Nithin S
221IT085

IT301 Lab Assignment 2

Note: While calculating the time taken to run the program, I have taken the timezone parameter as NULL, as the timezone doesn’t matter when I subtract the end and start time.

Q1. Implement the parallel version of calculation of Pi (π) using #pragma omp parallel. The formula for calculation of Pi (π) is

$$\int_0^1 \frac{4.0}{(1+x2)} dx = \pi$$

Run the parallel code and take the execution time with 1, 2, 4, 8, 16, 32 threads. Record the timing.

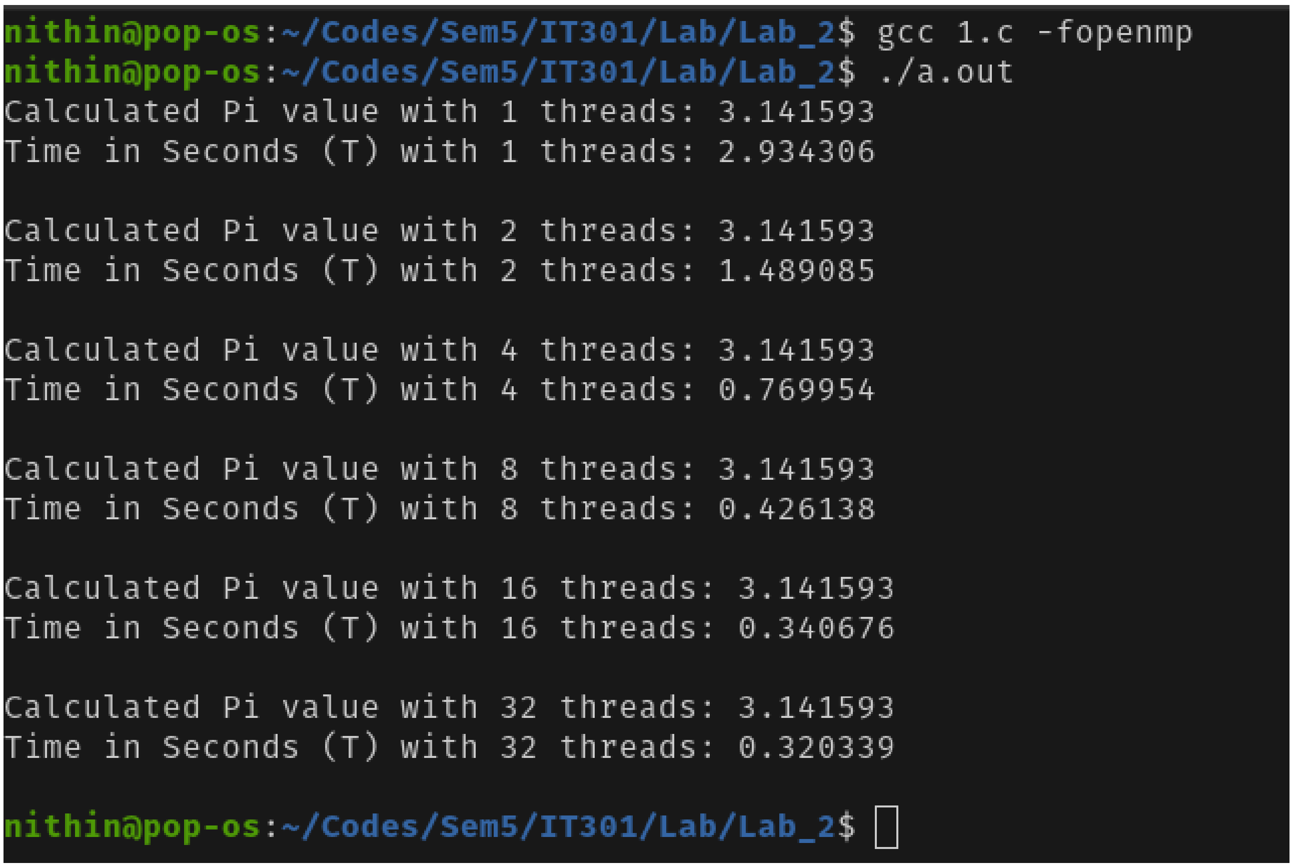
Code :

```
#include <stdio.h>
#include <omp.h>
#include <sys/time.h>
#include <stdlib.h>

int main() {

   struct timeval TimeValue_Start;
   struct timeval TimeValue_Final;
   long time_start, time_end;
   double time_overhead;


   long long num_steps = 1000000000;
   double step = 1.0 / (double)num_steps;
   double pi;


   int thread_counts[] = {1, 2, 4, 8,16,32};
```

```
    int num_cases = sizeof(thread_counts) / sizeof(thread_counts[0]);

    for (int t = 0; t < num_cases; t++) {
        int num_threads = thread_counts[t];
        omp_set_num_threads(num_threads);

        double sum = 0.0;


        gettimeofday(&TimeValue_Start, NULL);


        #pragma omp parallel
        {
            double local_sum = 0.0;
            #pragma omp for
            for (long long i = 0; i < num_steps; i++) {
                double x = (i + 0.5) * step;
                local_sum += 4.0 / (1.0 + x * x);
            }
            #pragma omp atomic
            sum += local_sum;
        }


        gettimeofday(&TimeValue_Final, NULL);


        pi = step * sum;
        printf("Calculated Pi value with %d threads: %lf\n", num_threads,
pi);


        time_start = TimeValue_Start.tv_sec * 1000000 +
TimeValue_Start.tv_usec;
        time_end = TimeValue_Final.tv_sec * 1000000 +
TimeValue_Final.tv_usec;
        time_overhead = (time_end - time_start) / 1000000.0;


        printf("Time in Seconds (T) with %d threads: %lf\n\n",
num_threads, time_overhead);
```

```
    }

    return 0;
}
```

