CS301: HIGH PERFORMANCE COMPUTING
ASSIGNMENT TWO

Submitted By:
Shaleen Kumar Gupta (201301429)
Visharad Bansal (201301438)

Q1.
PROBLEM STATEMENT

Calculate the value of pi by sum of series.

METHOD

pi= 4*[1 - (1/3) + (1/5) – (1/7) + ...]

COMPLEXITY

The serial code runs in O(n) time, where n is the number of integration steps.
OpenMP is used to divide the serial code among 4 threads, giving a theoretical speedup of 4x

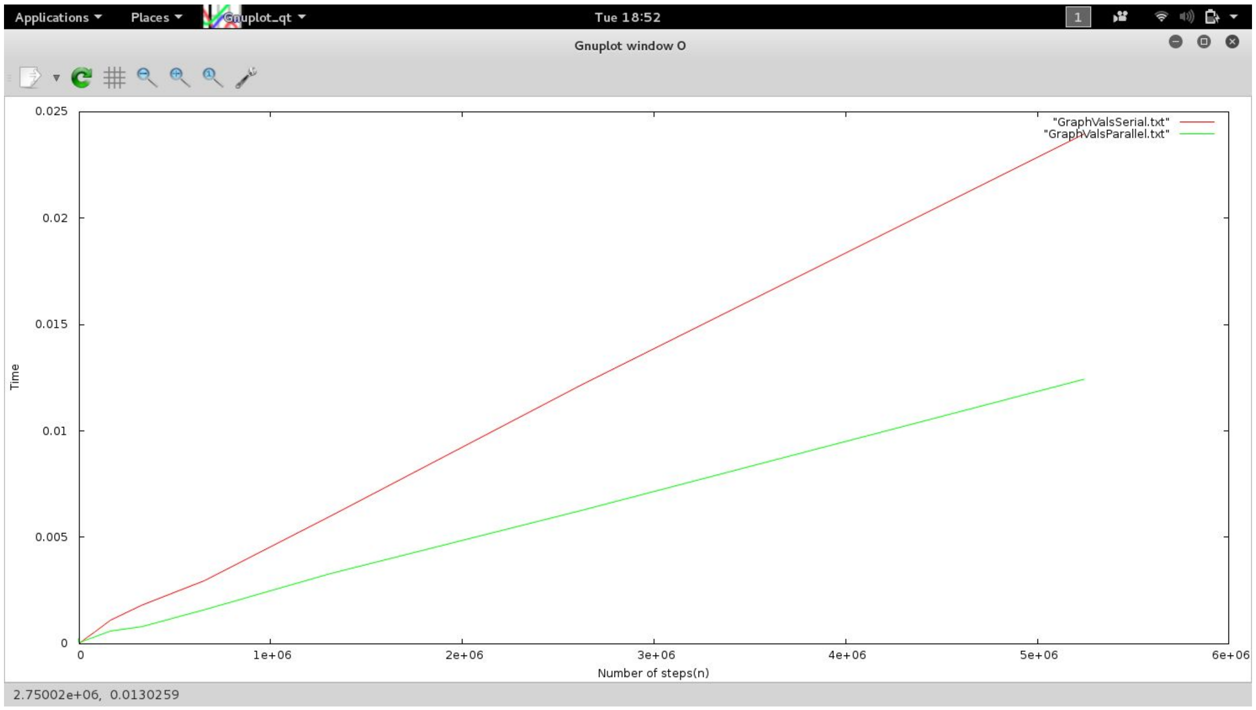
OPTIMIZATION STRATEGY

2 out of 4 threads (thread 0 and 2) are assigned all the positive terms of the series while thread 1 and 3 sum up all the negative terms. Then a reduction clause is used to calculate the final sum.

OBSERVATIONS AND PROBLEMS FACED

Using the OpenMp reduction clause gives a much better time compared to using an array to store local sums. This is probably because of the load and store overheads associated with using an array. Also, rather than declaring the variables inside each thread, they were declared outside in the serial part and made private to each thread. This also helped improve the time for the parallel code.

However, each step still requires a division to be performed, which is an intensive task. There are probably other methods such as Monte Carlo which will give an even better performance.



Comparison of serial and parallel times

HARDWARE DETAILS

No. of Processors	: 3
vendor_id	: Genuine Intel
CPU family	: 6
model	: 60
model name	: Intel(R) Core(TM) i5-4590 CPU @ 3.30GHz
stepping	: 3
CPU MHz	: 800.000
cache size	: 6144 KB
siblings	: 4
CPU cores	: 4

