Final Project Report- Hitesh Kumar Dasika (hdasika)

CSCI-B524: Parallelism in Programming Languages and Systems

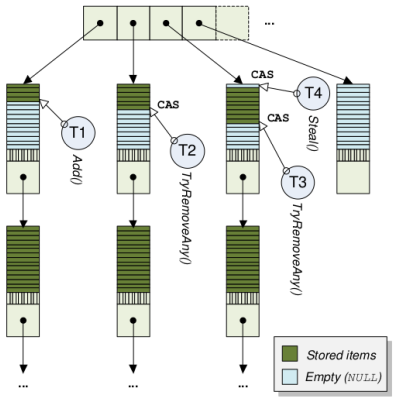
A Lock Free Algorithm for Concurrent Bags

This paper talks about designing an algorithm that supports multiple producers consumers and as well as dynamic collection sizes. The algorithm was designed to thrive for disjoint access parallelism and also exploit distributed design for the algorithm. Producer consumer collections are frequently used in pipelined design patterns for parallel applications. Bag data structure is used when insertion order is of no importance. Hence this algorithm can be used in applications which don't require any particular order to put data or remove data from the data structure.

The claims made by this algorithm are:

- It has distributed design which allows for disjoint access parallelism
- It makes use of the thread local storage
- It is dynamic in size and it also provides lock free memory management
- It only requires atomic primitives available in contemporary systems

The algorithm used in this paper is bag and it is implemented by using a linked list of arrays where each thread is working on its own array block. The data structure is dynamic because new array blocks can be added to the existing blocks dynamically . Hence each thread is ideally working on a linked list of array blocks.



The algorithm is designed to use thread local storage as mentioned in the claims. So each thread has some variables which are local to them. These thread local variables store the specific locations in the array blocks where the insertions or deletions are happening.

The algorithm also talks about disjoint access parallelism(DAP). DAP states that there should be no contention on shared objects when two transactions are accessing disjoint data sets. The algorithm has a very distributed design and each thread has its own part of area to work on and hence it followed DAP.

The algorithm makes use of CAS operations when updating any state or any variable related to the data structure. The CAS operations can be easily implemented in any contemporary systems.

Implementation

As this algorithm is designed around producer consumer pattern, two major functions it provides are Add and TryRemoveAny.

Add

This function is used to add an item to the data structure. Each thread has its own block to add the item. The thread has local variables which store the threadHead and threadBlock values. With the help of these values, the next position to add the item can be identified.

Algorithm 1 $Add(item)$

```
1: if threadHead has reached end of array then
2:     Allocate new block and add it in the linked list before threadBlock
       and set threadBlock to it
3:     threadHead ← 0
4: end if
5: threadBlock[threadHead] ← item
6: threadHead ← threadHead + 1
```

The edge conditions which needs to be handled is when the block is full. A new block needs to be added and the local pointers in the thread needs to be updated. This is also handled using CAS operations. The advantage for this algorithm is that at a time only one thread will try to insert into the block. There can be many consumers, but producer is only one for that block.

TryRemoveAny

This method is used to remove items from the bag data structure.

Algorithm 2 $TryRemoveAny()$

```
1: loop
2:     if threadHead < 0 then
3:         if threadBlock is last block in linked list then
4:             return Steal()
5:         end if
6:         threadBlock ← next array block in list
7:         threadHead ← last array position
8:     end if
9:     item ← threadBlock[threadHead]
10:    if item ≠ NULL and
       CAS(threadBlock[threadHead], item, NULL) then
11:        return item
12:    else
13:        threadHead ← threadHead − 1
14:    end if
15: end loop
```

Algorithm 3 $Steal()$

```
1: loop
2:     if stealHead has reached end of array then
3:         stealBlock ← next block in linked list or next list
4:         stealHead ← 0
5:     end if
6:     item ← stealBlock[stealHead]
7:     if item ≠ NULL and
       CAS(stealBlock[stealHead], item, NULL) then
8:         return item
9:     else
10:        stealHead ← stealHead + 1
11:    end if
12: end loop
```

It also takes the help of the local variables in the thread local storage to get the current state of the referenced data block. If a thread referencing a block runs out of data to consume, it should not wait and hence there is another function called steal which goes to other neighboring array blocks and steals

the elements from that block. There are thread local variables that are stored for stealing. They store the most recent place from which the stealing happened. The above mentioned pseudo code clearly elaborates how stealing happens.

Delete Block

This algorithm boasts about lock free memory management and rightly so it has special mentions in the algorithm about how to free up the memory in the program. When a block is deleted, the memory should be freed up else there will be other threads which will try to access the block. Hence the authors suggests to use hazard pointers. Hazard pointers are set when a thread visits a block. Hence memory is not relinquished until all the hazard pointer references are removed.

Not only that, when a block is being removed, there is a chance due to steal operation, some other thread might try and access it. Hence the algorithm makes use of two pointers that are mentioned for every block. When a block is empty, the thread first sets one of the flag in the prev block indicating that the next block is ready to be removed. The thread which first notices the flag will try and remove the block by changing the references. The second marker is used when the thread other then designated thread steals element from the block. When the block becomes empty, the other thread sets the flag telling the designated thread that this block should be removed and you should update your thread local variables.