Output:



```
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_2$ gcc 1.c -fopenmp
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_2$ ./a.out
Calculated Pi value with 1 threads: 3.141593
Time in Seconds (T) with 1 threads: 2.934306

Calculated Pi value with 2 threads: 3.141593
Time in Seconds (T) with 2 threads: 1.489085

Calculated Pi value with 4 threads: 3.141593
Time in Seconds (T) with 4 threads: 0.769954

Calculated Pi value with 8 threads: 3.141593
Time in Seconds (T) with 8 threads: 0.426138

Calculated Pi value with 16 threads: 3.141593
Time in Seconds (T) with 16 threads: 0.340676

Calculated Pi value with 32 threads: 3.141593
Time in Seconds (T) with 32 threads: 0.320339

nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_2$ 
```

Analysis:

This openmp C code calculates the value of pi using numerical integration (specifically, the midpoint rule) and measures the execution time with different numbers of threads using OpenMP for parallel processing.

The code uses gettimeofday to record the start and end time of each calculation, determining the time taken to compute π with varying numbers of threads.

The loop over num_cases sets the number of threads for each iteration. Inside the parallel region (#pragma omp parallel), each thread calculates a portion of the sum that approximates π. The

#pragma omp atomic directive ensures that updates to the sum variable from multiple threads are performed atomically.

The code evaluates the performance by running the computation with different thread counts (1, 2, 4, 8, 16, 32) and prints both the computed value of π and the time taken for each case.

By comparing the execution time across different thread counts, the program effectively demonstrates how the computation scales with increasing parallelism.

Q2. Develop an OpenMp program for matrix multiplication (C=A*B). Analyze the speedup and efficiency of the parallelized code. Vary the size of your matrices from 250, 500, 750, 1000, and 2000 and measure the runtime with one thread. For each matrix size, change the number of threads from 2,4,8., and plot the speedup versus the number of threads. Compute the efficiency.

Code:

```
#include <stdio.h>
#include <omp.h>
#include <sys/time.h>
#include <stdlib.h>

void matrix_multiply(double **A, double **B, double **C, int size) {
  #pragma omp parallel for
  for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
      C[i][j] = 0;
      for (int k = 0; k < size; k++) {
        C[i][j] += A[i][k] * B[k][j];
      }
    }
  }
}

double **allocate_matrix(int size) {
  double **matrix = (double **)malloc(size * sizeof(double *));
  for (int i = 0; i < size; i++) {
    matrix[i] = (double *)malloc(size * sizeof(double));
  }
  return matrix;
}

void free_matrix(double **matrix, int size) {
  for (int i = 0; i < size; i++) {
    free(matrix[i]);
  }
  free(matrix);
}

int main() {
  struct timeval TimeValue_Start;
  struct timeval TimeValue_Final;
  long time_start, time_end;
  double sequential_time, parallel_time, speedup, efficiency;

  int sizes[] = {250, 500, 750, 1000, 2000};
  int num_threads_options[] = {1, 2, 4, 8};

  FILE *output_file = fopen("speedup_data.csv", "w");
  fprintf(output_file,
"MatrixSize,NumThreads,SequentialTime,ParallelTime,Speedup\n");

  for (int s = 0; s < 5; s++) {
    int size = sizes[s];
    printf("<-------------------------------------------------------------------->\n");
    printf("Matrix Size: %d\n", size);

    double **A = allocate_matrix(size);
    double **B = allocate_matrix(size);
    double **C = allocate_matrix(size);


    for (int i = 0; i < size; i++) {
      for (int j = 0; j < size; j++) {
        A[i][j] = rand() % 100;
        B[i][j] = rand() % 100;
      }
    }

    for (int t = 0; t < 4; t++) {
```

