ECSE 420 - A1 Report

Alexander Liu - 261117058, Anthony Zhao - 260944810

1.1 Sequential Implementation

We start with our two input matrices and compute bounds represented by the rows, common, and cols integers. The common integer represents the constraint that the number of columns in matrix a must equate the number of rows in matrix b. We take the transpose of b to leverage cache locality and speed up the runtime performance of our sequential implementation (this step is not strictly necessary, but we thought it would be interesting to include). After these steps, we can iterate over both input matrices to compute the final by using standard matrix multiplication from linear algebra. Figure 1 illustrates the implementation in MatrixMultiplication.java.

```
public static Double[][] sequentialMultiplyMatrix(
            Double[][] a,
            Double[][] b) {
        validateInputMatrices(a, b);

        Integer rows = a.length;
        Integer common = a[0].length;
        Integer cols = b[0].length;

        Double[][] bTranspose = computeTranspose(b);
        Double[][] res = new Double[rows][cols];

        for (int r = 0; r < rows; r++) {
            for (int c = 0; c < cols; c++) {
                double sum = 0.0;

                for (int i = 0; i < common; i++) {
                    sum += a[r][i] * bTranspose[c][i];
                }

                res[r][c] = sum;
            }
        }

        return res;
    }
```

Figure 1. Sequential matrix multiplication implementation

To validate this method, a test suite at MatrixMultiplicationTest.java (see appendix) was created. We used gradle as our project's build tool, so we could run the tests for each of the project's modules.The goal with unit testing was to ensure proper functionality, so we cover multiple cases like matrices with negative values, zero matrices, rectangular matrices, matrices with mismatching sizes, etc.

1.2 Parallel Implementation

The parallel implementation uses a row-stripped approach. See Figure 2 below for the concrete implementation.

```
public static Double[][] parallelMultiplyMatrix(
            Double[][] a,
            Double[][] b,
            Integer numThreads) {
        validateInputMatrices(a, b);

        Integer rows = a.length;
        Integer cols = b[0].length;

        Double[][] bTranspose = computeTranspose(b);
        Double[][] res = new Double[rows][cols];

        ExecutorService executor = Executors.newFixedThreadPool(numThreads);
        Future<?>[] futures = new Future<?>[rows];

        for (int r = 0; r < rows; r++) {
            futures[r] = executor.submit(
                    new MatrixMultiplicationTask(r, a, bTranspose, res));
        }

        for (Future<?> f : futures) {
            try {
                f.get();
            } catch (Exception e) {
                throw new RuntimeException(e);
            }
        }

        executor.shutdown();
        return res;
    }
```

Figure 2. Parallel matrix multiplication implementation

In the sequential implementation, we would iterate over each row of matrix a and compute the dot products for that row against all columns of matrix b. In this new implementation, we create a fixed thread pool bounded by the number of threads defined by NUMBER_THREADS. Each worker thread in the pool runs a task class called MatrixMultiplicationTask (see appendix) that takes a single row in matrix a and performs all computations against all the columns in matrix b. Then, each worker thread updates the resulting res matrix accordingly. Since we are only reading from input matrices, and the write operations to the resulting matrix are independent (i.e. different rows write to different locations), we don't need explicit synchronization between workers of the thread pool. You may find the implementation of `MatrixMultiplicationTask` in Figure 3 below.

```java
public class MatrixMultiplicationTask implements Runnable {

    Integer rowIx; // Index of row in matrix A
    Double[][] a;
    Double[][] b;
    Double[][] res;

    /**
     * Constructs a matrix multiplication task for a specific row.
     *
     * @param rowIx the index of the row in matrix A to compute
     * @param a     the first matrix (matrix A)
     * @param b     the second matrix (matrix B, should be transposed)
     * @param res   the result matrix where computed values will be stored
     */
    public MatrixMultiplicationTask(
            Integer rowIx,
            Double[][] a,
            Double[][] b,
            Double[][] res) {
        this.rowIx = rowIx;
        this.a = a;
        this.b = b;
        this.res = res;
    }

    /**
     * Computes one row of the matrix multiplication result.
     * This method is called when the task is executed by a thread.
     * It calculates all column values for the assigned row index.
     */
    public void run() {
        Integer common = a[0].length;
        Integer cols = res[0].length;

        for (int col = 0; col < cols; col++) {
            double sum = 0.0;

            for (int i = 0; i < common; i++) {
                sum += a[rowIx][i] * b[col][i];
            }

            res[rowIx][col] = sum;
        }
    }
}
```

Figure 3. Parallel matrix multiplication task definition

Similar to the sequential solution, validation is performed via the same unit test suite. You will see in each unit test that we test both the sequential and parallel implementations to

ensure that they are both consistent with respect to what we expect, but also consistent among themselves.

