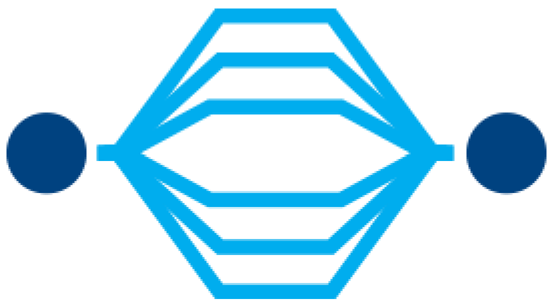
KINGDOM OF SAUDI ARABIA
Ministry of Education
Taibah University
College of Computer Science and Engineering
(Girls Section)

المملكة العربية السعودية
وزارة التعليم
جامعة طيبة
كلية علوم و هندسة الحاسبات
(قسم الطالبات)

CS424 – Introduction to Parallel Computing

Final Lab Project



by

Amal Saad Aljabri 3550343

Supervised by

Ms. Ebtesam Aljohani

Semester II 2019/2020

Table of Contents:

1 The first program

1.1 Description of the problem and the sequential solution

I would like to iterate loop at 1,000 times and in each time adds one to a variable SUM that initially is 0. The following figure 1.1 shows the flowchart of the serial solution to add one to a variable SUM. The following figure 1.2 shows the code of the serial solution. This serial solution is slow and inefficient. I plan to solve the problem and improve the serial solution by parallelism and that is through launches 1,000 threads and each thread adds 1 to a variable SUM. This will be implemented through the use of the java multithreading approach.

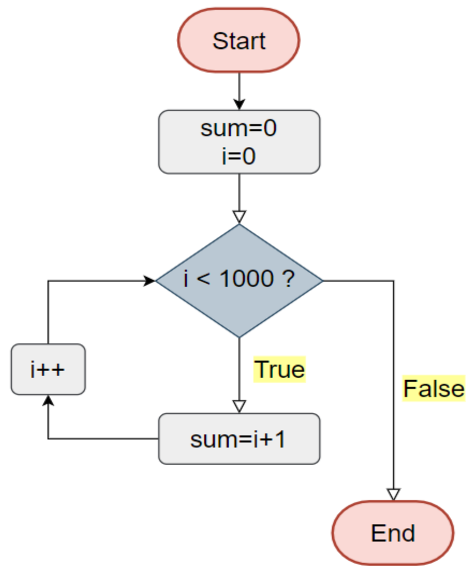


Figure 1.1 The flowchart of the serial solution

```java
public static void Serial_AddOne() {
    int SUM = 0;
    for (int i = 0; i < 1000; i++) {
        SUM = i + 1;
    }
    System.out.println(SUM);
}
```

Figure 1.2 The code of the serial solution

1.2 Parallel algorithm design

I will talk about parallel design to launch 1,000 threads and each thread adds 1 to a variable SUM. The program creates 1,000 threads executed in a thread pool executor. I will use data parallelization for distributing the data across different parallel threads, data parallelism is achieved when each thread performs the same task on this different pieces of distributed data, this task is an instance of the Runnable interface, also called a runnable object. The SUM is initially 0. When all the threads are finished, the SUM should be 1,000 but the output is unpredictable. This because run the program without synchronization. This is a common problem, known as a race condition, in multithreaded programs. To avoid race conditions, it is necessary to prevent more than one thread from simultaneously entering a certain part of the program, known as the critical region. I used three different techniques to achieve synchronization. The following figure 1.3 shows these different techniques to achieve synchronization.

