Parallel and Distributed Computing

Assignment 1

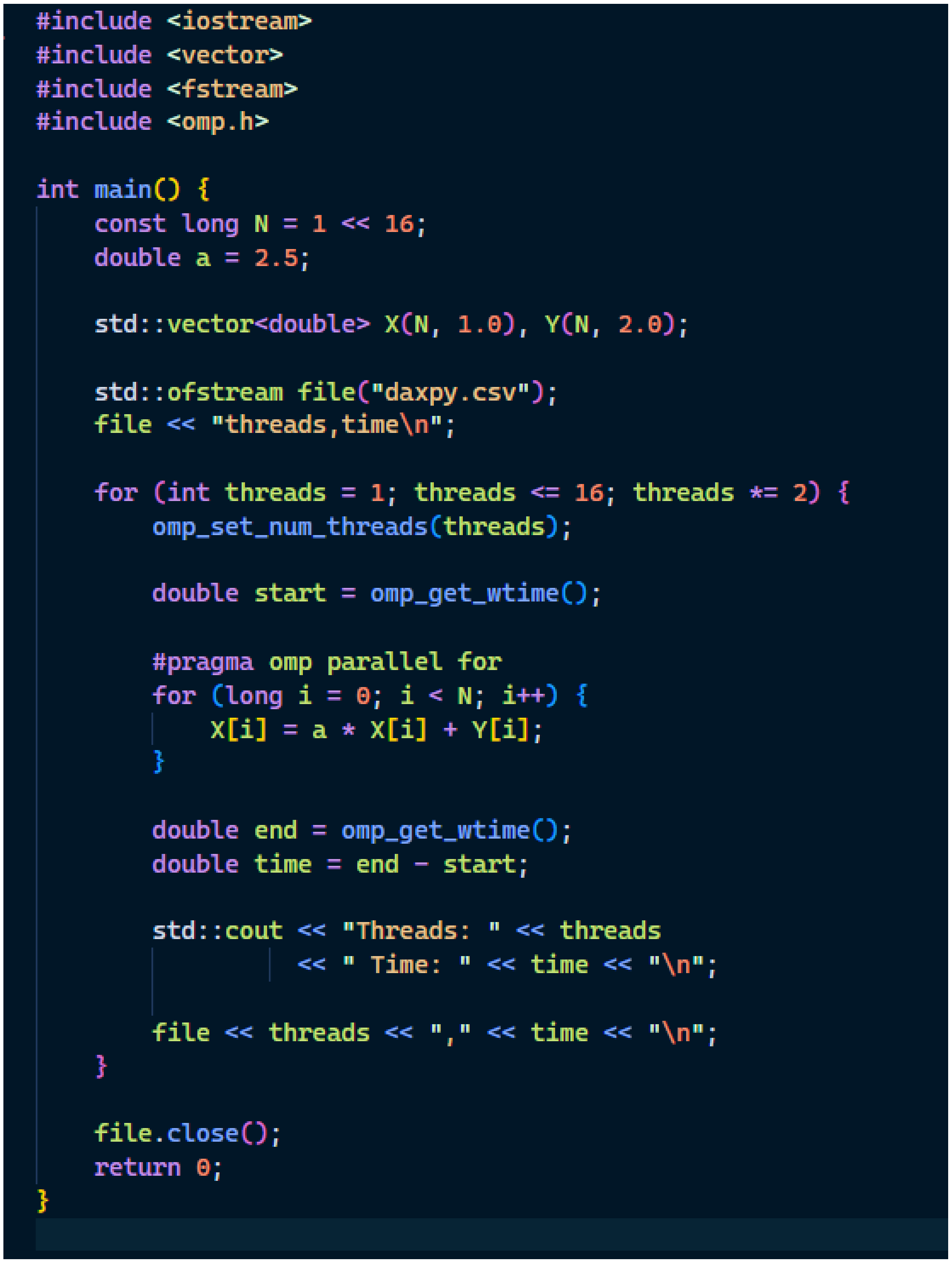
Question 1: DAXPY Loop

Problem Statement:

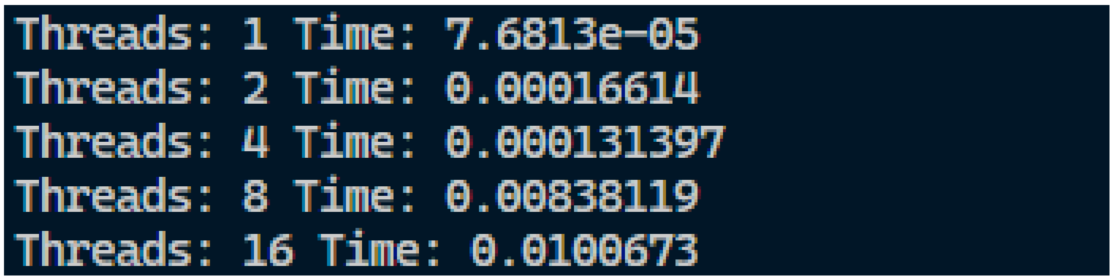
DAXPY Loop: D stands for Double precision, A is a scalar value, X and Y are one-dimensional vectors of size 2^{16} each, and P stands for Plus. The operation performed in one iteration is:

$$X[i] = a \times X[i] + Y[i]$$

The objective is to compare the speedup (in execution time) gained by increasing the number of threads starting from a 2-thread implementation.



