Assignment 3 ECSE 420

Elvric Trombert
260673394
Simon Zheng
260744353

December 5, 2018

---

1. (a) t_0 shows that time it takes for reading an element in the array that is in the cache. L' is the number of elements that can be contained in cache simultaneously: if $s = 0$ then $L' = 4$. If $s = 1, 2$ then $L' = 2$. If $s > 2$ then $L' = 1$.

   (b) t_1 indicates the time it takes to read an element from main memory as the next index cannot be access through the cache.

   (c) 1 means that all the element in the array can be cached at once and accessed in constant time, thus the time does not change. It is the lowest time possible since accessing something from the cache is faster then accessing it from main memory.

   2 increases exponentially as the more cache miss we get the more time it takes to access an index on our array thus the ratio of cache miss and cache hit increases. The more cache miss we get the greater our average access time.

   3, is the case where only one element can be cached at a time thus when accessing the next element we always have to go to main memory which hence causes every access to have the same average access time. It is highest average access time as accessing something from main memory takes more time than accessing something from the cache.

   (d) Padding technique in ALock degrades the overall performance of the lock as in order get the lock, each thread must go to main memory to retrieve the value in the array index associated with it, since that information cannot be kept in the cache. Thus this slows down the lock as in order to get the lock each thread will make a call to main memory which takes more time than accessing its cache.

2. The class implementing the *contains* is called BoundedLockBasedQueue.java.

   The TestContains class is the one that will test whether contains work appropriately. After talking to the TA, he mentioned that since the only concurrent part that we had to achieve was in the contains method. Thus in order to test the contains method we added an add method that itself however is not concurrent.

   Our test method performs the following: we randomly add integers from 0 to 14 where the chances of adding a new number after adding a new number decreases (at first they are

1/2 then after adding the first element 1/3 and so on). This ensure that upon every try we get different integers on different list sizes (ensuring that the results cannot be hard coded and work on different lists with different sizes and item values). When a number is added to the list we mention it in a print statement.

Then we concurrently launch all the contains call on the integers from 0 to 14 and print out the result of the contains method. To make the results more easily readable we compare the contains output with an array that is set to false when the number i was not added to the list and true when the number was added to the list. We then print at "Test passed" or "Test fails" based on these results at the end.

Our implementation is correct since our test passes which means that the lock are correctly acquired and released by the method and that the implementation does not cause any false reads or deadlocks. This is due to the fact that we acquire the lock of the previous and current nodes which ensure that the current node we are reading is part of the list, as to remove or add a node in the list these two same locks must also be acquired by the add or remove method for them to remove the current node we are accessing or add a new node for the add method.

3. The class is called BoundedLockBasedQueue.java

   (b) The first difficulty comes when enqueuing. Here, we must make sure before enqueuing that there is enough space in the queue for that, so first we must retrieve the *tail* and *head* values and ensure that $head - tail! = queueMaxSize$. If that passes then the issue is, when adding the object to the queue, we must make sure that another thread did not add an object before us, which implies testing that again and so on. Thus, the only way is to use a test-and-set approach afterwards, which makes it more complex than with the lock. Furthermore, each thread has no way of knowing (without accessing the tail and head) whether it can perform its tasks. This implies that the CPU will be used by threads that just idle, getting the tail and head values while not making any progress, compares to the lock where the thread will just wait to be awaken by the condition.

   The second difficulty is the exact same idea for dequeuing: in this case, if $head == tail$, then we must wait for an enqueue to happen, but even if it did, when dequeuing, we also have to check that, since by our last verification the queue was not dequeued by another dequeuer. Thus, we must test-and-set before being able to dequeue an object. Again, that involves a lot of CPU idle time just getting and testing variables instead of doing useful work.

4. The test class is called MatrixVectorMultiplication.java

   (c) Given that we run into the following error message "Exception in thread main java.lang.OutOfMemoryError: Java heap space" when running the code with 2000 side splitting the matrix up to 1x1 or 2x1 or 1x2. We run the code on the server with 2000 matrix splitting the matrix until it reaches 250. With this the parallel running time is 1181563085 nano seconds and the sequential time is 31571979 nano seconds. Here the parallel runtime is greater than the sequential we believe that this is due to the fact that even with the split to 250 the 8 cores provided by the server take too much time to divide the work than just actually solving the multiplication thus being slower than the sequential part. This we get a "speed up" of $\frac{31571979}{1181563085} = 0.0267$ which is a slow down again this is due to the fact that our operation technically requires a lot more thread than just 8 to perform faster than the sequential application.