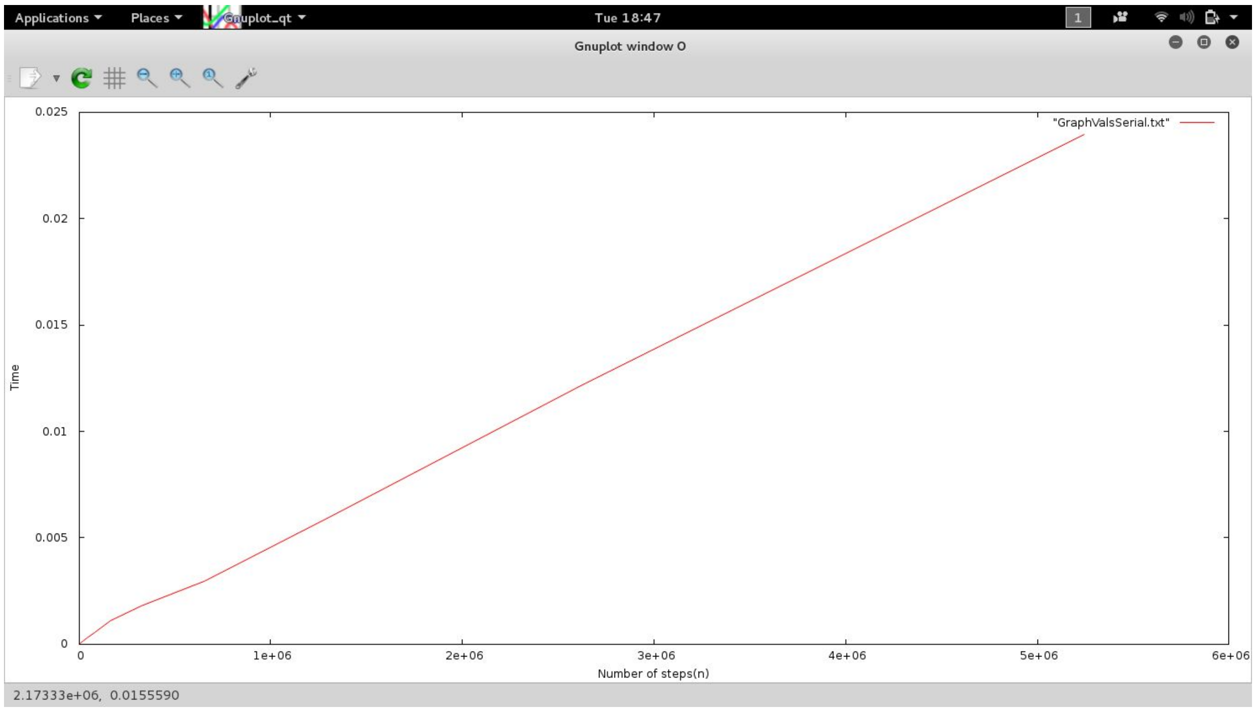
INPUT PARAMETERS

Number of steps (n).
n was taken from 10 to 10,000

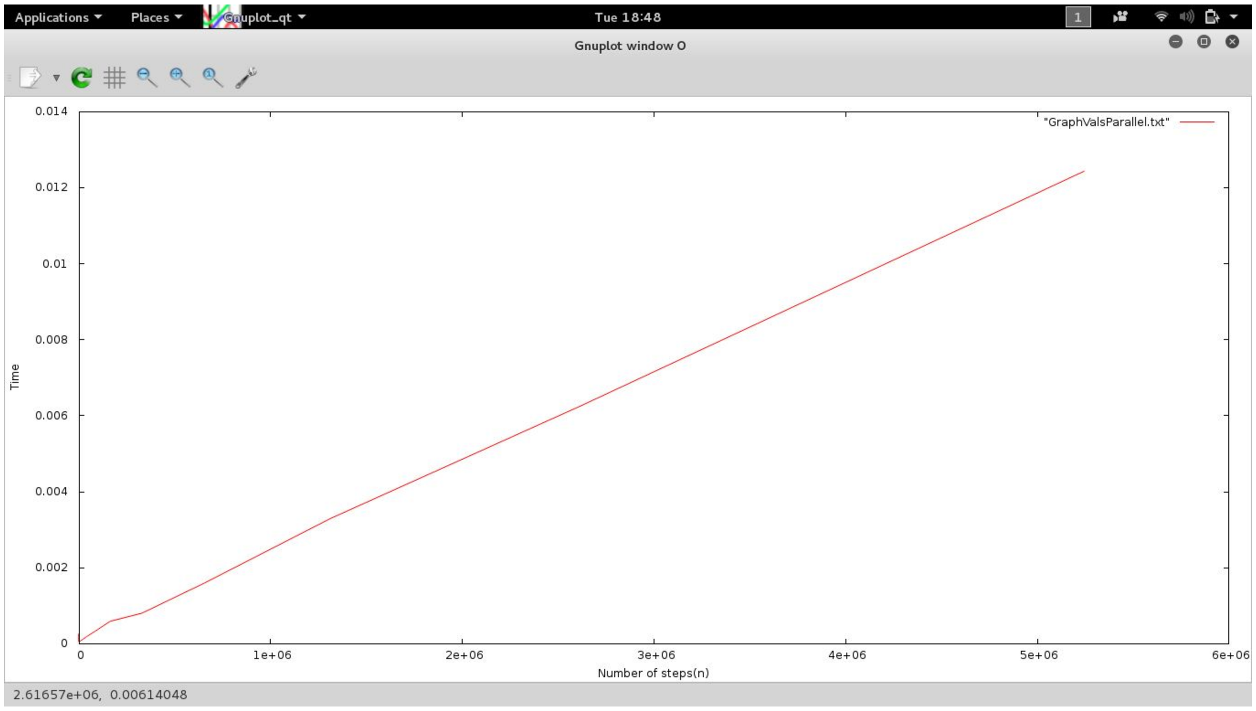
OBSERVATIONS

Both serial and parallel codes gave almost linear speedup with respect to the problem size, with a few kinks, specially for small values of n.

Time was calculated using omp_get_wtime() library function include in the "omp.h" header file.



Serial Scaling

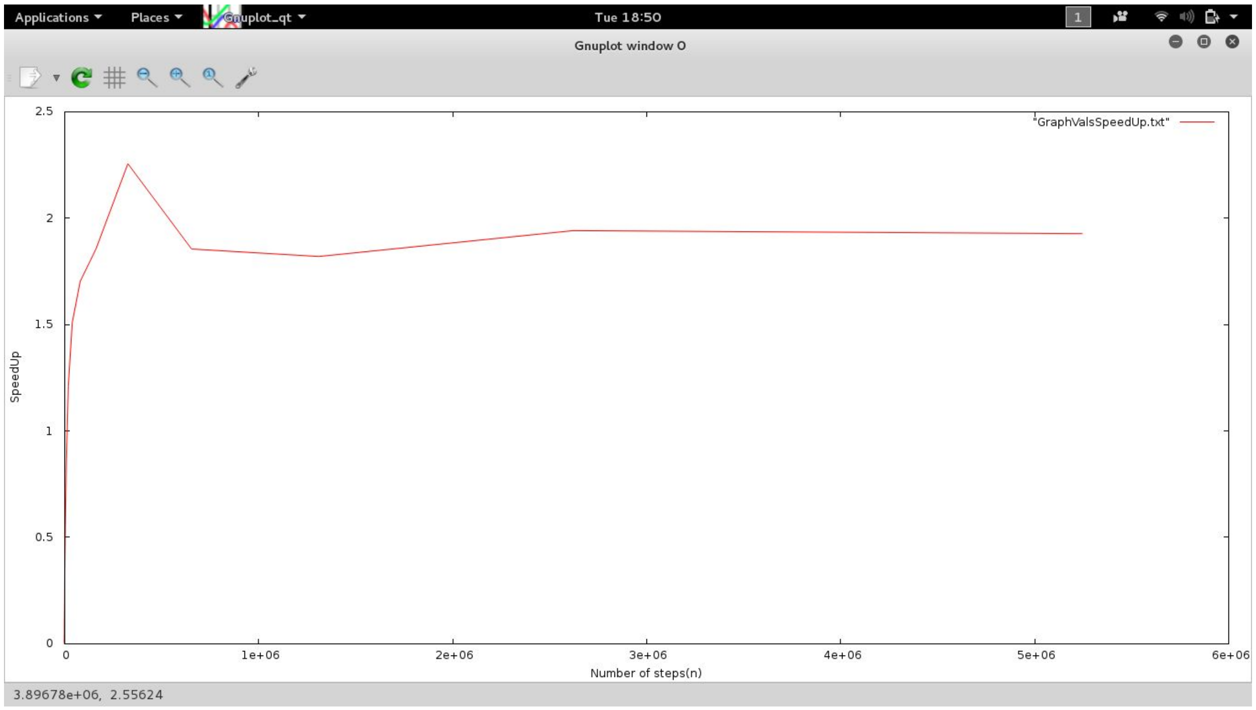


Parallel Scaling

SPEED UP

The maximum speed up achieved was ~2.2
Speed Up was calculated as:

Speed Up= (Time taken by serial code) / (Time taken by parallel code)


Applications
Places
Gnuplot_qt
Tue 18:50
Gnuplot window 0
"GraphValsSpeedUp.txt"
SpeedUp
Number of steps(n)
0
0.5
1
1.5
2
2.5
0
1e+06
2e+06
3e+06
4e+06
5e+06
6e+06
3.89678e+06, 2.55624

Speed Up v/s N

Q2.

PROBLEM STATEMENT

Calculate the Nth Fibonacci number.

METHOD

This is the basic algorithm:

```
fib[0]=fib[1]=1;
fib[n]=fib[n-1]+fib[n-2]
```

Matrix method:

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n = \begin{pmatrix} F_{n-1} & F_n \\ F_n & F_{n+1} \end{pmatrix}$$

COMPLEXITY

After memoization, the code runs in O(n) time. An array fib[] is used to store the values. The matrix implementation runs in O(n) time, as the multiplication is for a 2x2 matrix.

