Numerical Pi Estimation

Course Level:
CS1

PDC Concepts Covered:

PDC Concept	Bloom Level
Concurrency	C
Sequential dependency	C
Data race	C
Synchronization	A

Programming Knowledge Prerequisites:
Basic programming knowledge in Java is required for this lab. More specifically, the skills below are needed to complete this lab.

- Variable declaration
- If-else
- For loop
- Functions/methods

Tools Required:
Java: Java Development Kit (JDK) 8 or later and Pyjama

Prolog and Review:

One of the important concepts in programming is the **if-else** statement. Using the if-else statement, a programmer the decide which actions to take in the program. Following is the if-else syntax.

```
if(boolean_expression) {
   /* statement(s) will execute if the boolean expression is true */
} else {
   /* statement(s) will execute if the boolean expression is false */
}
```

If the Boolean expression evaluates to true, then the **if block** is executed, otherwise **else block** of code is executed. Likewise, another important programming concept is looping. Using looping, a programmer can do a set of statements multiple times. A common looping structure in programming languages is the **for-loop**. Consider the following example.

```
for ( initialization; condition; increment ) {
    statement(s);
}
```

The **initialization** is executed once upon entering the loop. Next, the **condition** is evaluated. If it is true then the body of the loop is executed, otherwise loop exits, and the program continues with the next statement that occurs after the for-loop. After the body executes, control flow jumps to **increment** statement. Typically, the increment statement updates a loop control variable that is used in the condition to determine is the loop should continue. The condition gets evaluated again and the process repeats.

Problem description:

In this lab, you will be writing a program to approximate the value of the mathematical constant Pi, typically known by its mathematical symbol π. By definition, Pi is the ratio of the circumference of a circle to its diameter. This ratio remains the same regardless of the size of the circle. Moreover, Pi is an irrational number, which means it is a real number with a nonrepeating decimal expansion. In other words, it is not representable as an integer ratio and the decimal point goes forever [1].

One of the simplest methods to calculate Pi accurately to a great number of decimal places is the Gregory-Leibniz series. Each time a new term is added the result gets closer and closer to Pi. The series is represented as follow.

$$\pi = 4\left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \cdots\right] = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1}$$

With the help of a computer, a large number of terms can be used in estimating Pi. The actual value of Pi up to 20 digits is 3.14159265358979323846. The table below is generated from a serial implementation of the Gregory-Leibniz series. Notice that as more terms are added to the series, the approximation of Pi approaches the actual value of Pi, but the computation time increases noticeably.

Number of Terms	pi	Time (Milliseconds)
10	3.04183961892940	-
100	3.13159290355855	-
1,000	3.14059265383979	-
10,000	3.14149265359003	1.0
100,000	3.14158265358971	8.0
1,000,000	3.14159165358977	23.0
10,000,000	3.14159255358979	235.0
100,000,000	3.14159264358932	1,768.0
1,000,000,000	3.14159265258805	12,406.0

Fortunately, even personal computers consist of multi-core processors. Leveraging the parallelism using the multiple cores can reduce the computation time. Therefore, because more terms can be computed in the same amount of time as the serial version, parallelization can improve accuracy.

Methodology:

Using Gregory-Leibniz series, you will construct a data parallel solution. In other words, you will divide the input data between the processors, and each processor will compute its part or the answer. Fortunately, once the data is partitioned among the processors, the processors can compute their terms independently without the need to share any part of their answers until the very end of the computation. Such computations are known at *embarrassingly parallel* and are typically much easier to program than computations that need to share data.

The data that you will partition is the terms in the series. You could partition the terms such that each processor is given one term. However, such a partitioning would give poor accuracy, unless you have thousands of processors. Therefore, the data needs to be partitioned when the number of terms is more than the number of processors. You will group the terms, and then assign each group of the partitioned terms to a processor. This is also known as *data decomposition* or *data parallelization*. A simple example is presented in Figure 1, where a 12 term Gregory-Leibniz series is shown. The figure depicts three processors, including a processor designated as master processor. The computation occurs in two phases. First, the work is divided equally among the three processors by the master processor. Each processor sums four terms in the series. Second, the global sum, initialized as zero before the thread creation, gets updated serially by the local part (my_sum). Synchronization among the threads is required at this point to ensure that multiple threads are not updating the global sum simultaneously.

