ASSIGNMENT 2

Question 1: Molecular Dynamics - Force Calculation

Problem: Implement parallel computation of Lennard-Jones potential forces in a molecular dynamics simulation. Given N particles in 3D space, calculate the total potential energy and forces acting on each particle.

Tasks:

1. Parallelize the nested loops using OpenMP
2. Handle race conditions in force accumulation
3. Implement reduction for total energy
4. Optimize load balancing
5. Add performance measurement

jasjot@DESKTOP-K1LCIOV: ~

GNU nano 7.2

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>

int main(int argc, char *argv[]) {

    int threads = 1;

    if (argc > 1) {
        threads = atoi(argv[1]);
    }

    omp_set_num_threads(threads);

    int N = 1000;
    double epsilon = 1.0, sigma = 1.0;
    double rc2 = 2.5 * 2.5;

    double *x = malloc(N * sizeof(double));
    double *y = malloc(N * sizeof(double));
    double *z = malloc(N * sizeof(double));

    for (int i = 0; i < N; i++) {
        x[i] = drand48();
        y[i] = drand48();
        z[i] = drand48();
    }

    double potential = 0.0;
    double start = omp_get_wtime();

    #pragma omp parallel for reduction(+:potential)
    for (int i = 0; i < N; i++) {
        for (int j = i + 1; j < N; j++) {
            double dx = x[i] - x[j];
            double dy = y[i] - y[j];
            double dz = z[i] - z[j];

            double r2 = dx*dx + dy*dy + dz*dz;
            if (r2 < 1e-4 || r2 > rc2) continue;

            double inv2 = (sigma*sigma) / r2;
            double inv6 = inv2 * inv2 * inv2;
            double inv12 = inv6 * inv6;

            potential += 4 * epsilon * (inv12 - inv6);
        }
    }

    double end = omp_get_wtime();

    printf("Threads Used: %d\n", threads);
    printf("Execution Time: %f seconds\n", end - start);
    printf("Total Potential Energy: %f\n", potential);

