ECSE 420 Parallel Computing **Dennis Liu #260581270**
Assignment #3 **Yijie Zhou #260622641**

Question 1

Deadlock is possible when re-using its own node instead of its predecessor node. For instance, let Thread1 and Thread2 be our two concurrent threads, and Qnode1, Qnode2 be Qnode of each thread.

Deadlock case:

Step1: Thread1 and Thread2 didn't acquired the lock:

Global Queue: Tail →Null

Step2: Thread1 try to acquire the lock, and it did.

Tail →Qnode1(true)

Step3: Thread2 try to acquire the lock, however it must wait because Thread1 have it.

Tail →Qnode2(true) →Qnode1(true)

Step4: Thread1 releases the lock

Tail →Qnode2(true) →Qnode1(false)

Step5: Somehow right before Thread2 check if Qnode1 release the lock, Thread1 try to re-acquire the lock. Qnode1's lock flag is true now.

Tail →Qnode1(true) →Qnode2(true) →Qnode1(true)

Because Qnode2 points to Qnode1 as its predecessor, and Qnode1 didn't know. Qnode2 thinks that Qnode1 still holds the lock, and Qnode1 have to wait Qnode2 to release it.

Deadlock occurs and no threads will be able to acquire the lock.

Question 2.1

Code is available in fine_grainedList.java

Question 2.2

Test is available in test_fine_grained.java

We first lock the head as predecessor and then lock its next node as current node, then we iterate by unlock predecessor and lock the next node of current node. This is the same linearization techniques used in both addition and removal methods. In this way, they follow the same order of acquiring locks as they scroll through the list.

Question 3.1

Implementation can be found in bounded.java and tested in test_boundedlock.java

Question 3.2

The main problem is to execute the insertions and removals when the queue is full or empty. How to pause a thread without using lock is very difficult. The big problem is how to put the threads to standby and how to return them to the situations where they were interrupted.

Question 4.1

Implemented in MatrixVectorSequentialMultiply.java. Can be tested in Question4.java with testSequential() method.

Question 4.2

Listing 1: Parallel Vector Adder

```

public class VectorParallelAdd implements Callable<double[]>{
  double[] a, b, result;
  static ExecutorService exec;

  // constructor
  public VectorParallelAdd( double[] a, double[] b, ExecutorServi
  this.a = a;
  this.b = b;
  this.exec = exec;
  }

  // call() returns result for future.get()
  public double[] call() throws Exception {
    int n = a.length;
    result = new double[n];

    if (n == 1) {
      // base case one element in each array
      result[0] = a[0] + b[0];

    } else {
      // split vectors into halves
      double[] a1 = Arrays.copyOfRange( a, 0, n/2 );
```

```
      double[] a2 = Arrays.copyOfRange( a, n/2, n );
      double[] b1 = Arrays.copyOfRange( b, 0, n/2 );
      double[] b2 = Arrays.copyOfRange( b, n/2, n );

      // create new VectorAdd jobs with array halves, pass to sam
      Future<double[]> f1 = exec.submit( new VectorParallelAdd( a
      Future<double[]> f2 = exec.submit( new VectorParallelAdd( a

      // get array results when completed
      while (!f1.isDone() || !f2.isDone()) {
      }
      double[] c1 = f1.get();
      double[] c2 = f2.get();

      // concatenate two halves of the result
      result = DoubleStream.concat(Arrays.stream(c1), Arrays.stre

    }
    return result;
  }
}
```

Listing 2: Parallel Matrix Vector Multiplier

```
public class MatrixVectorParallelMultiply implements Callable<dou
  double[][] a;
  double[] b, result;
  int n, m;
  static ExecutorService exec;

  // constructor
  public MatrixVectorParallelMultiply( double[][] a, double[] b,
  this.a = a;
  this.b = b;
  this.exec = exec;
  n = a.length;
  m = a[0].length;
  result = new double[n];