OPTIMIZATION STRATEGY

IMPLEMENTATION 1

For every call to the function fibonacci(n) (which returns the nth fibonacci number), two threads are spawned: one which calls fibonacci(n-1) and the second which calls fibonacci(n-2). Then the threads are made to synchronize via a wait call, and the master adds up the two values and returns the answer.

To reduce the time taken, fibonacci numbers are stored in the array fib[] the first time they are calculated. The function then simply checks if fib[n] is non zero and returns it.

Also, for n<20 the parallel code was reduced to serial, in the hope of a better speedup.

IMPLEMENTATION 2

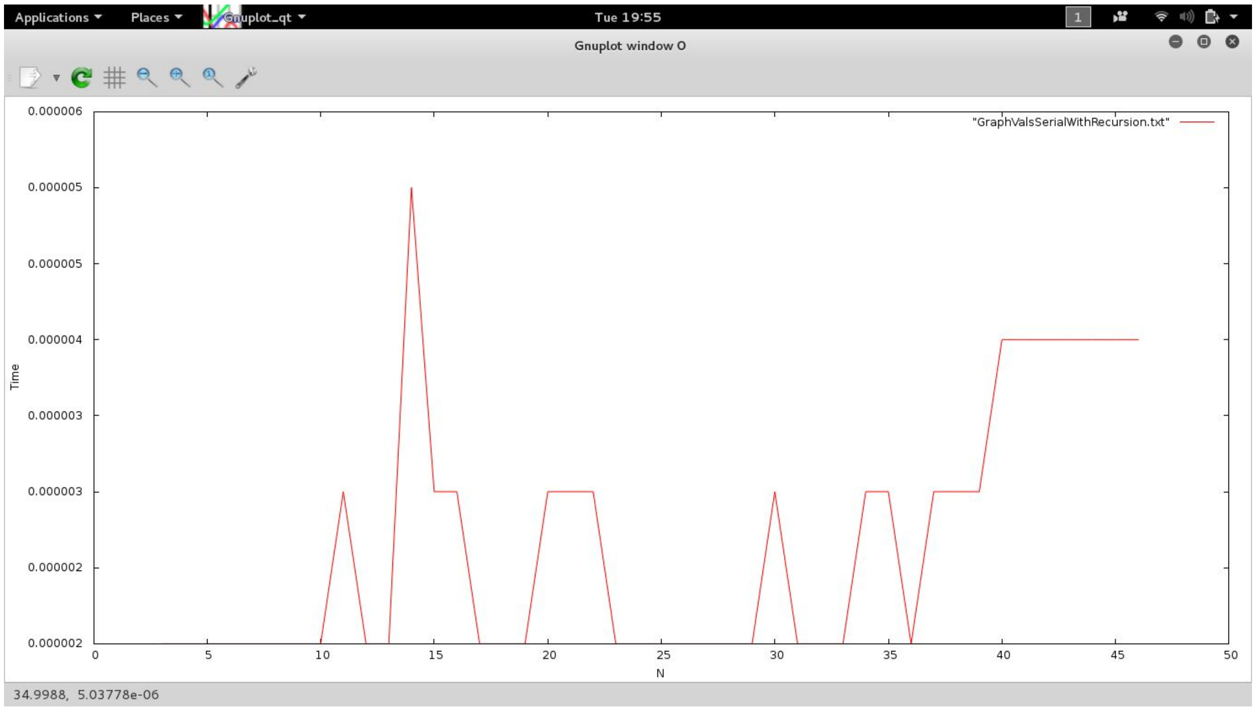
Parallel matrix multiplication was used to calculate the nth fibonacci number as per the formula described earlier. The outer loop was parallelized and the *#pragma for* construct was used.

OBSERVATIONS AND PROBLEMS FACED

Spawning of new threads and subsequent parallelization of the the code is useful for computation intensive programs. The parallel code for fibonacci fails to give a speedup because the overhead (the master creating new threads, dividing tasks and synchronization at the end) is much more compared to the task performed by each thread, which is in most cases just to call a function and return a value. As a result, the serial code always gave a better time compared to the parallel code.

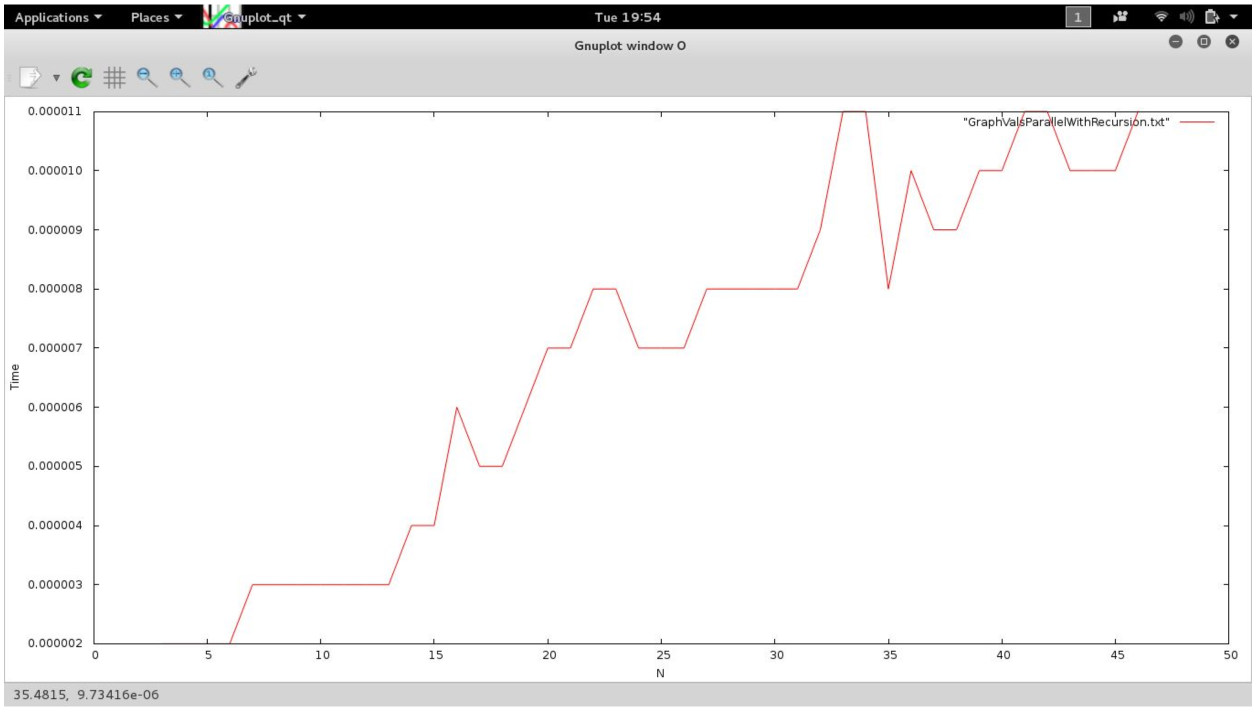
There is also the size limitation. The datatype long long was used to store the numbers, and it only stored upto the 45th fibonacci number accurately before overflowing. Maybe with a higher value of n a speedup > 1 will be observed.

