ECSE 420 - Parallel Computing

Assignment 3

Arian Omidi - 260835976
Namdar Nejad - 260893536

April 12, 2022

1 Memory Access & Anderson Lock

1.1

Since the average time per access remains constant in the green graph, we can conclude that we are fetching each element of the array from the memory and are able to store it in the cache without doing extra operations (write, replace, ... due to the cache misses). So L' represents the size of the cache which is 4 words here and t_0 is the average access time to the array that is entirely in the memory.

1.2

If the array size is bigger than the cache size, we will be facing cache misses at some points. So t_1 is the average access time to the array when fetching elements that are not in the cache.

1.3

Part 1: The entire array can fit into the cache and we have an empty cache so the access time is constant and rather small.

Part 2: In this case, we are slowly facing cache misses and the access time is not constant. As the stride increase so does the access time.

Part 3: In this section, we have reached a point where the cache is entirely full and we are only having cache misses. So the access time is constant but quite high.

1.4

Using the padding technique would increase the cache misses as we would be filling the cache with extra data that is not useful, thus degrading the overall performance of the lock.

2 Examining the Fine-Grained Algorithm

2.1 Contains Implementation

The implementation for the `contains()` can be found in `Q2Contains.java` below in **Appendix A**.

The fine-grained algorithm described in Chapter 9.5 improves concurrency by locking individual nodes instead of the entire list. We need a "lock coupling" protocol to do our operations on the list. In the `contains()` method, we iterate through the list until we find the key we are looking for, and check at each step compare it with the key of the node.

2.2 Test Implementation

The test code can be found in `Q2Test.java` below in **Appendix A**.

In the test method we create a number of threads that add a certain number of items to the list. After completing the list we check if it contain a given element and print the result.

3 Bounded Queue

3.1 Lock-Based Queue

The lock-based bounded queue implementation can be found below in **Appendix B** under `BoundedQueue.java`.

3.2 Lock-Free Queue

The lock-free bounded queue implementation can be found below in **Appendix B** under `LockFreeBoundedQueue.java`.

We ran into difficulties defining the criteria for allowing queuing and dequeuing as we had to ensure that the data was not being overwritten. To overcome this challenge, we made all of the pointers and the array into Atomic Objects as well as adding another Atomic Integer to keep track of the current size.

4 Matrix Vector Multiplication

4.1 Sequential Implementation

The sequential multiplication implementation can be found below in **Appendix C** under `SequentialMultiplier.java`.

The naive sequential matrix-vector multiplication was implemented by calculating the dot product between each row of the matrix and the vector. Thus, this method achieves a $O(n^2)$ runtime complexity.

4.2 Parallel Implementation

The highly parallel and practical parallel multiplication implementations can be found below in **Appendix C** under `ParallelMultiplier.java` and `PracticalParallelMultiplier.java` respectively.

For the parallel matrix-vector multiplication we implemented two different methods: a highly parallel multiplier and a practical parallel multiplier. As the name suggests the highly parallel multiplier, has a $\Theta(\log n)$ critical path. However, a side-effect of its high parallelism is that it need many processors to achieve this performance. Since at each step the algorithm must create 2 new threads, the overhead of this method is huge. Thus, while this approach is theoretically optimal, in practice it is orders of magnitude slower than the sequential multiplier.

To remedy this, we implemented a practical parallel multiplier which uses the simple methods we implemented in Assignment 1 to greatly speed up the performance of matrix-vector multiplication.

4.3 Sequential vs Parallel Comparison

Since our highly parallel multiplier crashes our computers when multiplying matrices of size 200, we used our practical parallel multiplier for this comparison.