1.3 Benchmarking

We added two methods to benchmark execution time in `MatrixMultiplication.java`: benchmarkSequential and benchmarkParallel (see appendix). The number of workers in the thread pool was set to 5 given the precondition that we have multiple processors.

Both sequential and parallel implementations were run against the same inputs. To ensure consistency, we ran the benchmarks multiple times and manually verified that the runtimes were on the same scale for each matrix size. To check the plausibility of our results, we relied on intuition. For example, small matrices should run very quickly whereas larger matrices naturally take longer. We also checked that the runtimes of the parallel solution scaled better than the sequential one as matrix size scales (more details in section 1.5).

1.4 Varying Number of Threads

We implemented the runBenchmarkByThreads in `MatrixMultiplication.java` (see appendix) method to collect a baseline runtime for our sequential solution as well as execution times for our parallel solution while varying the number of threads in the pool between 1 and 40. The upper bound of 40 was chosen because I had a 2 hour window where I had to go on a bike ride and could run the benchmarks on my machine. Refer to Figure 4 for the parallel runtimes.

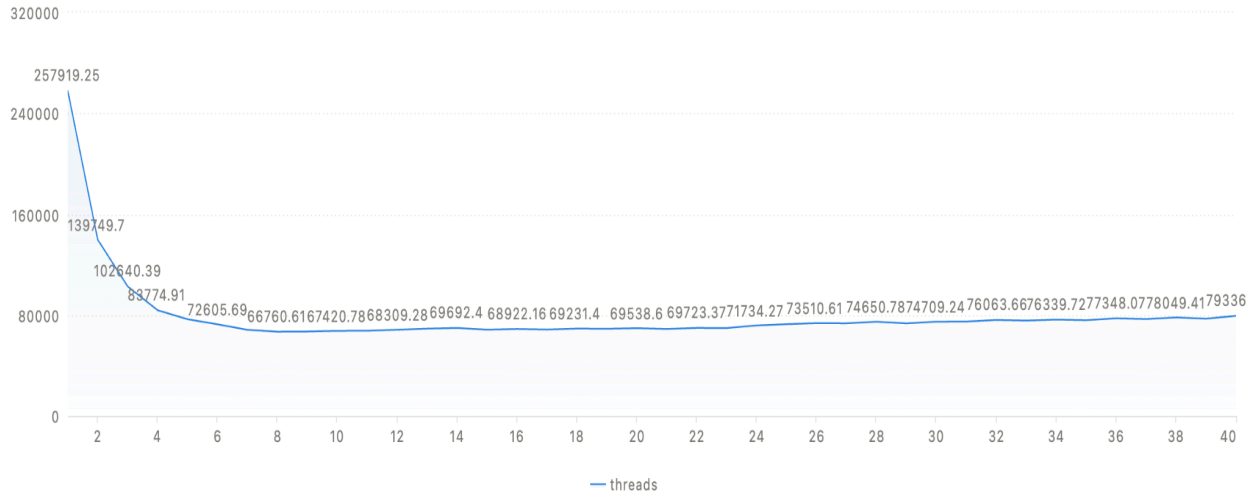

Figure 4. Runtime (ms) plotted against number of threads

From the output of our runBenchmarkByThreads method (see appendix), our sequential baseline took ~258s to multiply two 4000x4000 matrices. Our parallel implementation with a single worker thread exhibits virtually the same runtime (~258s). At 2 workers, we have a runtime of ~140s which yields a speed-up of 258s / 140s ~= 1.84x. At 4 threads, we have a speed up of 258s / 84s ~= 3.07x. At 8 threads, we have our fastest runtime with a speed-up of 258s / 67s ~= 3.85x. After 8 workers, the runtime plateaus and eventually worsens. We can deduce that the speed-up also plateaus and becomes worse.