Graph 1: Serial Implementation with Recursion



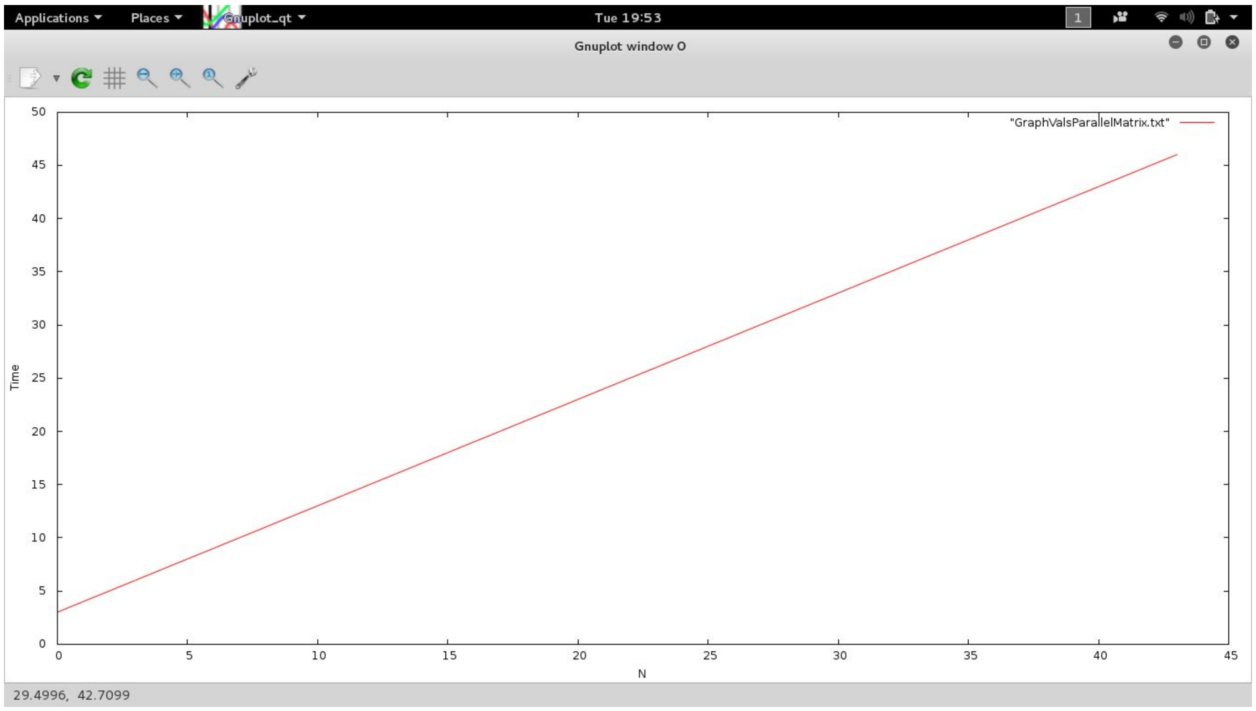
Due to memoization, we observe certain spikes

Graph 2: Parallel Implementation with Recursion



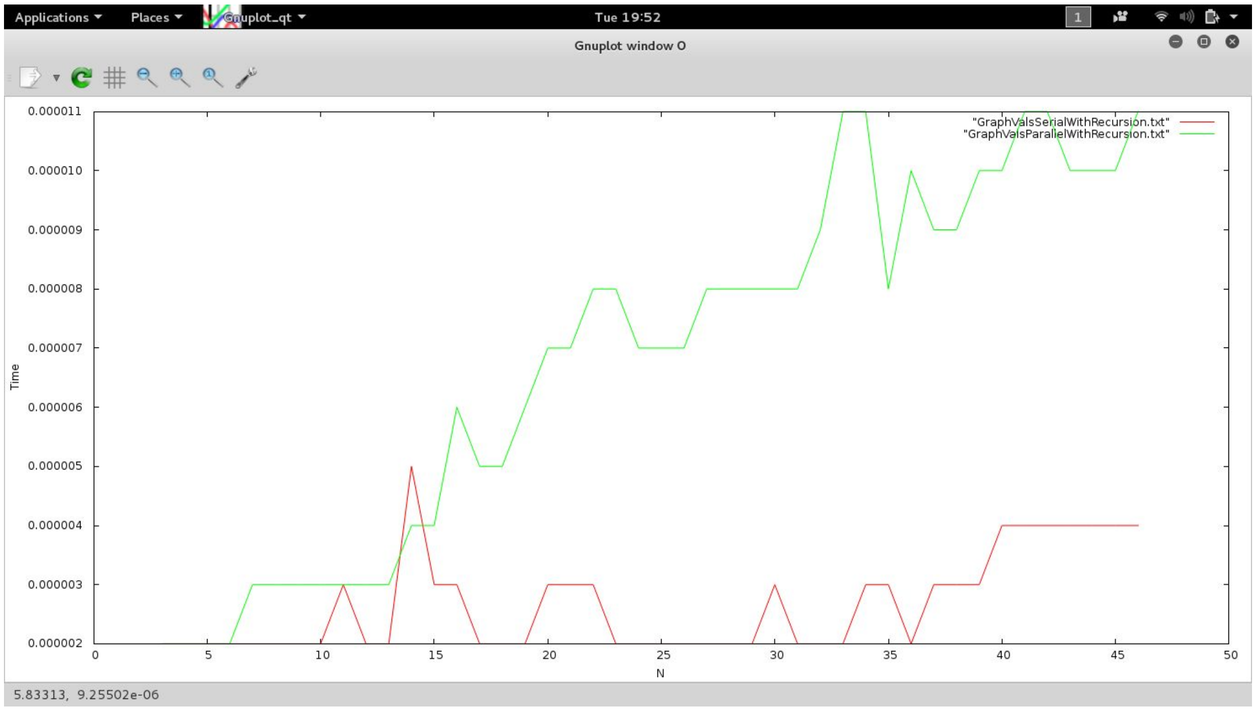
Here we observe a more or less linearly increasing time due to the high number of threads spawned

