Alexander Harris - 260688155
Abbas Yadollahi - 260680343

ECSE 420
December 4, 2018

Assignment 3 Report

Q1:

1.1
For time t0, this represents the average access time for the array when the entire array is within memory. Therefore, L' represents the cache size, with is 4 words in this case.

1.2
Since L' represents the cache size, than L>L' means that there will be a cache miss, since we cannot fit everything in memory. Therefore t1 represents the average time to access the array when accessing an element that is not within the cache.

1.3
Part 1 of the graph is the case where the entire array fits within the cache, therefore the access time will be constant. Part 2 is where not every access is a cache miss, therefore the access time is not constant. This makes sense because as s increases along 2 so does the access time. Part 3 is where every element we try to access results in a cache miss, therefore the access time is once again constant.

1.4
In our case, padding the array elements would degrade the overall performance as this would greatly increase the rate of cache misses, which would likely erase any performance benefit from using the Anderson Lock.

Q2:

2.1

The code for our implementation of the *contains()* method can be found in the Appendix under the *FineList* class. This method is similar to the add and remove methods where we step through the list until finding the appropriate key, however we simply check if the current node's key is equal to the one of the item we are looking for, and return the result.

2.2

Our test code for the *contains()* method can be found under the *TestList* class in the Appendix. This test simply creates a number of threads that add items into the list, stopping at a set amount of items. After adding, the thread sleeps for a few milliseconds, then checks if the list contains the element it just added. If it does, it removes the element from the list. If any of the add, remove or contains operations fail, the thread will print an error message. As a result, when we print the list after every thread has completed we should obtain an empty list.

We ran this test with 4, 8 and 16 threads on a 4-core machine with 10, 100 and 1000 items. Each time we obtained an empty list and no error messages, so we can assume that our implementation is correct based on our testing.

Q3:

3.1
Please see the code under *LockBasedQueue* for our implementation.

3.2
The main issue we ran into was with the *size*, *head* and *tail* variables. We had to make sure that the values of these variables were correct when being accessed, so we represented them using Atomic Integers. As a result, we were able to make sure that the queue has not empty during dequeuing and not full during enqueuing.

Q4:

4.1

Our sequential matrix vector multiplication can be found in the *sequentialMultiply()* method in the *MatrixVectorMultiplication* class. This is the naive $O(n^2)$ implementation which consists of stepping through the rows and columns of the matrix and summing up the products of the matrix rows and vector. It is interesting to note that this is much faster than the sequential matrix multiplication algorithm developed in Assignment 1, which just serves to illustrate the difference in computation time between an $O(n^3)$ algorithm and an $O(n^2)$ algorithm.

4.2

Since an algorithm with critical path $\Theta(log_2 n)$ would imply spawning thousands of threads proportional to the matrix size, we have decided to implement a different version as this type of algorithm would be highly inefficient on any modern processor. Instead, we will implement an algorithm which divides the matrix rows into smaller batches and assigns a batch to each thread.

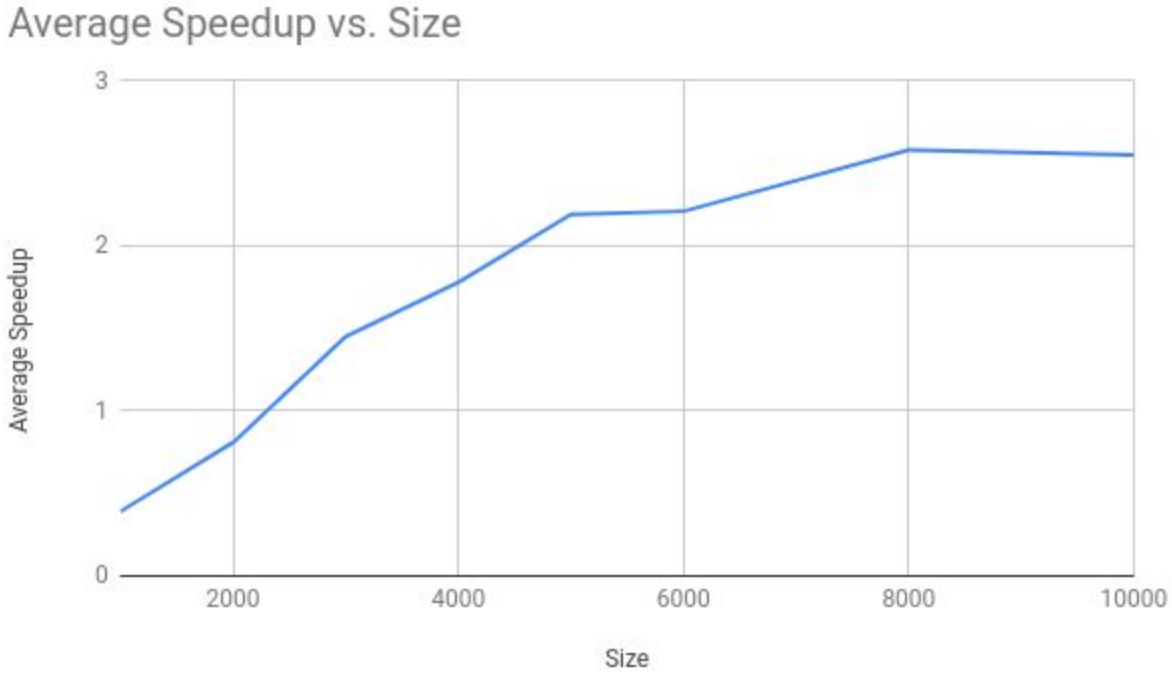
Our parallel matrix vector multiplication method can be found under *parallelMultiply()* in the *MatrixVectorMultiplication* class. This algorithm works by splitting the matrix by rows into individual vectors and assigning a subset of those to each thread. We then compute the product of the row and vector and store it in our result vector. Since the indexes in the result vector that we are accessing do not overlap, we do not need to implement any kind of synchronization since there should be no race conditions.

