Assignment 3

Spiros-Daniel Mavroidakos - 260689391
Jastaj Virdee - 260689027

December 4, 2018

Contents

1 Question 1

1.1

From the question, we know the following.

- We are reading L words from cache
- Cache Line is 4 words

L' is the number of words that can be stored in the cache given a maximum word seperation of stride s. A stride can be a maximum size of $\left\lfloor \frac{L}{2} \right\rfloor$. We also know that the cache can have several cache lines. We will denote the amount of cache lines by k.
Knowing this, we can come up with the following equation for L'.

$$L' = \left\lceil \frac{4k}{\left\lfloor \frac{L}{2} \right\rfloor} \right\rceil \tag{1}$$

Furthermore, t0 shows the cache access latency as it does take some time to read from the cache. Given that $L < L'$, the stride size will either be 0 or a size that fits within the case. Therefore, all the time delay for access comes from reading the case.

1.2

Given that L > L', the stride will be large enough that we cannot only read from the cache. The words will be too seperated to read only from cache. However, at t1, we have peaked in the maximum read time. This implies that the stride is larger than the actual cache size. That is why it is a constant line. t1 will therefore be showing the memory access latency since we are no longer reading from the cache. t1 might also take into account the time to read the cache as we need to generate a cache miss to read from other storage.

1.3

Part 1 is when L < L' so as stated in section 1.1, all of the data of the array will fit into the cache and will therefore be accessed at the time that it takes to read the cache. This is why it is the lowest of the 2 curves.
Part 2 is when L > L' however, the stride has not become so big that we cannot use the cache in a meaningful way. Throughout part 2, the stride is continuously increasing which is why the curve is increasing as you must read from cache and then main memory. At the end of part 2 (the peak value where it becomes constant) is where part 3 begins and when the stride has become so large that the stride takes up all of the cache and you will always incur a cache miss and have to then go to main memory which is why it is a peak. Part 2 has a mix between some values are in cache and some are not while part 3 is where everything is not in the cache.

1.4

If we were to pad the array such that distinct elements are mapped to distinct cache lines then the value of L' will surely be affected. From 1.1, we know that we have k cache lines and that every

cache line can contain 4 words. However, by padding we reduce the amount of words that a cache line can hold. It can now only hold 1 word. Therefore, we would get the following revised equation:

$$L' = \left\lceil \frac{k}{\left\lfloor \frac{L}{2} \right\rfloor} \right\rceil \tag{2}$$

Because of this, performance degredation will happen since the value of L' will be reduced. A smaller L' means that we will have more cache misses as we can store less words in the cache and therefore will have to read from main memory more often which is noticeably more costly than reading from a cache.

2 Question 2

2.1

All of the source code can be found in the appendix in the specified sections:

- The implementation for a node can be found in section A.1.
- The contains method implementation can be found in section A.2

2.2

All of the source code can be found in the appendix in the specified sections:

- The implementation for the contains Test can be found in section A.3
- The output of the test can be found in Section A.4.

The implementation of the contains method was done using a hand-over-hand locking style. This assured that you would lock the previous node and the current node. This is needed to ensure that when you are selecting the next node, that you do not get preempted by another operation that came after you. For example, if you only lock the current node, then it is possible that you unlock the current node and then you set your next node but you have not locked this node. Now, the thread gets suspended and another one comes along and removes the node you currently have selected. You resume and now you are on an orphan node. This is not safe and is why you need to lock both the previous and current. You release the previous and then lock the current node's next node. Like this, the current node blocks anyone from overtaking you.

The test implemented here if several threads tried to check for the existance of the same node. From the output, you can clearly see that hand-over-hand locking is occuring due to the order of locks and release. Furthermore, the locks are working as expected since no lock is being locked without an unlock occuring prior. The output clearly demonstrates a previous and current node being locked. Then the previous node is unlocked and the current node's next node is locked. It can also be noticed that the released previous node is then being locked by another waiting node. Since the add and remove methods that have been seen in class are implemented using the same hand-over-hand method, those operations cannot interfere with the contains method as they are designed to be thread safe.

The test setup is relatively simple, 3 nodes are set up each with a different value: Node a has a value of 1, node b has a value of 2 and node c has a value of 3. The head node's next value is set up to be node a. Node a points to b, b points to c and finally c points to the tail node which marks the end of the list. From the source code, the contains method will return false only if we reach the tail node as all other nodes have been checked for equality. For simplicity, each node is going to be searching for the value of 3. From the output, we can observe that each thread will find the value at a different time and not all at the same time. If the output would show lock duplication before an unlock or at least 2 threads finding the value conccurently then we would know that the implementation is flawed. However, this is not the case and therefore, the test passes.

