Implementing and Verifying a Concurrent Heap Algorithm

Shouzheng Yang

Department of Computer Science and Engineering, York University
4700 Keele Street, Toronto, Ontario, Canada, M3J 1P3

Abstract. We implement an efficient concurrent heap algorithm in Java (jdk 1.6) originally proposed by Hunt *et al*. The algorithm successfully avoids deadlock and allows insertions and deletions proceeding in opposite directions as typical sequential heaps do. A bit-reversal technique is applied to scatter consecutive insertion accesses across the leaves of the tree. The code is carefully refined from the original pseudo-code to improve the level of concurrency. Our experimental results of running the implementation on Java PathFinder and the Intel Manycore Testing Lab further verify the correctness and evaluate the performance of the algorithm, respectively. In a word, although there is a great amount of overhead cost of the concurrent implementation, the algorithm can achieve more concurrency as the number of cores or threads increases.

1 Introduction

Heap, as one of the widely used data structures, plays very important roles in a variety of algorithms such as branch and bound [1], sorting, multi-processor scheduling and so on. A binary heap is a binary tree in which the key (priority value) at any node is higher than that of any of its child node(s). Many basic data structures such as array and linked-list can be used to store a binary heap. The paper [2] discusses a concurrent algorithm of binary heap based on an array due to the convenience of arrays for implementation. In an array, the root of the heap occupies position 1, and the left and the right child of any node at position i occupy position 2i and 2i+1, respectively. Items can exist in level L only if level L-1 is full. The basic operations on a binary heap are insertion (enqueue) and deletion (dequeue), which are inserting a new item into the heap and returning the highest priority item from the heap, respectively. In a typical sequential implementation, both insertion and deletion can be achieved within O(log n) time on an n-item heap.

We implement the algorithm in Java with the help of a few concurrency mechanisms but mainly relying on the `java.util.concurrent.locks.ReentrantLock` class for this lock-based algorithm. A few efforts have been made to improve concurrency both in the scope of the algorithm design and the implementation phase. For example, the code is highly refined and no lock will be acquired until we do immediately need it. We also verify some of the key issues in the algorithm to ensure the correctness of the algorithm and explore some important issues in

the algorithm such as a particular way of using locks, etc. In this paper, we will describe the algorithm in detail, discuss our implementation and experimental methodology, analyze the testing results and show our efforts of exploring Java PathFinder to verify the implementation.

2 Related work

A number of concurrent algorithms on binary heaps based on different techniques have been proposed so far. As pointed out in [2], Biswas and Browne [3] proposed a method relying on the presence of a FIFO work queue to store sub-operations placed by processes performing insertions and deletions on a heap. In this way, the FIFO work queue allows consecutive insertions and deletions by a simple window-locking scheme. Rao and Kumar [4] have introduced a scheme that manipulates insertions in a top-down direction, the same as the direction of deletions, in order to avoid deadlocks. Jones [5] has proposed a concurrent priority queue algorithm using a skew heap whose binary tree representation does not require that all the intermediate levels are fully filled. Some researchers focus on non-blocking algorithms, either lock-free or wait-free. Such algorithms ensure that at least one operation will make progress no matter the contention or interleaving caused by concurrent operations. For example, Israeli and Rappoport [6] have proposed a wait-free algorithm for concurrent priority queues, which makes use of strong atomic synchronization primitives. Barnes [7] attempts to introduce another wait-free algorithm that uses existing atomic primitives. Another method of executing many insertions (or deletions) in parallel can be achieved by using pipelining [8, 9], which is not suitable for frequently alternating insertions and deletions. Sundell and Tsigas [10] present a lock-free priority queue that is based on a sorted skip-list. It employs an aided strategy that allows a task to continue even though another task has unfinished work. In addition, a new programming paradigm called STM [11] has recently introduced a new concept for handling concurrency. The basic idea of STM draws from the experience of transactions. Transactions are atomic blocks. If a conflict between two transactions occurs, the computation is discarded and the blocks are re-executed from the beginning. In the paper [12], a STM-based binary heap has been implemented and evaluated.

3 The Algorithm

Basically, the algorithm is a lock-based concurrent heap algorithm. It places mutual-exclusive locks on the heap's size and each node of the heap. A tag is associated with each node indicating its state. States can be `EMPTY` (currently the node is NULL), `AVAILABLE` (normal state by default), or in a transient state denoted by the Process Identifier `(pid)` of the inserting process if the node is being carried on the way to its place.