This method is far better performance wise than spawning a unique thread for each row in the matrix. There are currently no modern CPUs that have 2000 threads, so creating this many would only increase resource costs and make it much more complicated for the OS scheduler to assign CPU time to each thread. In practice, it is much more efficient to batch a subset of the matrix rows into a single thread. In this case, we divide the number of rows of the matrix equally among our available threads.

4.3

In order to test our matrix multiplication method, we wrote a simple program that first generates a random matrix A and random vector b. Then we compute the result using the sequential multiplication method, store it in res1 and measure the computation time (end1). Then we compute using the parallel method and store the result in res2 and the computation time in end2. To compute the speedup between the parallel and sequential methods, we divide end1, the sequential execution time, by end2, the parallel execution time. Since we expect end1 to be larger than end2, this gives us a ratio expressing the difference in speed between the parallel and sequential algorithms. We also check to see if $res1 == res2$.

When testing our algorithms using a matrix size of 2000x2000, vector size of 2000 and 4 threads, we discovered that our parallel algorithm was actually slower than our sequential algorithm. This is likely due to the extra time needed to create the threads and scheduling decisions. As we increase the size, we should start to see a speedup. The following chart shows the average speedup of our parallel algorithm over 10 iterations versus the size of the matrix/vector:



As we can see, our parallel algorithm is able to achieve speeds greater than our sequential algorithm for matrix/vector sizes above 3000. However, performance does begin to level out after 8000. This is likely due to the limited amount of threads we have, a machine with a higher thread count could theoretically still benefit at this stage.

4.4

We can see from observation that our algorithm performs $\Theta(n^2)$ work, and looking at the sequential implementation proves this. When it comes to the critical path, since in our algorithm there are no data dependencies, the critical path does not exist. Therefore, we can say that the parallelism of our algorithm is equal to the work our algorithm performs, which means it should see significant speedup for larger matrix sizes on machine with more CPU threads.

Appendix:

FineList:

```
public boolean contains(T item) {
        ListNode prev = null;
        ListNode curr;

        int key = item.hashCode();
        head.lock();

        try {
            prev = head;
            curr = prev.next;
            curr.lock();
            try {
                while (curr.key < key) {
                    prev.unlock();
                    prev = curr;
                    curr = curr.next;
                    curr.lock();
                }
                return curr.key == key;
            } finally {
                curr.unlock();
            }
        } finally {
            prev.unlock();
        }
    }
```