Graph 3: Parallel Matrix Implementation

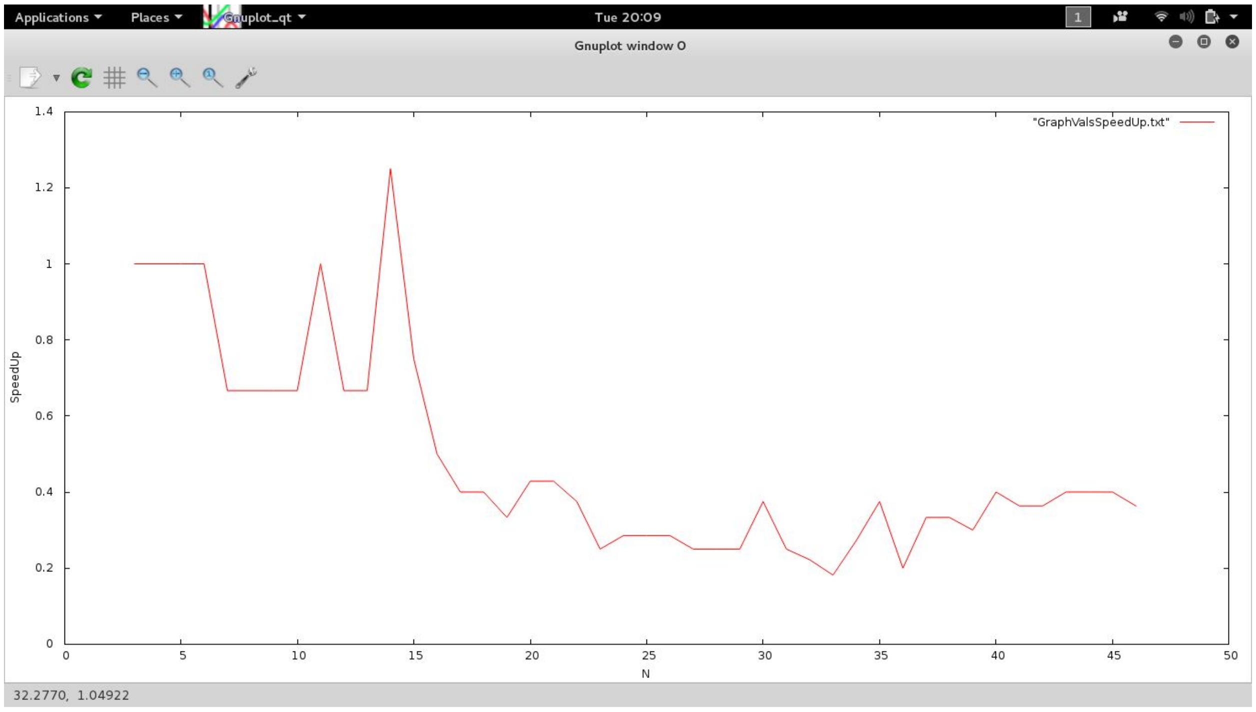


Implementation using the matrix method takes a much longer time compared to recursive implementations. This is because of the intensive matrix multiplications (high number of saxpy operations)

Graph 4: Serial and Parallel Implementation with recursion



The SpeedUp Curve



The speedup is less than 1. It hovers around 1 for n<20 as both codes follow the serial implementation. For n>20, the parallel overheads is too high and brings the speedup to an average of 0.5

HARDWARE DETAILS

No. of Processors	: 3
vendor_id	: Genuine Intel
CPU family	: 6
model	: 60
model name	: Intel(R) Core(TM) i5-4590 CPU @ 3.30GHz
stepping	: 3
CPU MHz	: 800.000
cache size	: 6144 KB
siblings	: 4
CPU cores	: 4

INPUT PARAMETERS

Number of steps (n).
n was taken upto 45.

Q3.
PROBLEM STATEMENT

Write a serial code and a parallel code for performing bubble sort and odd-even sort

METHOD

Bubble sort, is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. Although the algorithm is simple, it is too slow and impractical for most problems even when compared to insertion sort.

Odd–even sort is a sorting algorithm developed originally for use on parallel processors with local interconnections. It is a comparison sort related to bubble sort, with which it shares many characteristics. It functions by comparing all odd/even indexed pairs of adjacent elements in the list and, if a pair is in the wrong order (the first is larger than the second) the elements are switched. The next step repeats this for even/odd indexed pairs (of adjacent elements). Then it alternates between odd/even and even/odd steps until the list is sorted.

THEORETICAL SPEEDUP

OpenMP is used to divide the serial code among 4 threads, giving a theoretical speedup of 4x

OPTIMIZATION STRATEGY

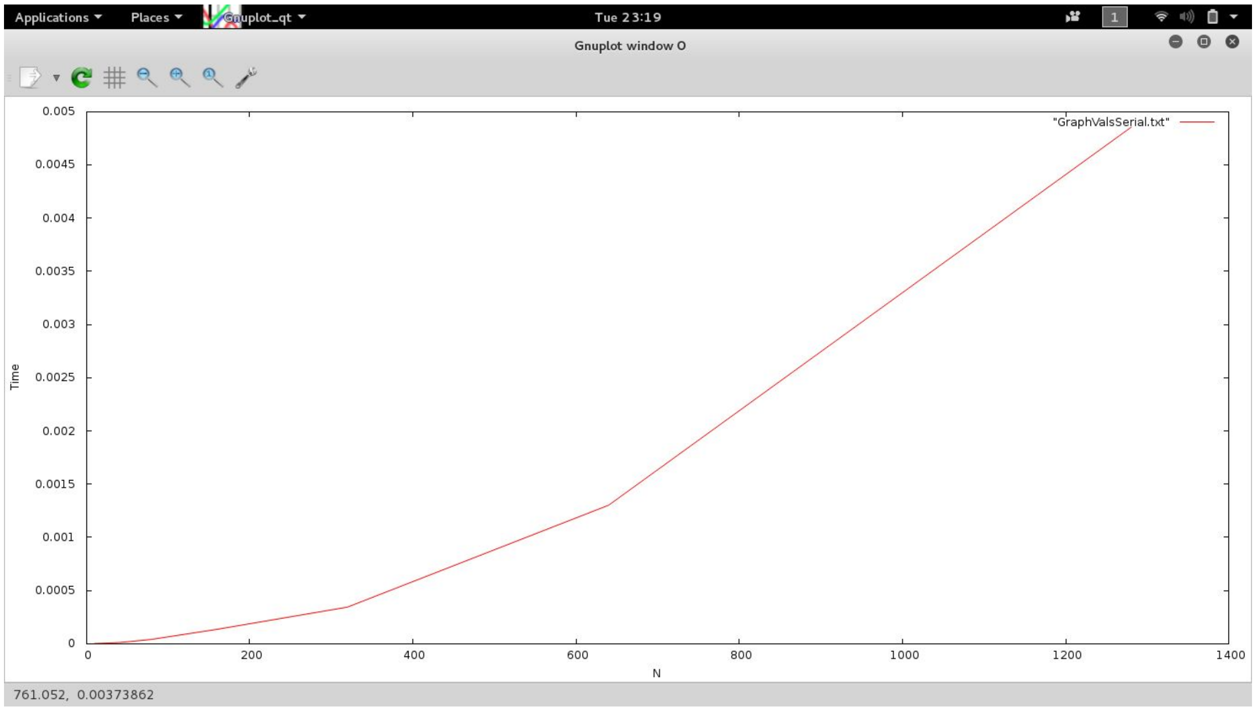
Algorithm for Parallel Bubble Sort

```
1. For $k = 0$ to $n-2$
2. If $k$ is even then
3.         for $i = 0$ to $(n/2)-1$ do in parallel
4.         If $A[2i] > A[2i+1]$ then
5.          Exchange $A[2i] \leftrightarrow A[2i+1]$
6. Else
7.         for $i = 0$ to $(n/2)-2$ do in parallel
8.         If $A[2i+1] > A[2i+2]$ then
9.          Exchange $A[2i+1] \leftrightarrow A[2i+2]$
10. Next $k$
```