The two basic operations, deletion and insertion, are both processed in the same directions as a typical sequential heap algorithm does. A delete operation

starts by popping the root node and replacing it with the one in the rightmost position in the lowest level of the heap. Then the delete operation will "heapify" the heap. To handle concurrency, locks on individual nodes are placed during the phase of "heapifying." In each step, it first acquires the lock on the root node of the sub-tree and then the locks on its left and right child's will be further acquired. If swapping is not performed, the delete operation is completed. Otherwise, the lock on the swapped child is retained for the next step while the locks on the parent and the un-swapped child are released. This series of steps will be continue until swapping is not necessary or a leaf node has been reached. An insert operation first adds the newly inserted node to the leftmost vacant position in the last level of the heap. It then compares the priority of the inserted node with its parent's and performs swapping if necessary. To handle concurrency, each step of the bottom-up comparison, the lock on the parent node is first acquired followed by the lock on the inserted node. Both locks will be released after comparison and swapping. In this way, although the lock on the inserted node always needs to be re-acquired again in every step, locks are acquired in the same order as the delete operation in order to avoid deadlocks.

In addition, the algorithm employs a bit-reversal technique [13] to reduce lock contention. In a typical definition of a heap, nodes in the lowest level are compact to the left. In the algorithm, it relaxes this restriction since this does not affect any main property of binary heaps. Under the relaxed model, consecutive insertions will traverse different sub-trees. For example, in the third level of a heap (nodes 8-15, where node 1 is the root), eight consecutive insertions would start from the nodes 8, 12, 10, 14, 9, 13, 11 and 15, respectively. Hence, lock contention is reduced because any two consecutive insertion paths do not have any common node other than the root. This insertion sequence is achieved by reversing the binary representation (without the first bit) of the current heap size plus one.

Above description is the main idea of insert and delete operation. Details of the algorithm such as the usage of tags will be discussed in detail in Section 3.2 via analysis of the implementation.

3.1 The ReentrantLock

As can be seen from the related work section, different heap algorithms require different techniques of concurrency. A number of alternative techniques are available in Java such as semaphore, monitor, compare and swap, etc. Since the algorithm is based on locks, `java.util.concurrent.locks.ReentrantLock` is the main mechanism that the implementation relies on. According to the documentation of the Java class library[1], the ReentrantLock implements the interface Lock in Java and provides the same functionalities as synchronized methods and blocks, but with extended capabilities. Here, the same functionalities refer to two aspects. One is "visible" which can guarantee that variable modification is

[1] See `download.oracle.com/javase/6/docs/api/java/util/concurrent/locks/ReentrantLock.html`

immediately visible to all threads. Since each thread has its local memory, visibility is important for ensuring that threads are always reading the most recently updated values. The other is mutual exclusion.

Besides these, the ReentrantLock constructor offers a "fairness" option. By default, it is unfair. If fair is set, threads are able to obtain a lock in the order that they request. In this way, a newly requesting thread may wait for all the other previous threads that wait on a particular lock but the lock could be currently free. This situation happens that when the previous requesting threads tried to acquire the lock, the lock was somehow being held, and when the lock becomes free, the processors have not turned to any of these previous requesting threads. Therefore, the unfair option results in better performance. Since our algorithm does not rely on fair queuing, the implementation of the algorithm uses the "unfair" option to avoid significant performance cost due to the overhead of suspending and resuming jobs.