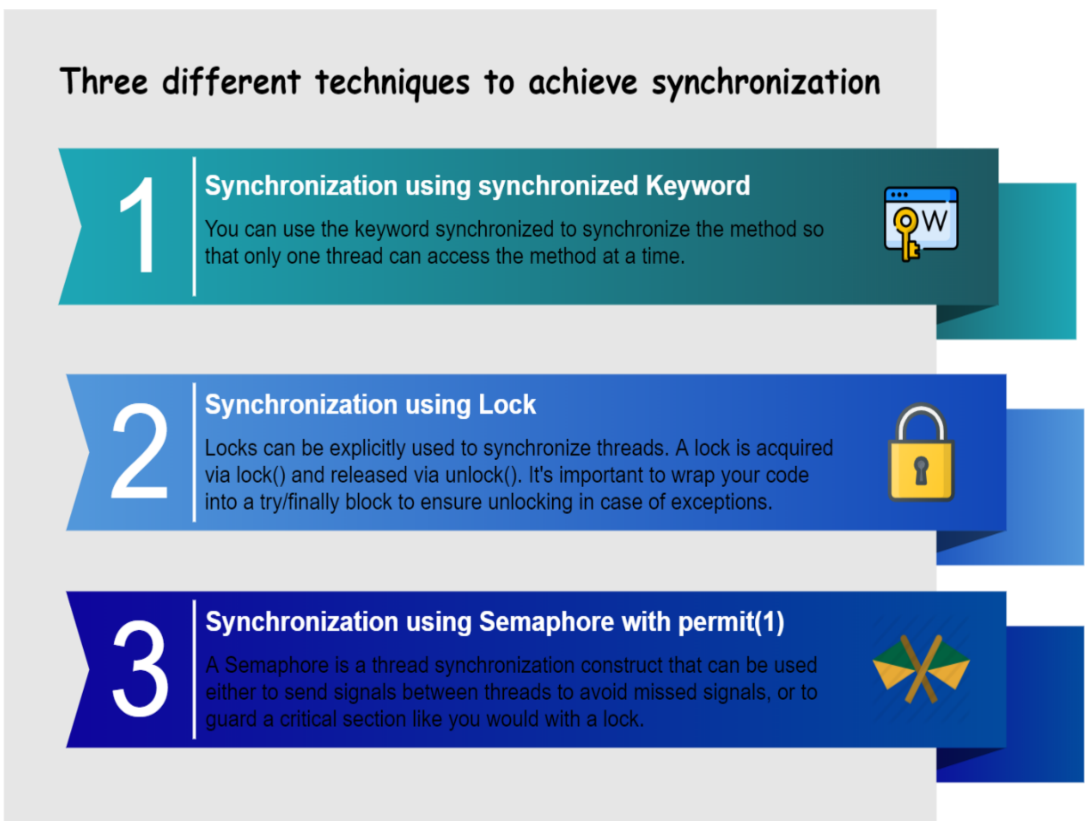


Figure 1.3 Three different techniques to achieve synchronization

The following figure 1.4 shows the class diagram.