1.5 Varying Input Size

We implemented the runBenchmarkByMatrixSizes method in `MatrixMultiplication.java` (see appendix) method to collect runtimes for our sequential and parallel solutions while varying the matrix sizes as specified by the assignment guidelines. Refer to Figure 5 for the runtimes vs input sizes.

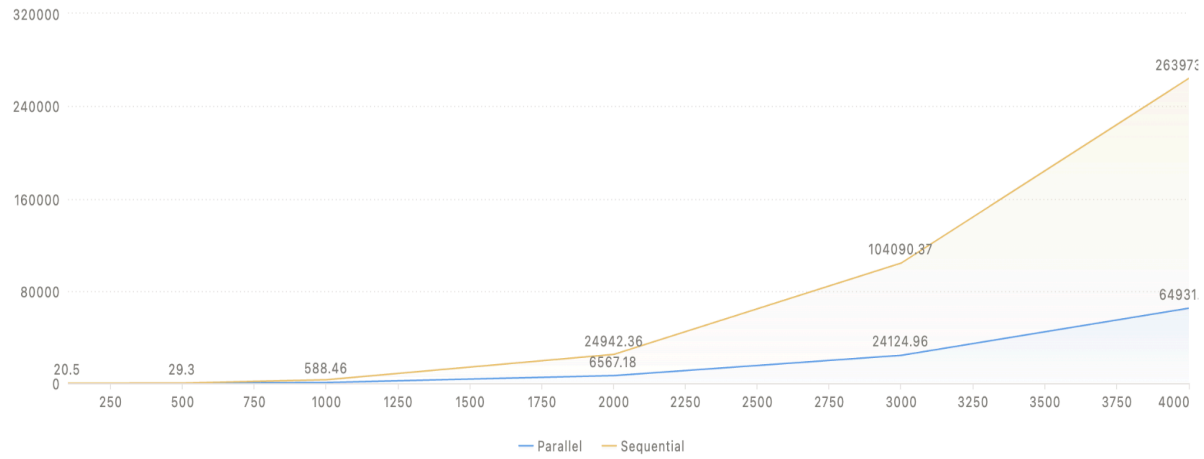

Figure 5. Runtimes (ms) plotted against input size for sequential and parallel implementations

We used 8 threads in our thread pool to compute these runtimes as 8 threads yielded the best performance from Section 1.4. You may consult the appendix to see the output of the runBenchmarkByMatrixSizes method.

1.6 Interpretations

In Figure 4, we notice a sharp increase in performance as we go from 1 worker to 8, followed by a plateau and then a slight loss in runtime as the number of threads increases beyond 8. Having a pool of 1 worker is the same as having a sequential implementation (with a little extra overhead) which explains the nearly identical runtimes. As the number of workers increases, more physical processor cores on the machine are leveraged. The reason performance plateaus beyond 8 workers is because the machine the benchmarks were run on (my macbook) only has 8 physical cores. As such, having a large amount of workers hinders performance as runtime cost is incurred through management overhead that comes with such a large number of workers.

In Figure 5, we notice that the parallel solution scales much better than the sequential one. The reason is that more threads gives us more workers to solve smaller parts of the larger problem in parallel. For small matrices, the sequential solution exhibits similar/better runtime. This is because the costs related to thread creation/management actually outweigh the performance gains of parallelism at such small matrix sizes. We have also seen in class that parallel solutions generally tend to underperform in comparison to sequential ones when the input problem is small. We have deduced from section 1.4 that the best speed-up we obtained is with 8 workers at 3.85x ~ 4x. Within the plot, we notice this speed-up relation to be generally true (and especially visible) for larger matrix sizes.