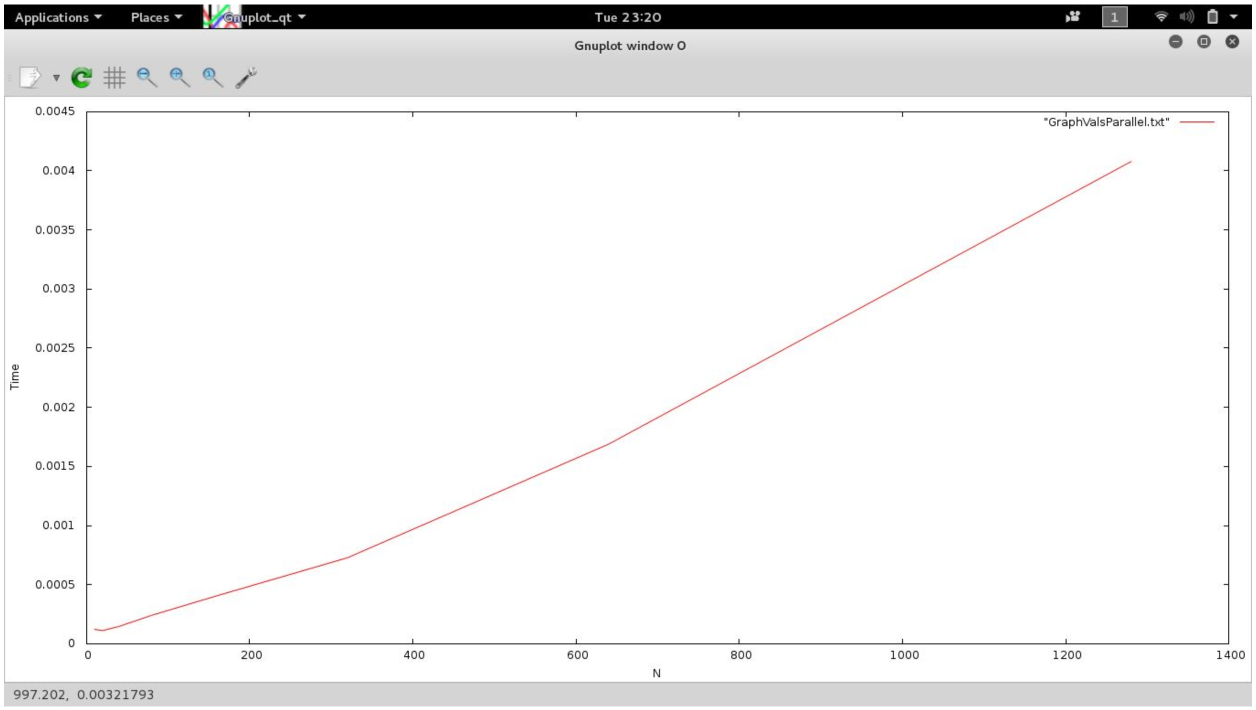
COMPLEXITY

Steps 1-10 is a one big loop that is represented n -1 times. Therefore, the parallel time complexity is $O(n)$. If the algorithm, odd-numbered steps need $(n/2)$ - 2 processors and even-numbered steps require $(n/2)$ - 1 processors. Therefore, this needs $O(n)$ processors.

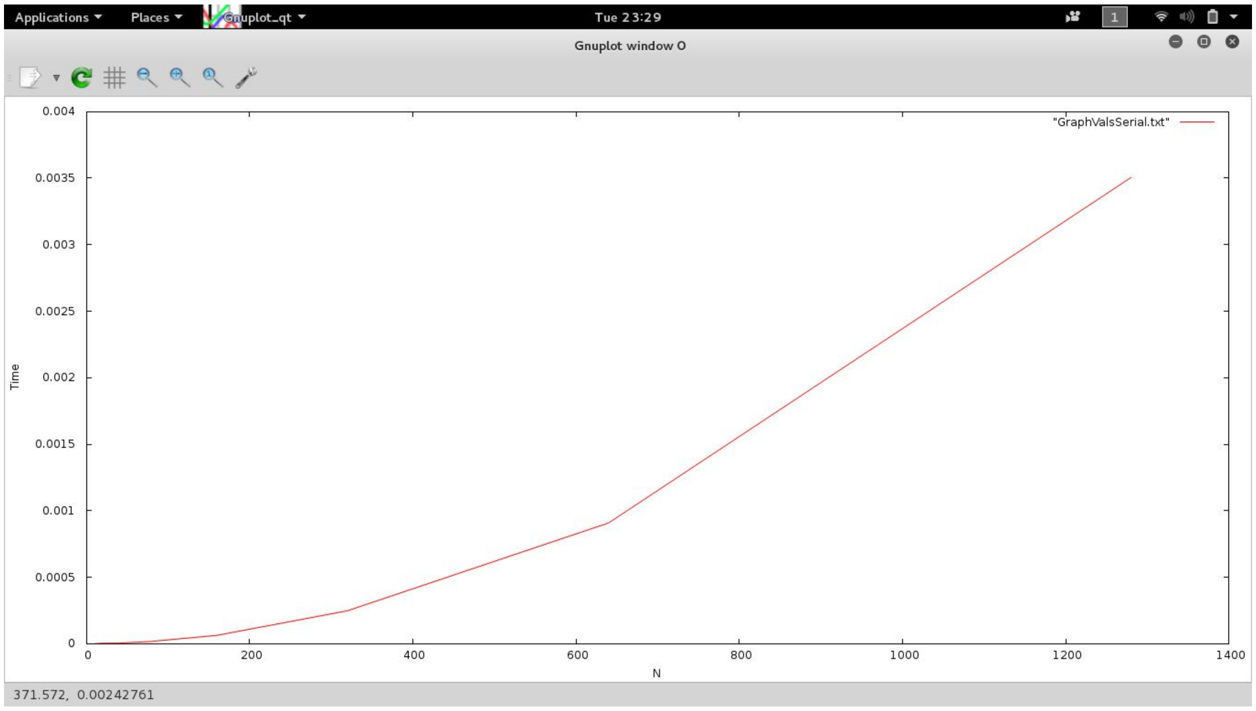
OBSERVATIONS AND PROBLEMS FACED



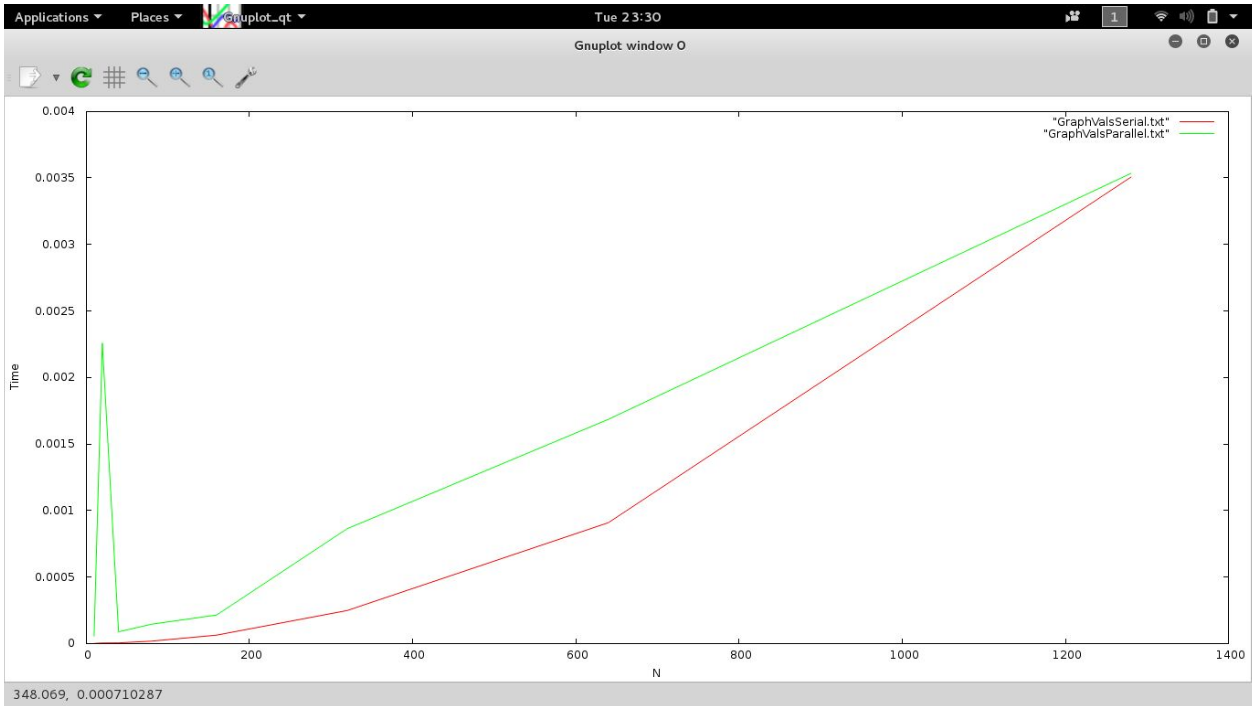
Time taken vs Problem Size for Bubble Sort in Serial Implementation



