random access read).

- In the experiments, we varied the number of threads, starting from 1 and exponentially increased their number to 32. (Number of threads ranged as [1,2,4,8,16,32])

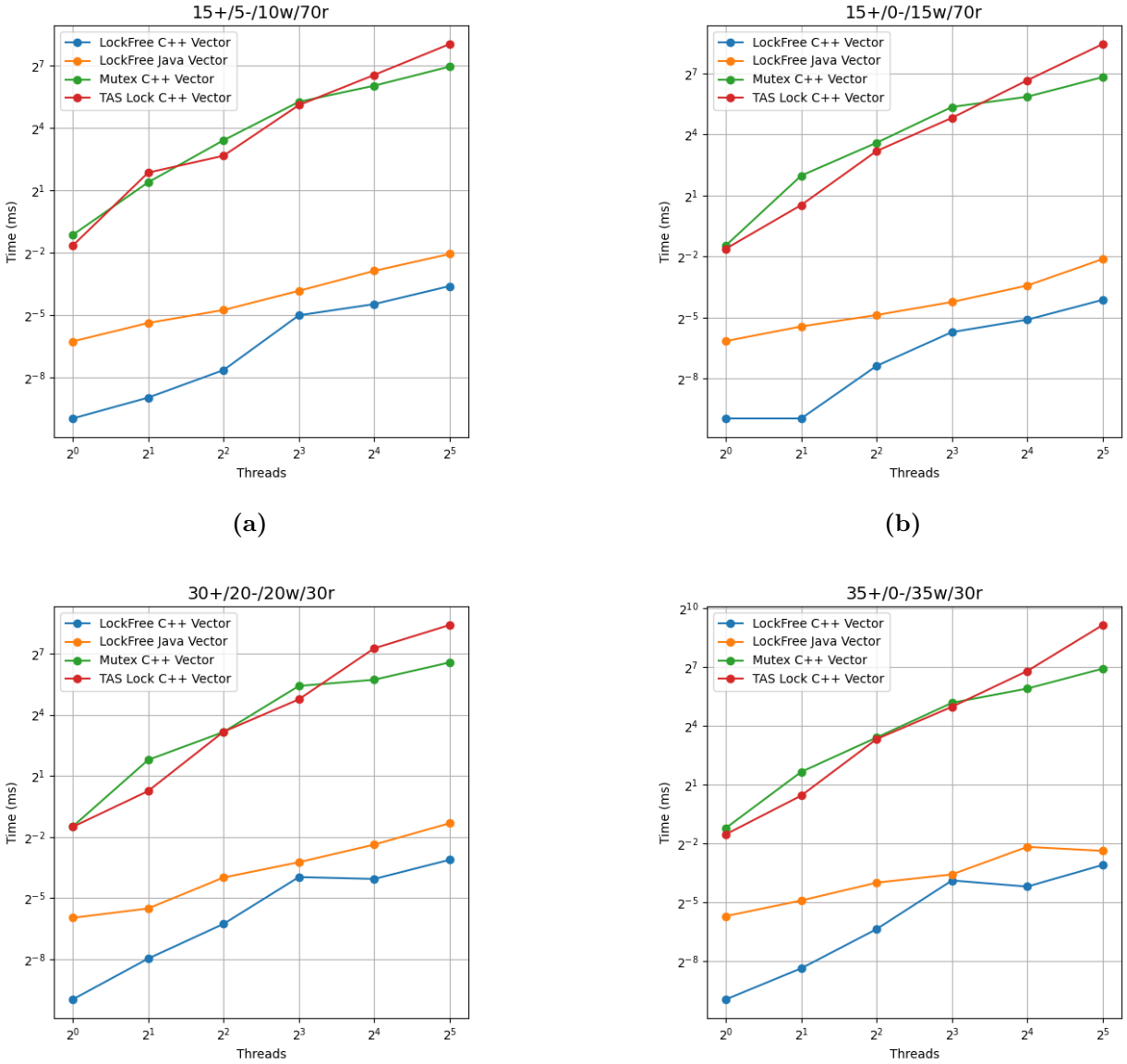
- Every active thread executed 10,000 operations on the shared vector. This included some percentage of `+` operator: **push_back**, `-` operator: **pop_back**, `w` operator: **write** and `r` operator: **read**.