2.1 Causing Deadlocks

```
public class Deadlock {

    private static class DeadlockThread1 extends Thread {

        public void run() {
            synchronized (lock1) {
                System.out.println("Thread 1 holding lock 1...");

                try {
                    Thread.sleep(1000); // sleep to increase chances of deadlock
                } catch (InterruptedException e) {
                    // ignore exception
                }

                System.out.println("Thread 1 waiting for lock 2...");

                synchronized (lock2) {
                    System.out.println("Thread 1 holding locks 1 and 2...");
                }
            }
        }
    }

    private static class DeadlockThread2 extends Thread {

        public void run() {
            synchronized (lock2) {
                System.out.println("Thread 2 holding lock 2...");

                try {
                    Thread.sleep(1000); // sleep to increase chances of deadlock
                } catch (InterruptedException e) {
                    // ignore exception
                }

                System.out.println("Thread 2 waiting for lock 1...");

                synchronized (lock1) {
                    System.out.println("Thread 2 holding locks 1 and 2...");
                }
            }
        }
    }

    public static Object lock1 = new Object();
    public static Object lock2 = new Object();

    public static void main(String[] args) {
        DeadlockThread1 t1 = new DeadlockThread1();
```

```
        DeadlockThread2 t2 = new DeadlockThread2();

        t1.start();
        t2.start();
    }
}
```

Figure 6. Program causing a deadlock

We will give a brief overview of how the Deadlock.java as seen in Figure 6 program demonstrates a deadlock. Suppose there are two shared resources: A & B . Naturally, we would implement locks to ensure proper synchronization and safe access to these shared resources (in this example, there are no resources, but we have 2 locks). Suppose now we have two threads: t1 & t2. t1 tries to access A, then B. t2 on the other hand tries to access B, then A. Given this, there is an interleaving where t1 acquires A, t2 acquires B, and both are stuck waiting for each other to relinquish the second resource they both need. This interleaving leads to a deadlock — both threads are stuck waiting and nothing happens!

2.2 Avoiding Deadlocks

In general, we can modify a program using some design patterns to avoid deadlocks. The simplest fix would be to implement resource ordering. This means that both threads would try acquiring the locks in the same order. Thus, if a thread can't obtain the first lock, it blocks which removes the risk of contention with other resources downstream. Another solution would be to implement a non-blocking acquisition. For example, a thread would try to obtain a lock with a defined timeout interval. If the timeout expires, then the thread backs off by relinquishing its locks. Yet another solution would be to reduce the scope of the lock. By minimizing the critical section, we reduce time holding the lock and reduce the chances of lock contention.

By far the easiest solution is to implement resource ordering. You may find the previous program modified with resource ordering to prevent deadlocks from occurring called ResourceOrdering.java in Figure 7 below.

```
public class ResourceOrdering {

    private static class ResourceOrderingThread extends Thread {

        private Integer threadNum;

        public ResourceOrderingThread(Integer threadNum) {
            this.threadNum = threadNum;
        }

        public void run() {
            synchronized (lock1) {
                System.out.println(
                    String.format("Thread %d holding lock 1...", threadNum)
                );
```

```
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    // ignore exception
                }

                System.out.println(
                    String.format("Thread %d waiting for lock 2...", threadNum)
                );

                synchronized (lock2) {
                    System.out.println(
                        String.format(
                            "Thread %d holding locks 1 and 2...",
                            threadNum
                        )
                    );
                }
            }
        }
    }

    public static Object lock1 = new Object();
    public static Object lock2 = new Object();

    public static void main(String[] args) {
        ResourceOrderingThread t1 = new ResourceOrderingThread(1);
        ResourceOrderingThread t2 = new ResourceOrderingThread(2);

        t1.start();
        t2.start();
    }
}
```

Figure 7. Resource ordering example

3.1 Preventing Simultaneous Chopstick Access

The program prevents two philosophers from holding the same chopstick by using Java's synchronized blocks with each chopstick object as a monitor lock. The JVM guarantees that only one thread can hold the lock on any given object at a time. When a philosopher enters a synchronized (chopstick) block, other philosophers trying to access that same chopstick must wait until the lock is released when the first philosopher exits the synchronized block. The following code in Figure 8 demonstrates the implementation for using the synchronized keyword.

```java
private void pickUpChopsticks() {
    // all philosophers try to pick up left chopstick first, which would eventually create deadlock
    System.out.println("Philosopher " + id + " is trying to pick up left chopstick");
    synchronized (leftChopstick) {
        System.out.println("Philosopher " + id + " picked up left chopstick");

        System.out.println("Philosopher " + id + " is trying to pick up right chopstick");
        synchronized (rightChopstick) {
            System.out.println("Philosopher " + id + " picked up right chopstick");
        }
    }
}
```