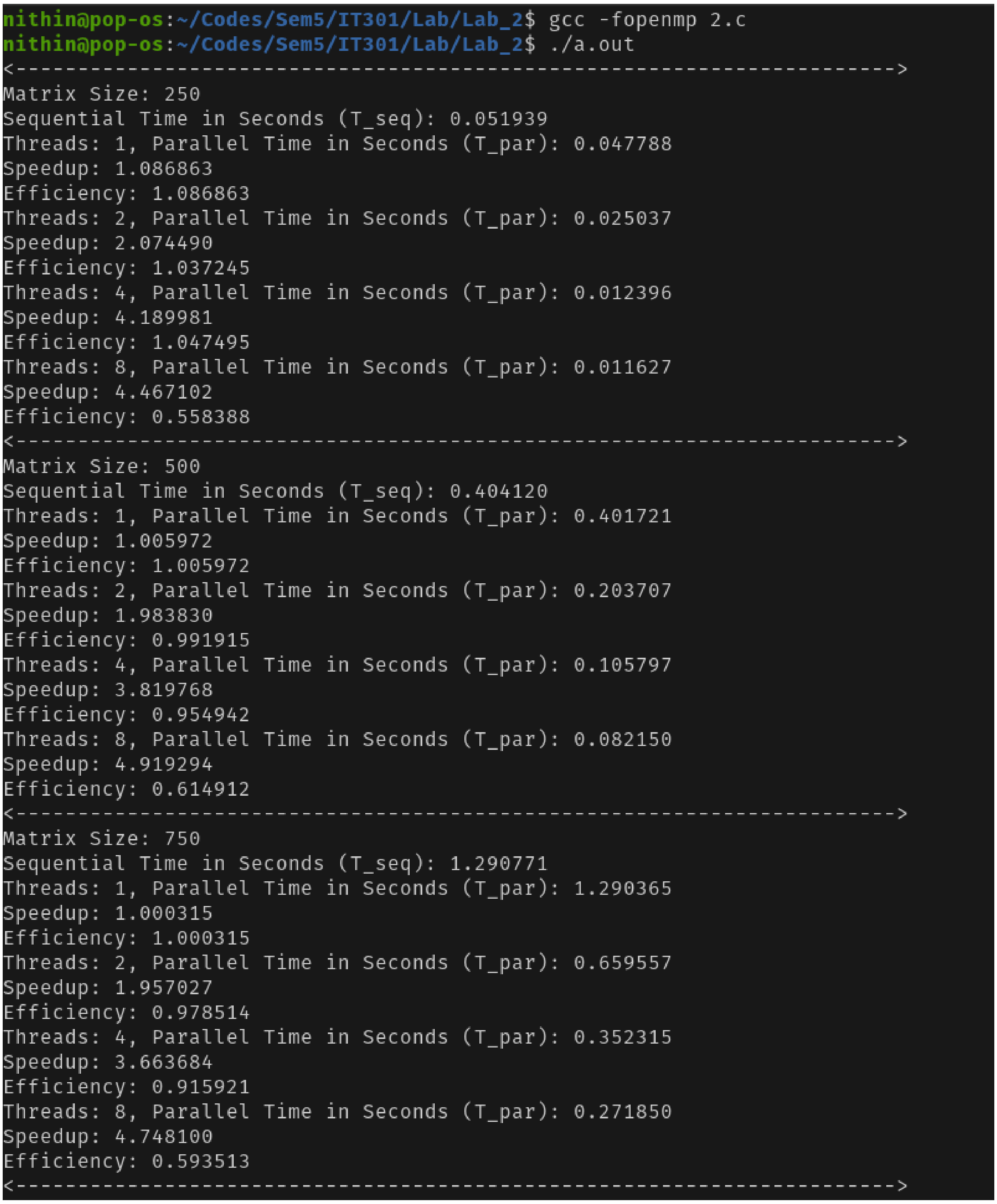
```
        int num_threads = num_threads_options[t];
        omp_set_num_threads(num_threads);


        if (num_threads == 1) {
            gettimeofday(&TimeValue_Start, NULL);
            matrix_multiply(A, B, C, size);
            gettimeofday(&TimeValue_Final, NULL);

            time_start = TimeValue_Start.tv_sec * 1000000 +
TimeValue_Start.tv_usec;
            time_end = TimeValue_Final.tv_sec * 1000000 +
TimeValue_Final.tv_usec;
            sequential_time = (time_end - time_start) / 1000000.0;

            printf("Sequential Time in Seconds (T_seq): %lf\n",
sequential_time);
        }


        gettimeofday(&TimeValue_Start, NULL);
        matrix_multiply(A, B, C, size);
        gettimeofday(&TimeValue_Final, NULL);

        time_start = TimeValue_Start.tv_sec * 1000000 +
TimeValue_Start.tv_usec;
        time_end = TimeValue_Final.tv_sec * 1000000 +
TimeValue_Final.tv_usec;
        parallel_time = (time_end - time_start) / 1000000.0;

        printf("Threads: %d, Parallel Time in Seconds (T_par): %lf\n",
num_threads, parallel_time);


        speedup = sequential_time / parallel_time;


        fprintf(output_file, "%d,%d,%lf,%lf,%lf\n", size, num_threads,
sequential_time, parallel_time, speedup);

        printf("Speedup: %lf\n", speedup);
    }
```

```
        free_matrix(A, size);
        free_matrix(B, size);
        free_matrix(C, size);
    }

    fclose(output_file);
    return 0;
}
```

