# Lab-2 Report

This report talks about the calculating the value of Pi($\pi$) using Wallis Method and Leibniz Series. I implement, both the ways of calculating Pi($\pi$) serially and in a multi-threaded fashion.
Since, the precision of double or long double is limited, we expect not to see accuracy beyond 15 decimal digits when calculating $\pi$.

I have chosen Wallis and Leibniz Series to calculate the value of $\pi$, because these methods were slow in comparison to other methods such as Ramanujan formula. This will help me in studying the effects of using multi-threading more and garner experience out of speeding up the calculation using multi-threading.

## Wallis Method

[Reference 1 - (19)]

$$\frac{\pi}{2} = \prod_{n=1}^{\infty} \frac{4n^2}{4n^2-1} = \prod_{n=1}^{\infty} \left( \frac{2n}{2n-1} \cdot \frac{2n}{2n+1} \right)$$
$$= \left(\frac{2}{1} \cdot \frac{2}{3}\right) \cdot \left(\frac{4}{3} \cdot \frac{4}{5}\right) \cdot \left(\frac{6}{5} \cdot \frac{6}{7}\right) \cdot \left(\frac{8}{7} \cdot \frac{8}{9}\right) \cdot \cdots$$

This method calculates value of $\pi$ using a product series. Shown below is an implementation in C.

```c
long double CalPiFromWallisFormula(long int n)
{
        long double product=1.0;
        long double pi,term;
        for(long double i=1; ((long int)i)<n; i++){
                if(product == 0.0 ) break;
                term=(2*i)/(2*i-1);
                product=product*term;
                term=(2*i)/(2*i+1)*1.0;
                product=product*term;
        }
        long double pi = 2 * product;
        return pi;
}
```

## Leibniz Series

[Reference 1 - (6) ]

$$1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots = \frac{\pi}{4}$$

It is derived from the taylor series and uses a sum series to calculate the value of $\pi$.
Shown below is an implementation in C.

```c
long double CalPiFromLeibnizFormula(long int n)
{
        long double pi;
        long double sum=0.0;
        long double term;
        for(long int i=0;i<n;i++){
                term= pow(-1,i)/(2*i+1);
                sum+= term;
        }
        pi = 4 * sum;
        return pi;
}
```

## Approach for Multi-thread execution

Approach M : Uses *mutex* as a synchronising medium.
Approach S : Uses *semaphore* as a synchronising medium.

For multi-thread execution, I divided the number of terms and gave each thread its share of computations to perform. After each thread computes its calculations, I then combine the results in a global variable.
I also found that Approach S produced better timings on multi-threading executions than Approach M, due to unknown reasons. Even though, general text on the web, mentions mutexes as being lightweight. At the end, I show results using Approach S.
A thread job for Wallis method is shown below.

```
void* ThreadJobWallis(void* threadCount)
{
        long tCount = (long) threadCount;
        long int startTerm = (gTermsLeibniz/gNumOfThreads)*tCount;
        long int endTerm = (gTermsLeibniz/gNumOfThreads)*(tCount+1);
        long double tProduct = GetProductWallis(startTerm,endTerm);
        pthread_mutex_lock(&gSumMutex);
        gThreadProductWallis = gThreadProductWallis*tProduct;
        pthread_mutex_unlock(&gSumMutex);
        return NULL;
}
```

## Results

Values of ($\pi$) generated on *2000000000 terms* of series :

```
Actual  π                = 3.14159265358979311599796346854 4
Leibniz π                = 3.14159265308979655188098556450 9
Leibniz π (MT)           = 3.14159265308979522785329252521 1
Wallis  π                = 3.14159265319710102925057637612 8
Wallis  π (MT)           = 3.14159265313596882383297137497 6
```

```
(MT : On multi-threading variant)
```

Accuracy

```
Leibniz Serial = 10 decimal digits
Leibniz MT     = 10 decimal digits
Wallis  Serial = 10 decimal digits
Wallis  MT     = 10 decimal digits
```