(d) i. Calculate the work of the add function for vectors let $n = 2^m$:

$$\begin{aligned}
A_1(2^m) &= 2^1 A_v(2^{m-1}) + \Theta(1) \\
A_1(2^{m-1}) &= 2^1 A_v(2^{m-2}) + \Theta(1) \\
A_1(2^{m-2}) &= 2^1 A_v(2^{m-3}) + \Theta(1) \\
A_1(2^m) &= 2^2 A_v(2^{m-2}) + 2\Theta(1) + \Theta(1) \\
A_1(2^m) &= 2^3 A_v(2^{m-3}) + 4\Theta(1) + 2\Theta(1) + \Theta(1) \\
2^0 + 2^1 + 2^2 + 2^3 + .. + 2^m &= 2^{m+1} - 1 \\
A_1(2^m) &= 2^m * 1 + (2^{m+1} - 1)\Theta(1) \\
A_1(n) &\approx n + 2n\Theta(1) \\
\Theta(n) &= 3n
\end{aligned}$$

Calculating the critical path length:

$$\begin{aligned}
A_\infty(2^m) &= A_v(2^{m-1}) + \Theta(1) \\
A_\infty(2^{m-1}) &= A_v(2^{m-2}) + \Theta(1) \\
A_\infty(2^{m-2}) &= A_v(2^{m-3}) + \Theta(1) \\
A_\infty(2^m) &= A_v(2^{m-2}) + 1\Theta(1) + \Theta(1) \\
A_\infty(2^m) &= A_v(2^{m-3}) + 1\Theta(1) + 1\Theta(1) + 1\Theta(1) \\
A_\infty(2^m) &= 1 + m\Theta(1) \\
A_\infty(n) &\approx m\Theta(1) \\
\Theta(\log(n)) &= m
\end{aligned}$$

ii. Calculate the work of the multiply method for matrix vector multiplication $n = 2^m$:

$$
\begin{aligned}
M_1(2^m) &= 2^2 M_v(2^{m-1}) + \Theta(2^m) \\
M_1(2^{m-1}) &= 2^2 M_v(2^{m-2}) + \Theta(2^{m-1}) \\
M_1(2^{m-2}) &= 2^2 M_v(2^{m-3}) + \Theta(2^{m-2}) \\
M_1(2^m) &= 2^4 M_v(2^{m-2}) + 2^2\Theta(2^{m-1}) + \Theta(2^m) \\
M_1(2^m) &= 2^6 M_v(2^{m-3}) + 2^4\Theta(2^{m-2}) + 2^2\Theta(2^{m-1}) + \Theta(2^m) \\
M_1(2^m) &\approx 2^6 M_v(2^{m-3}) + 2^2 * 2^m + 2 * 2^m + 2^m \\
M_1(2^m) &\approx 2^6 M_v(2^{m-3}) + (2^2 + 2 + 2^0)2^m \\
2^0 + 2^1 + 2^2 + 2^3 + .. + 2^m &= 2^{m+1} - 1 \\
M_1(2^m) &\approx (2^m)^2 + (2^{m+1} - 1)2^m \\
M_1(2^m) &\approx (2^m)^2 + 2^{2m+1} - 2^m \\
M_1(n) &\approx n^2 + n^2 * 2 - n \\
\Theta(n^2) &= 3n^2 - n
\end{aligned}
$$

Calculate the critical path the multiply method for matrix vector multiplication $n = 2^m$:

$$\begin{aligned}
M_\infty(2^m) &= M_v(2^{m-1}) + \Theta(m) \\
M_\infty(2^{m-1}) &= M_v(2^{m-2}) + \Theta(m-1) \\
M_\infty(2^{m-2}) &= M_v(2^{m-3}) + \Theta(m-2) \\
M_\infty(2^m) &= M_v(2^{m-3}) + \Theta(m-2) + \Theta(m-1) + \Theta(m) \\
M_\infty(2^m) &\approx M_v(2^{m-3}) + m - 2 + m - 1 + m \\
1 + 2 + 3 + 4 + .. + n &= \frac{(n+1) * n}{2} \\
M_\infty(2^m) &\approx \frac{(m+1) * m}{2} \\
\Theta(\log(n)^2) &= \frac{m^2 + m}{2}
\end{aligned}$$

iii. The parallelism achieved is therefore:

$$\frac{M_1}{M_\infty} = \Theta(\frac{n^2}{\log(n)^2})$$

This is less then the Θ running time required for matrix multiplication which makes sense since, every time we multiply, we "create" 4 new threads compared to 8 with matrix multiplication, thus, in our case, for a 2000×2000 matrix, we would need about 364770 cores to run the multiplication at its fastest speed. More cores would not improve the speed up.