TestList:

```
package ca.mcgill.ecse420.a3;

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

public class TestList {
    private static final int NUMBER_THREADS = 4;
    private static final int NUM_ITEMS = 1000;
    private static final int THREAD_ITEMS = NUM_ITEMS / NUMBER_THREADS;

    private static FineList<Integer> fineList = new FineList<>();

    public static void main(String[] args) {

        ExecutorService executor =
Executors.newFixedThreadPool(NUMBER_THREADS);

        for (int i = 0; i < NUMBER_THREADS; i++) {
            executor.execute(new ListRunnable(i * THREAD_ITEMS));
        }

        executor.shutdown();
        try {
            executor.awaitTermination(10, TimeUnit.SECONDS);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println(fineList.toString());
    }

    private static class ListRunnable implements Runnable {
        int cnt;

        private ListRunnable(int cnt) {
            this.cnt = cnt;
        }

        public void run() {
```

```
            for (int i = 0; i < THREAD_ITEMS; i++) {
                if (!fineList.add(cnt + i)) {
                    System.out.println("[FAIL] Could not add: " + (cnt +
i));
                }
                try {
                    Thread.sleep(3);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                if (fineList.contains(cnt + i)) {
                    if (!fineList.remove(cnt + i)) {
                        System.out.println("[FAIL] Could not remove: " +
(cnt + i));
                    }
                } else {
                    System.out.println("[FAIL] Could not find: " + (cnt +
i));
                }
            }
        }
    }
}
```

LockBasedQueue:

```
package ca.mcgill.ecse420.a3;

import java.util.concurrent.atomic.AtomicInteger;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

// Bounded lock-based blocking queue
public class LockBasedQueue<T> {

 public Object[] queue;                 // Array version of linked list
 public int head;                       // First entry in queue
 public int tail;                       // Last entry in queue
 public int maxSize;                    // Max number of objects allowed
in queue
 public AtomicInteger size;             // Number of used slots in queue
 public ReentrantLock eLock, dLock;     // Lock out other
enqueuers/dequeuers
 public Condition notEmpty, notFull;    // When queue is not empty or not
full

 public LockBasedQueue(int maxSize) {
   this.queue = new Object[maxSize];
   this.head = 0;
   this.tail = this.head;
   this.maxSize = maxSize;
   this.size = new AtomicInteger(0);
   this.eLock = new ReentrantLock();
   this.dLock = new ReentrantLock();
   this.notFull = eLock.newCondition();
   this.notEmpty = dLock.newCondition();
 }

 // Add object to the end of the queue
 public void enqueue(T value) {
   if (value == null) {
```

```
      throw new NullPointerException();
    }

    boolean wakeUpDequeuers = false;
    eLock.lock();

    try {
      while (this.size.get() == this.maxSize) {
        try {
          this.notFull.await();
        } catch (InterruptedException e) {}
      }

      add(value);
      if (this.size.getAndIncrement() == 0) {
        wakeUpDequeuers = true;
      }
    } finally {
      this.eLock.unlock();
    }

    if (wakeUpDequeuers) {
      this.dLock.lock();
      try {
        this.notEmpty.signalAll();
      } finally {
        this.dLock.unlock();
      }
    }
  }

  // Remove and return the head of the queue
  public T dequeue() {
    T value;
    boolean wakeUpEnqueuers = false;
    this.dLock.lock();
```

```
  try {
    while (this.size.get() == 0) {
      try {
        this.notEmpty.await();
      } catch (InterruptedException e) {}
    }

    value = remove();
    if (this.size.getAndDecrement() == this.maxSize) {
      wakeUpEnqueuers = true;
    }
  } finally {
    this.dLock.unlock();
  }
  if (wakeUpEnqueuers) {
    this.eLock.lock();
    try {
      this.notFull.signalAll();
    } finally {
      this.eLock.unlock();
    }
  }
  return value;
}

public void add(T value) {
  final Object[] items = this.queue;
  items[this.tail] = value;
  if (++this.tail == items.length) {
    this.tail = 0;
  }
}

public T remove() {
  final Object[] items = this.queue;
```

```
    T value = (T) items[this.head];
    items[this.head] = null;
    if (++this.head == items.length)
        this.head = 0;
    return value;
  }
}
```

LockFreeQueue:

```
package ca.mcgill.ecse420.a3;

import java.util.concurrent.atomic.AtomicInteger;
import java.util.concurrent.atomic.AtomicReferenceArray;

// Bounded lock-free blocking queue
public class LockFreeQueue<T> {

 public AtomicReferenceArray<T> queue;
 public AtomicInteger head, tail, size;
 public int maxSize;

 public LockFreeQueue(int maxSize) {
   this.maxSize = maxSize;
   this.queue = new AtomicReferenceArray<>(maxSize);
   this.head = new AtomicInteger(0);
   this.tail = new AtomicInteger(0);
   this.size = new AtomicInteger(0);
 }

 // Add object to the end of the queue
 public void enqueue(T value) {
   int size = this.size.get();
   while (size == this.maxSize || !this.size.compareAndSet(size, size +
1)) {
     size = this.size.get();
   }

   this.queue.set(this.tail.getAndIncrement(), value);

   if (this.tail.get() == this.maxSize) {
     this.tail.set(0);
   }
 }
```

```
 // Remove and return the head of the queue
 public T dequeue() {
   int size = this.size.get();
   while (size == 0 || !this.size.compareAndSet(size, size - 1)) {
     size = this.size.get();
   }
   T value = this.queue.getAndSet(this.head.getAndIncrement(), null);

   if (this.head.get() == this.maxSize) {
     this.head.set(0);
   }

   return value;
 }
}
```