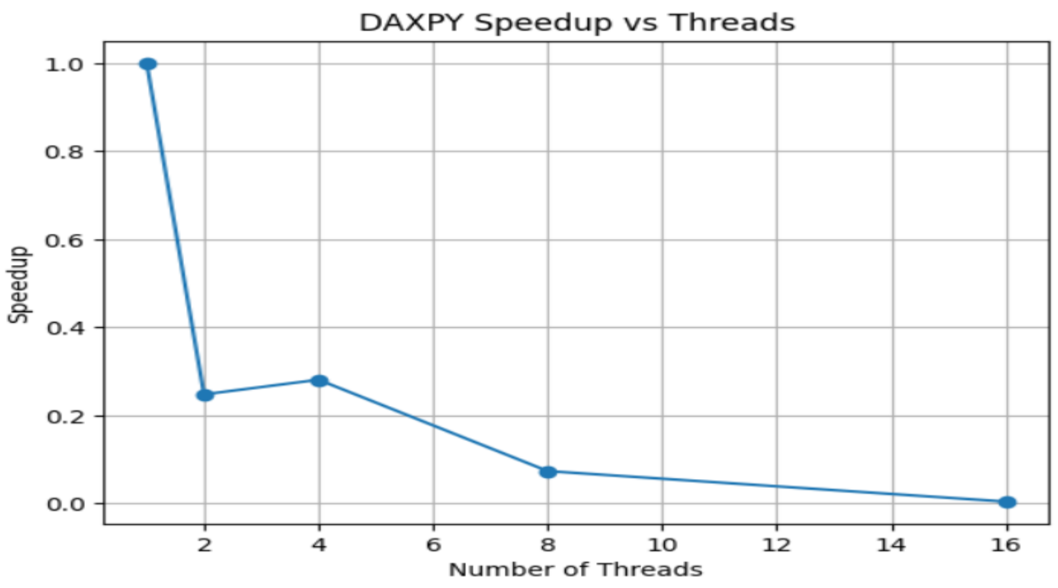
```cpp
#include <iostream>
#include <vector>
#include <fstream>
#include <omp.h>

int main() {
    const long N = 1 << 16;
    double a = 2.5;

    std::vector<double> X(N, 1.0), Y(N, 2.0);

    std::ofstream file("daxpy.csv");
    file << "threads,time\n";

    for (int threads = 1; threads <= 16; threads *= 2) {
        omp_set_num_threads(threads);

        double start = omp_get_wtime();

        #pragma omp parallel for
        for (long i = 0; i < N; i++) {
            X[i] = a * X[i] + Y[i];
        }

        double end = omp_get_wtime();
        double time = end - start;

        std::cout << "Threads: " << threads
                  << " Time: " << time << "\n";

        file << threads << "," << time << "\n";
    }

    file.close();
    return 0;
}
```



```
Threads: 1 Time: 7.6813e-05
Threads: 2 Time: 0.00016614
Threads: 4 Time: 0.000131397
Threads: 8 Time: 0.00838119
Threads: 16 Time: 0.0100673
```

Observation

From the obtained results and the speedup graph, it is observed that the performance improves when the number of threads is increased from 1 to 4. The maximum speedup is achieved around 3–4 threads, after which the performance starts to degrade as the number of threads is increased further. When using 8 and 16 threads, the execution time increases significantly and the speedup drops sharply. This behavior indicates that the DAXPY operation is highly memory-bound, and increasing the number of threads beyond a certain point leads to excessive memory access contention, thread management overhead, and scheduling costs. As a result, additional threads do not contribute to useful computation and instead reduce overall performance.