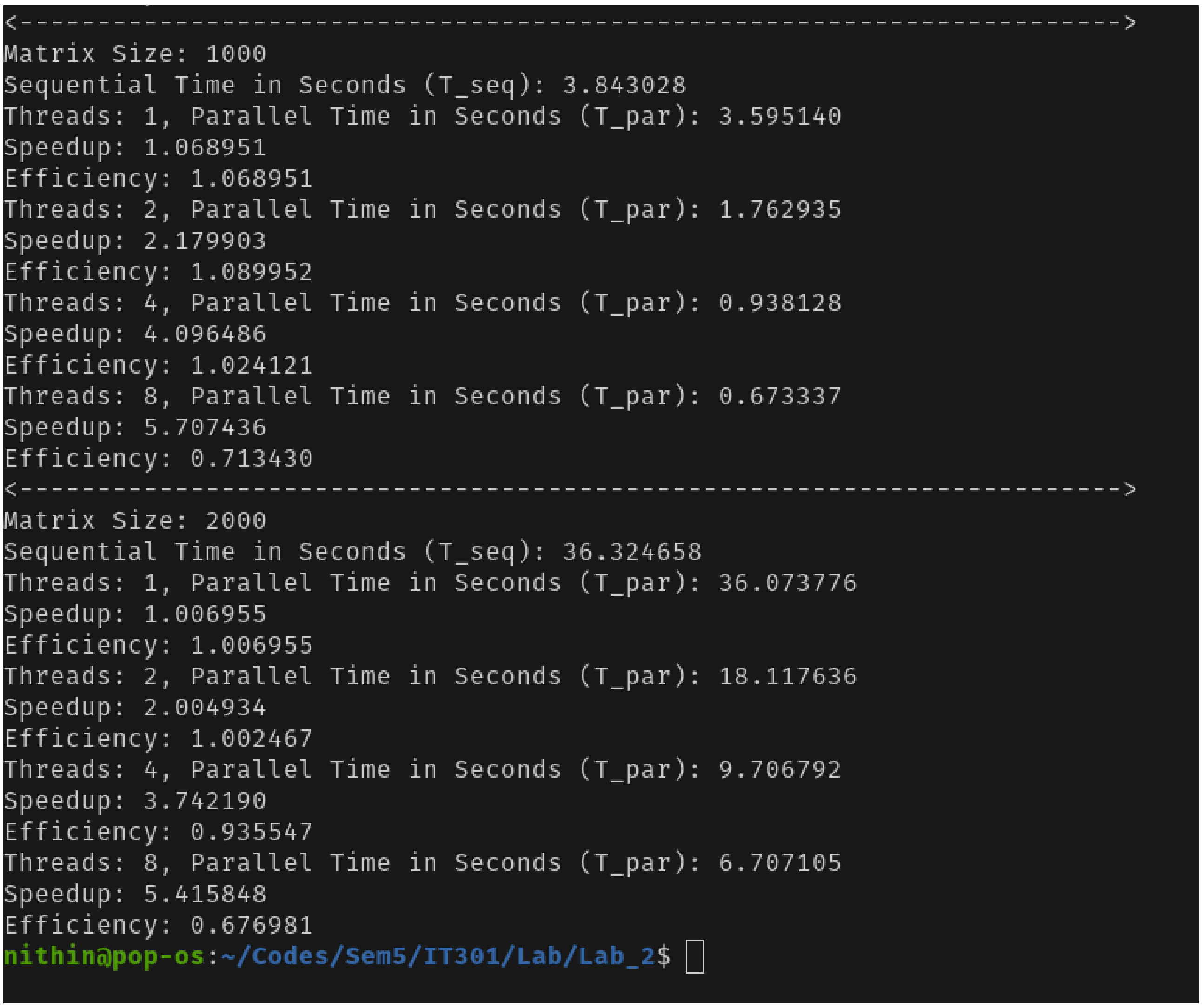
Python Code to plot the graph

```
import pandas as pd
import matplotlib.pyplot as plt


data = pd.read_csv('speedup_data.csv')


sizes = data['MatrixSize'].unique()
for size in sizes:
    subset = data[data['MatrixSize'] == size]
    plt.plot(subset['NumThreads'], subset['Speedup'], marker='o',
label=f'Size: {size}')


plt.title('Speedup vs. Number of Threads')
plt.xlabel('Number of Threads')
plt.ylabel('Speedup')
plt.legend()
plt.grid(True)


plt.savefig('speedup_vs_threads.png')
```