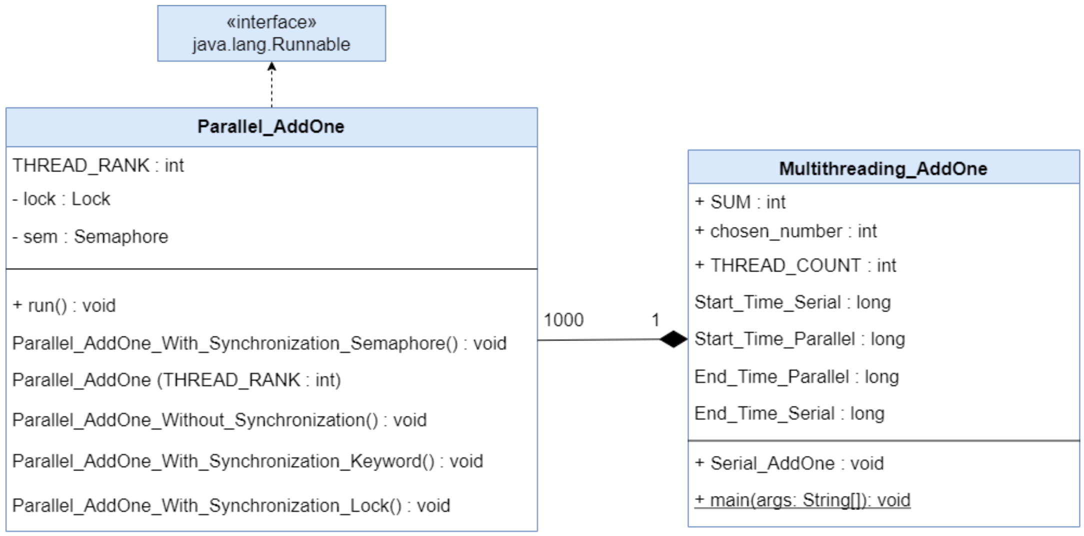


Figure 1.4 The class diagram

1.3 Parallel code

For the full source code, please visit GitHub

In this code, a menu will appear for the user at the beginning when running the program, through which he chooses how to implement the Parallel_AddOne without synchronization or with synchronization in three different techniques. The menu is as follows:

- `When click 1: Parallel_AddOne with synchronization using synchronized keyword.`
- `When click 2: Parallel_AddOne with synchronization using lock.`
- `When click 3: Parallel_AddOne with synchronization using semaphore--> permit(1).`
- `When Click any number: Parallel_AddOne without synchronization.`

The following figures 1.5 and 1.6 shows code of the program.

```java
import java.util.Scanner;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Semaphore;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Multithreading_AddOne {

    public static int THREAD_COUNT = 1000;
    public static int SUM = 0;
    public static int chosen_number;

    public static void Serial_AddOne() {
        int SUM = 0;
        for (int i = 0; i < 1000; i++) {
            SUM = i + 1;
        }
        System.out.println(SUM);
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("---------------------------------------------------------------------------------------");
        System.out.println("                     Please, choose the appropriate option!");
        System.out.println("---------------------------------------------------------------------------------------");
        System.out.println("                            The results of Parallel: ");
        System.out.println("---------------------------------------------------------------------------------------");
        System.out.println("     Click 1: Parallel AddOne With Synchronization Using synchronized Keyword ");
        System.out.println("---------------------------------------------------------------------------------------");
        System.out.println("            Click 2: Parallel AddOne With Synchronization Using Lock ");
        System.out.println("---------------------------------------------------------------------------------------");
        System.out.println("     Click 3: Parallel AddOne With Synchronization Using Semaphore --> permit(1) ");
        System.out.println("---------------------------------------------------------------------------------------");
        System.out.println("            Click ANY NUMBER: Parallel AddOne Without Synchronization ");
        System.out.println("---------------------------------------------------------------------------------------");

        chosen_number = sc.nextInt();

        System.out.println("---------------------------------------------------------------------------------------");
        System.out.print("                           Serial AddOne, SUM = ");
        long Start_Time_Serial = System.nanoTime();
        Serial_AddOne();
        long End_Time_Serial = System.nanoTime();
        System.out.println("---------------------------------------------------------------------------------------");
        System.out.println("                   Run-Times of Serial Code is " + (End_Time_Serial - Start_Time_Serial)
                + " nanoseconds");
        System.out.println("---------------------------------------------------------------------------------------");
        ExecutorService executor = Executors.newCachedThreadPool();

        long Start_Time_Parallel = System.nanoTime();

        for (int thread = 0; thread < THREAD_COUNT; thread++) {
            executor.execute(new Parallel_AddOne(thread));
        }

        executor.shutdown();

        while (!executor.isTerminated()) {
        }

        long End_Time_Parallel = System.nanoTime();

        System.out.println("                            Parallel AddOne, SUM = " + SUM);
        System.out.println("---------------------------------------------------------------------------------------");
        System.out.println("                             Number of threads is " + THREAD_COUNT);
        System.out.println("---------------------------------------------------------------------------------------");
        System.out.println("                 Run-Times of Parallel Code is " + (End_Time_Parallel - Start_Time_Parallel)
                + " nanoseconds");
        System.out.println("---------------------------------------------------------------------------------------");
    }
```