3 Question 3

3.1

The contents of the (circular) array-based queue will be very similar to the one shown in class (which used a linked-list). Notable differences would be that the head and tail will be represented by integers, representing the index of the array. Both will be initialized to zero. The array will also need to be created using the 'int capacity' variable to set the size.
The equeue method will also be very similar in that both implementations will lock the 'enqLock' before attempting to enqueue an object, and will release it once it is finished. It will also wait while the queue is full and notify blocked dequeuers if the queue was empty. The difference comes when one attempts to enqueue an item. Rather than creating a new node, the item will be placed in the array at index '(tail+1)%capacity','tail' and 'size' will then be incremented. However in the case that the queue is empty, the item will be placed at index 'tail%capacity' instead.
Similarly, the dequeue method will lock the 'deqLock' before attempting to dequeue an object, and release it once it is finished. It will then wait while the queue is empty. To dequeue an object, the item at the 'head%capacity' index of the array will be saved, and later returned, 'head' will be incremented and 'size' will be decremented. However 'head' will not be incremented if the queue only had one item (and is now empty). Finally the method will notify the blocked enqueuers if the queue was full.

3.2

To perform a lock-free enqueue, one would have to perform a logical enqueue enqueue by performing a Compare and Set between the item at the 'tail' index and the new item. One would then have to perform a physical enqueue by performing a CAS between the 'tail' index and the index of the new item. However these two steps are not atomic, so the 'tail' index may either point to the actual last item or the penultimate item. For the original implementation using linked-lists, if one were to find a trailing tail one would fix it by checking if the tail node has a non-null next field, and then performing a CAS on the tail and tail.next field. This is where a problem arises when trying to use arrays because checking if the item at the index after 'tail' is not null does not not guarantee we have a trailing tail. Because we are using a circular queue, it is possible that the 'tail' index is in fact pointing to the correct index, but the next index may be non-null.
To solve this, one would have to switch to a linear, fixed size array. Enqueuing would consist of placing the item at the first index which has a null field. Dequeuing would consist of saving and later returning the item in the first index, and shifting all the other items to the previous position.

4 Question 4

4.1

The algorithm for sequential matrix vector multiplication involves computing each entry of the result vector by performing the dot product on the rows of the matrix with the vector. The code can be found in section B.1.

4.2

The algorithm used of parallel matrix vector multiplication involved splitting up the problem into multiple subtasks. Specifically, each subtask was responsible for calculating one entry of the result vector. To do this it, the dot product was performed on one row of the matrix and the vector. The work and critical path will be discussed in part 4. The code can be found in Appendix B.2.

4.3

To test the execution times of the sequential and parallel methods, a 2000 by 2000 randomly generated matrix was created as well as a randomly generated 2000-length vector. The time was recorded and the sequential method was executed. Once finished, the time was recorded again and the difference between the two times was calculated to determine the time of execution. The same was done for the parallel method, where 3 threads were used. The execution time of the sequential method was 0.142s, the parallel time was 0.109s. The speedup on P processors is the time taken by one processor divided the time taken by P processors. In this example it would be the time of the sequential method divided by the time of the parallel method. Hence the speed up on 3 processors is 1.30. The code can be found in section B.3.

4.4

Each subtask performs n multiplications and n-1 additions. There are n subtasks so the work is $n*(n+n-1)$ which is $giO(n^2)$. Since all the critical paths can be executed in parallel, the critical path would be the cost of one subtask, so $n+n-1$ which is $O(n)$. The parallelism is the work divided by the critical path: $O(n^2/n) = O(n)$.

A Question 2 Code

A.1 Node Implementation

```
1  import java.util.concurrent.locks.ReentrantLock;
2
3  public class Node<T>{
4
5      public ReentrantLock nodeLock = new ReentrantLock();
6
7      public T item;
8      public int key;
9      public volatile Node next;
10
11     public void lock()
12     {
13         this.nodeLock.lock();
14         System.out.println("Node with key = " + key + " locked");
```

```
    }

    public void unlock()
    {
        System.out.println("Node with key = " + key + " unlocked");
        this.nodeLock.unlock();
    }

}
```