3.2 Algorithm Details

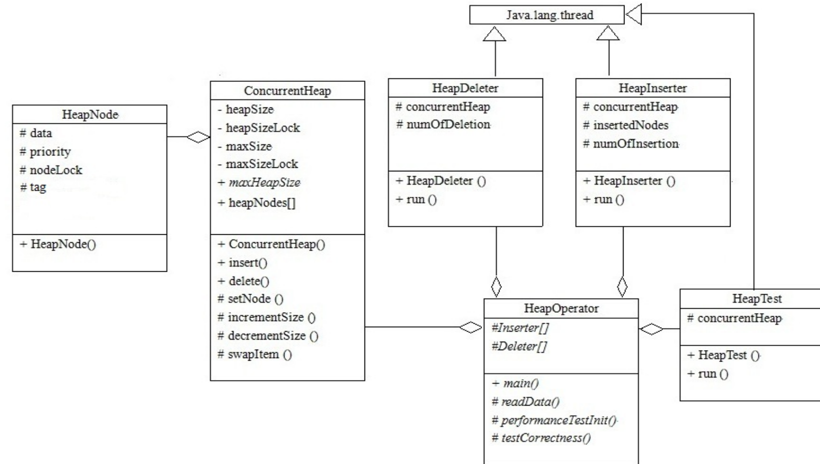

Figure 1

Figure 1 shows the class-diagram as an overview of the implementation. ConcurrentHeap is the class that supports concurrent insertions (insert()) and deletions (delete()) of an array-based binary heap. It associates a number of heap nodes, which are defined by the HeapNode class and these heap nodes form the array-based binary heap data structure in the ConcurrentHeap class. Each node has its own data, priority, tag and node lock. The heap size and the corresponding size locks are further declared in the ConcurrentHeap class. As the bit-reversal technique is applied, nodes in the lowest level of the heap tree may not be compact. MaxSize is a variable used for possible maximal size of the heap tree. The variables size and maxSize are maintained in the methods incrementSize() or decrementSize() where the bit-reversal technique is applied.

Besides the description of the basic ideas of insertion and deletion mentioned before, a number of cases need to be considered in order to ensure the correctness

of the algorithm as well as avoiding deadlock. As this is the most intricate part of the algorithm, we explain it using code, in particular, the use of tags for insertion.

```
heapNode[parent].nodeLock.lock();
if (heapNode[parent].tag==HeapNode.EMPTY){
    // ......
    heapNode[parent].nodeLock.unlock();
    hasParentLock=false;
}
else if (heapNode[parent].tag==Thread.currentThread().
   getId()){
    // ......
    heapNodes[parent].nodeLock.unlock();
    hasParentLock=false;
}
else if (heapNode[parent].tag!=HeapNode.AVAILABLE){
    // ......
    heapNodes[parent].nodeLock.unlock();
    hasParentLock=false;
}
else hasParentLock=true;
heapNode[child].nodeLock.lock();
if (heapNode[child].tag==Thread.currentThread().getId()){
    if (hasParentLock) ...
    else ...
}
else if(hasParentLock)heapNode[parent].nodeLock.unlock();
heapNodes[child].nodeLock.unlock();
```

As can be seen from the code snippet, the usage of tags for insertion is in the following manner.

1. If the tag of the parent node is EMPTY, the insert operation is completed because the inserted item must have been moved to the root of the heap. So we terminate this insert operation.
2. If the tag of the parent node is the current thread's pid, the inserted node must have been moved one position upwards by a delete operation. So we move the inserting thread one position upwards in pursuit of the inserted item.
3. If the tag of the parent node is NOT AVAILABLE, that means the value must be a pid of another thread. In this case, we release the lock on the parent node and give the upper inserting thread priority to finish its insertion first.
4. If all above cases are not met, the tag of the parent node must be AVAILABLE. We then try to acquire the locks of its child's nodes. If the tag of the child node is the current thread's pid, then no interference has occurred and the insertion operation can proceed normally. Otherwise, we move the

inserting thread upwards because the inserted item must have been moved by delete operations upwards at least twice.

From the above code snippet we can also see one of the efforts made from the original paper to improve concurrency. In the pseudo-code written in the paper, a child lock will be soon acquired after the acquisition of its parent node. This does no good to concurrency. Hence, in our implementation, only essential statements are being put within the lock holding period and no lock will be acquired until we do really need them.

The usage of tags for deletion is much more straightforward. A lock of a node can be acquired as long as it is not EMPTY. We do not need to consider other cases for deletion since insert operations have addressed all the conditions. Another point in the algorithm that needs to be emphasized is that we don't release the lock of the heap size until we have acquired the lock of the first corresponding manipulated node. For insertion this means we need to acquire the node at the inserting position before we release the lock of the heap size, while for deletion this refers to the node that is going to replace the root in the first step. This guarantees that the heap size is kept unchanged before it starts working on the right node. A verification test is done to check the importance of obeying this lock manipulation sequence in Section 4.5.