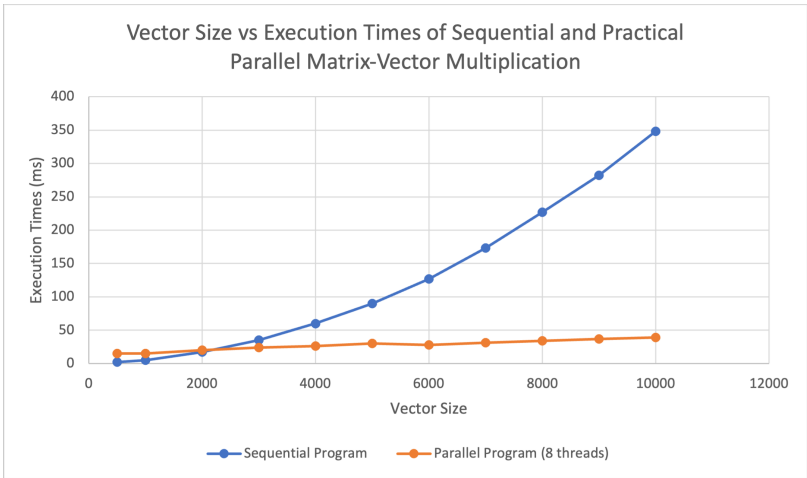

Figure 1: Vector size vs. Execution time

As seen in Figure 1, we compared the execution time of the sequential and parallel multipliers, running on 8 threads, with differing vector sizes. We observed that with a vector size, N, less than 2000, the sequential multiplier was more efficient due to its minimal overhead. However, for $N > 2000$, the parallel multiplier greatly outperformed the sequential multiplier which grew at and exponential level.

At around $N = 2000$, the two methods converged and produced similar execution times, with the sequential multiplier finishing in 17ms while the parallel multiplier finished in 20ms. Thus, as observed in Figure 2, the speed up at $N = 2000$ is equal to 0.85.

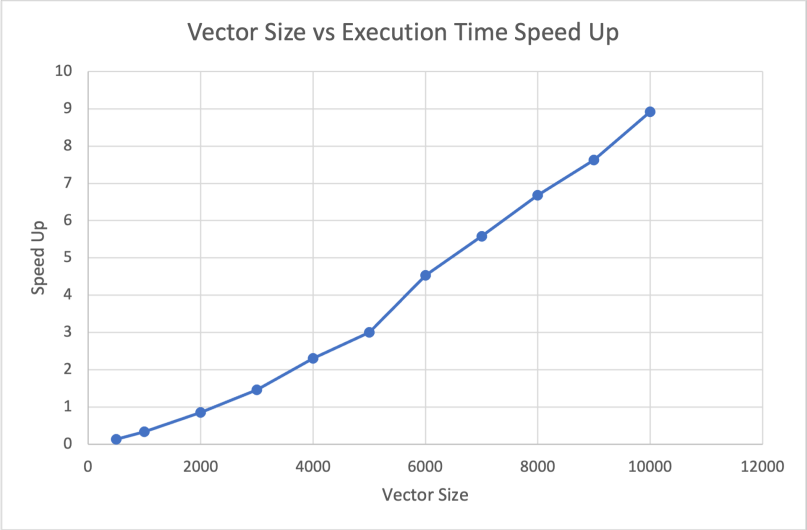

Figure 2: Vector size vs. Execution time speed up

4.4 Work and Critical Path

The highly parallel multiplier leverages the fact that if $C = M \cdot V$ then

$$\begin{pmatrix} C_0 \\ C_1 \end{pmatrix} = \begin{pmatrix} M_{00} & M_{01} \\ M_{10} & M_{11} \end{pmatrix} \begin{pmatrix} V_0 \\ V_1 \end{pmatrix} = \begin{pmatrix} M_{00} \cdot V_0 + M_{01} \cdot V_1 \\ M_{10} \cdot V_0 + M_{11} \cdot V_1 \end{pmatrix} \tag{1}$$

Therefore, at each step we are halving the vector and halving the matrix in two dimensions. Thus we will have $\Theta(N) \cdot \Theta(N) + \Theta(N) = \Theta(N^2)$ work nodes from dividing the problem to the base case of one element times one element. The critical path is only $\Theta(\log_2 N)$ as we are halving the size of the vector at each step. Thus, it will only take $\log_2 N$ steps to reach the base case.

Since the work is $\Theta(N^2)$ and the critical path is $\Theta(\log_2 N)$, the parallelism is equal to $\Theta(\frac{N^2}{\log_2 N})$. Since N^2 grows much faster than $\log_2 N$, as N gets larger the program can become nearly completley parallel.

Appendices

A Question 2 Code

Q2Contains.java

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

public class Q2Contains {
   public boolean contains(T item){
      int key = item.hashCode();

      ListNode prev;
      ListNode curr;

      if(head.next != Null){
         curr = head.next;
         prev = head;

         head.lock();
         curr.lock();
      }

      try{
         try{
            while(true){
               prev.unlock();
               prev = curr;
               curr = curr.next;
               curr.lock();

               if (curr.key == key){
                  return true;
               }
            }
         }
         finally{
            prev.unlock();
            curr.unlock();
         }
      }
```