Figure 1.5 Parallel_AddOne program code

```java
    static class Parallel_AddOne implements Runnable {
        private static Lock lock = new ReentrantLock();
        private static Semaphore sem = new Semaphore(1);
        int THREAD_RANK;

        Parallel_AddOne(int THREAD_RANK) {

            this.THREAD_RANK = THREAD_RANK;
        }

        @Override
        public void run() {

            if (chosen_number == 1) {
                Parallel_AddOne_With_Synchronization_Keyword();
            } else if (chosen_number == 2) {
                Parallel_AddOne_With_Synchronization_Lock();
            } else if (chosen_number == 3) {
                Parallel_AddOne_With_Synchronization_Semaphore();
            } else {
                Parallel_AddOne_Without_Synchronization();
            }
        }

        void Parallel_AddOne_Without_Synchronization() {
            SUM = SUM + 1;
        }

        static synchronized void Parallel_AddOne_With_Synchronization_Keyword() {
            SUM = SUM + 1;
        }

        void Parallel_AddOne_With_Synchronization_Lock() {
            lock.lock();
            try {
                SUM = SUM + 1;
            } finally {
                lock.unlock();
            }
        }

        void Parallel_AddOne_With_Synchronization_Semaphore() {
            try {
                sem.acquire();
                SUM = SUM + 1;
            } catch (InterruptedException exc) {
                System.out.println(exc);
            }

            sem.release();
        }

    }
}
```

Figure 1.6 Parallel_AddOne program code

1.4 Sample output

The following tables 1.1, 1.2, 1.3 and 1.4 shows the output of parallel code.

Table 1.1 The output of parallel_AddOne without synchronization

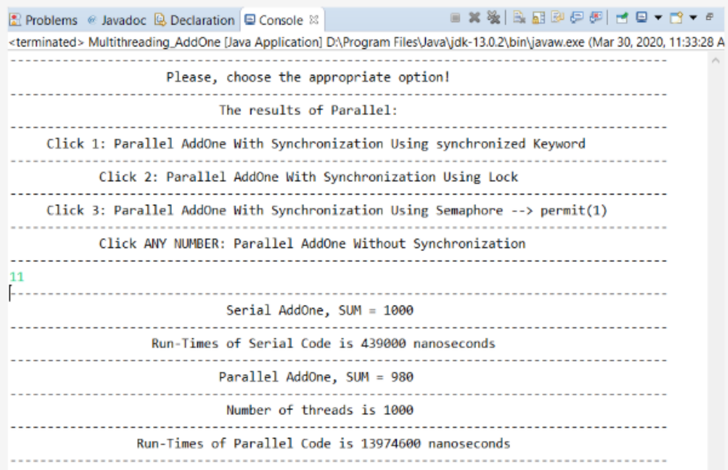
How to implement	Capture the output screen
Parallel_AddOne without synchronization	

Table 1.2 The output of parallel_AddOne with synchronization using synchronized keyword

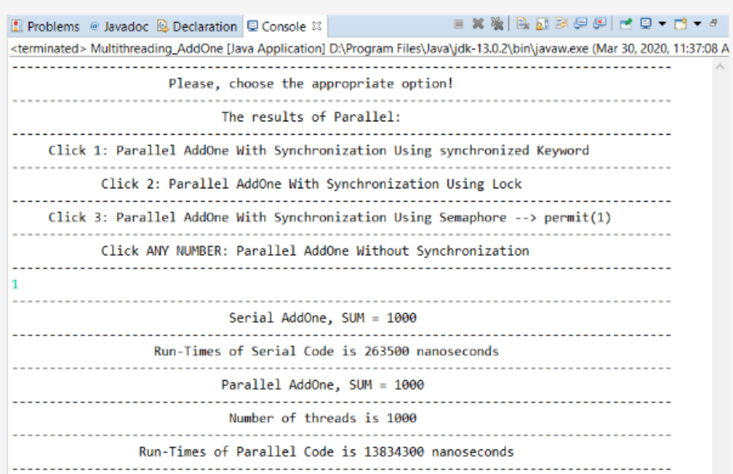
How to implement	Capture the output screen
Parallel_AddOne with synchronization using synchronized keyword	