Conclusion

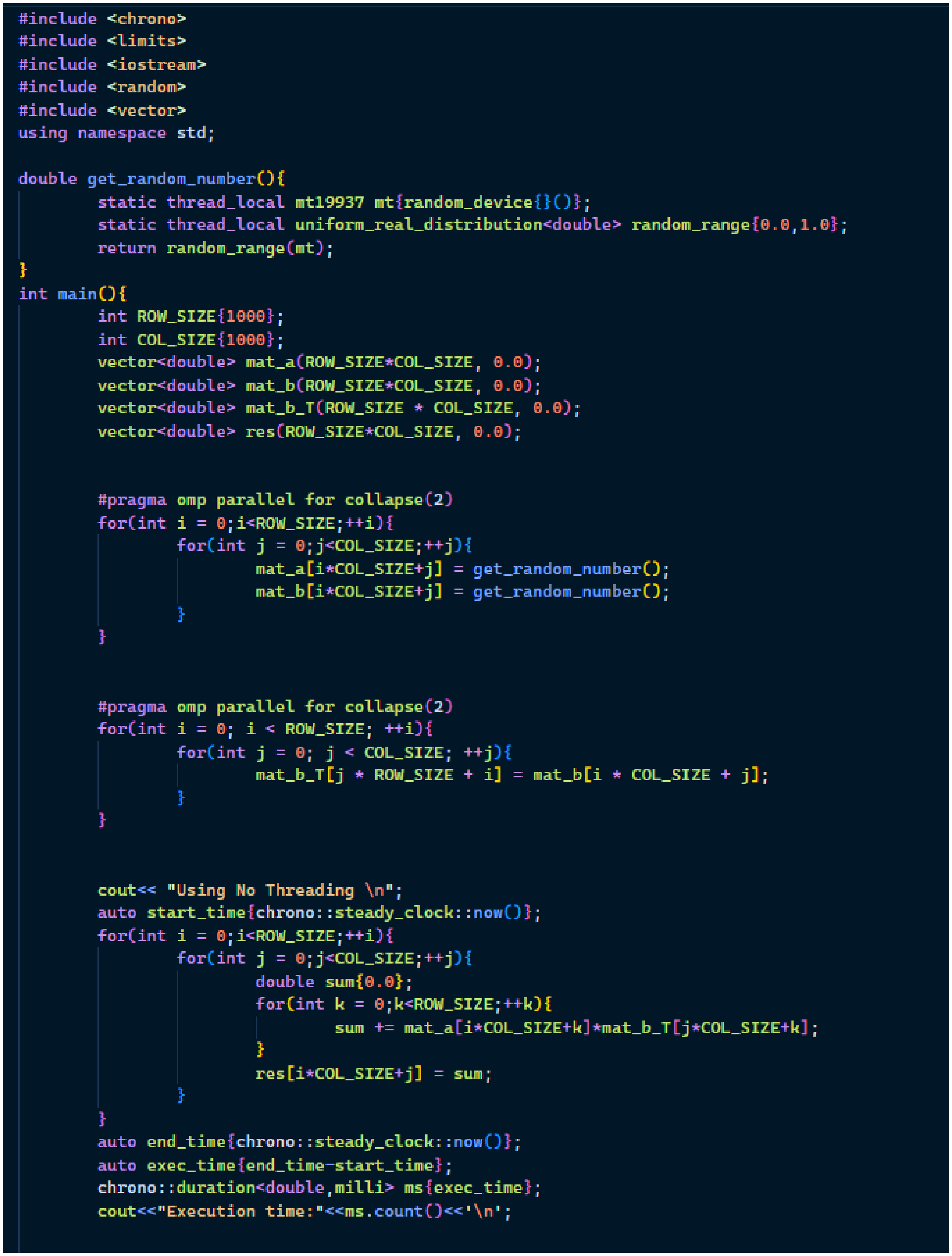
The DAXPY parallel implementation demonstrates limited scalability due to its dependence on memory bandwidth rather than computational intensity. While parallelization provides performance benefits up to a small number of threads, the speedup saturates and eventually decreases once the available memory bandwidth is fully utilized. Beyond the optimal thread count, the overhead of thread creation, synchronization, and cache coherence outweighs the benefits of parallel execution. Therefore, increasing the number of threads does not always guarantee better performance, and an optimal thread count—typically close to the number of physical CPU cores—must be chosen for efficient execution.