```
      finally{
         return false;
      }
   }
}
```

Q2Test.java

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

import java.util.concurrent.*;

public class Q2Test {
    private static final int THREADS = 4;
    private static final int ITEMS = 1000;
    private static final int NUM_TESTS = 30;

    private static Q2Contains<Integer> ourList = new Q2Contains<>();

    public static void main(String[] args) {
        Random rand = new Random();

        ExecutorService executor = Executors.newFixedThreadPool(THREADS);


        for (int i = 0; i < NUM_TESTS; i++) {
            int testNum = rand.nextInt(ITEMS);
            ListRunnable runList = new ListRunnable(testNum);
            executor.execute(runList);
        }

        executor.shutdown();
    }

    private static class ListRunnable implements Runnable {
        int checkIndex;

        private ListRunnable(i) {
            this.checkIndex = i;
        }
```

```
        public void run() {
            for (int i = 0; i < ITEMS / THREADS; i++) {
                try {
                    ourList.add(i);
                }
            }

            if (ourList.contains(checkIndex)) {
                System.out.println("Item" + checkIndex + " found");
            } else {
                System.out.println("Item" + checkIndex + " not found");
            }
        }
    }
}
```

B Question 3 Code

BoundedQueue.java

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

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

/**
 *  Structure inspired by Chapter 10.3 of The Art of Multicore Programming
 **/
public class BoundedQueue<T> {

    private final T[] queue;
    private volatile int head;
    private volatile int tail;

    private final Lock deqLock = new ReentrantLock();
    private final Lock enqLock = new ReentrantLock();
    private final Condition notEmpty = deqLock.newCondition();
    private final Condition notFull = enqLock.newCondition();

    public BoundedQueue(int capacity){
```

```
        queue = (T[]) new Object[capacity];
        head = 0;
        tail = 0;
    }

    public void enq(T x) {
        boolean mustWakeDequeuers = false;

        enqLock.lock();
        try {
            while (tail - head == queue.length) {
                notFull.await();
            }

            if (tail == head) {
                mustWakeDequeuers = true;
            }

            queue[tail % queue.length] = x;
            tail++;
        } catch (InterruptedException ignored) {
        } finally {
            enqLock.unlock();
        }

        if (mustWakeDequeuers) {
            deqLock.lock();
            try {
                notEmpty.signalAll();
            } finally {
                deqLock.unlock();
            }
        }
    }

    public T deq() {
        boolean mustWakeEnqueuers = false;
        T result = null;

        deqLock.lock();
        try {
            while (tail == head) {
```

```
                notEmpty.await();
            }

            if (tail - head == queue.length) {
                mustWakeEnqueuers = true;
            }

            result = queue[head % queue.length];
            head++;
        } catch (InterruptedException ignored) {
        } finally {
            deqLock.unlock();
        }

        if (mustWakeEnqueuers) {
            enqLock.lock();
            try {
                notFull.signalAll();
            } finally {
                enqLock.unlock();
            }
        }
        return result;
    }
}
```

LockFreeBoundedQueue.java

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

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

public class LockFreeBoundedQueue<T> {
    private final AtomicReferenceArray<T> queue;

    private final AtomicInteger head, tail, size;
    private final int capacity;

    public LockFreeBoundedQueue(int capacity) {
        this.capacity = capacity;
```

```
        queue = new AtomicReferenceArray<>(capacity);

        head = new AtomicInteger(0);
        tail = new AtomicInteger(0);
        size = new AtomicInteger(0);
    }

    public void enq(T x) {
        int curSize = size.get();
        while (curSize == capacity || !size.compareAndSet(curSize, curSize + 1))
            curSize = size.get();

        queue.set(tail.getAndIncrement(), x);

        if (tail.get() == capacity) {
            tail.set(0);
        }
    }

    public T deq() {
        int curSize = size.get();
        while (curSize == 0 || !size.compareAndSet(curSize, curSize - 1))
            curSize = size.get();

        T result = queue.getAndSet(head.getAndIncrement(), null);

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

        return result;
    }
}
```

C Question 4 Code

Matrix.java

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

public class Matrix {
```

```
int dim;
double[][] data;
int rowDisplace, colDisplace;

public Matrix(int d) {
    dim = d;
    rowDisplace = colDisplace = 0;
    data = new double[d][d];
}

public Matrix(double[][] matrix) {
    data = matrix;
    rowDisplace = colDisplace = 0;
    dim = matrix.length;
}

private Matrix(double[][] matrix, int x, int y, int d) {
    data = matrix;
    rowDisplace = x;
    colDisplace = y;
    dim = d;
}

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

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

public int getDim() {
    return dim;
}

public Matrix[][] split() {
    Matrix[][] result = new Matrix[2][2];
    int newDim = dim / 2;

    result[0][0] = new Matrix(data, rowDisplace, colDisplace, newDim);
    result[0][1] = new Matrix(data, rowDisplace, colDisplace + newDim, newDim);
    result[1][0] = new Matrix(data, rowDisplace + newDim, colDisplace, newDim);
```