Table 1.3 The output of parallel_AddOne with synchronization using lock

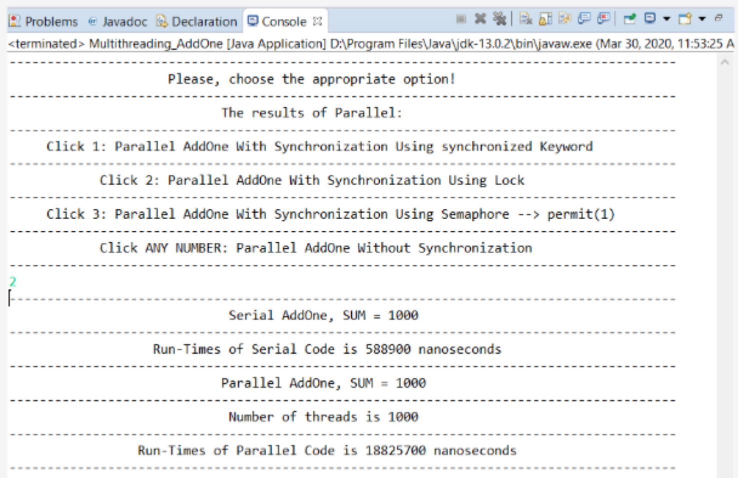
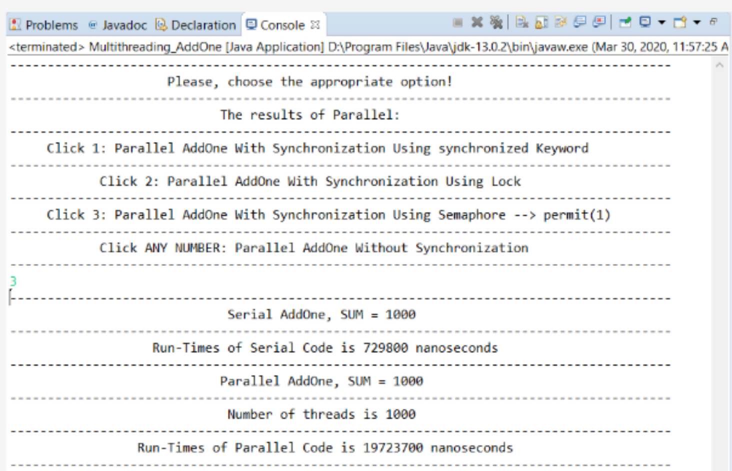
How to implement	Capture the output screen
Parallel_AddOne with synchronization using lock	

Table 1.4 The output of parallel_AddOne with synchronization using semaphore

How to implement	Capture the output screen
Parallel_AddOne with synchronization using semaphore	

1.5 Performance evaluation

I will evaluate performance when THREAD_COUNT = 1,000. through the following steps:

1. Runtime's account of serial and parallel code.

   RunTime of paralle code = End_Time_Parallel − Start_Time_Parallel

   RunTime of Serial code = End_Time_Serial − Start_Time_Serial

2. Measure the relation between the serial and the parallel run-times is the speedup.

   $$\mathbf{S(n,p)} = \frac{T_{serial(n)}}{T_{parallel(n,p)}}$$

3. Measure of parallel performance is parallel efficiency.

   $$\mathbf{E(n,p)} = \frac{S(n,p)}{p}$$

The following table 1.5 shows the results of run-times of serial and parallel code (times are in nanoseconds).