- We measured the CPU time (in milliseconds) that all threads needed in order to complete.

- We illustrate the performance of the concurrent vectors with a distribution of $+ : 15\%, - : 5\%, w : 10\%, r : 70\%$ on Figure 6a. Similarly, Figure 6c demonstrates the performance results with a distribution containing predominantly writes, $+ : 30\%, - : 20\%, w : 20\%, r : 30\%$. In these diagrams, the number of threads are plotted along the x-axis (in a logarithmic scale), while the time needed to complete all operations is shown along the y-axis (measured in milliseconds, in a logarithmic scale).

These workloads define different type of usescases which covers almost all the scenarios.

Now, we can observe that the time for executing the above workloads was lowest in Java based lock free vector. For spinning locks, this time is highest as expected. And for Mutex which is queue based lock, this time is somewhat lesser. As we know that Spinning locks are unscalable which is observable from plots. TAS lock with vector performed similar to Mutex till 8 but then starts degrading. As it can be observable that in all types of workloads, lock free vector outperforms any lock based implementation. We will discuss the performance difference between in C++ and Java in next section.



(a) **(b)**

(c) **(d)**

Figure 6: Performance Results: Intel Core

5 Further Extension

As C++ based implementation suffers from ABA problem. I have also created a java based implementation. We can observe that time in Java is larger than C++. I think that my implementation where I typecast the pointer in long long * to atomically swap could be a reason for faster execution. In Java, there are Atomic references which are used to perform cas operations. Also, in background lock free reference counting creates an overhead. C++ based implementation achieved faster performance. But the disadvantage part is that it suffers from ABA problem. The plot shows that for 100% push_backs (all operations create contention) C++ implementation works best. It achieves similar performace to Intel TBB concurrent vector implementation in C++.

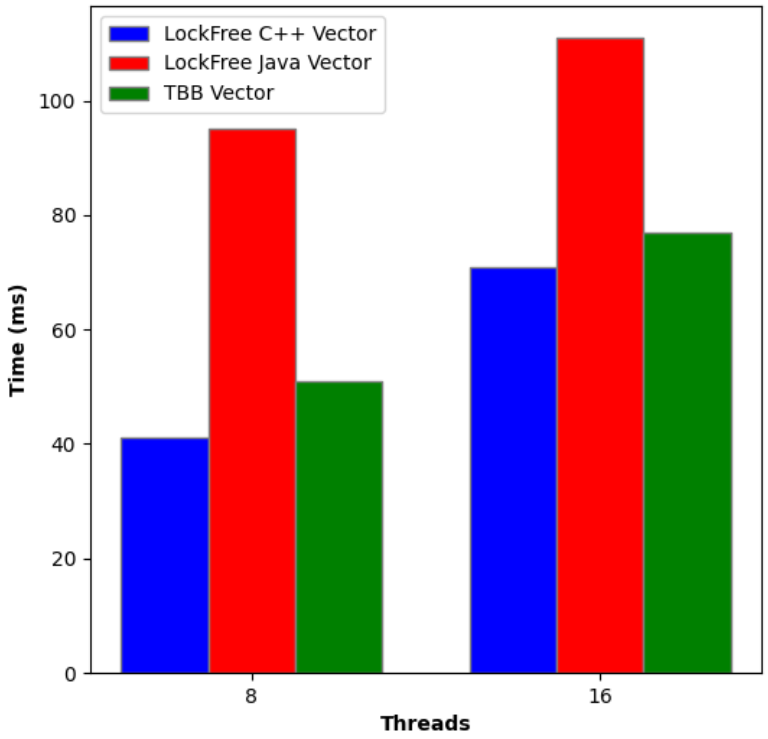


Figure 7: Time measured in milliseconds for Workload is 100% pushback in Lockfree Java, C++ and TBB vectors