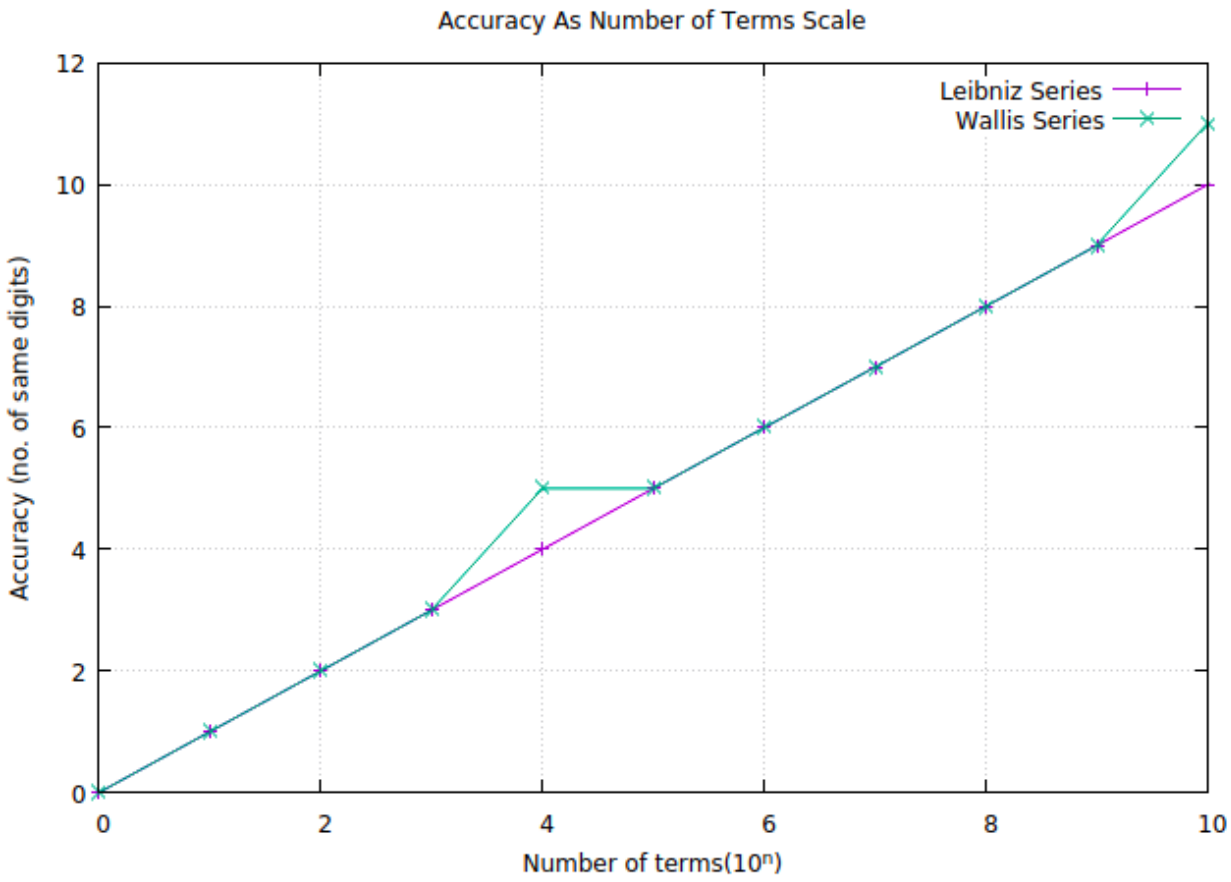
Accuracy (in bits)

```
Leibniz Serial = 25 decimal bits
Leibniz MT     = 25 decimal bits
Wallis  Serial = 25 decimal bits
Wallis  MT     = 25 decimal bits
```

Accuracy (in absolute percentage)

```
Leibniz Serial = 100.000000%
Leibniz MT     = 100.000000%
Wallis  Serial = 100.000000%
Wallis  MT     = 100.000000%
```