Table 1.5 The run-times of serial and parallel code (times are in nanoseconds)

	Serial	Parallel			
		Without Synchronization	With Synchronization		
			synchronized Keyword	Lock	Semaphore
Run Times	729800	13974600	13834300	18825700	19723700

The following table 1.6 shows the speedups of parallel code.

Table 1.6 The speedups of parallel code

	Parallel			
	Without Synchronization	With Synchronization		
		synchronized Keyword	Lock	Semaphore
speedup	0.052	0.05275	0.038766	0.037

The following table 1.7 shows the efficiencies of parallel code.

Table 1.7 The efficiencies of parallel code

	Parallel			
	Without Synchronization	With Synchronization		
		synchronized Keyword	Lock	Semaphore
efficiency	0.000052	0.00005275	0.00003877	0.000037

2 The second program

2.1 Description of the problem and the sequential solution

I would like to find the factorial of any number using loop. Factorials are very simple things. They are just products, indicated by an exclamation mark. For instance, "four factorial" is written as "4!" and means 1×2×3×4 = 24. In general, "n!" and means the product of all the whole numbers from 1 to n; that is, n! = 1×2×3×...×n. The following figure 2.1 shows the flowchart of the serial solution to find the factorial of any number using loop. The following figure 2.2 shows the code of the serial solution. I plan to improve the performance of this serial solution by parallelism. This will be implemented through the use of the java multithreading approach.

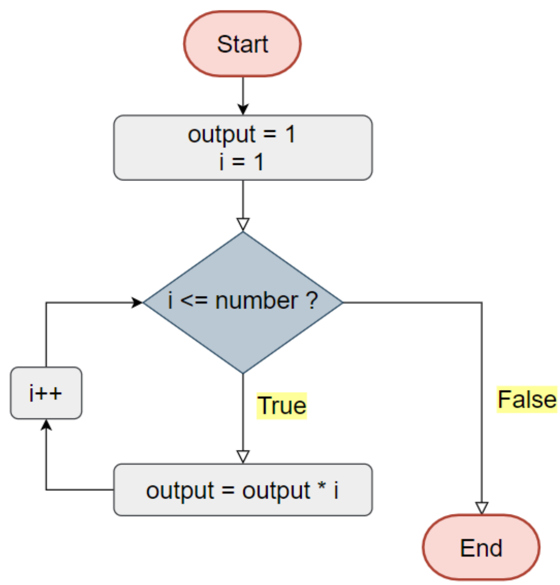


Figure 2.1 The flowchart of the serial solution

```
public static void Serial_Factorial(int num) {
    long output = 1;
    int i;
    for (i = 1; i <= num; i++) {
        output = output * i;
    }
    System.out.println(output);
}
```

Figure 2.2 The code of the serial solution

2.2 Parallel algorithm design

I will talk about a parallel design to find the factorial of any number. The program creates threads executed in a thread pool executor. I will use data parallelization for distributing the data across different parallel threads, data parallelism is achieved when each thread performs the same task on this different pieces of distributed data, this task is an instance of the Runnable interface, also called a runnable object. I will parallelize to the loop and split iterations between THREAD_COUNT. Each THREAD_RANK will have iterations to compute multiplication on. And stores the results for these iterations for this THREAD_RANK into my_output variable. When all THREAD_RANK iterations are finished, will stores the results for all THREAD_RANK into output variable and compute multiplication on. The following figure 2.3 shows the class diagram.