3.3 Other Implementation Related Concerns

In addition to the implementation of a heap, HeapOperator, as shown in the class diagram (Figure 1), is the class which contains the main method. It acts as a coordinator class. Within the main method, it employs a concurrent heap as well as a number of instances of HeapDeleter and HeapInserter objects. Each of these deleters or inserters, inherits from java.lang.Thread class, and has its own run() method to invoke the insert() and delete() methods written in the ConcurrentHeap. Hence, in class HeapDeleter and HeapInserter, there need to be a reference to the concurrent heap. The variables "numOfDeletion" and "numOfInsertion" specify the number of deletions or insertions that are performed each time when a new thread is created. Furthermore, a number of inserters and deleters will be created in the main method and start their threads in the main method.

Apart from using reentrant locks, other concurrency mechanisms used in the implementation are the `volatile` keyword and atomic variables. The `volatile` keyword ensures the declared variable always to be visible to all threads. Atomic variables support lock-free thread-safe programming on single variables and they are lighter-weight in terms of performance compared to synchronized blocks or reentrant locks. Most hardware infrastructures nowadays also support such atomic operations. In the implementation, atomic variables are used for the purpose of testing such as time counter, etc.

Also, CyclicBarrier is applied in the implementation to increase concurrency for testing purpose. Threads of inserters and deleters are waiting for each other until they all reach a common barrier and then start their run() methods. In

addition, in the main() method, the program waits for all inserters and deleters finishing their jobs before starting the next round of testing.

4 Testing and Verification

4.1 Check Heap Property During Runtime

As shown in the class diagram (Figure 1), we implement the HeapTest class which aims at checking the heap property during runtime. Basically, it creates another thread and always checks whether the heap still maintains the definition of a heap while the program is running. More concretely, it checks whether a parent node has higher priority than its children's priority during the runtime for any nodes. Since there may be some nodes under processing at the time that HeapTest is running, HeapTest ignores those nodes with tag value other than AVAILABLE. With regard to the implementation of the HeapTest class, it is designed based on ReentrantLock like the ConcurrentHeap class.

4.2 Verification Preparations

In order to verify the algorithm and our implementation, we employ Java PathFinder[2](JPF) and test a few key issues in the algorithm. Before we start running JPF, we make a few preparations in order to verify the code within a reasonable running time since the state space which JPF relies on for searching different interleavings and variable states is quite large. All the following steps are aiming at reducing the state space. First, we either remove unrelated variables for each specific experiment or use the `FilterField` annotation in JPF to filter some running variables that are not related to the core of the algorithm, such as loop counters, etc. Second, we decrease the number of nodes to be at most twenty nodes in the heap and most of our experiments are running with about five nodes. Finally, the experiments that we have made are based on at most two inserter threads and two deleter threads, and for each thread, involves no more than five operations, either insert or delete. From the class diagram we can see that it is easy to make these changes by setting certain values to the variable such as "numOfInsertion", etc.

4.3 Data Race Detection

A data race always occurs when one event makes a change to its next state before a second one has had sufficient time to latch. As far as we know, in terms of bad programming, there are two possibilities that could lead to a data race. The first one is lack of a correct timing mechanism, especially an inappropriate synchronization manner, so that events could happen in unpredictable orders. For example, this could result in an ambiguous order of reading/writing a particular piece of data. The second possibility relates to visibility. Even if events

[2] `http://babelfish.arc.nasa.gov/trac/jpf/wiki`

are executed in a desired way, a data race could still occur if a certain piece of data is not immediately visible to other threads after a change has been made to it because it is not guaranteed that the reading thread will see a value written by another thread on a timely basis in Java. Therefore, based on these two possibilities, we try to find data races in the algorithm.

Inappropriate Scheduling We first consider an inappropriate scheduling as the reason leading to data races. In the algorithm, besides the program flow, the synchronization manner plays the role to schedule the threads and ensure the correctness of the executing order of manipulating the heap during runtime. So, our experiment first runs JPF with the `PreciseRaceDetector` listener to detect any existing data races. Nothing is found since whenever the algorithm needs to access a node, it locks the node to guarantee both mutual exclusion and visibility. We then remove the lock of the root node of the heap and run JPF again in order to show the importance of the lock. Not to our surprised, JPF successfully finds a data race at the root node. As we can see in Figure 2, two threads are executing two delete operations at the same time. Thread-1 tries to replace the root node value with the rightmost node in the last level of the heap. So it performs a write operation indicated as `putfield`. Meanwhile, thread-2 tries to read the tag value of the root node at one of the very first steps of deletion to check if the root is EMPTY, indicated as `getfield`. Hence, a data race is found from this ambiguous order of read/write of the root node.

```
====================================================== error #1
gov.nasa.jpf.listener.PreciseRaceDetector
race for field shouzheng.cool6490a.HeapNode@17c.tag
  Thread-1 at shouzheng.cool6490a.ConcurrentHeap.swapItem(ConcurrentHeap.java:271)
              "(shouzheng\cool6490a\ConcurrentHeap.java:271)"  : putfield
  Thread-2 at shouzheng.cool6490a.ConcurrentHeap.delete(ConcurrentHeap.java:151)
              "(shouzheng\cool6490a\ConcurrentHeap.java:151)"  : getfield
```