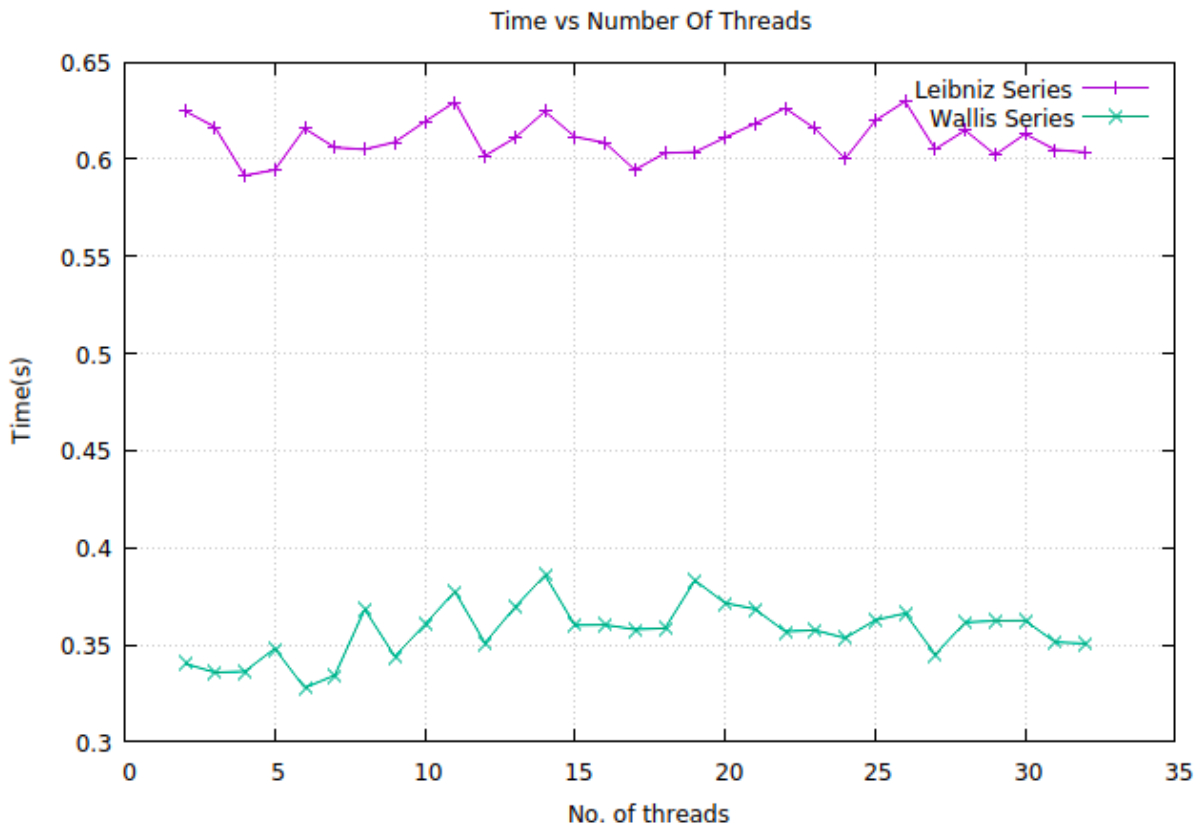
The below graph shows how accuracy of Pi($\pi$) increases with increase in number of terms for Wallis method and Leibniz Series.