Figure 8. Implementation of pickUpChopsticks() method in pickUpChopsticks.java

Why Deadlock Will Eventually Occur

Deadlock occurs eventually because all philosophers follow the same strategy: pick up the left chopstick first, then the right chopstick. When all philosophers simultaneously grab their left chopstick, each one holds exactly one chopstick and needs their right chopstick to proceed. However, each philosopher's right chopstick is actually another philosopher's left chopstick, creating a circular wait where everyone waits for a resource held by someone else. Since no philosopher will release their chopstick until they finish eating, and no one can eat without both chopsticks, they all become permanently blocked. The following screenshot in Figure 9 shows the output of the program that created a deadlock.

```
Philosopher 1 is thinking
Philosopher 0 is thinking
Philosopher 3 is thinking
Philosopher 2 is thinking
Philosopher 4 is thinking
Philosopher 1 is trying to pick up left chopstick
Philosopher 3 is trying to pick up left chopstick
Philosopher 2 is trying to pick up left chopstick
Philosopher 2 picked up left chopstick
Philosopher 1 picked up left chopstick
Philosopher 4 is trying to pick up left chopstick
Philosopher 0 is trying to pick up left chopstick
Philosopher 4 picked up left chopstick
Philosopher 0 picked up left chopstick
Philosopher 1 is trying to pick up right chopstick
Philosopher 4 is trying to pick up right chopstick
Philosopher 3 picked up left chopstick
Philosopher 3 is trying to pick up right chopstick
Philosopher 2 is trying to pick up right chopstick
Philosopher 0 is trying to pick up right chopstick
```

Figure 9. Deadlock output from running DiningPhilosophersDeadlock.java

As we can see, all philosophers are trying to pick up the right chopstick and everyone got stuck. This creates a deadlock.

3.2 Deadlock Prevention Solution

We decided to use ReentrantLock to prevent deadlock. More specifically, we added a timeout-based lock acquisition with ReentrantLock and implemented an all-or-nothing strategy where philosophers must acquire both chopsticks within the timeout period or

release any they're holding. Each philosopher uses tryLock(100, TimeUnit.MILLISECONDS) instead of blocking indefinitely on lock acquisition. If a philosopher successfully acquires their left chopstick but cannot get their right chopstick within 100ms, they immediately release the left chopstick in the finally block and retry after a brief pause. This approach breaks the "hold and wait" deadlock condition because no philosopher holds a chopstick indefinitely while waiting for another. Even if all philosophers simultaneously acquire their left chopstick, the timeout mechanism forces them to release it and retry, preventing the circular wait that causes deadlock. The provided code snipped in Figure 10 illustrates a significant logic alteration designed to prevent deadlock when picking up chopsticks.

```
private boolean pickUpChopsticks() throws InterruptedException {
    boolean leftAcquired = false;
    boolean rightAcquired = false;
    try {
        leftAcquired = leftChopstick.tryLock(timeoutMs, java.util.concurrent.TimeUnit.MILLISECONDS);
        if (leftAcquired) {
            System.out.println("Philosopher " + id + " picked up left chopstick.");
            rightAcquired = rightChopstick.tryLock(timeoutMs, java.util.concurrent.TimeUnit.MILLISECONDS);
            if (rightAcquired) {
                System.out.println("Philosopher " + id + " picked up right chopstick.");
                return true;
            } else {
                System.out.println("Philosopher " + id + " failed to pick up right chopstick.");
                return false;
            }
        } else {
            // Failed to acquire left chopstick
            return false;
        }
    } finally {
        if (!rightAcquired && leftAcquired) {
            leftChopstick.unlock();
            System.out.println("Philosopher " + id + " put down left chopstick.");
        }
    }
}
```

Figure 10. pickUpChopsticks() with ReentrantLock in DiningPhilosophers.java

Is Starvation Possible?

Yes, starvation is still possible with the basic ReentrantLock solution. The basic ReentrantLock does not provide fairness guarantees, so some philosophers might consistently lose the competition for chopsticks to their neighbours. A philosopher could theoretically keep failing to acquire both chopsticks due to unlucky timing while other philosophers successfully eat repeatedly. Since there's no mechanism to prioritize philosophers who have been waiting longer or failed multiple attempts, individual philosophers could be starved indefinitely even though the system as a whole continues making progress.