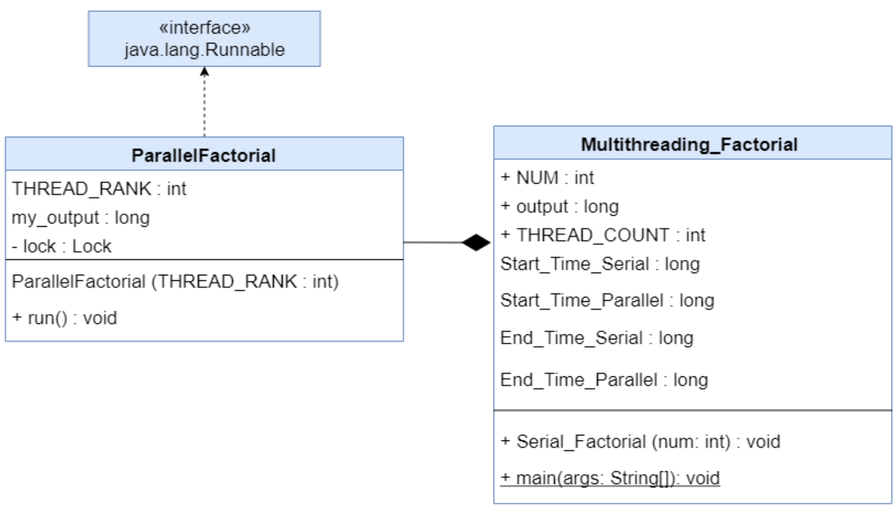


Figure 2.3 The class diagram

2.3 Parallel code

For the full source code, please visit GitHub

The following figures 2.4 and 2.5 shows code of the program.

```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Multithreading_Factorial {

    public static int NUM = 18;
    public static long output = 1;
    public static int THREAD_COUNT = 4;

    public static void Serial_Factorial(int num) {
        long output = 1;
        int i;
        for (i = 1; i <= num; i++) {
            output = output * i;
        }
        System.out.println(output);
    }

    public static void main(String[] args) {
        System.out.println("---------------------------------------------------------");
        System.out.print("         Serial Factorial of " + NUM + " is: ");
        long Start_Time_Serial = System.nanoTime();
        Serial_Factorial(NUM);
        long End_Time_Serial = System.nanoTime();
        System.out.println("---------------------------------------------------------");
        System.out.println("   Run-Times of Serial Code is " + (End_Time_Serial - Start_Time_Serial) + " nanoseconds");

        ExecutorService executor = Executors.newCachedThreadPool();

        long Start_Time_Parallel = System.nanoTime();

        for (int thread = 0; thread < THREAD_COUNT; thread++) {
            executor.execute(new Parallel_Factorial(thread));
        }

        executor.shutdown();

        while (!executor.isTerminated()) {
        }

        long End_Time_Parallel = System.nanoTime();

        System.out.println("---------------------------------------------------------");
        System.out.println("         Parallel Factorial of " + NUM + " is: " + output);
        System.out.println("---------------------------------------------------------");
        System.out.println("                  Number of threads is " + THREAD_COUNT);
        System.out.println("---------------------------------------------------------");
        System.out.println(
                "    Run-Times of Parallel Code is " + (End_Time_Parallel - Start_Time_Parallel) + " nanoseconds");
        System.out.println("---------------------------------------------------------");
    }
```