Discussing now the implementation for the sequential time, it is two basic for-loops that just iterate through the matrix and the vector and perform the multiplications.
For the parallel part, we followed the approach given to us in class for the matrix multiplication.

We created a Vector and Matrix class and added *split* methods to be able to "virtually" split a vector in half and a matrix in 4 without having to copy the matrix entries into a smaller matrix over and over again (the idea of doing this with a *split* method in its own class as opposed to inline came from the book). Then, using these methods, for the *add* Task we cut each vector into 2 pieces recursively passing it to the executor as submissions, until we reach a 1 by 1 vector. Then, we perform the addition.

For the matrix vector multiplication, we created two new vectors that will hold the result of the multiplication and addition to the left and right side of the matrix, independently. Then, we split the matrix in 4 and split the two new vectors in 2 (along with the vector the matrix is being multiplied with). We then recursively pass each quadrant, with its corresponding answer vector-half along with the multiplication vector-half. Once done, we pass the two answer vectors with the final answer vector to the add vector task.

Appendix

0.1 BoundedLockBasedQueue

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

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

public class BoundedLockBasedQueue<Item> {
  private Object[] items;
  private ReentrantLock deqLock;
  private ReentrantLock enqLock;
  private Condition notFull;
  private Condition notEmpty;
  private int head = 0;
  private int tail = 0;

  public BoundedLockBasedQueue(int size) {
    this.items = new Object[size];
    deqLock = new ReentrantLock();
    enqLock = new ReentrantLock();
    notFull = enqLock.newCondition();
    notEmpty = deqLock.newCondition();
  }

  public void enqueue(Item item) throws InterruptedException {
    boolean mustWakeDequeue;
    enqLock.lock();
    try {
      while (tail - head == items.length) {
        System.out.println("Queue is full");
        notFull.await();
      }
      items[tail % items.length] = item;
      tail++;
      mustWakeDequeue = tail - head == 1;
    } finally {
      enqLock.unlock();
    }
    if (mustWakeDequeue) {
      try {
        deqLock.lock();
        notEmpty.signalAll();
      } finally {
        deqLock.unlock();
      }
    }
  }

  public Item dequeue() throws InterruptedException {
    Item itemReturn;
    boolean mustWakeEnq;
    deqLock.lock();
    try {
      while (tail == head) {
        System.out.println("Queue is Empty");
        notEmpty.await();
      }
      itemReturn = (Item) items[head % items.length];
      head++;
      mustWakeEnq = tail - head == items.length - 1;
    } finally {
      deqLock.unlock();
    }
    if (mustWakeEnq) {
      try {
        enqLock.lock();
        notFull.signalAll();
      } finally {
        enqLock.unlock();
      }
    }
    return itemReturn;
  }
}
```

0.2 FineGrainedList

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

import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class FineGrainedList<Item> {
  private LockableNode<Item> head;

  public FineGrainedList() {
    head = null;
  }

  public boolean contains(Item item) {
    if (head == null) {
      return false;
    }
    int key = item.hashCode();
    head.lock();
    LockableNode pred = head;
    try {
      LockableNode<Item> curr = pred.next;
      /* if the list only contains the head then check the head key
       * if the head key == the item key then return true else we have to iterate
       */
      if (curr == null || pred.key == key) {
        return pred.key == key;
      }
      curr.lock();
      try {
        while (curr.key < key && curr.next != null) {
          pred.unlock();
          pred = curr;
          curr = curr.next;
          curr.lock();
        }
        if (curr.key == key) {
          return true;
        }
      } finally {
        curr.unlock();
      }
    } finally {
      pred.unlock();
    }
    return false;
  }

  // non concurrent add method
  public boolean add(Item item) {
    // initializing the head if the list is null
    if (head == null) {
      head = new LockableNode<>(item);
      return true;
    }
    // edge case when the list only contains 1 node
    if (head.next == null) {
      if (head.key < item.hashCode()) {
        head.next = new LockableNode<>(item);
        return true;
      } else if (head.key == item.hashCode()) {
        return false;
      } else {
        LockableNode<Item> newNode = new LockableNode<>(item);
        newNode.next = head;
        head = newNode;
        return true;
      }
    }

    LockableNode<Item> pre = head;
    LockableNode<Item> cur = head.next;

    while (cur != null) {
      if (cur.key < item.hashCode()) {
        pre = cur;
        cur = pre.next;
      } else if (cur.key == item.hashCode()) {
        return false;
      } else {
        LockableNode<Item> newItem = new LockableNode<>(item);
        pre.next = newItem;
        newItem.next = cur;
        return true;
      }
    }

    pre.next = new LockableNode<>(item);
    return true;
  }
}

class LockableNode<Item> {
  protected Item item;
  protected int key;
  protected Lock lock = new ReentrantLock();
  protected LockableNode<Item> next;

  public LockableNode(Item item) {
    this.item = item;
    this.key = item.hashCode();
  }

  public void lock() {
    lock.lock();
  }
  public void unlock() {
    lock.unlock();
  }
}
```