Algorithm 4 $DeleteBlock()$

```
1: if stealPrev ≠ NULL then
2:     if CAS(stealPrev.next, stealBlock, stealBlock + mark2) then
3:         Set mark1 on stealBlock.next using CAS
4:         if stealBlock.next has mark2 then
5:             Set mark1 on stealBlock.next.next using CAS
6:         end if
7:         repeat
8:             if stealPrev.next is not referencing stealBlock then
9:                 UpdateStealPrev()
10:            end if
11:        until stealPrev = NULL or CAS(stealPrev.next, stealBlock + mark2, stealBlock.next − mark1)
12:        stealBlock ← next block in linked list or next list
13:    end if
14:    UpdateStealPrev()
15: else
16:    stealPrev ← stealBlock
17:    stealBlock ← next block in linked list or next list
18: end if
```

My Implementation

As the paper suggested that the algorithm can be implemented in any contemporary platforms, I tried implementing in Java. I was able to simulate all the features that are mentioned in the algorithm. Java also provides Atomic operations with the help of CAS. I made use of those for CAS operations. For data structure, I made use of a linked list from the collections framework. The linked list consists of Block objects. Block objects have data elements called nodes of specific block size I.e. an array of node elements and variables for marking while deletion.

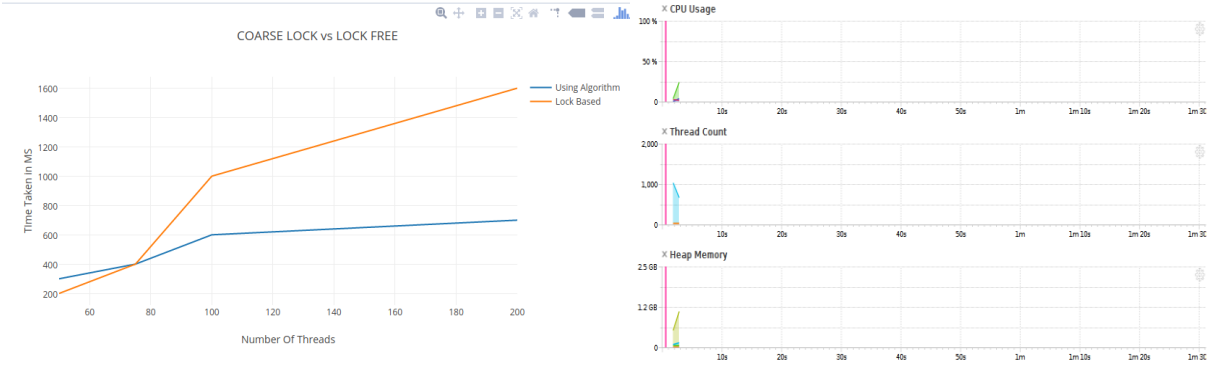
Review on the Paper

This paper talks about implementing a producer consumer paradigm on a bag data structure. I did not find a good use case for a specific bag data structure in the parallel design paradigm. There are many different ways in which the aforementioned operations in the algorithm can be handled. For example, in delete case, we can just maintain a global count variable and whenever a block is deleted, the count can be changed instead of using hazard pointers and marking them. Instead of using a linked list like structure, where the next pointer is needs to be handled, a dynamic array can be used. It reduces the number of points that needs to be handled. This is my opinion on the algorithm and I am not sure if I am right or wrong as I have not coded it.

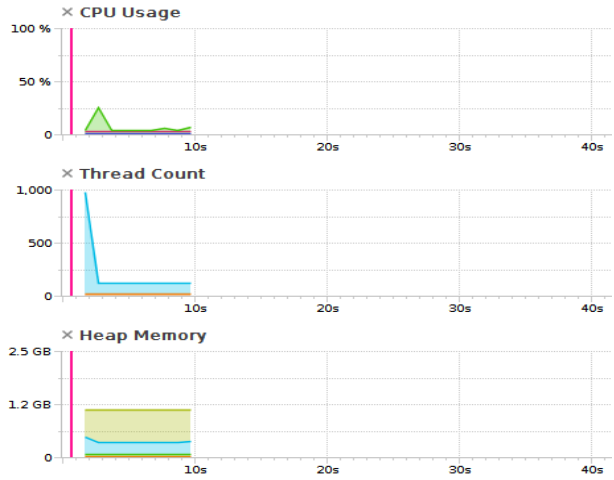
Benchmarking

The plan was to compare the algorithm with a lock based algorithm and analyze the performance. According to the plan, a lock based algorithm for a producer consumer problem which uses same kind of data structure was implemented. The locked algorithm implements reentrant lock. It puts data into the queue.

This graph is made with the help of data generated by having 50% producers and 50% consumers. You can see here The graph on the right shows the CPU usage , memory usage at the time when all the threads are active. It shows that CPU Usage is not that high and the load is evenly distributed. Also, heap memory usage is pretty decent as the memory management is properly done in the algorithm.



The below graph shows the different characteristics when a lock based algorithm is being run. You can observer the clear difference between the CPU usage and memory usage here.



I have compared results between the two algorithms by varying the number of producers and consumers I.e. 1 producer N-1 consumers, N/2 producers N/2 Consumers,1 Consumers and N-1 Producers. In all the cases the graph looks similar and lock free is taking less time and hence I summarized my benchmarking results by showing a single graph and also showing the memory usage in different algorithms. I hereby conclude that the performance of a lock free algorithm is better when considering huge sets of data and also many number of threads. This is also evident in the memory and cpu usage as shown above.