Question 2: Matrix Multiplication

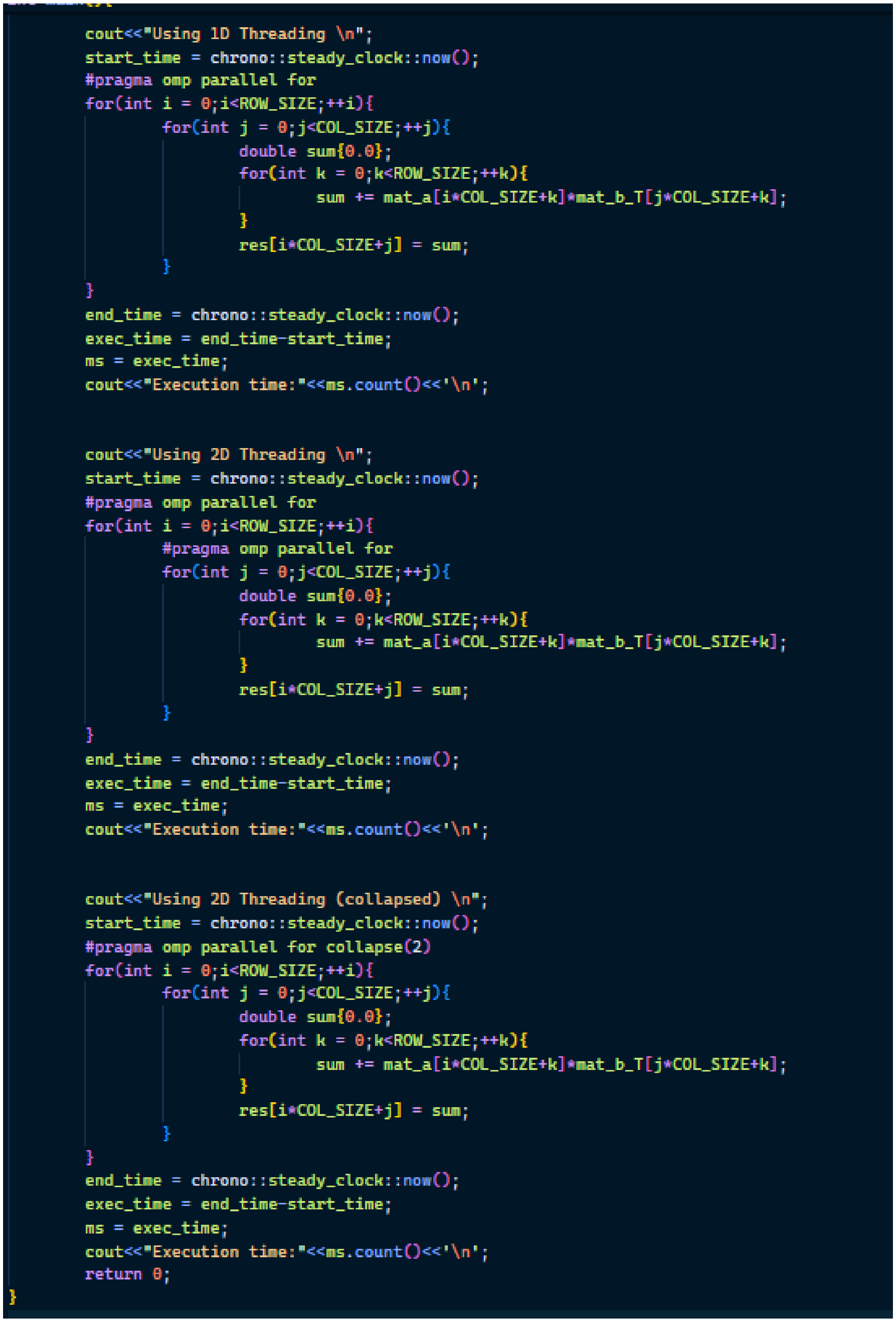
Problem Statement:

Build a parallel implementation of multiplication of large matrices (e.g., 1000×1000). Repeat the experiment from Question 1 and analyze work partitioning among threads using:

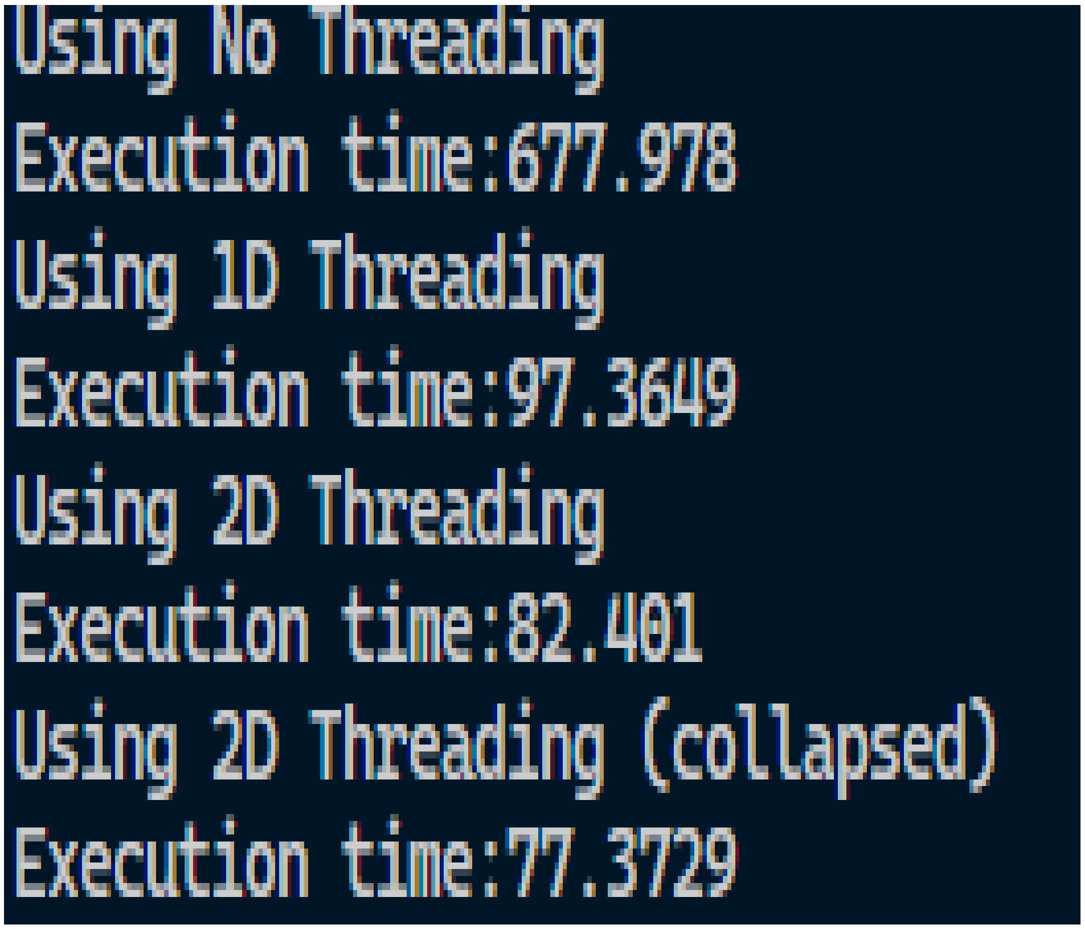
- 1D threading
- 2D threading



```cpp
#include <chrono>
#include <limits>
#include <iostream>
#include <random>
#include <vector>
using namespace std;

double get_random_number(){
        static thread_local mt19937 mt{random_device{}()};
        static thread_local uniform_real_distribution<double> random_range{0.0,1.0};
        return random_range(mt);
}
int main(){
        int ROW_SIZE{1000};
        int COL_SIZE{1000};
        vector<double> mat_a(ROW_SIZE*COL_SIZE, 0.0);
        vector<double> mat_b(ROW_SIZE*COL_SIZE, 0.0);
        vector<double> mat_b_T(ROW_SIZE * COL_SIZE, 0.0);
        vector<double> res(ROW_SIZE*COL_SIZE, 0.0);


        #pragma omp parallel for collapse(2)
        for(int i = 0;i<ROW_SIZE;++i){
                for(int j = 0;j<COL_SIZE;++j){
                        mat_a[i*COL_SIZE+j] = get_random_number();
                        mat_b[i*COL_SIZE+j] = get_random_number();
                }
        }


        #pragma omp parallel for collapse(2)
        for(int i = 0; i < ROW_SIZE; ++i){
                for(int j = 0; j < COL_SIZE; ++j){
                        mat_b_T[j * ROW_SIZE + i] = mat_b[i * COL_SIZE + j];
                }
        }


        cout<< "Using No Threading \n";
        auto start_time{chrono::steady_clock::now()};
        for(int i = 0;i<ROW_SIZE;++i){
                for(int j = 0;j<COL_SIZE;++j){
                        double sum{0.0};
                        for(int k = 0;k<ROW_SIZE;++k){
                                sum += mat_a[i*COL_SIZE+k]*mat_b_T[j*COL_SIZE+k];
                        }
                        res[i*COL_SIZE+j] = sum;
                }
        }
        auto end_time{chrono::steady_clock::now()};
        auto exec_time{end_time-start_time};
        chrono::duration<double,milli> ms{exec_time};
        cout<<"Execution time:"<<ms.count()<<'\n';
```