Figure 2.4 Parallel_Factorial program code

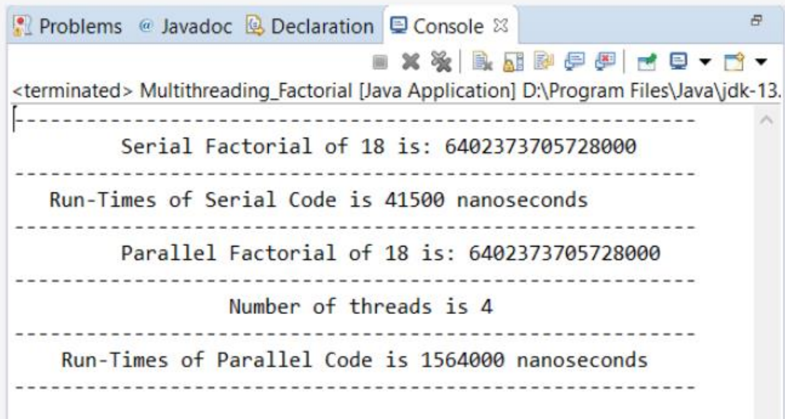
```
    static class Parallel_Factorial implements Runnable {
        private static Lock lock = new ReentrantLock();

        int THREAD_RANK;
        long my_output = 1;

        Parallel_Factorial(int THREAD_RANK) {

            this.THREAD_RANK = THREAD_RANK;
        }


        @Override
        public void run() {

            int local_n = (NUM + THREAD_COUNT - 1) / THREAD_COUNT;
            int first = local_n * THREAD_RANK;
            int last = Math.min(first + local_n, NUM);

            // System.out.println("THREAD_RANK "+THREAD_RANK);
            for (int i = first; i < last; i++) {
                // System.out.println((i+1)+"*"+my_output);
                my_output = my_output * (i + 1);
                // System.out.println(my_output);

            }
            lock.lock();
            // System.out.println(output+"*"+my_output);
            output = output * my_output;
            lock.unlock();
        }
    }
}
```

Figure 2.5 Parallel_Factorial program code

2.4 Sample output

The following table 2.1shows the output of parallel code.

Table 2.1 The output of Parallel_Factorial

The program	Capture the output screen
Compute a factorial of a given number in parallel	

2.5 Performance evaluation

I will evaluate performance with different number of THREAD_COUNT and NUM through the following steps:

1. Runtime's account of serial and parallel code.

   RunTime of paralle code = **End_Time_Parallel − Start_Time_Parallel**

   RunTime of Serial code = **End_Time_Serial − Start_Time_Serial**

2. Measure the relation between the serial and the parallel run-times is the speedup.

   $$\mathbf{S(n,p)} = \frac{\mathbf{T_{serial(n)}}}{\mathbf{T_{parallel(n,p)}}}$$

3. Measure of parallel performance is parallel efficiency.

   $$\mathbf{E(n,p)} = \frac{\mathbf{S(n,p)}}{\mathbf{p}}$$

The following table 2.2 shows the results of run-times of serial and parallel code (times are in nanoseconds).

Table 2.2 The run-times of serial and parallel code (times are in nanoseconds)

	The value of NUM			
THREAD_COUNT	5	10	15	20
1	3634900	3898500	3828600	4044800
2	1153700	1303600	1190200	969800
4	1521900	1511900	1567500	1513400
8	1778800	1688700	1824700	1601200
16	2029600	1858400	1974700	2022700
32	2523600	3405300	2760300	2383800

The following table 2.3 shows the speedups of parallel code.

Table 2.3 The speedups of parallel code

	The value of NUM			
THREAD_COUNT	5	10	15	20
1	1	1	1	1
2	3.1506	2.99	3.21677	4.171
4	2.388	2.6785	2.442488	2.67265
8	2.043	2.3085	2.098	2.526
16	1.7909	2.0977	1.9388	1.9997
32	1.440	1.14483	1.387	1.6968

The following table 2.4 shows the efficiencies of parallel code.

Table 2.4 The efficiencies of parallel code

	The value of NUM			
THREAD_COUNT	5	10	15	20
1	1	1	1	1
2	1.5753	1.495	1.608385	2.0855
4	0.597	0.669625	0.610622	0.6681625
8	0.255375	0.2885625	0.26225	0.31575
16	0.1119	0.1311	0.121175	0.12498
32	0.045	0.03576	0.0433	0.0530

References

- Pacheco, P., "An Introduction to Parallel Programming", Morgan Kaufman. 2011.
- GeeksforGeeks. 2020. Factorial Of An Array Of Integers - Geeksforgeeks. [online] Available at: <https://www.geeksforgeeks.org/factorial-of-an-array-of-integers/> [Accessed 30 March 2020].
- Liang, Y., “Introduction to java programming”, Prentice Hall, 2014.