Figure 2

Non-instant Visibility In the second experiment, we try to run JPF to find any program bug regarding to visibility. Since the ReentrantLock and most of the other synchronization mechanisms in Java support instant visibility, the `volatile` keyword is only used in the program to assist the output generation. More concretely, in the program, we define a volatile output array to record the time spent per 10000 insertions for performance measurement. Each time a 10000-insertion is finished, we record the elapsed time into the output array and increment the array subscript index (current size). No lock is involved to ensure synchronization for the array but the subscript index is set to be of type AtomicInteger. (In this way, there hides a bug actually but the program keeps running correctly. We will discuss the bug at the end of this section.) So, we remove the volatile keyword and try to test if JPF can find a data race.

The answer is no. After careful consideration, it makes sense because a data race requires read/write or write/write performed simultaneously. This does not exist in our program for the output array. Furthermore, the program contains a mechanism such as the ReentrantLock which always updates the threads' local memory. This can be a side-effect that influences our test for finding a data race related to missing the volatile keyword. However, out of curiosity how JPF reacts to the `volatile` keyword, we have further designed a simple example as follows.

```
public class VolatileDataRace {
    // without volatile keyword
    static int v=0;
    public static void main(String args[]) {
        v= 1;
        new MyThread().start();
    }
    public static class MyThread extends Thread {
        public void run() {
            assert v==1;
        }
    }
}
```

In the example, if the variable is not set to be volatile, v=1 could be either visible or invisible to MyThread during the runtime when it prints out the value of v. So, we expect that JPF could find an error in the above example, but JPF detects nothing. Due to lack of proper documentation, we don't know if JPF sets up multiple states to correlate the two versions of variables in main memory and local memory, but anyway, as far as we understand, this might be a future improvement for JPF.

Finally, regarding the hidden bug, the program seems to always run correctly because the subscript index (AtomicInteger) is obtained and incremented in an atomic way. Furthermore, it is very unlikely that the program accesses the same array slot twice to write the time spent for two consecutive 1000-insertion before the subscript gets incremented because a 10000-insertion takes much longer time. However, in theory, this is a bug that may happen depending on the Java scheduler. In order to produce the bug, we set the interval sufficiently small, i.e. one insertion. Errors due to simultaneously write/write, can be seen if we add some assertions of the array and run JPF to enumerate all different interleavings.

However, to our surprise, although JPF can successfully enumerate all different interleavings, it still cannot report the data race. Later, we found the fact that JPF reports data races on a single variable but not on array elements unless a certain property is set, because by default JPF does not want to cause a serious state space explosion. Finally, JPF is able to detect the data race on this array after we add the property `"cg.threads.break_arrays=true"` for the data race listener.

4.4 Deadlock Detection

We run our program in JPF with the listener `DeadlockAnalyzer`. As expected, no deadlock is found since the algorithm is designed with particular concerns on avoiding deadlock. More precisely, although the fact that concurrent insertions and deletions proceed in opposite directions, they all acquire the parent node lock and the child node lock in the same order. In addition, the insert operations release the parent node lock and child node lock at the end of each parent-child comparison iteration and re-acquire the parent node lock again which is the child node lock in the next iteration to avoid deadlock.