```
        result[1][1] = new Matrix(data, rowDisplace + newDim, colDisplace + newDim, newDim);

        return result;
    }
}
```

Vector.java

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

import java.util.Arrays;

public class Vector {
    int dim;
    double[] data;
    int displace;

    public Vector(int d) {
        dim = d;
        displace = 0;
        data = new double[d];
    }

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

    private Vector(double[] vector, int x, int d) {
        data = vector;
        displace = x;
        dim = d;
    }

    public double get(int i) {
        return data[i + displace];
    }

    public void set(int i, double value) {
        data[i + displace] = value;
```

```
    }

    public int getDim() {
        return dim;
    }

    public Vector[] split() {
        Vector[] result = new Vector[2];
        int newDim = dim / 2;

        result[0] = new Vector(data, displace, newDim);
        result[1] = new Vector(data, displace + newDim, newDim);

        return result;
    }

    public double[] getData() {
        return Arrays.copyOf(data, data.length);
    }
}
```

SequentialMultiplier.java

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

public class SequentialMultiplier {
    /**
     * Returns the result of a sequential matrix-vector multiplication
     * The matrix and vector are randomly generated
     *
     * @param M is the matrix
     * @param v is the vector
     * @return the result of the multiplication
     */
    public static double[] multiply(double[][] M, double[] v) {
        // check if the dimensions are correct
        // the number of columns in M should be equal to the number of elements in v
        if (M[0].length != v.length) {
            try {
                throw new Exception("Invalid input");
            } catch (Exception e) {
```

```
                e.printStackTrace();
            }
        }

        int m = M.length;
        int n = v.length;
        double[] prod = new double[m];

        /*
        multiply each row of matrix a with the vector
         */
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                prod[i] += M[i][j] * v[j];
            }
        }

        return prod;
    }

}
```

PracticalParallelMultiplier.java

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

import ca.mcgill.ecse420.a3.utils.Matrix;
import ca.mcgill.ecse420.a3.utils.Vector;

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

public class ParallelMultiplier {

    private static ExecutorService executor;

    /**
     * Returns the result of a concurrent matrix-vector multiplication
     * The matrix and vector are randomly generated
     *
```

```
     * @param M is the matrix
     * @param v is the vector
     * @return the result of the multiplication
     */
    public static double[] multiply(double[][] M, double[] v) {
        int m = M.length;
        int _n = M[0].length;
        int n = v.length;

        // check if the dimensions are correct
        // the number of columns in M should be equal to the number of elements in v
        if (n != _n || m != n) {
            try {
                throw new Exception("Invalid input");
            } catch (Exception e) {
                e.printStackTrace();
            }
        }

        // Creating threads
        executor = Executors.newCachedThreadPool();

        Matrix M_matrix = new Matrix(M);
        Vector v_vector1 = new Vector(v);
        Vector v_vector = new Vector(v);
        Vector c_vector = new Vector(n);

        try {
            Future<?> future = executor.submit(new MultiplyTask(M_matrix, v_vector, c_vector));
            future.get();
        } catch (Exception e) {
            e.printStackTrace();
        }

        executor.shutdown();

        return c_vector.getData();
    }


    public static class MultiplyTask implements Runnable {
```

```
    private Matrix M;
    private Vector v, c, lhs, rhs;

    public MultiplyTask(Matrix M, Vector v, Vector c) {
        this.M = M;
        this.v = v;
        this.c = c;

        this.lhs = new Vector(M.getDim());
        this.rhs = new Vector(M.getDim());
    }

    /**
     * Multiplies the given row and inputs the result in the prod matrix
     */
    public void run() {
        try {
            if (M.getDim() == 1) {
                c.set(0, M.get(0, 0) * v.get(0));
            } else {
                Matrix[][] splitM = M.split();
                Vector[] splitV = v.split();
                Vector[] splitLHS = lhs.split();
                Vector[] splitRHS = rhs.split();

                Future<?>[][] future = (Future<?>[][]) new Future[2][2];
                for (int i = 0; i < 2; i++) {
                    future[i][0] = executor.submit(new MultiplyTask(splitM[i][0], splitV[0],
                        splitLHS[i]));
                    future[i][1] = executor.submit(new MultiplyTask(splitM[i][1], splitV[1],
                        splitRHS[i]));
                }

                for (int i = 0; i < 2; i++) {
                    for (int j = 0; j < 2; j++) {
                        future[i][j].get();
                    }
                }

                Future<?> done = executor.submit(new AdditionTask(lhs, rhs, c));
                done.get();
            }
```

```
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

}

public static class AdditionTask implements Runnable {

    private Vector a, b, c;

    public AdditionTask(Vector a, Vector b, Vector c) {
        this.a = a;
        this.b = b;
        this.c = c;
    }

    public void run() {
        try {
            if (a.getDim() == 1) {
                c.set(0, a.get(0) + b.get(0));
            } else {
                Vector[] splitA = a.split();
                Vector[] splitB = b.split();
                Vector[] splitC = c.split();

                Future<?>[] future = (Future<?>[]) new Future[2];

                for (int i = 0; i < 2; i++) {
                    future[i] = executor.submit(new AdditionTask(splitA[i], splitB[i], splitC[i]));
                }

                for (int i = 0; i < 2; i++) {
                    future[i].get();
                }
            }
        } catch (
                Exception ex) {
            ex.printStackTrace();
        }
    }
```

```
    }
}
```