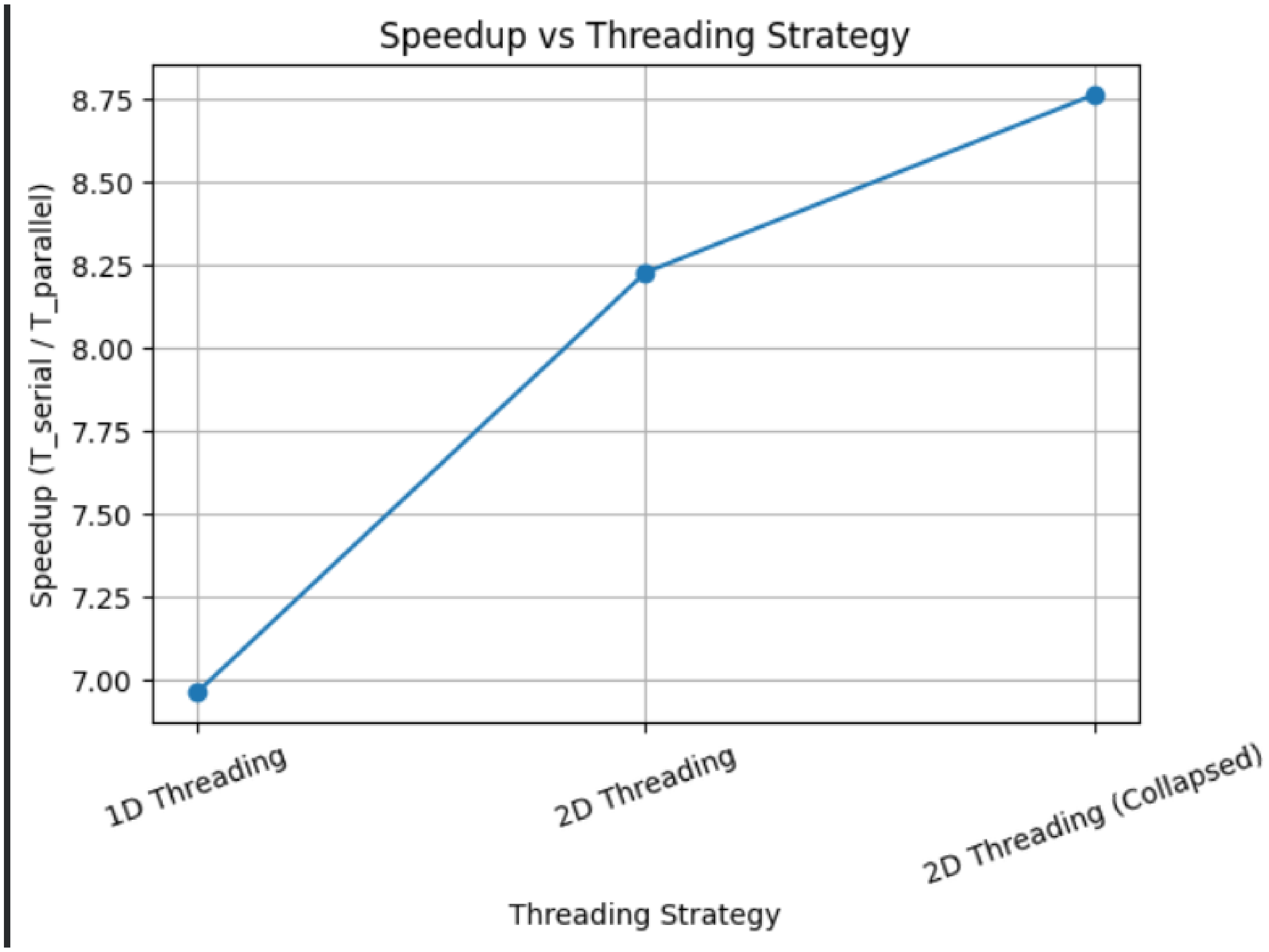
```cpp
    cout<<"Using 1D Threading \n";
    start_time = chrono::steady_clock::now();
    #pragma omp parallel for
    for(int i = 0;i<ROW_SIZE;++i){
        for(int j = 0;j<COL_SIZE;++j){
            double sum{0.0};
            for(int k = 0;k<ROW_SIZE;++k){
                sum += mat_a[i*COL_SIZE+k]*mat_b_T[j*COL_SIZE+k];
            }
            res[i*COL_SIZE+j] = sum;
        }
    }
    end_time = chrono::steady_clock::now();
    exec_time = end_time-start_time;
    ms = exec_time;
    cout<<"Execution time:"<<ms.count()<<'\n';


    cout<<"Using 2D Threading \n";
    start_time = chrono::steady_clock::now();
    #pragma omp parallel for
    for(int i = 0;i<ROW_SIZE;++i){
        #pragma omp parallel for
        for(int j = 0;j<COL_SIZE;++j){
            double sum{0.0};
            for(int k = 0;k<ROW_SIZE;++k){
                sum += mat_a[i*COL_SIZE+k]*mat_b_T[j*COL_SIZE+k];
            }
            res[i*COL_SIZE+j] = sum;
        }
    }
    end_time = chrono::steady_clock::now();
    exec_time = end_time-start_time;
    ms = exec_time;
    cout<<"Execution time:"<<ms.count()<<'\n';


    cout<<"Using 2D Threading (collapsed) \n";
    start_time = chrono::steady_clock::now();
    #pragma omp parallel for collapse(2)
    for(int i = 0;i<ROW_SIZE;++i){
        for(int j = 0;j<COL_SIZE;++j){
            double sum{0.0};
            for(int k = 0;k<ROW_SIZE;++k){
                sum += mat_a[i*COL_SIZE+k]*mat_b_T[j*COL_SIZE+k];
            }
            res[i*COL_SIZE+j] = sum;
        }
    }
    end_time = chrono::steady_clock::now();
    exec_time = end_time-start_time;
    ms = exec_time;
    cout<<"Execution time:"<<ms.count()<<'\n';
    return 0;
}
```



```
Using No Threading
Execution time:677.978
Using 1D Threading
Execution time:97.3649
Using 2D Threading
Execution time:82.401
Using 2D Threading (collapsed)
Execution time:77.3729
```