0.3 TestContains

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

import java.util.Random;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class TestContains {
  private static ExecutorService exec = Executors.newCachedThreadPool();

  public static void main(String[] args) throws ExecutionException, InterruptedEx
    Random ran = new Random();
    int bound = 2;
    FineGrainedList<Integer> intList = new FineGrainedList<>();
    boolean testResult = true;
    boolean[] answer = new boolean[15];
    for (int i = 0; i < 15; i++) {
      /* chance to add a number depends on previous
       * number added starting at 1/2 then 1/3 and so on as we add
       * more numbers to the list.
       */
      if (ran.nextInt(bound) == 0) {
        bound++;
        System.out.println("Adding number " + i + " to the list");
        intList.add(i);
        answer[i] = true;
      } else {
        answer[i] = false;
      }
    }
    for (int i = 0; i < 15; i++) {
      int testNum = i;
      Future<Boolean> containsResult = exec.submit(() -> intList.contains(testNum
      boolean ans = containsResult.get();
      System.out.println("The method contains with param " + i + " outputs " + ar
      if (answer[i] != ans) {
        testResult = false;
      }
    }
    if (testResult) {
      System.out.println("Test passed");
    } else {
      System.out.println("Test failed");
    }
    exec.shutdown();
  }
}
```

0.4 Matrix

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

public class Matrix {
  int rowDim;
  int columnDim;
  double[][] data;
  int rowDisplace;
  int columnDisplace;

  public Matrix(int rowDim, int columnDim, int rowDisplace, int columnDisplace, d
    this.rowDim = rowDim;
    this.columnDim = columnDim;
    this.rowDisplace = rowDisplace;
    this.columnDisplace = columnDisplace;
    this.data = data;
  }

  public Matrix(double[][] data) {
    this.rowDim = data.length;
    this.columnDim = data[0].length;
    this.data = data;
    this.rowDisplace = 0;
    this.columnDisplace = 0;
  }

  public double get(int row, int column) {
    return data[row + rowDisplace][column + columnDisplace];
  }

  public void set(int row, int column, double value) {
    data[row + rowDisplace][column + columnDisplace] = value;
  }
```

```
public Matrix[][] split4() {
  Matrix[][] newMatrix = new Matrix[2][2];
  int newRowDim = rowDim / 2;
  int newColDim = columnDim / 2;

  // Case where the matrix has an even number of rows
  if (rowDim % 2 == 0) {
    // Case where the matrix has an even number of columns
    if (columnDim % 2 == 0) {
      newMatrix[0][0] = new Matrix(newRowDim, newColDim, rowDisplace, columnDi
      newMatrix[0][1] =
          new Matrix(newRowDim, newColDim, rowDisplace, columnDisplace + newCol
      newMatrix[1][0] =
          new Matrix(newRowDim, newColDim, rowDisplace + newRowDim, columnDispl
      newMatrix[1][1] =
          new Matrix(
              newRowDim, newColDim, rowDisplace + newRowDim, columnDisplace + n
    }
    /* Case where the matrix has an odd number of columns the
     * the top right and bottom right matrices will then have an extra column
     */
    else {
      newMatrix[0][0] = new Matrix(newRowDim, newColDim, rowDisplace, columnDi
      newMatrix[0][1] =
          new Matrix(newRowDim, newColDim + 1, rowDisplace, columnDisplace + ne
      newMatrix[1][0] =
          new Matrix(newRowDim, newColDim, rowDisplace + newRowDim, columnDispl
      newMatrix[1][1] =
          new Matrix(
              newRowDim,
              newColDim + 1,
              rowDisplace + newRowDim,
              columnDisplace + newColDim,
              data);
    }
  }
  /* Case where the matrix has an odd number of rows the
   * the bottom left and bottom right matrices will then have an extra row
   */
  else {
    if (columnDim % 2 == 0) {
      newMatrix[0][0] = new Matrix(newRowDim, newColDim, rowDisplace, columnDi
      newMatrix[0][1] =
          new Matrix(newRowDim, newColDim, rowDisplace, columnDisplace + newCol
      newMatrix[1][0] =
          new Matrix(newRowDim + 1, newColDim, rowDisplace + newRowDim, column
        newMatrix[1][1] =
            new Matrix(
                newRowDim + 1,
                newColDim,
                rowDisplace + newRowDim,
                columnDisplace + newColDim,
                data);
      }
      /* Case where the matrix has an odd number of columns the
       * the top right and bottom right matrices will then have an extra column
       */
      else {
        newMatrix[0][0] = new Matrix(newRowDim, newColDim, rowDisplace, columnDi
        newMatrix[0][1] =
            new Matrix(newRowDim, newColDim + 1, rowDisplace, columnDisplace + ne
        newMatrix[1][0] =
            new Matrix(newRowDim + 1, newColDim, rowDisplace + newRowDim, column
        newMatrix[1][1] =
            new Matrix(
                newRowDim + 1,
                newColDim + 1,
                rowDisplace + newRowDim,
                columnDisplace + newColDim,
                data);
      }
    }
    return newMatrix;
  }

  public int getRowDim() {
    return rowDim;
  }

  public int getColumnDim() {
    return columnDim;
  }
}
```