Therefore, based on the algorithm, we attempt to verify that releasing and re-acquiring the parent node lock after swapping at each iteration is essential to guarantee deadlock freedom. According to this, we simply remove the corresponding code that releases the parent node lock if a swap is performed, and JPF successfully locates the code stuck in the insert() and delete() methods that cause the deadlock.

```
====================================================== thread ops #1
   3       2       0     trans    insn         loc              : stmt
------- ------- ------- --------------------------------------------------
W:457     |       |       7      invokespecial java\util\concurrent\locks\ReentrantLock.java:49 : (java\util\concurrent\locks\ReentrantLock.java:49)
  |     W:508     |       6      invokespecial java\util\concurrent\locks\ReentrantLock.java:49 : (java\util\concurrent\locks\ReentrantLock.java:49)
  |       |     W:443     5      invokevirtual shouzheng\cool6490a\HeapOperator.java:133 : (shouzheng\cool6490a\HeapOperator.java:133)
  |       S       |       4
  S       |       |       3
```

Figure 3

Further, we attempt to combine JUnit[3] and JPF to detect deadlocks. A code snippet is shown below:

```
public class JPFInJUnit extends TestJPF{
    public static void main(String[] testMethods){
        runTestsOfThisClass(testMethods);
    }
    @Test public void Test1()...
    @Test public void Test2()...
    @Test
    public void TestDeadLock()
    {
        if (!isJPFRun())
        {
            /* running outside the VM of JPF */
        }
        if (verifyDeadlock("+listener=.listener.
            DeadlockAnalyzer",
             "+deadlock.format=essential",
```

[3] www.junit.org

```
            "+report.console.property_violation=error,
               trace,snapshot"))
        {
            // Invoke threads for insertion and deletion.
            System.out.println("There is no deadlock!");
        }
        else System.out.println("A deadlock is found!");
    }
}
```

First, the JUnit test class needs to extend `gov.nasa.jpf.util.test.TestJPF` in order to combine them together. Second, we need to create a main method with the statement runTestsOfThisClass(command line arguments) inside. These two steps together ensure that all tests are running in a JUnit-JPF combined environment. Third, a few verification methods such as deadlock detection, unhandled exception detection and property violation are available in JPF for JUnit test. Listeners can be further specified to perform the verification. Fourth, as in the code snippet, isJPFRun() is a method provided by JPF to give options to specify whether running the code inside the normal JVM or the VM of JPF.

Finally, we test the deadlock problem again that releasing and re-acquiring the parent node lock after swapping is essential to guarantee deadlock freedom. The same result is reported by JPF as in Figure 3.

4.5 Testing With the Help of JPF

In this section, we show an example how can we conduct better testing with JPF's power of state space searching. As mentioned in Section 3.2, there is an important point in the code, where the lock of the heap size is released after the lock of the next corresponding manipulated node is acquired. We can easily understand this from the code snippet of the delete() method.

```
heapSizeLock.lock();
// Apply the bit-reversal technique to calculate
// the corresponding node index that will be used
// to replace the root node.
bottom=decrementSize(heapSize);

heapNodes[bottom].nodeLock.lock();
heapSizeLock.unlock();

// Retrieve bottom node information.
heapNodes[bottom].nodeLock.unlock();
```

Failure of this locking sequence may result in an error. An example of the error is shown in Figure 4. From the figure we can see that a delete operation pops the root node and tries to replace it with node-3 which is indicated as the "bottom" node in the code. If we switch the two highlighted lines (lock bottom node and release heap size lock) in the above code, it could happen

that, right after the heap size lock is released, the Java scheduler turns to an insert operation and the insert operation will put the newly inserted node in the position of node-3. Therefore, we loose the node with value 2 in this example.

As far as we know, it is not easy for JPF to directly report such bugs in the code because JPF does not know the specific idea of the algorithm. In addition, this bug occurs without any deadlock or data race phenomenon, etc. However, we can still find such bugs with the help of JPF. More concretely, we add an assertion of the heap size to the program when JPF enumerates all interleavings. For this example, the assertion of the remaining nodes that equals to three fails. Hence, the capability of state space searching provides JPF additional strong power for program testing.

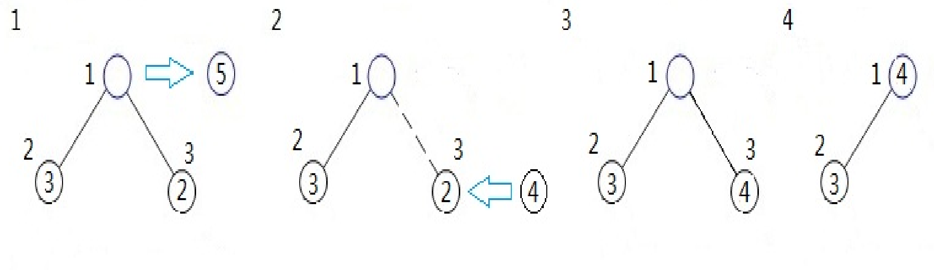

Figure 4

5 Algorithm Performance Evaluation

Evaluation for performance is performed on the Intel Manycore Testing Lab (MTL) with memory set to be 12G. Two experiments are designed. Each experiment takes 100 rounds for testing and the first few results are discarded in order to give the JVM a chance to optimize the code. The Java garbage collector is called before each round to eliminate some side effects. Experiments are as follow.