Observation

The execution time reduces significantly when moving from the serial implementation to parallel implementations. One-dimensional threading provides a substantial speedup over the no-threading case, while two-dimensional threading further improves performance by distributing the workload more evenly among threads. The best performance is achieved using collapsed 2D threading, which minimizes load imbalance and thread idle time, resulting in the highest speedup among all strategies.



Conclusion

Matrix multiplication benefits greatly from parallelization due to its compute-intensive nature. Compared to 1D threading, 2D threading—especially with loop collapsing—utilizes CPU cores more efficiently and reduces scheduling overhead. This experiment demonstrates that selecting an appropriate parallelization strategy is crucial for maximizing performance, and finer-grained work distribution leads to better scalability.

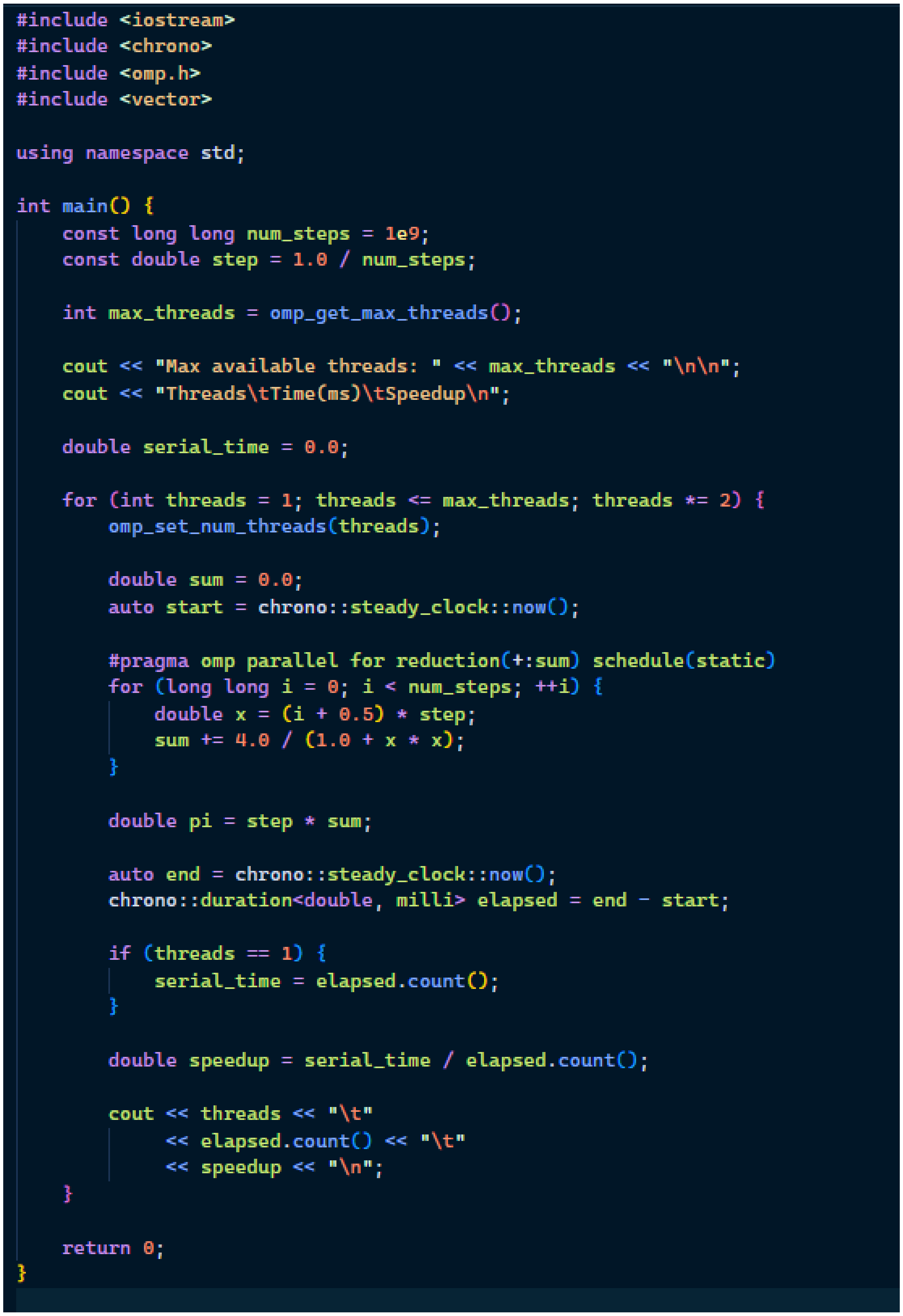
Question 3: Calculation of π

Problem Statement:

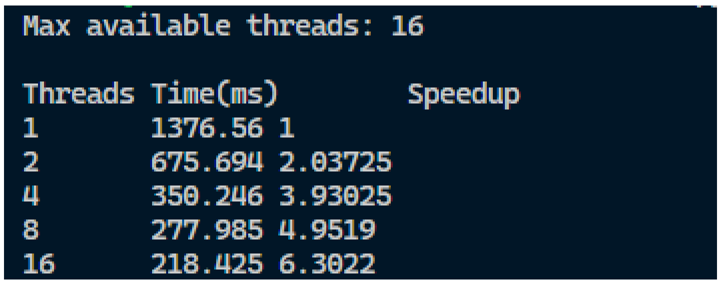
The value of π is computed using numerical integration:

$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx$$

The integral is approximated using the rectangle method and parallelized using OpenMP reduction.