Preventing Starvation

We prevent starvation by adding the fairness property to ReentrantLock using ReentrantLock(true). Fair ReentrantLocks maintain a FIFO queue of waiting threads and guarantees that the thread that has been waiting the longest will acquire the lock next. This ensures that no philosopher can be indefinitely bypassed by others when competing for

chopsticks. When multiple philosophers are waiting for the same chopstick, the fair lock prioritizes the one who started waiting first, preventing any philosopher from being consistently unlucky in the competition for resources.

```
for (int i = 0; i < numberOfPhilosophers; i++) {
    // chopsticks[i] = new ReentrantLock(); // without fairness, starvation possible
    chopsticks[i] = new ReentrantLock(fair:true); // adding fairness
}
```

Figure 11. ReentrantLock with fairness policy enabled

How to Test for Starvation-Free Behaviour?

We can test for lack of starvation by tracking how many times each philosopher eats and reporting statistics every 10 meals. In a starvation-free system, all philosophers should have roughly similar eating counts over time with no philosopher having significantly fewer meals than others. We introduced another variable called eatCount to keep track of the number of times each philosopher eats. When we stop the program manually, we can check if every philosopher has eaten a similar amount of times. Figure 12 illustrates output from a test run.

```
Philosopher 1 is eating.
Philosopher 1 has eaten 13 times.
Philosopher 3 put down chopsticks.
Philosopher 4 picked up left chopstick.
Philosopher 3 is thinking.
Philosopher 4 picked up right chopstick.
Philosopher 4 is eating.
Philosopher 4 has eaten 12 times.
Philosopher 2 is thinking.
Philosopher 3 picked up left chopstick.
Philosopher 1 put down chopsticks.
Philosopher 1 is thinking.
Philosopher 0 picked up left chopstick.
Philosopher 3 picked up right chopstick.
Philosopher 3 is eating.
Philosopher 4 put down chopsticks.
Philosopher 3 has eaten 10 times.
Philosopher 0 picked up right chopstick.
Philosopher 4 is thinking.
Philosopher 0 is eating.
Philosopher 2 picked up left chopstick.
Philosopher 0 has eaten 12 times.
Philosopher 3 put down chopsticks.
Philosopher 3 is thinking.
Philosopher 4 picked up left chopstick.
Philosopher 2 picked up right chopstick.
Philosopher 2 is eating.
Philosopher 2 has eaten 10 times.
```

Figure 12. Test output for starvation-free check

4.1 Amdahl's Law pt. 1

Suppose the sequential part of a program accounts for 40% of the program's execution time on a single processor. Find an expression for the overall speedup that can be achieved by running the program on an *n*-processor machine. (The expression should be a formula for speedup as a function of *n*.) Show all intermediate steps. What is the limit of the speed-up if there is no bound to how many processors your system can have?

Solution:

Given that, $S = 40\% = 0.4$, we know $P = 1 - S = 1 - 0.4 = 0.6$. By applying Amdahl's law, we have

$$\text{Speedup} = \frac{1}{S + \frac{P}{n}} = \frac{1}{0.4 + \frac{0.6}{n}}$$

Let's simplify it:

$$\text{Speedup} = \frac{1}{\frac{0.4n+0.6}{n}}$$

$$\text{Speedup} = \frac{n}{0.4n + 0.6}$$

$$\text{Speedup} = \frac{n}{0.2(2n + 3)}$$

$$\text{Speedup} = \frac{5n}{2n + 3}$$

Finally, we have

$$\text{Speedup}(n) = \frac{5n}{2n + 3}$$

If there is no bound to the number of processors we can have,

$$\lim_{n\to\infty} \frac{5n}{2n + 3} = \lim_{n\to\infty} \frac{5}{2 + \frac{3}{n}} = \frac{5}{2} = 2.5$$

4.2 Amdahl's Law pt. 2

Now suppose the sequential part accounts for 30% of the program's computation time. Let *s_n* be the program's speedup on *n* processors, assuming the rest of the program is perfectly parallelizable. Your supervisor tells you to double this speedup: the revised program should have speedup *s'_n* >= 2*s_n*. You advertise for a programmer to replace the sequential part with an improved version that decreases the sequential time percentage by a factor of *k.* Find an expression for the value of *k* you should advertise for. Show all intermediate steps.