MatrixVectorMultiplication:

```
package ca.mcgill.ecse420.a3;

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

public class MatrixVectorMultiplication {
   private static final int NUM_THREADS = 4;
   private static final int MATRIX_SIZE = 2000;
   private static final int NUM_ITER = 10;
   private static final Random RAND = new Random();

   public static void main(String[] args) {
       double speed_ave = 0;
       for (int i = 0; i < NUM_ITER; i++) {
           long start;
           double[][] A = generateRandomMatrix(MATRIX_SIZE, MATRIX_SIZE);
           double[] b = generateRandomVector(MATRIX_SIZE);

           start = System.nanoTime();
           double[] res1 = sequentialMultiply(A, b);
           double end1 = (double) (System.nanoTime() - start);
           System.out.println("Elapsed time: " + end1);

           start = System.nanoTime();
           double[] res2 = parallelMultiply(A, b);
           double end2 = (double) (System.nanoTime() - start);
           System.out.println("Elapsed time: " + end2);

           double speed = Math.round(end1 / end2 * 100.0) / 100.0;
           speed_ave += speed;

           System.out.println(Arrays.toString(res1));
           System.out.println(Arrays.toString(res2));
           System.out.println("Same result: " + Arrays.equals(res1, res2));
           System.out.println("Parallel was " + speed + "x faster than sequential!");
       }
       speed_ave /= NUM_ITER;
       System.out.println("\nAverage speedup was: " + (Math.round(speed_ave * 100.0) /
100.0) + "x for " + NUM_ITER + " iterations");
   }

   public static double[] parallelMultiply(double[][] matrix, double[] vector) {
       ExecutorService executor = Executors.newFixedThreadPool(NUM_THREADS);
```

```
        double[] res = new double[MATRIX_SIZE];

        for (int i = 0; i < NUM_THREADS; i++) {
            executor.execute(new ParallelMultiply(matrix, vector, res, i));
        }

        executor.shutdown();
        try {
            executor.awaitTermination(10, TimeUnit.SECONDS);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return res;
    }

    private static class ParallelMultiply implements Runnable {
        double[][] matrix;
        double[] vector;
        double[] res;
        int rowNum;
        int rows;
        int cols;

        private ParallelMultiply(double[][] matrix, double[] vector, double[] res, int
rowNum) {
            this.matrix = matrix;
            this.vector = vector;
            this.res = res;
            this.rowNum = rowNum;
            rows = matrix.length;
            cols = matrix[0].length;
        }

        @Override
        public void run() {

            // Starting at rowNum, skip num threads every iteration
            for (int i = rowNum; i < rows; i += NUM_THREADS) {
                // Same algorithm as sequential multiply
                res[i] = 0;
                for (int j = 0; j < cols; j++) {
                    res[i] += matrix[i][j] * vector[j];
                }
            }
        }
    }

    public static double[] sequentialMultiply(double[][] matrix, double[] vector) {
        int rows = matrix.length;
        int cols = matrix[0].length;
```

```
        double[] res = new double[rows];

        for (int i = 0; i < rows; i++) {
            res[i] = 0;
            for (int j = 0; j < cols; j++) {
                res[i] += matrix[i][j] * vector[j];
            }
        }
        return res;
    }

    private static double[][] generateRandomMatrix(int numRows, int numCols) {
        double[][] matrix = new double[numRows][numCols];
        for (int row = 0; row < numRows; row++) {
            for (int col = 0; col < numCols; col++) {
                matrix[row][col] = (double) ((int) (RAND.nextInt(10) + 1));
            }
        }
        return matrix;
    }

    private static double[] generateRandomVector(int numElems) {
        double[] vector = new double[numElems];
        for (int i = 0; i < numElems; i++) {
            vector[i] = (double) ((int) (RAND.nextInt(10) + 1));
        }
        return vector;
    }
}
```