```

```
    }

  //call() returns result for future.get()
   public double[] call() throws Exception {

     if (n == 1 && m == 1) {

       // base case one element in each vector
       result[0] = a[0][0] * b[0];

     } else if (n == 1 && m == 2) {

       // base case 1 x 2 vector multiply with 2 x 1 vector
       result[0] = a[0][0] * b[0] + a[0][1] * b[1];

     } else if (n == 2 && m == 1) {

       // base case 2 x 1 vector multiply with 1 x 1 vector
       result[0] = a[0][0] * b[0];
       result[1] = a[1][0] * b[0];

     } else {

       // split matrix into 4, vector into 2
       double[][] a11 = deepCopy( a, 0, n/2, 0, m/2 );
       double[][] a12 = deepCopy( a, 0, n/2, m/2, m );
       double[][] a21 = deepCopy( a, n/2, n, 0, m/2 );
       double[][] a22 = deepCopy( a, n/2, n, m/2, m );
       double[] b11 = Arrays.copyOfRange( b, 0, m/2 );
       double[] b21 = Arrays.copyOfRange( b, m/2, m );

       // create new MatrixVectorParallelMultiply jobs with matrix
       Future<double[]> f1 = exec.submit( new MatrixVectorSequentia
       Future<double[]> f2 = exec.submit( new MatrixVectorSequentia
       Future<double[]> f3 = exec.submit( new MatrixVectorSequentia
       Future<double[]> f4 = exec.submit( new MatrixVectorSequentia

       // get array results when completed
       while (!f1.isDone() || !f2.isDone() || !f3.isDone() || !f4.i
```

```java
    }

    // add multiplication vector results
    Future<double[]> f5 = exec.submit( new VectorParallelAdd( f1
    Future<double[]> f6 = exec.submit( new VectorParallelAdd( f3

    // get array results when completed
    while (!f5.isDone() || !f6.isDone()) {
    }

    // concatenate two halves of the result
    result = DoubleStream.concat(Arrays.stream(f5.get()), Arrays

  }
  return result;
}

private double[][] deepCopy (double[][] in, int startRow, int en

  double[][] out = new double[endRow - startRow][endCol - startC

  for(int i = startRow; i < endRow; i++)
    for(int j = startCol; j < endCol; j++)
      out[i - startRow][j - startCol] = in[i][j];

  return out;
}
}
```

The parallel vector adder operates in constant time in its base case and $\Theta(n)$ when concatenating its results. It also splits the task into two separate parallel vector adder tasks each with size n/2.

Thus its work is:

$$A_1(n) = 2A_1(n/2) + \Theta(1) = \Theta(n) \tag{1}$$

Its critical path length is:

$$A_1(n) = A_1(n/2) + \Theta(1) = \Theta(log\ n) \tag{2}$$

The parallel matrix vector multiplier calls the parallel vector adder twice in each combination step. It also splits the task into four separate parallel matrix vector multiplier tasks each with size n/2.

Thus its work is:

$$M_1(n) = 4M_1(n/2) + 2\Theta(n) = \Theta(n^2) \tag{3}$$

Its critical path length is:

$$M_1(n) = M_1(n/2) + \Theta(log\ n) = \Theta(log^2\ n) \tag{4}$$

Question 4.3

Implemented in MatrixVectorParallelMultiply.java. Can be tested against sequential multiplier in Question4.java with verify() method. The execution time for the sequential multiplier is 12 milliseconds, while the execution time for the parallel multiplier is 90 milliseconds. This is likely due to the copying operation of the method that operates in $\Theta(n)$ time when splitting the matrix and vector into quarters and halves. If implemented in a language such as Python with in-place array slicing functionality, the parallel method should out-perform the sequential method.

The parallel method uses 4 threads, and has a serial fraction of 1/3. Theoretically it should give a speed up of:

$$S_{speedUp}(n) = \frac{1}{(1-p) + \frac{p}{n}} = \frac{1}{(1-\frac{1}{3}) + \frac{\frac{1}{3}}{4}} = \frac{4}{3} \tag{5}$$

Question 4.4

The sequential multiplier operates in $\Theta(n^2)$

Thus the parallel multiplier has work:

$$M_1(n) = 4\Theta(n^2) + 2\Theta(n) = \Theta(n^2) \tag{6}$$

Its critical path length is:

$$M_1(n) = \Theta(n^2) + \Theta(log\ n) = \Theta(n^2) \tag{7}$$

Its parallelism is equal to its max speedup which is the time needed on one processor divided by the time needed on as many processors as needed. 4 maximum processors are needed, and the time needed for a large multiplication problem is roughly 4 times with one processor than with 4. Thus the parallellism is roughly 4.