A.2 Contains Implementation

```
import java.util.LinkedList;
import java.util.*;
import java.util.concurrent.*;
import java.util.concurrent.locks.ReentrantLock;

public class FineGrainedContains<T>
{
    public Node head = new Node();

    public Node tail = new Node();

    public boolean contains(T item){
        // basic locking for hand-over-hand locking
        Node pred = head;
        pred.lock();
        Node curr = pred.next;
        curr.lock();
        int key = item.hashCode();

        try
        {
            while (curr.key <= key)
            {
                // check if the item has been found
                if (curr.item == item)
                {
                    return true;
                }

                // Hand over hand locking
                pred.unlock();
                pred = curr;

                // end if the tail id reached
                if(curr.next != tail)
                {
                    curr = curr.next;
                    curr.lock();
                }
                else
                {
                    curr = curr.next;
                    curr.lock();
                    break;
                }
```

```
46              }
47          }
48          finally
49          {
50              pred.unlock();
51              curr.unlock();
52          }
53
54          // nothing was found
55          return false;
56      }
57
58  }
```

A.3 Contains Test

```
1   public class ContainsTest{
2       public static FineGrainedContains test = new FineGrainedContains<Object>();
3
4       public static void main(String args[]){
5
6           // Create the linkedList structure
7           Node c = new Node();
8           c.item = 3;
9           c.key = c.item.hashCode();
10          c.next = test.tail;
11
12          Node b = new Node();
13          b.item = 2;
14          b.key = b.item.hashCode();
15          b.next = c;
16
17
18          Node a = new Node();
19          a.item = 1;
20          a.key = a.item.hashCode();
21          a.next = b;
22
23          test.head.next = a;
24
25          // setup and run the tester threads
26          Tester[] testers = new Tester[5];
27
28          for(int i = 0; i < testers.length; i++)
29          {
30              testers[i] = new Tester();
31              testers[i].start();
32          }
33      }
34
35      /**
36       * Class that will run the contains method for a thread
37       */
38      private static class Tester extends Thread {
39          public void run() {
40              System.out.println("Thread with id = " + this.getId() + " Searching for
                    value = 3.");
```

```
41
42              boolean found = test.contains(3);
43
44              if (found)
45              {
46                  System.out.println("Thread with id = " + this.getId() + " found value"
                        );
47              }
48              else
49              {
50                  System.out.println("Thread with id = " + this.getId() + " did not find
                        value");
51              }
52          }
53      }
54
55  }
```

A.4 Contains Test Output

```
1   Thread with id = 13 Searching for value = 3.
2   Thread with id = 15 Searching for value = 3.
3   Thread with id = 14 Searching for value = 3.
4   Thread with id = 16 Searching for value = 3.
5   Thread with id = 12 Searching for value = 3.
6   Node with key = 0 locked
7   Node with key = 1 locked
8   Node with key = 0 unlocked
9   Node with key = 2 locked
10  Node with key = 1 unlocked
11  Node with key = 3 locked
12  Node with key = 2 unlocked
13  Node with key = 3 unlocked
14  Node with key = 0 locked
15  Node with key = 1 locked
16  Thread with id = 16 found value
17  Node with key = 0 unlocked
18  Node with key = 2 locked
19  Node with key = 0 locked
20  Node with key = 1 unlocked
21  Node with key = 3 locked
22  Node with key = 1 locked
23  Node with key = 2 unlocked
24  Node with key = 0 unlocked
25  Node with key = 3 unlocked
26  Node with key = 0 locked
27  Node with key = 2 locked
28  Thread with id = 15 found value
29  Node with key = 1 unlocked
30  Node with key = 3 locked
31  Node with key = 1 locked
32  Node with key = 2 unlocked
33  Node with key = 0 unlocked
34  Node with key = 3 unlocked
35  Node with key = 0 locked
36  Node with key = 2 locked
37  Thread with id = 14 found value
```

```
Node with key = 1 unlocked
Node with key = 3 locked
Node with key = 1 locked
Node with key = 2 unlocked
Node with key = 0 unlocked
Node with key = 3 unlocked
Node with key = 2 locked
Thread with id = 12 found value
Node with key = 1 unlocked
Node with key = 3 locked
Node with key = 2 unlocked
Node with key = 3 unlocked
Thread with id = 13 found value
```

B Question 4 Code

B.1 Sequential Implementation

```
package ca.mcgill.ecse420.a1;
public class SeqMatrixVectorMult {

        public static double[][] matrix;
        public static double[] vector;
        public static  int MATRIX_SIZE;

        public SeqMatrixVectorMult(double[][] matrix, double[] vector, int MATRIX_SIZE
            ){
                this.matrix = matrix;
                this.vector = vector;
                this.MATRIX_SIZE = MATRIX_SIZE;
        }

    /**
         * @return the result of the multiplication
         * */
        public static double[] sequentialMultiplication() {
                double[] c = new double[MATRIX_SIZE];
                for (int i = 0; i < MATRIX_SIZE; i++) {
                        for (int j = 0; j < MATRIX_SIZE; j++) {
                                c[i] = c[i] + matrix[i][j] * vector[j];
                        }
                }
        /*
                // if you want to print the result
                for (int i = 0; i < MATRIX_SIZE; i++) {
                        System.out.print(c[i] + " ");
                }
                System.out.println("");
                */
                return c;
        }
}
```