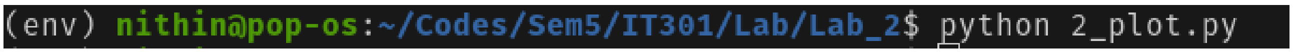
Output:



```
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_2$ gcc -fopenmp 2.c
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_2$ ./a.out
<------------------------------------------------------------------------>
Matrix Size: 250
Sequential Time in Seconds (T_seq): 0.051939
Threads: 1, Parallel Time in Seconds (T_par): 0.047788
Speedup: 1.086863
Efficiency: 1.086863
Threads: 2, Parallel Time in Seconds (T_par): 0.025037
Speedup: 2.074490
Efficiency: 1.037245
Threads: 4, Parallel Time in Seconds (T_par): 0.012396
Speedup: 4.189981
Efficiency: 1.047495
Threads: 8, Parallel Time in Seconds (T_par): 0.011627
Speedup: 4.467102
Efficiency: 0.558388
<------------------------------------------------------------------------>
Matrix Size: 500
Sequential Time in Seconds (T_seq): 0.404120
Threads: 1, Parallel Time in Seconds (T_par): 0.401721
Speedup: 1.005972
Efficiency: 1.005972
Threads: 2, Parallel Time in Seconds (T_par): 0.203707
Speedup: 1.983830
Efficiency: 0.991915
Threads: 4, Parallel Time in Seconds (T_par): 0.105797
Speedup: 3.819768
Efficiency: 0.954942
Threads: 8, Parallel Time in Seconds (T_par): 0.082150
Speedup: 4.919294
Efficiency: 0.614912
<------------------------------------------------------------------------>
Matrix Size: 750
Sequential Time in Seconds (T_seq): 1.290771
Threads: 1, Parallel Time in Seconds (T_par): 1.290365
Speedup: 1.000315
Efficiency: 1.000315
Threads: 2, Parallel Time in Seconds (T_par): 0.659557
Speedup: 1.957027
Efficiency: 0.978514
Threads: 4, Parallel Time in Seconds (T_par): 0.352315
Speedup: 3.663684
Efficiency: 0.915921
Threads: 8, Parallel Time in Seconds (T_par): 0.271850
Speedup: 4.748100
Efficiency: 0.593513
<------------------------------------------------------------------------>
```