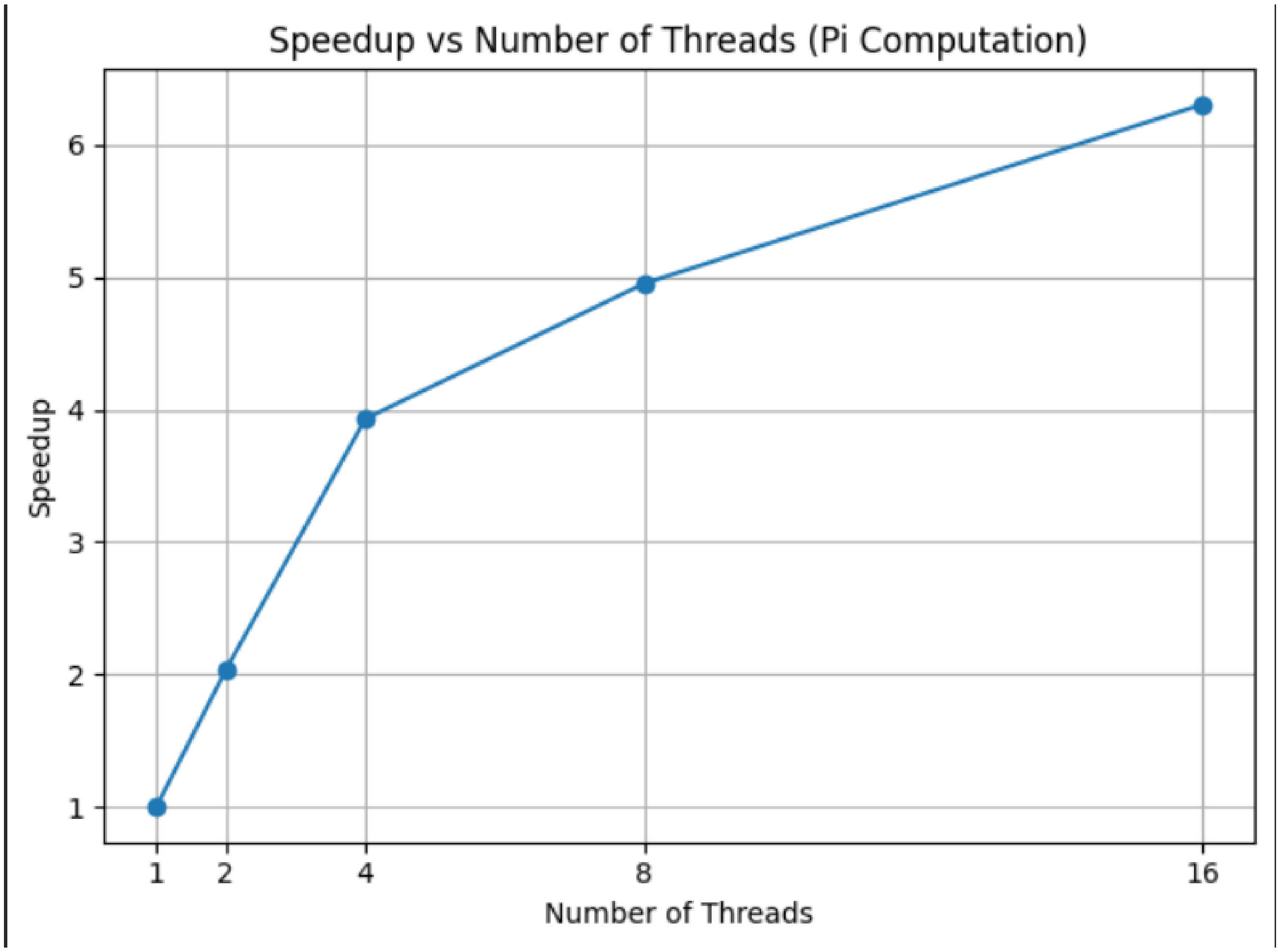
```cpp
#include <iostream>
#include <chrono>
#include <omp.h>
#include <vector>

using namespace std;

int main() {
    const long long num_steps = 1e9;
    const double step = 1.0 / num_steps;

    int max_threads = omp_get_max_threads();

    cout << "Max available threads: " << max_threads << "\n\n";
    cout << "Threads\tTime(ms)\tSpeedup\n";

    double serial_time = 0.0;

    for (int threads = 1; threads <= max_threads; threads *= 2) {
        omp_set_num_threads(threads);

        double sum = 0.0;
        auto start = chrono::steady_clock::now();

        #pragma omp parallel for reduction(+:sum) schedule(static)
        for (long long i = 0; i < num_steps; ++i) {
            double x = (i + 0.5) * step;
            sum += 4.0 / (1.0 + x * x);
        }

        double pi = step * sum;

        auto end = chrono::steady_clock::now();
        chrono::duration<double, milli> elapsed = end - start;

        if (threads == 1) {
            serial_time = elapsed.count();
        }

        double speedup = serial_time / elapsed.count();

        cout << threads << "\t"
             << elapsed.count() << "\t"
             << speedup << "\n";
    }

    return 0;
}
```



```
Max available threads: 16

Threads Time(ms)        Speedup
1       1376.56 1
2       675.694 2.03725
4       350.246 3.93025
8       277.985 4.9519
16      218.425 6.3022
```

Observation

The execution time decreases steadily as the number of threads increases, resulting in a near-linear improvement in speedup up to higher thread counts. Significant performance gains are observed when increasing threads from 1 to 4, and the speedup continues to improve up to 16 threads, although at a slower rate. This gradual reduction in performance gain indicates the presence of parallel overhead and reduction synchronization costs, which limit perfect linear scalability.



Conclusion

The parallel computation of π scales efficiently with increasing number of threads due to the independent nature of iterations and the use of OpenMP reduction. While ideal linear speedup is not achieved, the implementation effectively utilizes available CPU resources, demonstrating that data-parallel workloads with minimal dependencies benefit greatly from multithreading until synchronization overhead becomes dominant.

Submitted to : Dr Saif Nalband

Submitted by : Aunish Kr Yadav

Roll No. : 102303306