B.2 Parallel Implementation

```java
package ca.mcgill.ecse420.a1;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class ParMatrixVectorMult {

	public static double[][] matrix;
	public static double[] vector;
	public static int NUMBER_THREADS;
	public static int MATRIX_SIZE;

	public ParMatrixVectorMult(double[][] matrix, double[] vector,
			int MATRIX_SIZE, int NUMBER_THREADS){
		this.matrix = matrix;
		this.vector = vector;
		this.MATRIX_SIZE = MATRIX_SIZE;
		this.NUMBER_THREADS = NUMBER_THREADS;
	}

	/**
	 * @return the result of the multiplication
	 * */
	public static double[] parallelMultiplication() {
	double[] c = new double[MATRIX_SIZE];
	ExecutorService executor = Executors.newFixedThreadPool(NUMBER_THREADS);
	for (int i = 0; i < MATRIX_SIZE; i++) {
		executor.execute(new OneEntryMultiplication(matrix, vector, c, i,
		    MATRIX_SIZE));
	}
	executor.shutdown();
	try {
			executor.awaitTermination(30, TimeUnit.MINUTES);
		} catch (InterruptedException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
	/*
	// if you want to print the result
	for (int i = 0; i < MATRIX_SIZE; i++) {
			System.out.print(c[i] + " ");
		}
	System.out.println("");
		*/
	return c;
	}
}

//task class: one row of the matrix multiplied by the vector
//to produce one entry of the result vector.
class OneEntryMultiplication implements Runnable {
	private double[][] a;
	private double[] b, c;
	private int i, MATRIX_SIZE;
	public OneEntryMultiplication(double[][] m1, double[] m2, double[] m3,
			int index, int m_size) {
		a = m1; b = m2; c = m3;
```

```
58                  i = index; MATRIX_SIZE = m_size;
59          }
60          @Override
61          public void run() {
62                  for (int j = 0; j < MATRIX_SIZE; j++) {
63                          c[i] = c[i] + a[i][j] * b[j];
64                  }
65          }
66  }
```

B.3 Test Program

```
1   package ca.mcgill.ecse420.a1;
2
3   public class MatrixVectorMultTest {
4
5           private static final int NUMBER_THREADS = 3;
6           private static final int MATRIX_SIZE = 2000;
7
8           public static void main(String[] args) {
9                   double[][] matrix = generateRandomMatrix(MATRIX_SIZE, MATRIX_SIZE);
10                  double[] vector = generateRandomVector(MATRIX_SIZE);
11
12                  SeqMatrixVectorMult seq = new SeqMatrixVectorMult(matrix, vector,
                        MATRIX_SIZE);
13                  double start = System.currentTimeMillis();
14                  seq.sequentialMultiplication();
15                  double end = System.currentTimeMillis();
16                  System.out.println("Sequential: " + (end-start)/1000 + "s");
17
18                  ParMatrixVectorMult par = new ParMatrixVectorMult(matrix, vector,
                        MATRIX_SIZE, NUMBER_THREADS);
19                  start = System.currentTimeMillis();
20                  par.parallelMultiplication();
21                  end = System.currentTimeMillis();
22                  System.out.println("Parallel: " + (end-start)/1000 + "s");
23          }
24
25          /**
26           * Populates a matrix of given size with randomly generated integers between 0-10.
27           * @param numRows number of rows
28           * @param numCols number of cols
29           * @return matrix
30           */
31      private static double[][] generateRandomMatrix (int numRows, int numCols) {
32          double matrix[][] = new double[numRows][numCols];
33          for (int row = 0 ; row < numRows ; row++ ) {
34              for (int col = 0 ; col < numCols ; col++ ) {
35                  matrix[row][col] = (double) ((int) (Math.random() * 10.0));
36              }
37          }
38          return matrix;
39      }
40
41      /**
42       * Populates a vector of given size with randomly generated integers between 0-10.
43       * @param numCols number of cols
```

```
44        * @return vector
45        */
46      private static double[] generateRandomVector (int numCols) {
47          double vector[] = new double[numCols];
48          for (int col = 0 ; col < numCols ; col++ ) {
49              vector[col] = (double) ((int) (Math.random() * 10.0));
50          }
51          return vector;
52      }
53  }
```