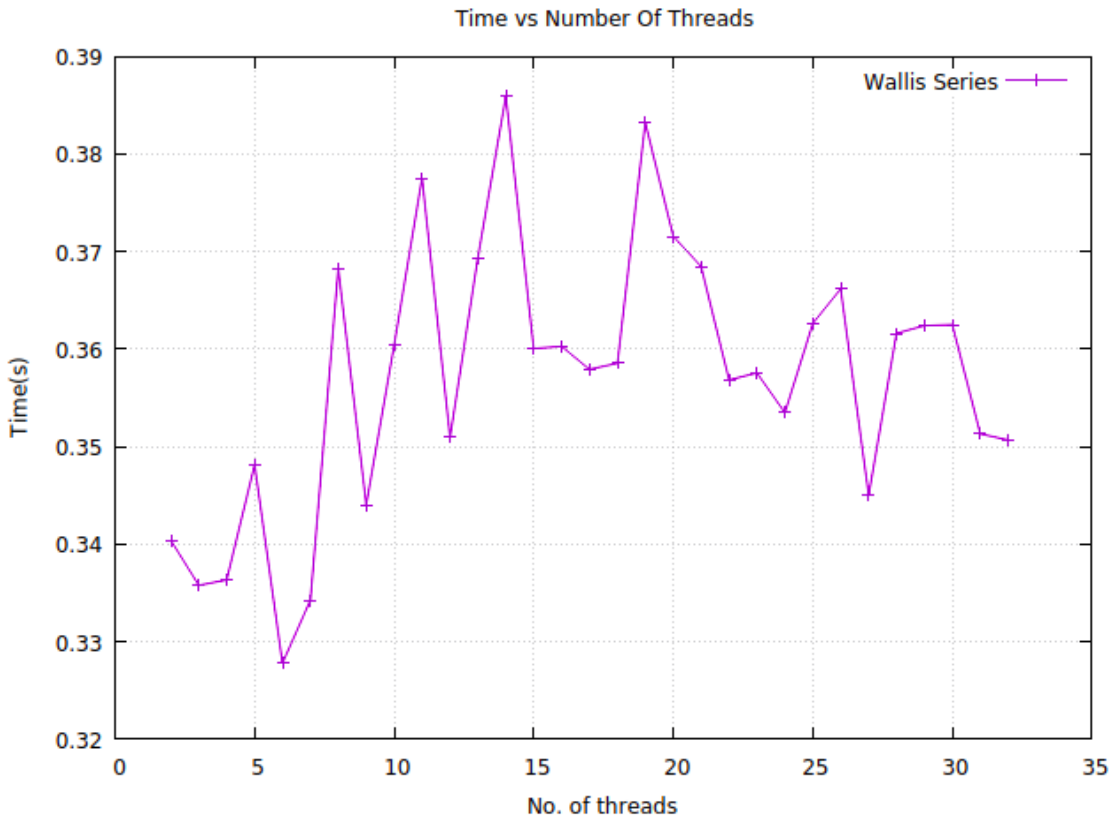

In the below graphs , we observe:

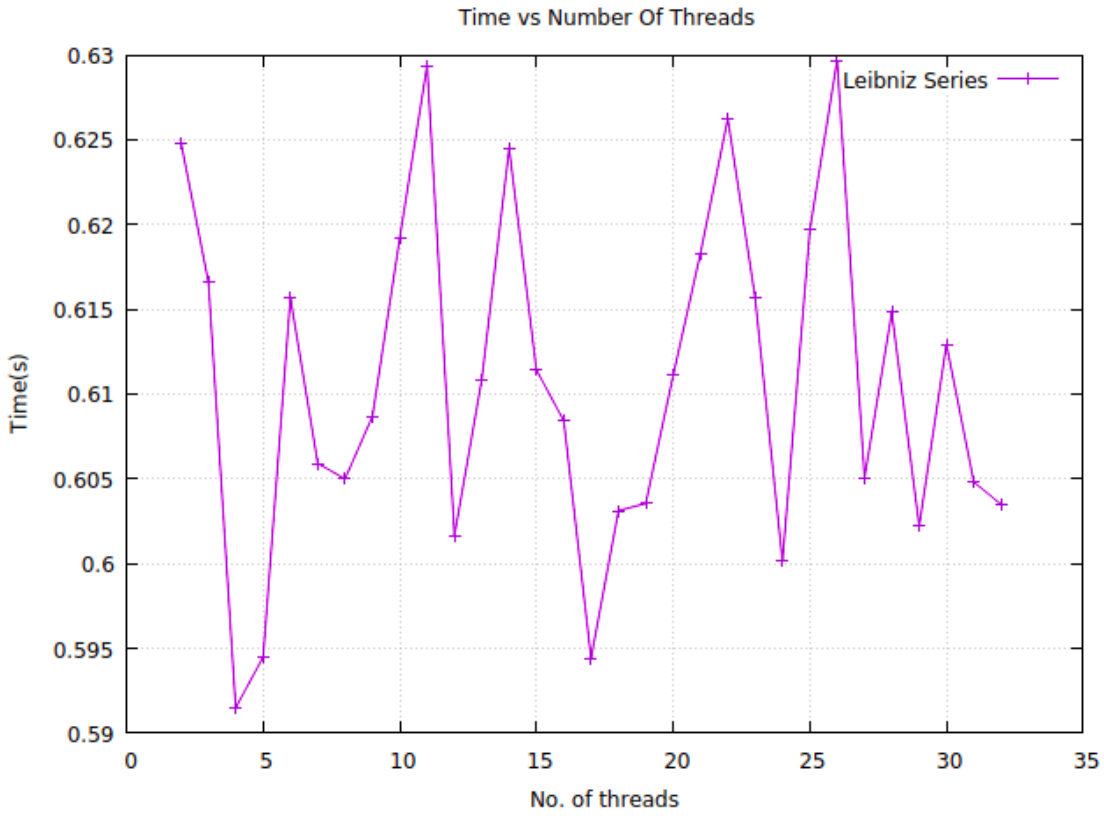
- We can see, the Wallis Product Series method converges faster than the Leibniz Sum series.
- We see that in general as the thread count increases beyond a threshold it starts to increase the overall time curve.

