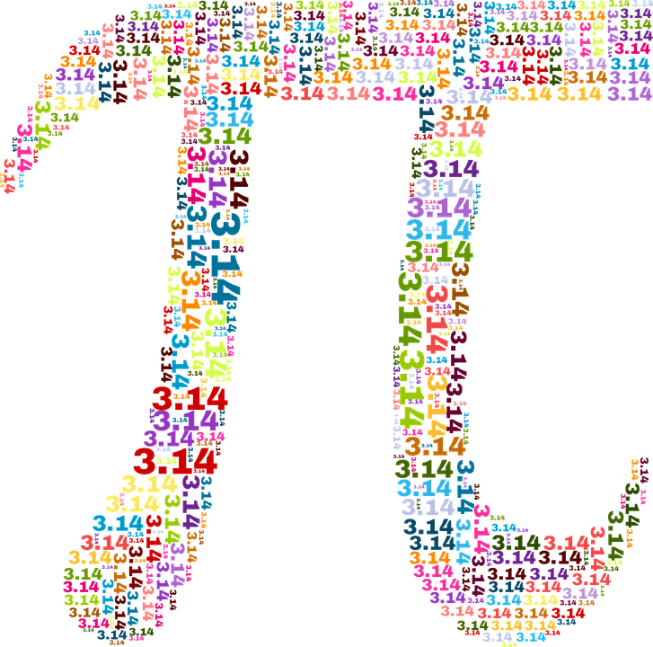


**CSC1300: LAB 9**

Numerical Pi Estimation



# Concepts

* Parallel Processing
* Branching
* Loops
* Functions

# tools required

You need a C++ compiler that is OpenMP capable. If you followed the “GETTING YOUR COMPUTER SET UP” instructions at the beginning of the semester, you should already have OpenMP installed.

Here is a link to getting OpenMP installed (Linux/Mac/Windows) if you have problems: <https://www.csc.tntech.edu/pdcincs/index.php/installation/>

# specifications

In this lab, you will be writing a program to approximate the value of the mathematical constant Pi, typically known by its mathematical symbol . 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 (<https://www.livescience.com/29197-what-is-pi.html>).

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 follows:

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.

## Data Parallel Solution

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 the figure below, 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 2**

**dfsff**

**()**

**Processor 1**

**dfsff**

**()**

**Processor 0 (Master)**

**dfsff**

**()**

**my\_sum**

**my\_sum**

**my\_sum**

**global\_sum**

**dfsff**

**()**

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

Note: The global sum cannot be computed before the completion of computing the local sum by the individual processor. This is called **sequential dependency**. Also, different processors are doing the same computation but on different pieces of data, which is called **data parallel**.

## Implementation:

Implement the code in parallel to get the estimated value of Pi using the Gregory-Leibniz series.

**You are given the serial version of this code** (**serial\_pi.cpp**).

Below is the *update\_sum* function from **serial\_pi.cpp** which sums all the terms of the series serially.

double update\_sum(int total\_terms)

{

double sum = 0.0;

double factor = 1;

for(int i = 0 ; i < total\_terms ; i++, factor = -factor)

{

sum += factor/(2.0 \* i + 1.0);

}

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.

Create a program named **parallel\_pi.cpp**. Your main function should remain the same. Re-write the update\_sum function to parallelize the code. You will need to **#include <omp.h>** at the top of your source file.

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:

#pragma 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.

#pragma omp critical

sum += my\_sum;

The above code is how we avoid **data race** when we update the global variable in OpenMP. A data race occurs when:

* two or more threads in a single process access the same memory location concurrently, and
* at least one of the accesses is for writing, and
* the threads are not using any exclusive locks to control their accesses to that memory.

To compile your **parallel\_pi.cpp** program, type the following:

g++ **-fopenmp** parallel\_pi.cpp

### Run Your Code

Create a blank text document named **test.txt**.

Now, run the serial version ten times and calculate the average time. Write “Serial Average Time: “ and then the average time on the first line of **test.txt**.

Next, run the parallel version ten times each with 2 to 8 threads and calculate the average time. Write “Parallel Average Time: “ and then the average time on the second line of **test.txt**.

Next, calculate the speedup based on the average times. Write “Speedup: “ and then the difference of the times on the third line of **test.txt**.

## What should you expect as your Results?

Below figure shows the computation time and value of Pi while using threads up to 8 in a C++ implementation. Both implementations are run on an Intel Core i7 laptop with 4 physical cores (and 8 virtual cores). The implementations show good speedup as we increase number of threads.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | **Number of Threads** | **Compute Time (in seconds)** | **Pi** | | 1 | 45.791 | 3.14159265308772 | | 2 | 23.928 | 3.14159265308789 | | 4 | 11.17 | 3.14159265308940 | | 6 | 5.992 | 3.14159265308899 | | 8 | 5.312 | 3.14159265308925 | |

# sample output

## Serial Output

Computation started..

Pi = 3.142

Computation time = 2.4 seconds

## Parallel output

Computation started..

Pi = 3.133

Computation time = 0.4 seconds

## Sample Text from text.txt

Serial Average Time: 2.36 seconds

Parallel Average Time: 0.40 seconds

Speedup (Serial / Parallel): 5.9

# What to Turn In

ZIP & upload the following files to the ilearn submission folder:

* parallel\_pi.cpp
* text.txt