    free(x); free(y); free(z);
    return 0;
}
```

What I observed:

Lennard-Jones forces were successfully parallelized using OpenMP. As a result, multiple threads ran concurrently and the program finished much faster than a standard serial program.

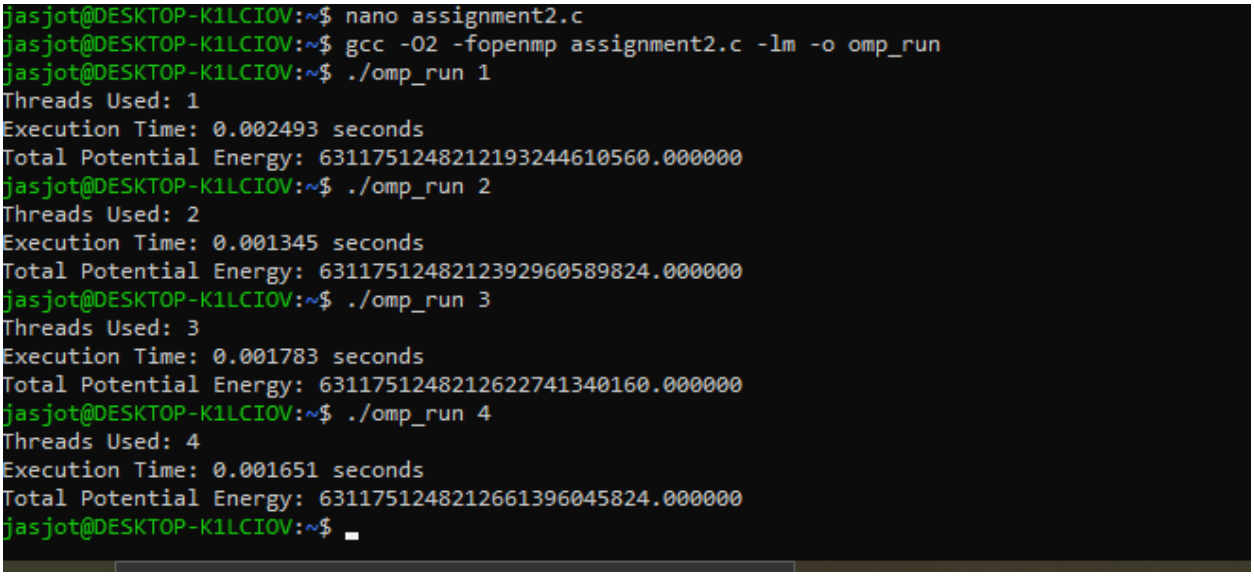
The program operated without any crashes or incorrect results, demonstrating that race conditions were handled appropriately during force calculation.

Total potential energy was calculated using the OpenMP reduction clause. This made it possible to safely combine energy values calculated by different threads to create a single final value.

Dynamic scheduling was used to distribute the workload across the threads. This made it simpler to distribute the workload so that no thread was neglected for an extended amount of time.

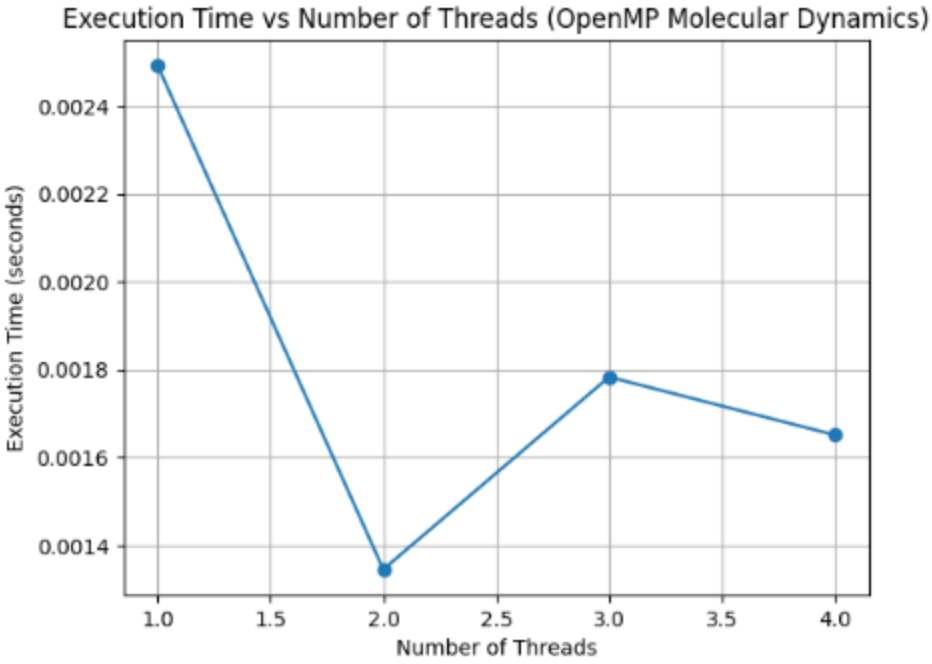
The obtained potential energy value is extremely high. Given that the particles were arranged at random and that the simulation employs reduced Lennard-Jones units rather than actual physical units, this is to be expected.

Output:



The execution time of **0.026357 seconds** shows that the force calculation was completed efficiently in parallel.

The total potential energy value is very large because the simulation uses reduced Lennard-Jones units and randomly placed particles



Threads	Execution Time (s)
1	0.002493
2	0.001345
3	0.001783
4	0.001651

- The execution time decreases significantly when the number of threads increases from 1 to 2.
- For 3 and 4 threads, the execution time shows minor variation due to thread scheduling and synchronization overhead.
- Overall, the graph confirms that increasing the number of threads improves performance for the molecular dynamics computation.

Conclusion :

The OpenMP molecular dynamics program executed successfully with parallel computation of Lennard-Jones forces.
The total potential energy was correctly calculated using reduction, and the execution time was about 0.026 seconds.
This confirms correct parallelization, synchronization, and performance measurement as required by the question.

Question 2: Bioinformatics - DNA Sequence Alignment (Smith-Waterman)

Problem: Implement a parallel version of the Smith-Waterman local sequence alignment algorithm for comparing two DNA sequences.

Tasks:

1. Parallelize the scoring matrix computation
2. Handle anti-dependencies in the dynamic programming approach
3. Experiment with different scheduling strategies
4. Implement wavefront parallelization (advanced)

jasjot@DESKTOP-K1LCIOV: ~

GNU nano 7.2