PracticalParallelMultiplier.java

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

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

public class PracticalParallelMultiplier {

    /**
     * Returns the result of a concurrent matrix-vector multiplication
     * The matrix and vector are randomly generated
     *
     * @param M is the matrix
     * @param v is the vector
     * @return the result of the multiplication
     */
    public static double[] multiply(double[][] M, double[] v, int threadCount) {
        int m = M.length;
        int _n = M[0].length;
        int n = v.length;

        // check if the dimensions are correct
        // the number of columns in M should be equal to the number of elements in v
        if (n != _n || m != n) {
            try {
                throw new Exception("Invalid input");
            } catch (Exception e) {
                e.printStackTrace();
            }
        }

        // Creating threads
        ExecutorService executor = Executors.newFixedThreadPool(threadCount);

        /*
```

```
        Create a thread for each row of a that is going to be multiplies, each thread will take take of
            one row multiplication
         */
        double[] prod = new double[m];
        for (int i = 0; i < m; i++) {
            executor.execute(new DotProduct(prod, M, v, i));
        }

        executor.shutdown();

        // waiting for threads to complete their execution
        try {
            executor.awaitTermination(m, TimeUnit.SECONDS);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        return prod;
    }

    public static class DotProduct implements Runnable {

        private double[] prod;
        private double[][] M;
        private double[] v;
        private final int row;

        /**
         * Constructor for the RowMultiply class
         *
         * @param prod product vector till now
         * @param M   matrix
         * @param v   vector
         * @param row row number that is being multiplies by the current thread
         */
        public DotProduct(double[] prod, double[][] M, double[] v, int row) {
            this.prod = prod;
            this.M = M;
            this.v = v;
            this.row = row;
        }
```

```
        /**
         * Multiplies the given row and inputs the result in the prod matrix
         */
        public void run() {
            prod[row] = 0;
            for (int i = 0; i < v.length; i++) {
                prod[row] += M[row][i] * v[i];
            }
        }
    }
}
```

MatrixVectorMultiplication.java

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

import java.util.Date;

public class MatrixVectorMultiplication {

    private static final int NUMBER_THREADS = 8;
    private static final int MATRIX_SIZE = 10000;


    public static void main(String[] args) {

        // Generate two random matrices, same size
        double[][] M = generateRandomMatrix(MATRIX_SIZE, MATRIX_SIZE);
        double[] v = generateRandomVector(MATRIX_SIZE);

        Date start, end;

        start = new Date();
        double[] out_seq = SequentialMultiplier.multiply(M, v);
        end = new Date();
        System.out.println("\nsequential multiplication (milli seconds): " + (end.getTime() -
            start.getTime()));

        start = new Date();
        double[] out_par = ParallelMultiplier.multiply(M, v);
        end = new Date();
```

```
        System.out.println("\nparallel multiplication (milli seconds): " + (end.getTime() -
            start.getTime()));

        start = new Date();
        double[] out_prac_par = PracticalParallelMultiplier.multiply(M, v, NUMBER_THREADS);
        end = new Date();
        System.out.println("\npractical parallel multiplication (milli seconds): " + (end.getTime() -
            start.getTime()));

    }

    /**
     * Populates a matrix of given size with randomly generated integers between 0-10.
     *
     * @param numRows number of rows
     * @param numCols number of cols
     * @return matrix
     */
    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;
    }

    /**
     * Populates a vector of given size with randomly generated integers between 0-10.
     *
     * @param num number of elements
     * @return vector
     */
    private static double[] generateRandomVector(int num) {
        double vector[] = new double[num];
        for (int i = 0; i < num; i++) {
            vector[i] = (double) ((int) (Math.random() * 10.0));
        }
        return vector;
    }
```

```
}
```