(Please turn over)




(Please turn over for more graphs)


Time vs Number Of Threads
Wallis Series
Time(s)
No. of threads
0.39
0.38
0.37
0.36
0.35
0.34
0.33
0.32
0
5
10
15
20
25
30
35


Time vs Number Of Threads
Leibniz Series
Time(s)
No. of threads
0.63
0.625
0.62
0.615
0.61
0.605
0.6
0.595
0.59
0
5
10
15
20
25
30
35

## Data

Accuracy Decimal Digits Data.

| Terms (10^n) | Leibniz Sum | Wallis Product |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 1 | 1 |
| 2 | 2 | 2 |
| 3 | 3 | 3 |
| 4 | 4 | 5 |
| 5 | 5 | 5 |
| 6 | 6 | 6 |
| 7 | 7 | 7 |
| 8 | 8 | 8 |
| 9 | 9 | 9 |
| 10 | 10 | 11 |

Threads vs Time Data. [Number of Terms=18018000]

| Thread Count | Leibniz Series Time(s) | Wallis Product Time(s) |
|---|---|---|
| 2 | 0.624821 | 0.340392 |
| 3 | 0.616579 | 0.335778 |
| 4 | 0.591496 | 0.336337 |
| 5 | 0.594485 | 0.348136 |
| 6 | 0.615681 | 0.327805 |
| 7 | 0.605916 | 0.334275 |
| 8 | 0.604995 | 0.368242 |
| 9 | 0.608686 | 0.34404 |
| 10 | 0.619177 | 0.360349 |
| 11 | 0.629382 | 0.377522 |
| 12 | 0.601677 | 0.350936 |
| 13 | 0.61084 | 0.369345 |
| 14 | 0.624493 | 0.385899 |
| 15 | 0.61142 | 0.360041 |
| 16 | 0.608412 | 0.360259 |
| 17 | 0.594425 | 0.357908 |

| | | |
|---|---|---|
| 18 | 0.603122 | 0.358541 |
| 19 | 0.603539 | 0.383172 |
| 20 | 0.61109 | 0.371543 |
| 21 | 0.618275 | 0.36846 |
| 22 | 0.626238 | 0.356812 |
| 23 | 0.615725 | 0.357537 |
| 24 | 0.600143 | 0.353479 |
| 25 | 0.619685 | 0.362619 |
| 26 | 0.629676 | 0.366157 |
| 27 | 0.605075 | 0.345007 |
| 28 | 0.614825 | 0.361589 |
| 29 | 0.602213 | 0.362387 |
| 30 | 0.612929 | 0.362474 |
| 31 | 0.604804 | 0.351295 |
| 32 | 0.603494 | 0.350663 |

References :


1. The Calculation and Analysis of PI in Computer Numerical Simulation by ZENG Shengda, RUAN Zhiyi, CAI Jianwei, LIN Lixin, WU Lurong. 2012
2. Implementation Example from CodeAnsar.
3. Wikipedia PI
4. Difference between Binary Semaphore and Mutex.