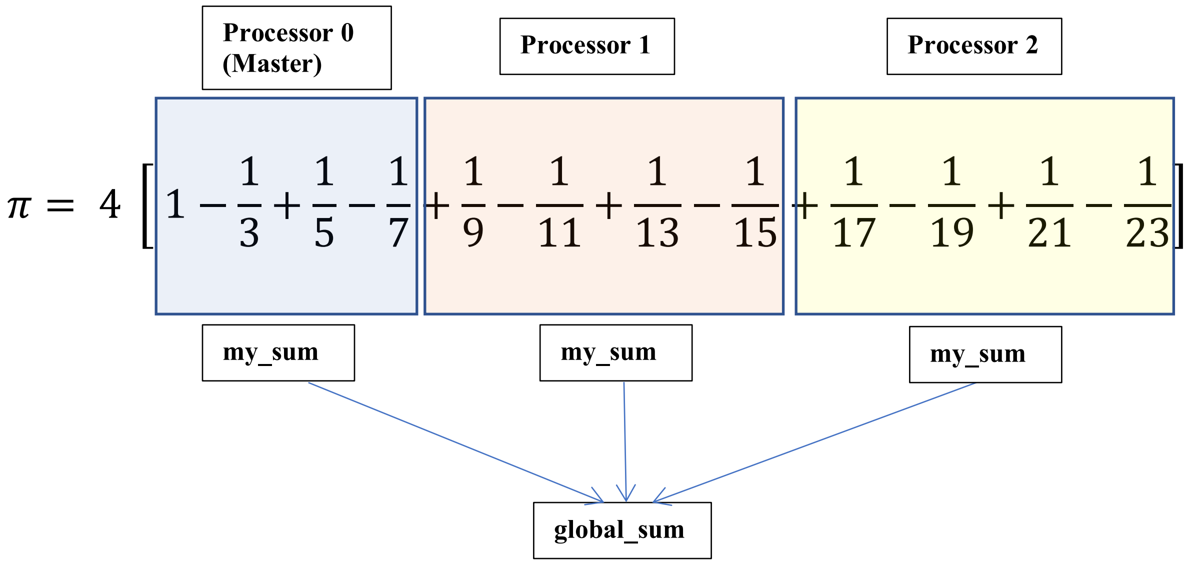

Processor 0 (Master)
Processor 1
Processor 2
$\pi = 4\left[1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\frac{1}{9}-\frac{1}{11}+\frac{1}{13}-\frac{1}{15}+\frac{1}{17}-\frac{1}{19}+\frac{1}{21}-\frac{1}{23}\right]$
my_sum
my_sum
my_sum
global_sum

Fig 1: Data Partitioning among the three processors

When the parallel execution is over, global sum is multiplied by four to get the approximation of Pi.

***Note to the instructor:** Instructor can illustrate **Sequential dependency** by pointing out that the global sum cannot be computed before the completion of computing the local sum by the individual processor. Instructor can illustrate **Data parallel** by pointing to that different processors are doing the same computation but on different pieces of data.

Implementation:

For this lab, you will be implementing the code in parallel (also in serial) to get the estimated value of Pi using the Gregory-Leibniz series. A pseudo code is given below that sums all the terms of the series serially.

```
Serial_Sum_Series(num_terms)
Inputs:
        num_terms  - Number of terms of the Gregory-Leibniz series
returns:
        sum – The sum of all the terms
begin
        set sum to 0
        set factor to 1.0
        loop index from 0 to num_terms
                set sum to sum + (factor/(2.0 * index + 1.0) )
                factor = -1 * factor
        return sum
```

In the Gregory-Leibniz series, every odd index term is negative; therefore, the ***factor*** variable gets multiplied by -1 to each odd index term. After calculating sum of the series, the value of Pi is generated by multiplying the sum with 4.