Time taken vs Problem Size for Bubble Sort in Parallel Implementation



Time taken vs Problem Size for Odd Even Sort in Serial Implementation



Time taken vs Problem Size for Odd Even Sort in Parallel Implementation

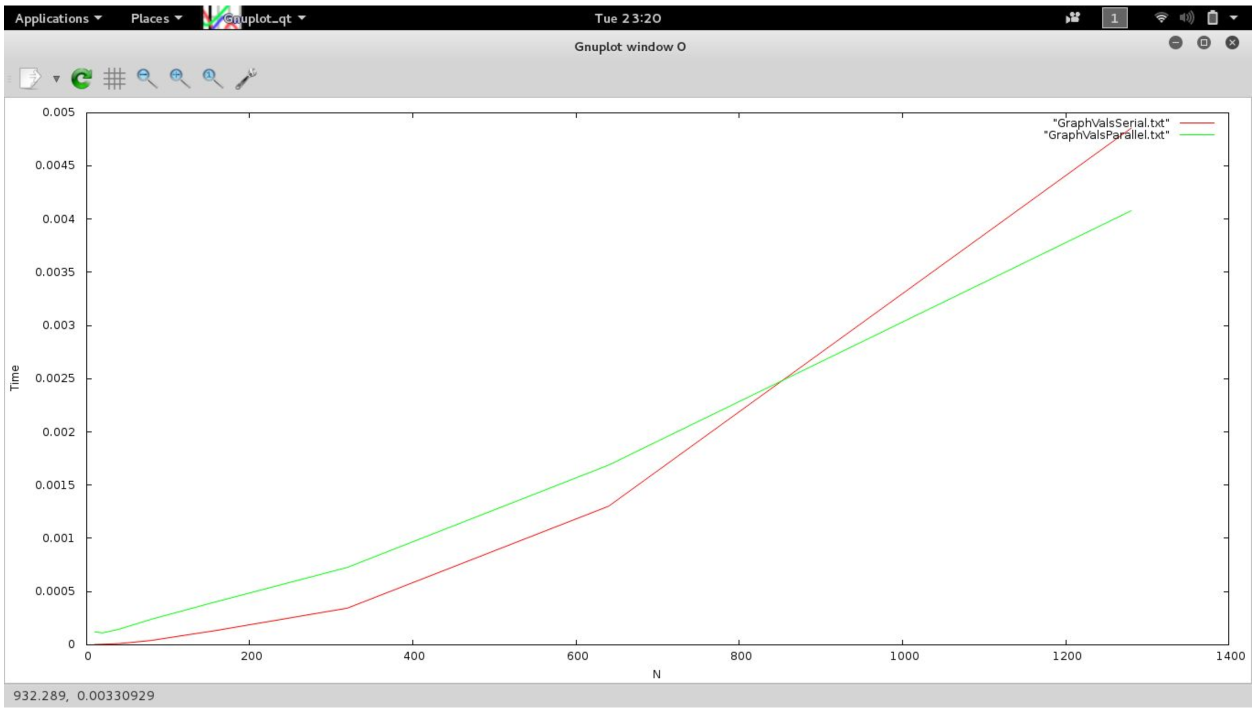
HARDWARE DETAILS

No. of Processors	: 3
vendor_id	: Genuine Intel
CPU family	: 6
model	: 60
model name	: Intel(R) Core(TM) i5-4590 CPU @ 3.30GHz
stepping	: 3
CPU MHz	: 800.000
cache size	: 6144 KB
siblings	: 4
CPU cores	: 4

INPUT PARAMETERS

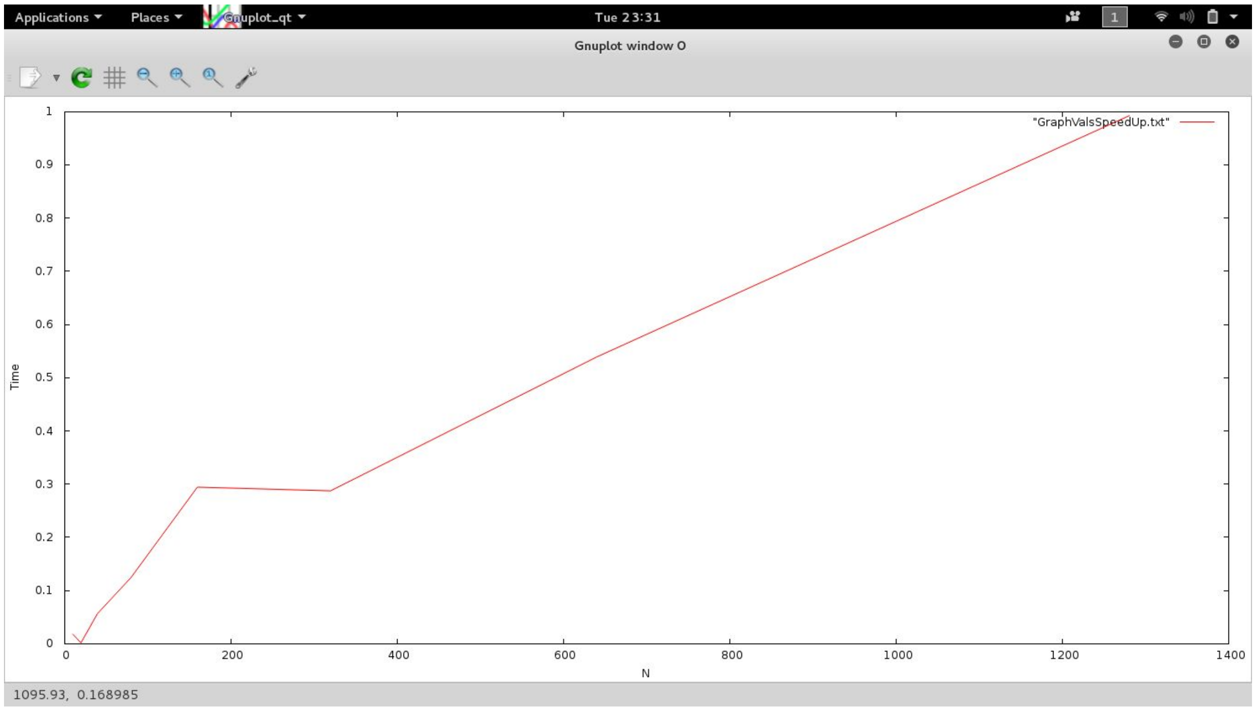
Number of steps (n).
n was taken from 100 to 2000 with n being doubled in each iteration.