Solution:

Given that, $s'_n = 2s_n$ and $\frac{0.3}{k}$.

Since the original speedup is $s_n = \frac{1}{0.3 + \frac{0.7}{n}}$ and we want to double it $s'_n = 2s_n = \frac{2}{0.3 + \frac{0.7}{n}}$.

With the new sequential time percentage $s = \frac{0.3}{k}$, update parallel time percentage accordingly $p = 1 - \frac{0.3}{k}$.

Rewrite the speedup $s'_n = \frac{1}{\frac{0.3}{k} + \frac{1 - \frac{0.3}{k}}{n}}$.

Want to solve k for this equation:

$$\frac{2}{0.3 + \frac{0.7}{n}} = \frac{1}{\frac{0.3}{k} + \frac{1 - \frac{0.3}{k}}{n}}$$

$$0.3 + \frac{0.7}{n} = 2\left[\frac{0.3}{k} + \frac{1 - \frac{0.3}{k}}{n}\right]$$

$$0.3 + \frac{0.7}{n} = \frac{0.6}{k} + \frac{2\left(1 - \frac{0.3}{k}\right)}{n}$$

$$0.3 + \frac{0.7}{n} = \frac{0.6}{k} + \frac{2 - \frac{0.6}{k}}{n}$$

$$k\left(0.3n + 0.7\right) = 0.6n + k\left(2 - \frac{0.6}{k}\right)$$

$$0.3kn + 0.7k = 0.6n + 2k - 0.6$$

$$0.3kn + 0.7k - 2k = 0.6n - 0.6$$

$$k(0.3n - 1.3) = 0.6(n - 1)$$

$$k = \frac{0.6(n-1)}{0.3n - 1.3}$$

$$k = \frac{0.6(n-1)}{0.1(3n - 13)}$$

$$k = \frac{6(n-1)}{3n - 13}$$

The expression for k is:

$$k = \frac{6(n-1)}{3n - 13} \text{ for } n > \frac{13}{3}$$

For n ≤ 4.33 processors, it's impossible to double the speedup, even with perfect parallelization.

4.3 Amdahl's Law pt. 3

Suppose the sequential time percentage could be decreased 3-fold, and when we do so, the modified program takes half the time of the original on *n* processors. What fraction of the overall execution time did the sequential part account for? Express your answer as a function of *n*. Show all intermediate steps.

Given information that the sequential time percentage is decreased 3-fold, then we have $S_{new} = \frac{S}{3}$, then the parallel time percentage is $P_{new} = \frac{2S}{3}$.

Let T be the original execution time, and T' be the new execution time. We want to find S as a function of n.

Original Speedup: $\frac{1}{S + \frac{1-S}{n}}$

New Speedup: $\frac{1}{\frac{S}{3} + \frac{1-\frac{S}{3}}{n}}$

Since execution time is inversely proportional to speedup, then:

$$\frac{T_{modified}}{T_{original}} = \frac{Speedup_{original}}{Speedup_{modified}}$$

$$\frac{Speedup_{original}}{Speedup_{modified}} = \frac{1}{2}$$

$$\frac{1}{\frac{S}{3} + \frac{1-\frac{S}{3}}{n}} = 2 \cdot \frac{1}{S + \frac{1-S}{n}}$$

$$\frac{1}{\frac{S}{3} + \frac{1-\frac{S}{3}}{n}} = \frac{2}{S + \frac{1-S}{n}}$$