To parallelize the serial code, you need to decompose the data as described in the methodology section. The directive that you will use to parallelize a section is given below.

```
//#omp parallel num_threads(thread_count)
```

Note that data decomposition is needed to so that every thread can get equal number of terms. This can be done by dividing the number of terms by the number of threads. In the case when the number of terms in the series are not evenly distributed, then either first or last thread is assigned some additional terms. Consider the following code block.

```
int num_parts = total_terms/thread_count;
int my_first_i = num_parts * my_rank;
int my_last_i;
if (my_rank == thread_count-1) my_last_i = total_terms;
else my_last_i = my_first_i + num_parts;
```

At this point, all the threads know their range of terms (**num_parts**) to sum in the Gregory-Leibniz series. The variable **my_first_i** is the index of the first term and **my_last_i – 1** is the index of the last term assigned to a thread. The last thread gets few additional terms in case when the number terms (**total_terms**) are not evenly divided by the number of threads (**thread_count**).

In the Gregory-Leibniz series, terms with an odd index are negative. Therefore, the first term of each thread is checked whether it lies in the even or odd position. The odd position terms are multiplied by negative one. An example is shown as below.

```
double factor;
if (my_first_i%2 == 0.0) factor = 1.0;
else factor = -1.0;
```

Each thread now calculates a partial sum using their assigned range of terms by the below code block. Each thread maintains a local copy of **my_sum** variable containing its partial sum of the series. Consider the following example.

```
double my_sum = 0.0;
for(i = my_first_i ; i < my_last_i ; i++, factor = -factor){
    my_sum += factor/(2.0 * i + 1.0);
}
```

After calculating the partial sum, each thread will want to update the global sum with their local sum. If all the threads add their local sum simultaneously to the global sum then the resultant value will be erroneous. Therefore, a synchronization method needs to be used to ensure there is only one update of the global sum by a thread at a time. OpenMP critical/atomic directive can be used to do that.

```
//#omp critical
sum += my_sum;
```

***Note to the instructor:** Above code snippet shows how to avoid **Data race** when update global variable in OpenMP. Moreover, instructor can talk about **Synchronization** at this point.

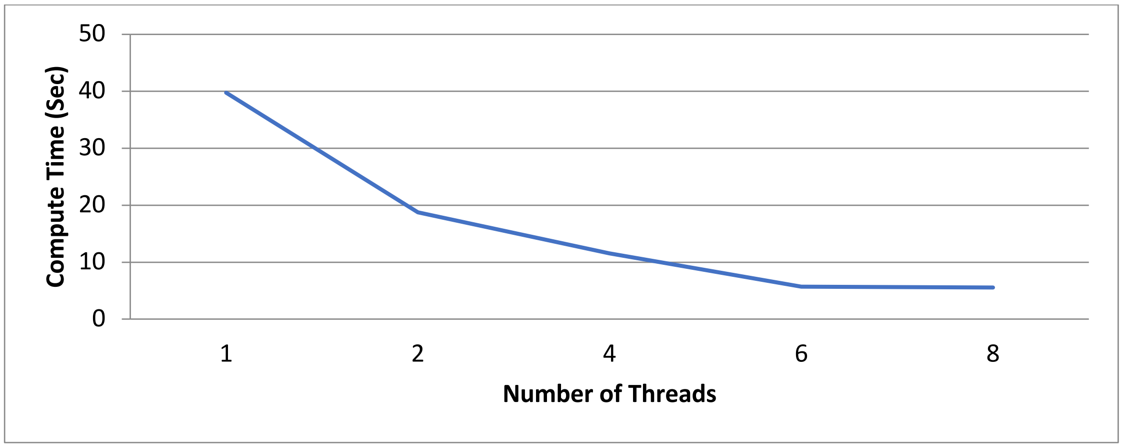
Now, run the serial version ten times and calculate the average time. Next, run the parallel version ten times each with 2 to 8 threads and record the average. Next, calculate the speedup based on the average times.

Results:

Below figure shows the computation time and value of Pi while using threads up to 8 in our implementation. The program is run on an Intel Core i7 laptop with 4 physical cores (and 8 virtual cores). It shows good speedup as we increase number of threads.

Speed up

Number of Threads	**Compute Time (in seconds)**	**Pi**
1	39.763	3.14159265308807
2	18.733	3.14159265308805
4	11.55	3.14159265308925
6	5.729	3.14159265308898
8	5.544	3.14159265308921



References:

1. https://www.livescience.com/29197-what-is-pi.html