# MULTI-THREADING TO SOLVE A BIG PROBLEM

## 1. INTRODUCTION

In this report, we will observe the code used to divide a large task among multiple threads and discuss the benefits of multi-threading.

## 2. JAVA CODE FOR MULTI-THREADING

This code defines the number of threads to be used as 4 and the range to be calculated by each thread as 250,000,000. The *compute()* method is used to calculate the sum of the integers in each thread and add the result to the *sum* in the *synchronized* block. By using a *synchronized* block, it is guaranteed that even if multiple threads attempt to access *sum* at the same time, another thread will not access it until processing is complete. The *join()* method is used to wait for each thread to finish. The final calculation result is stored in the *sum* and output by the parent thread (main thread).

Program.cs MultiThreadingSum.java

MultiThreadingSum.java

```java
public class MultiThreadingSum {
    private static final int THREAD_COUNT = 4;
    private static final int TASK_SIZE = 250000000;
    private static final int N = THREAD_COUNT * TASK_SIZE;
    private static long sum = 0;

    public static void main(String[] args) throws InterruptedException {
        Thread[] threads = new Thread[THREAD_COUNT];
        for (int i = 0; i < THREAD_COUNT; i++) {
            final int start = i * TASK_SIZE + 1;
            final int end = (i + 1) * TASK_SIZE;
            threads[i] = new Thread(() -> {
                long threadSum = compute(start, end);
                synchronized (MultiThreadingSum.class) {
                    sum += threadSum;
                }
                System.out.println("Thread " + Thread.currentThread().getId() + ": Computed sum from " + start + " to " + end + " = " + threadSum);
            });
            threads[i].start();
        }

        for (Thread thread : threads) {
            thread.join();
        }

        System.out.println("Parent: Sum of numbers from 1 to " + N + " = " + sum);
    }

    private static long compute(int start, int end) {
        long threadSum = 0;
        for (int i = start; i <= end; i++) {
            threadSum += i;
        }
        return threadSum;
    }
}
```

*Picture1. Java code for multi-threading*

```
misatoseki@MisatonoMacBook-Air Session10 % java MultiThreadingSum
Thread 14: Computed sum from 1 to 250000000 = 31250000125000000
Thread 17: Computed sum from 750000001 to 1000000000 = 218750000125000000
Thread 16: Computed sum from 500000001 to 750000000 = 156250000125000000
Thread 15: Computed sum from 250000001 to 500000000 = 93750000125000000
Parent: Sum of numbers from 1 to 1000000000 = 500000000500000000
misatoseki@MisatonoMacBook-Air Session10 %
```

*Picture2. Executed result.*

## 3. C# CODE FOR MULTI-THREADING

This code defines the number of threads to be used as 4 and the range to be calculated by each thread as 250,000,000. Each thread uses the *Compute()* method to calculate the sum of the integers in its range of responsibility, and uses the *Interlocked* class to add the result of the calculation to the *sum*. The *Interlocked* class is like a *synchronized* block in Java. This is a class that are guaranteed to prevent another thread from accessing a variable before the operation is complete, even if it is accessed by multiple threads at the same time. Once the threads have completed their operations, the *join()* method is used to wait for each thread to exit. The final calculation result is stored in the *sum* and is output by the parent thread (main thread).

Program.cs × MultiThreadingSum.java

Program.cs

```
using System;
using System.Threading;

class Program
{
    private static int THREAD_COUNT = 4;
    private static int TASK_SIZE = 250000000;
    private static int N = THREAD_COUNT * TASK_SIZE;
    private static long sum = 0;

    static void Main(string[] args)
    {
        Thread[] threads = new Thread[THREAD_COUNT];
        for (int i = 0; i < THREAD_COUNT; i++)
        {
            int start = i * TASK_SIZE + 1;
            int end = (i + 1) * TASK_SIZE;
            threads[i] = new Thread(() =>
            {
                long threadSum = Compute(start, end);
                Interlocked.Add(ref sum, threadSum);
                Console.WriteLine("Thread " + Thread.CurrentThread.ManagedThreadId + ": Computed sum from " + start + " to " + end + " = " + threadSum);
            });
            threads[i].Start();
        }

        foreach (Thread thread in threads)
        {
            thread.Join();
        }

        Console.WriteLine("Parent: Sum of numbers from 1 to " + N + " = " + sum);
    }

    static long Compute(int start, int end)
    {
        long threadSum = 0;
        for (int i = start; i <= end; i++)
        {
            threadSum += i;
        }
        return threadSum;
    }
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
misatoseki@MisatonoMacBook-Air Session10 % dotnet run
Thread 5: Computed sum from 250000001 to 500000000 = 93750000125000000
Thread 6: Computed sum from 500000001 to 750000000 = 156250000125000000
Thread 7: Computed sum from 750000001 to 1000000000 = 218750000125000000
Thread 4: Computed sum from 1 to 250000000 = 31250000125000000
Parent: Sum of numbers from 1 to 1000000000 = 500000000500000000
misatoseki@MisatonoMacBook-Air Session10 %
```

*Picture3. C# code for multi-threading and executed result*

## 4. WHAT I LEARNED

In the above code, four threads were used to compute a huge number of totals. This process could take a very long time if done in a single thread, but by using multithreading, the process could be completed in a much shorter time.

The concept of child/parent processes that we learned in the previous study is similar to the multithreading. The difference is that multithreading is a technique for achieving parallel processing **within a program**, while child/parent processes are techniques for achieving parallel processing between **different programs**.