0.5 Vector

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

public class Vector {
  int dim;
  double[] data;
  int colDisplace;
```

```
public Vector(double[] data, int colDisplace, int dim) {
  this.dim = dim;
  this.data = data;
  this.colDisplace = colDisplace;
}

public Vector(double[] data) {
  this.dim = data.length;
  this.data = data;
  this.colDisplace = 0;
}

public double get(int column) {
  return data[column + colDisplace];
}

public void set(int column, double value) {
  data[colDisplace + column] = value;
}

public void add(int column, double value) {
  data[colDisplace + column] += value;
}

public Vector[] split2() {
  Vector[] splitV = new Vector[2];
  // case where the split creats 2 even part
  if (dim % 2 == 0) {
    int newDim = dim / 2;
    splitV[0] = new Vector(data, colDisplace, newDim);
    splitV[1] = new Vector(data, colDisplace + newDim, newDim);
  }
  /* case where the split is not even:
   * The bottom part of the split in the vector will have 1 extra entry than th
   */
  else {
    int newDim = dim / 2;
    splitV[0] = new Vector(data, colDisplace, newDim);
    splitV[1] = new Vector(data, colDisplace + newDim, newDim + 1);
  }
  return splitV;
}

public int getDim() {
  return dim;
  }
}
```

0.6 MatrixVectorMultiplication

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

import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class MatrixVectorMultiplication {
  private static int MATRIX_SIZE = 2000;
  public static ExecutorService exec = Executors.newCachedThreadPool();

  public static void main(String[] args) throws ExecutionException, InterruptedEx
    double[] v = generateRandomVector(MATRIX_SIZE);
    double[][] m = generateRandomMatrix(MATRIX_SIZE, MATRIX_SIZE);
    measureParallelTime(m, v);
    measureSequentialTime(m, v);
    // System.out.println(Arrays.toString(sequentialMultiplyMatrixVector(m,v)));
    // System.out.println(Arrays.toString(parallelMultiplyMatrix(m, v)));
  }

  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) (Math.random() * 10.0));
      }
    }
    return matrix;
  }

  private static double[] generateRandomVector(int num) {
    double vector[] = new double[num];
    for (int row = 0; row < num; row++) {
      vector[row] = (double) ((int) (Math.random() * 10.0));
    }
    return vector;
  }

  public static void measureSequentialTime(double[][] matrix, double[] vector) {
    long startTime = System.nanoTime();
    sequentialMultiplyMatrixVector(matrix, vector);
    long endTime = System.nanoTime();
    long runTime = endTime - startTime;
    System.out.println("Runtime (sequential) : " + runTime + " ns");
  }

  public static void measureParallelTime(double[][] matrix, double[] vector)
      throws ExecutionException, InterruptedException {
    long startTime = System.nanoTime();
    parallelMultiplyMatrix(matrix, vector);
    long endTime = System.nanoTime();
    long runTime = endTime - startTime;
    System.out.println("Runtime (parallel) : " + runTime + " ns");
  }

  public static double[] sequentialMultiplyMatrixVector(double[][] matrix, double
    double[] result = new double[MATRIX_SIZE];
    for (int i = 0; i < MATRIX_SIZE; i++) {
      for (int j = 0; j < MATRIX_SIZE; j++) {
        result[i] += matrix[i][j] * vector[j];
      }
    }
    return result;
  }