Cross multiply:

$$S + \frac{1-S}{n} = 2\left(\frac{S}{3} + \frac{1-\frac{S}{3}}{n}\right)$$

$$S + \frac{1-S}{n} = \frac{2S}{3} + \frac{2-\frac{2S}{3}}{n}$$

Multiply both sides by 3n:

$$3n \cdot S + 3n \cdot \frac{1-S}{n} = 3n \cdot \frac{2S}{3} + 3n \cdot \frac{2 - \frac{2S}{3}}{n}$$

$$3nS + 3(1-S) = 2nS + 3\left(2 - \frac{2S}{3}\right)$$

$$3nS + 3 - 3S = 2nS + 6 - 2S$$

$$3nS - 2nS - 3S + 2S = 6 - 3$$

$$S(n-1) = 3$$

$$S = \frac{3}{n-1}, \text{where } n > 1$$

Appendix

Deadlock.java – see submitted source files

ResourceOrdering.java – see submitted source files

DiningPhilosophers.java – see submitted source files

DiningPhilosophersDeadlock.java – see submitted source files

DiningPhilosophersDeadlockTest.java – see submitted source files

DiningPhilosophersTest.java – see submitted source files

MatrixMultiplicationTask.java – see submitted source files

MatrixMultiplication.java – see submitted source files

MatrixMultiplicationTest – see submitted source files

Output of runBenchmarkByThreads

```
========== Running Benchmark by Threads Suite ==========
Sequential multiply took 257375.127 ms
```

```
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 1] Parallel multiply took 257919.245 ms
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 2] Parallel multiply took 139749.696 ms
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 3] Parallel multiply took 102640.388 ms
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 4] Parallel multiply took 83774.910 ms
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 5] Parallel multiply took 76647.784 ms
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 6] Parallel multiply took 72605.688 ms
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 7] Parallel multiply took 68229.204 ms
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 8] Parallel multiply took 66760.609 ms
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 9] Parallel multiply took 66901.969 ms
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 10] Parallel multiply took 67420.776 ms
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 11] Parallel multiply took 67487.678 ms
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 12] Parallel multiply took 68309.284 ms
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 13] Parallel multiply took 69196.340 ms
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 14] Parallel multiply took 69692.398 ms
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 15] Parallel multiply took 68381.723 ms
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 16] Parallel multiply took 68922.157 ms
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 17] Parallel multiply took 68494.445 ms
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 18] Parallel multiply took 69231.399 ms
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 19] Parallel multiply took 69076.667 ms
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 20] Parallel multiply took 69538.603 ms
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 21] Parallel multiply took 68941.253 ms
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 22] Parallel multiply took 69723.365 ms
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 23] Parallel multiply took 69629.931 ms
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 24] Parallel multiply took 71734.266 ms
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 25] Parallel multiply took 72735.991 ms
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 26] Parallel multiply took 73510.612 ms
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 27] Parallel multiply took 73395.429 ms
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 28] Parallel multiply took 74650.781 ms
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 29] Parallel multiply took 73337.002 ms
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 30] Parallel multiply took 74709.238 ms
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 31] Parallel multiply took 74820.939 ms
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 32] Parallel multiply took 76063.663 ms
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 33] Parallel multiply took 75585.870 ms
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 34] Parallel multiply took 76339.715 ms
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 35] Parallel multiply took 75893.245 ms
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 36] Parallel multiply took 77348.073 ms
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 37] Parallel multiply took 76781.887 ms
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 38] Parallel multiply took 78049.412 ms
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 39] Parallel multiply took 77061.408 ms
[MATRIX SIZE = 4000] [NUMBER OF THREADS = 40] Parallel multiply took 79336.152 ms
```

Output of runBenchmarkByMatrixSizes (PASSED logs for evaluating if sequential output is equivalent to parallel output)

```
========== Running Benchmark by Matrix Size Suite ==========
Benchmark 0: Matrix size = 100
Sequential multiply took 7.196 ms
Parallel multiply took 20.504 ms
Benchmark 1: Matrix size = 200
Sequential multiply took 27.311 ms
Parallel multiply took 30.963 ms
Benchmark 2: Matrix size = 500
Sequential multiply took 131.783 ms
```

```
Parallel multiply took 29.303 ms
Benchmark 3: Matrix size = 1000
Sequential multiply took 2984.684 ms
Parallel multiply took 588.456 ms
Benchmark 4: Matrix size = 2000
Sequential multiply took 24942.360 ms
Parallel multiply took 6567.175 ms
Benchmark 5: Matrix size = 3000
Sequential multiply took 104090.371 ms
Parallel multiply took 24124.961 ms
Benchmark 6: Matrix size = 4000
Sequential multiply took 263973.784 ms
Parallel multiply took 64931.440 ms
```