OBSERVATIONS



Time taken vs Problem Size for Bubble Sort in Parallel Implementation and Serial Implementation

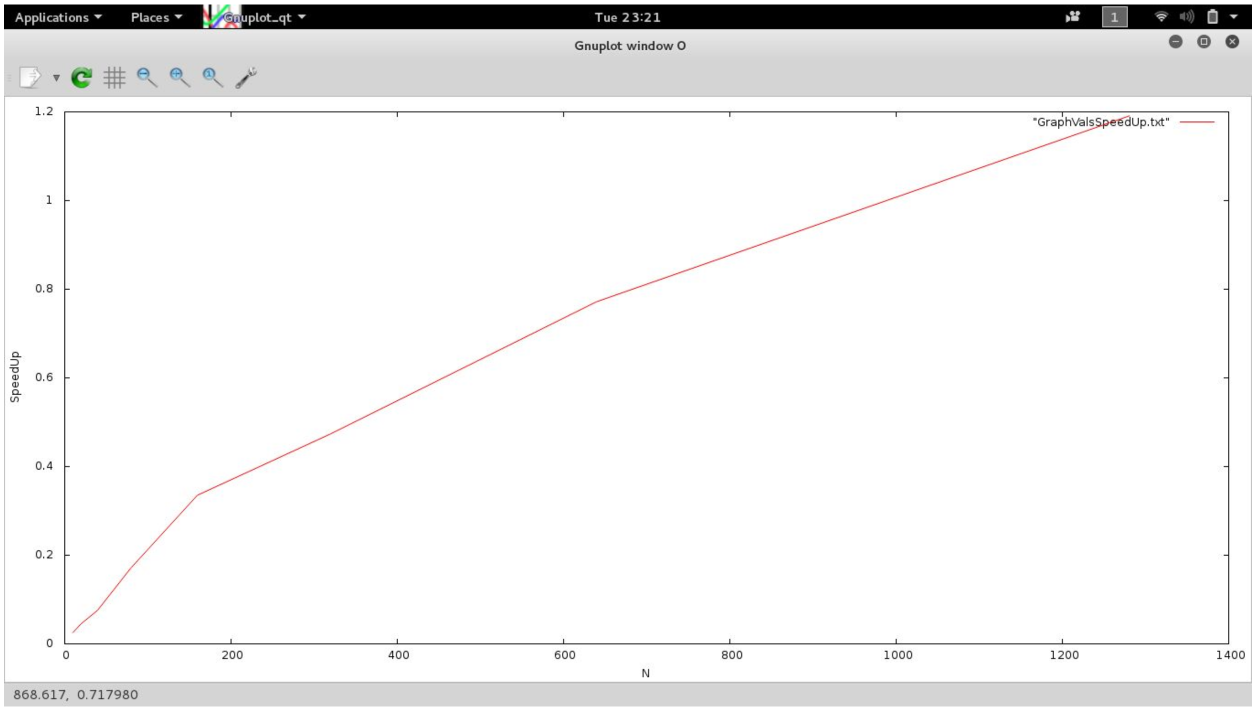
In the Bubble Sort implementations, we observed that the time taken in Parallel implementation is slightly higher for smaller size arrays while it goes below the Serial curve after a certain point. This depicts the expected outcome. For smaller input sizes, we experience more parallelization overheads than the optimization achieved.



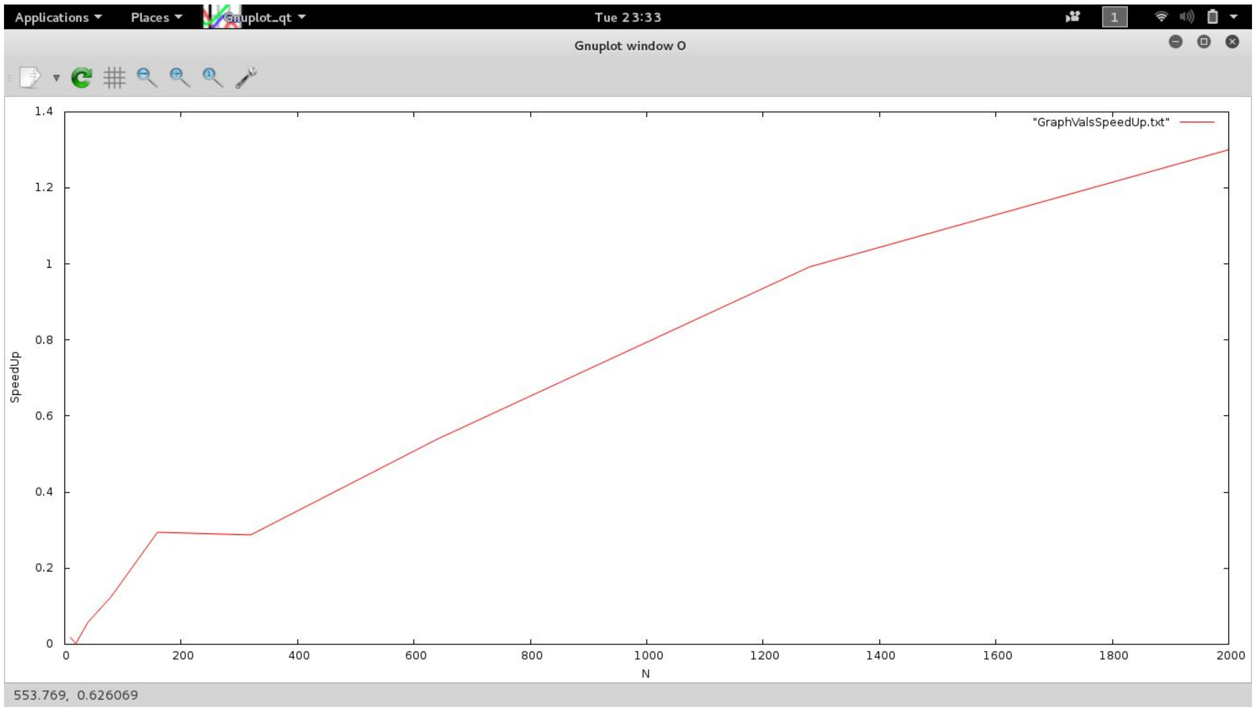
Time taken vs Problem Size for Odd Even Sort in Parallel Implementation and Serial Implementation

Time was calculated using omp_get_wtime() library function include in the "omp.h" header file.

SPEED UP



Problem Size vs SpeedUp for Bubble Sort in Parallel Implementation against Serial Implementation



Problem Size vs SpeedUp for Bubble Sort in Parallel Implementation against Serial Implementation

Speed Up was calculated as:

Speed Up= (Time taken by serial code) / (Time taken by parallel code)