  public static double[] parallelMultiplyMatrix(double[][] matrix, double[] vecto
      throws ExecutionException, InterruptedException {
    Matrix m = new Matrix(matrix);
    Vector v = new Vector(vector);
    double[] ans = new double[v.getDim()]; // To put the Vector into a correspond
    Vector vAns = new Vector(ans);
    exec.submit(new multiplyMatrixVectorTask(m, v, vAns)).get();
    exec.shutdown();
    return ans;
  }

  static class AddVectorTask implements Runnable {
    private Vector left;
    private Vector right;
    private Vector ans; // Left and right sides are merged/added together into or

    public AddVectorTask(Vector left, Vector right, Vector ans) {
      this.left = left;
      this.right = right;
      this.ans = ans;
    }
    @Override
    public void run() {
      try {
        if (left.getDim() == 1) { // Base case
          ans.set(0, left.get(0) + right.get(0)); // Put sum into corresponding v
        } else {
          // Do the task recursively by halving into top and bottom and waiting f
          // (promises)
          Vector[] leftSplit = left.split2();
          Vector[] rightSplit = right.split2();
          Vector[] ansSplit = ans.split2();
          Future<?> top = exec.submit(new AddVectorTask(leftSplit[0], rightSplit[
          Future<?> bottom =
              exec.submit(new AddVectorTask(leftSplit[1], rightSplit[1], ansSplit
          top.get();
          bottom.get();
        }
      } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
      }
    }
  }

  static class multiplyMatrixVectorTask implements Runnable {
    private Matrix m;
    private Vector v;
    private Vector ans;
    private Vector left;
    private Vector right;

    public multiplyMatrixVectorTask(Matrix m, Vector v, Vector ans) {
      this.m = m;
      this.v = v;
      this.ans = ans;
      this.left = new Vector(new double[ans.getDim()]);
      this.right = new Vector(new double[ans.getDim()]);
    }

    @Override
    public void run() {
      try {
        if (m.getColumnDim() == 250
            || m.getRowDim() == 250) { // Base case set to 250 (see report for ex
          for (int i = 0; i < m.getRowDim(); i++) {
            for (int j = 0; j < m.getColumnDim(); j++) {
              ans.add(i, m.get(i, j) * v.get(j));
```

```
                }
              }
              return;
            }
            Matrix[][] mSplited = m.split4();
            Vector[] vSplited = v.split2();
            Vector[] leftSplit = left.split2();
            Vector[] rightSplit = right.split2();
            Future<?> topLeft =
                exec.submit(new multiplyMatrixVectorTask(mSplited[0][0], vSplited[0],
            Future<?> topRight =
                exec.submit(new multiplyMatrixVectorTask(mSplited[0][1], vSplited[1],
            Future<?> bottomLeft =
                exec.submit(new multiplyMatrixVectorTask(mSplited[1][0], vSplited[0],
            Future<?> bottomRight =
                exec.submit(new multiplyMatrixVectorTask(mSplited[1][1], vSplited[1],
            topLeft.get();
            topRight.get();
            bottomLeft.get();
            bottomRight.get();
            Future<?> add = exec.submit(new AddVectorTask(left, right, ans));
            add.get();
          } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
          }
        }
      }
    }
```