The first one evaluates the performance of the algorithm under different levels of contention. Figure 5 shows the result. In all experiments, cores are equally loaded. We studied the performance of insertion-only, deletions-only and mixed insertion/deletion workloads. We set the number of threads always to be four times of the number of cores. In each round, every thread performs 10000 insert or delete operations. In insertion/deletion mixed workload, half of the threads are created for insertion and the other half are for deletion. The average throughput is measured per second.

As shown in Figure 5, insertion-only performs faster than deletion-only. One reason is that an insertion traverses less height of the tree in average compared to a deletion because deletions are likely to traverse the whole height of the tree. In addition, deletions suffer more from contention on the topmost nodes of the heap. This point is not only one of the reasons why insertion-only outperforms deletion-only, but also the reason why deletion-only increases slower and slower as the number of cores grows. In the case of alternating insertion and deletion,

contentions are raised due to both top-down and bottom-up direction operations. In addition, after a number of insertion/deletion cycles, the nodes remaining in the heap tend to have low priorities, so newly inserted nodes have to traverse longer and longer paths toward the root, which will reduce the throughput. This affects the line for insertion/deletion mixed in the diagram to become a little bit flat when the workload is increasing.

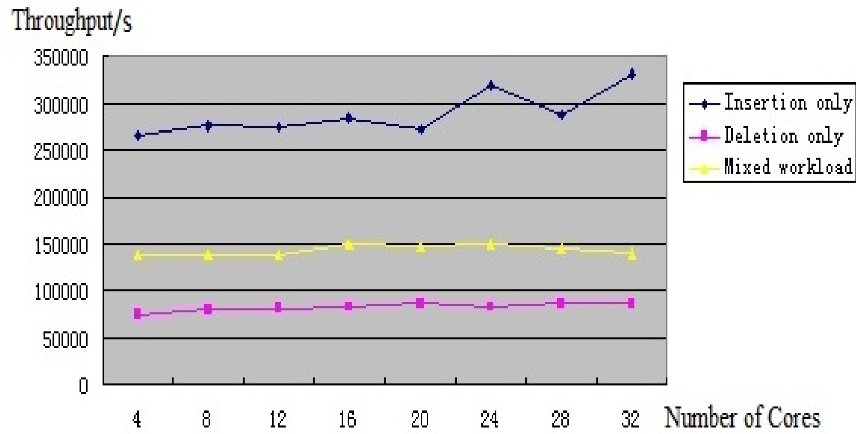

Figure 5

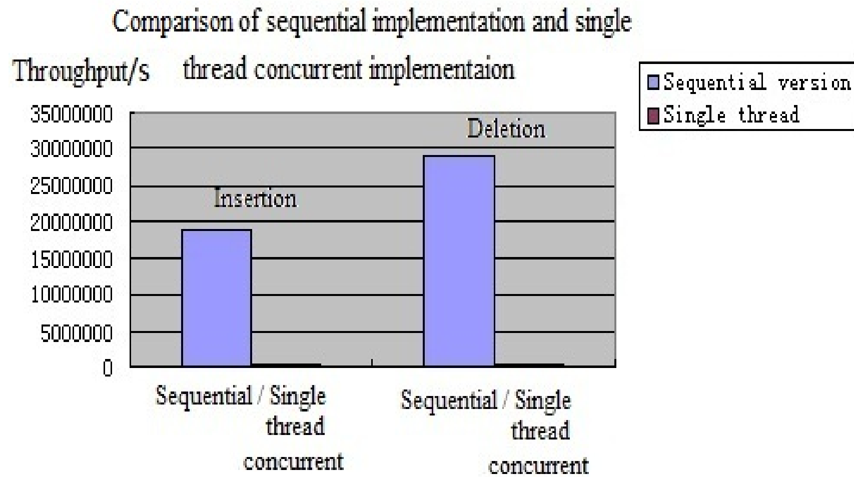

Figure 6

Anyway, one positive aspect from the result is that all of the three tests show that as the number of cores increases, the algorithm can achieve more or less concurrency. This confirms the result shown in the original paper [2].