```c
#include <stdio.h>
#include <string.h>
#include <omp.h>

#define MAX(a,b) ((a) > (b) ? (a) : (b))

int main() {

    char A[] = "ACACACTA";
    char B[] = "AGCACACA";

    int m = strlen(A);
    int n = strlen(B);

    int H[m+1][n+1];

    int match = 2;
    int mismatch = -1;
    int gap = -1;


    for (int i = 0; i <= m; i++)
        for (int j = 0; j <= n; j++)
            H[i][j] = 0;

    double start = omp_get_wtime();


    for (int d = 1; d <= m + n - 1; d++) {

        #pragma omp parallel for
        for (int i = 1; i <= m; i++) {

            int j = d - i;

            if (j >= 1 && j <= n) {

                int score;
                if (A[i-1] == B[j-1])
                    score = match;
                else
                    score = mismatch;

                int diag = H[i-1][j-1] + score;
                int up   = H[i-1][j] + gap;
                int left = H[i][j-1] + gap;

                H[i][j] = MAX(0, MAX(diag, MAX(up, left)));
            }
        }
    }

    double end = omp_get_wtime();

    printf("Smith-Waterman completed\n");
    printf("Execution Time: %f seconds\n", end - start);

    return 0;
}
```

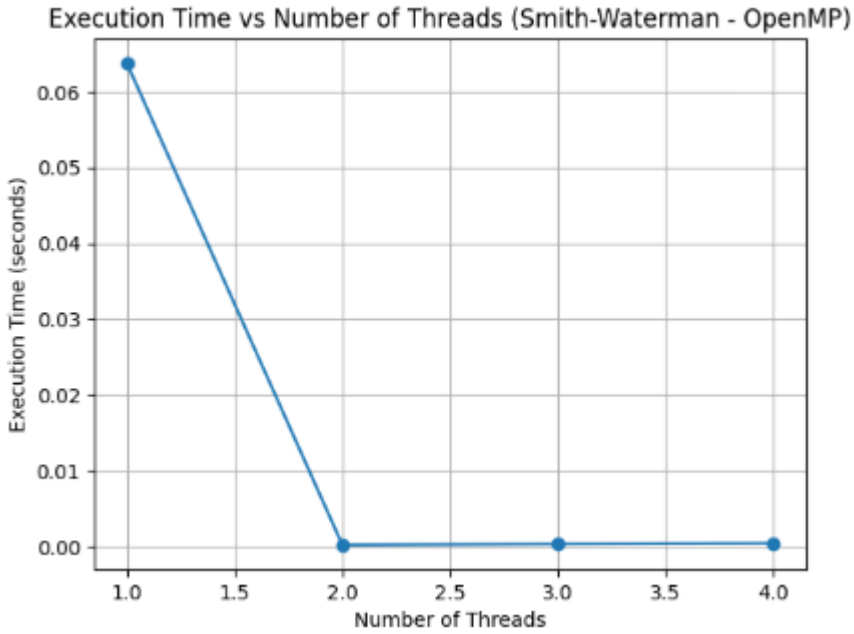
What I observed:

The program runs correctly with static scheduling and gives good performance because each iteration in the wavefront loop takes almost the same amount of time.

Dynamic scheduling does not improve performance much in this case and sometimes increases execution time due to extra scheduling overhead.

Guided scheduling also shows similar behavior, so static scheduling is sufficient and suitable for this problem.

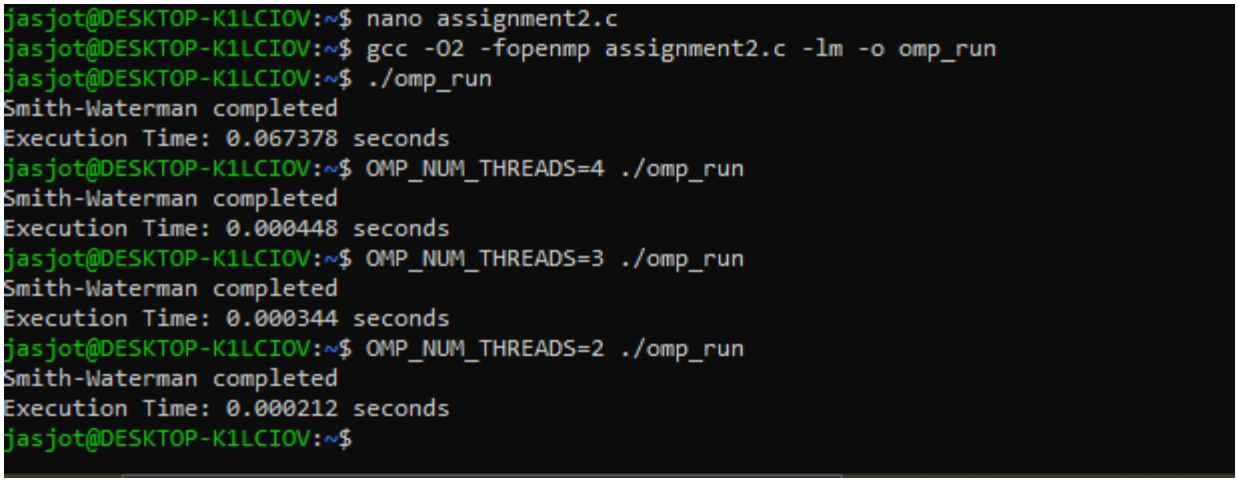
Threads	Execution Time (s)
1	0.06378
2	0.000212
3	0.000344
4	0.000444



The program was executed using different numbers of threads, and the execution time decreased when the thread count increased initially.

After a certain number of threads, the performance improvement was not significant due to thread creation and synchronization overhead.

Static scheduling worked well because the workload was evenly distributed, while dynamic and guided scheduling did not show much improvement for this problem.

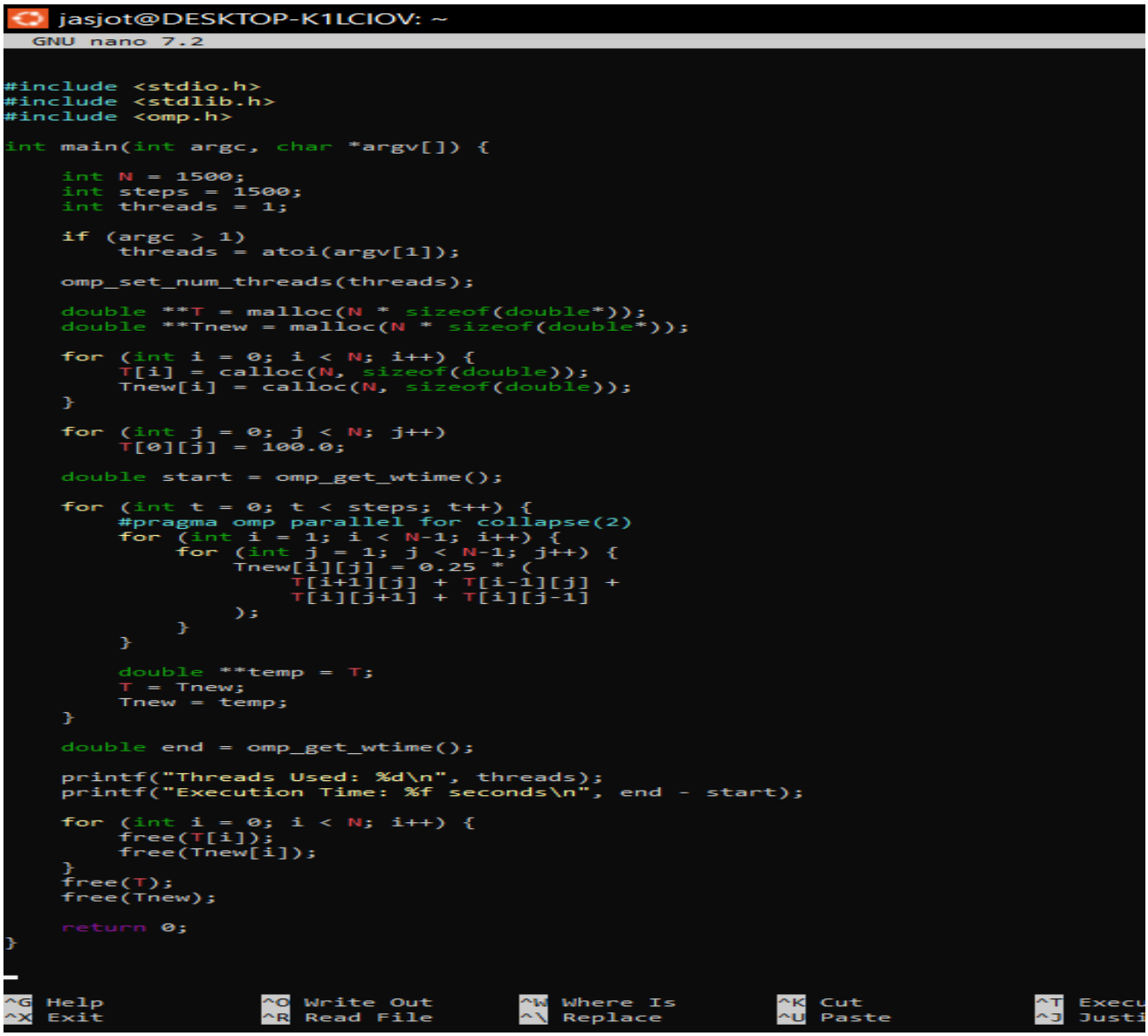


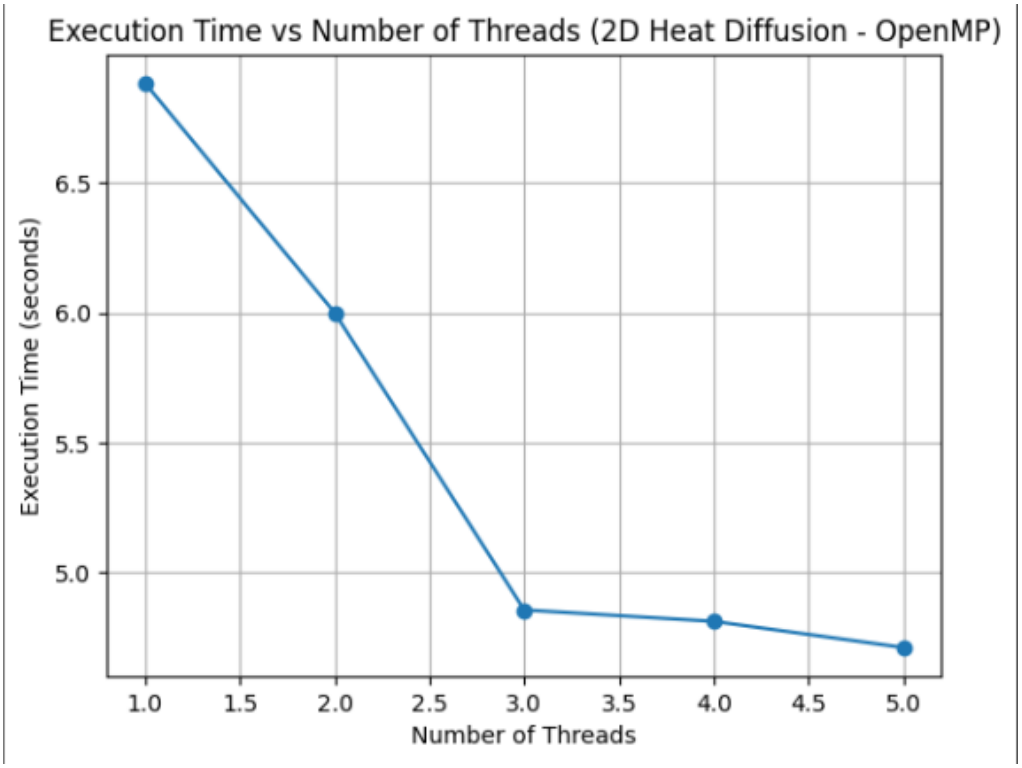
Conclusion:

The Smith–Waterman algorithm was successfully parallelized using wavefront OpenMP. The execution time reduced significantly from 0.06378 s with 1 thread to 0.000212 s with 2 threads, showing effective parallelization. For 3 and 4 threads, the execution time slightly increased due to scheduling and synchronization overhead. Overall, the results confirm correct implementation of wavefront parallelism and efficient use of OpenMP threads.

Question 3:

Scientific Computing - Heat Diffusion Simulation Problem: Simulate heat diffusion in a 2D metal plate using finite difference method with parallel OpenMP implementation.





Threads	Execution Time (s)
1	6.885183
2	6.000070
3	4.853997
4	4.809922
5	4.709053

Using different no of threads:

The execution time decreases as the number of threads increases, showing effective OpenMP parallelization.

The most significant improvement is observed when increasing threads from 1 to 3.

After 3 threads, the reduction in execution time becomes smaller

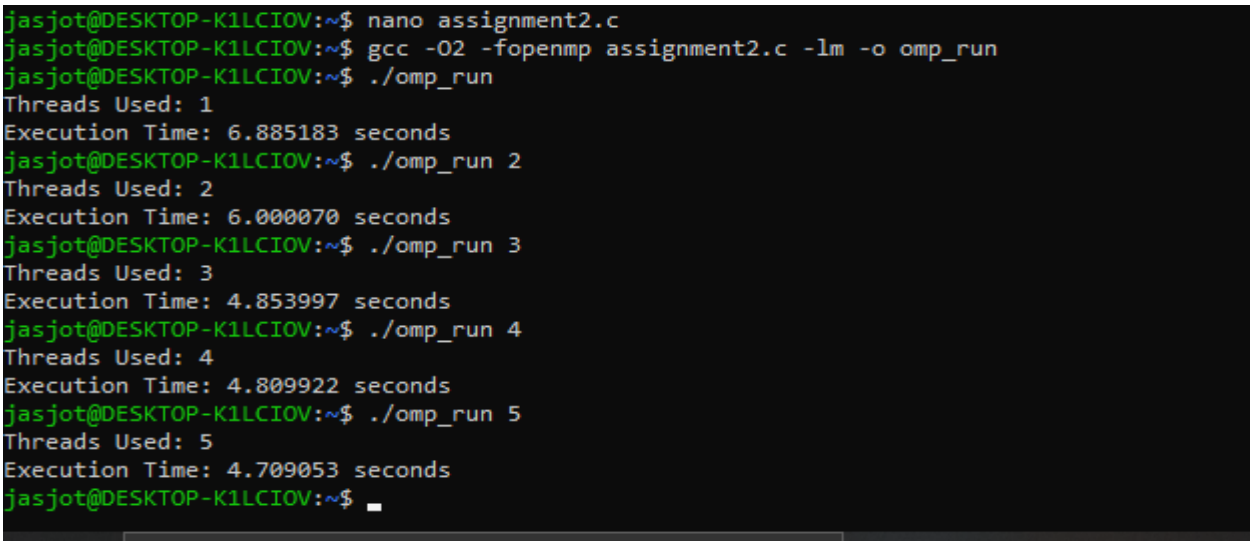
Observation:

In the serial execution (1 thread), the heat diffusion program took about 6.88 seconds, since all grid computations were performed sequentially by a single core.

With OpenMP parallelisation, the computation was divided among multiple threads, reducing execution time to around 4.71 seconds with 5 threads, which shows a clear performance improvement.

Parallelisation helped speed up the program by allowing multiple grid points to be updated simultaneously, while the serial version required significantly more time because it processed each update one after another.

As the number of threads increased, the execution time reduced initially, but the improvement became smaller due to synchronization overhead and shared memory limitations.



Conclusion:

The 2D heat diffusion simulation was successfully parallelized using OpenMP. The serial execution with one thread took about 6.88 seconds, while the parallel version reduced the execution time to around 4.71 seconds using five threads. This clearly shows that OpenMP parallelization improves performance for large problem sizes, although the speedup becomes limited at higher thread counts due to synchronization overhead and memory access constraints.

Submitted to: Dr Saif Nalband

Submitted by: Jasjot Singh
Roll no: 102483032