Figure 6 shows the result of the evaluation of overhead when using our concurrency mechanism. Besides a few facts such as concurrent implementation

requires more resource consumption can help to explain the high overhead, the cost of acquiring/releasing locks is undoubtedly the most essential reason for this result. Another clue which was found after the experiments may also confirm the reason. Since the main concurrency mechanism applied for the implementation is based on the Java's ReentrantLock. The book [14, page 282], suggests that using ReentrantLock is not a good idea for performance reason because it is not built into JVM in Java 1.5 so that no JVM optimization is gained for this kind of lock mechanism in Java 1.5. However, to be honest, I am not able to find out if it has been built into Java 1.6 or not. In addition, JPF re-implements the ReentrantLock and we have also tested this version. No significant performance improvement has been observed in terms of 100000 lock/unlock operations.

6 Conclusion

In the paper, we have addressed the algorithm that uses multiple locks to allow consistent accesses to an array-based binary heap. We have also discussed our ways of verifying and evaluating the algorithm on the Java PathFinder and the Intel Manycore Testing Lab, respectively. Our experiments of running the program on JPF systematically verify the correctness of the algorithm and show the importance of some issues in our implementation such as the way of guaranteeing deadlock freedom. On the other hand, along with our exploration of JPF, we hope that JPF could be improved in the future for a few things including the capability of checking algorithms of a larger number of threads within an acceptable time. Furthermore, the evaluation experiments show that the algorithm achieves more concurrency when the number of cores or threads increases. It is a successful sign for a concurrent algorithm although there exists a great amount of overhead cost of the concurrent implementation. However, some of the overhead cost is due to the concurrency mechanisms in Java. We hope that, if these costs such as the cost of using locks can be reduced in programming languages like Java, then concurrent algorithms would become more and more efficient and applicable in the future.

6.1 Acknowledgements

Thanks to the management, staff, and facilities of the Intel® Manycore Testing Lab[4].

References

1. Mohan, J.: Experience with Two Parallel Programs Solving the Traveling Salesman Problem. In: Proceedings of the 1983 International Conference on Parallel Processing, IEEE (1983) 191

[4] www.intel.com/software/manycoretestinglab

2. Hunt, G., Michael, M., Parthasarathy, S., Scott, M.: An efficient algorithm for concurrent priority queue heaps. Information Processing Letters **60**(3) (1996) 151–157
3. Biswas, J., Browne, J.: Simultaneous update of priority structures. Technical Report DOE/ER/25010-T2, Texas Univ., Austin (USA). Dept. of Computer Sciences (1987)
4. Rao, V., Kumar, V.: Concurrent Access of Priority Queues. IEEE Transactions on Computers **37**(12) (1988) 1657–1665
5. Jones, D.W.: Concurrent operations on priority queues. Communications of the ACM **32**(1) (1989) 132–137
6. Israeli, A., Rappoport, L.: Efficient Wait-Free Implementation of a Concurrent Priority Queue. In: Proceedings of the 7th International Workshop on Distributed Algorithms, Springer-Verlag (1993) 1–17
7. Barnes, G.: Wait-free algorithms for heaps. Computer Science and Engineering, University of Washington, Tech. Rep TR-94-12-07 (1994)
8. Quinn, M., Yoo, Y.: Data structures for the efficient solution of graph theoretic problems on tightly-coupled MIMD computers. In: Proceedings of the 1984 International Conference on Parallel Processing. (1984) 431–438
9. Quinn, M.: Designing Efficient Algorithms for Parallel Computers. McGraw-Hill Companies (1987)
10. Sundell, H., Tsigas, P.: Fast and lock-free concurrent priority queues for multi-thread systems. In: Proceedings of the 17th International Parallel and Distributed Processing Symposium, IEEE Computer Society (2003) 609–627
11. Shavit, N., Touitou, D.: Software transactional memory. In: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing, ACM (1995) 204–213
12. Dragicevic, K., Bauer, D.: A survey of concurrent priority queue algorithms. In: IEEE International Symposium on Parallel and Distributed Processing, 2008., IEEE 1–6
13. Leiserson, C., Rivest, R., Stein, C.: Introduction to algorithms. The MIT press (2001)
14. Brian, G., Tim, P., Joshua, B., Joseph, B., David, H., Doug, L.: Java concurrency in practice. Addison-Wesley (2006)