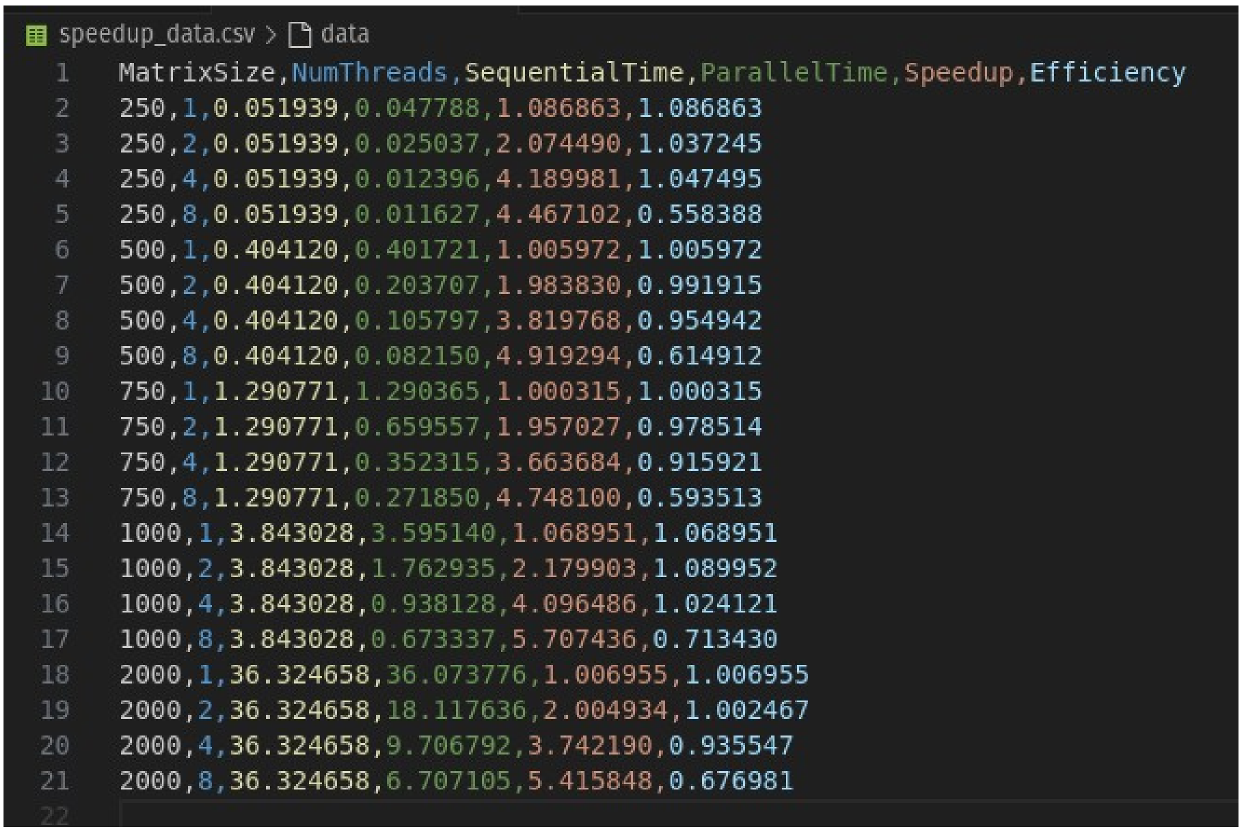


```
<------------------------------------------------------------------------->
Matrix Size: 1000
Sequential Time in Seconds (T_seq): 3.843028
Threads: 1, Parallel Time in Seconds (T_par): 3.595140
Speedup: 1.068951
Efficiency: 1.068951
Threads: 2, Parallel Time in Seconds (T_par): 1.762935
Speedup: 2.179903
Efficiency: 1.089952
Threads: 4, Parallel Time in Seconds (T_par): 0.938128
Speedup: 4.096486
Efficiency: 1.024121
Threads: 8, Parallel Time in Seconds (T_par): 0.673337
Speedup: 5.707436
Efficiency: 0.713430
<------------------------------------------------------------------------->
Matrix Size: 2000
Sequential Time in Seconds (T_seq): 36.324658
Threads: 1, Parallel Time in Seconds (T_par): 36.073776
Speedup: 1.006955
Efficiency: 1.006955
Threads: 2, Parallel Time in Seconds (T_par): 18.117636
Speedup: 2.004934
Efficiency: 1.002467
Threads: 4, Parallel Time in Seconds (T_par): 9.706792
Speedup: 3.742190
Efficiency: 0.935547
Threads: 8, Parallel Time in Seconds (T_par): 6.707105
Speedup: 5.415848
Efficiency: 0.676981
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_2$
```

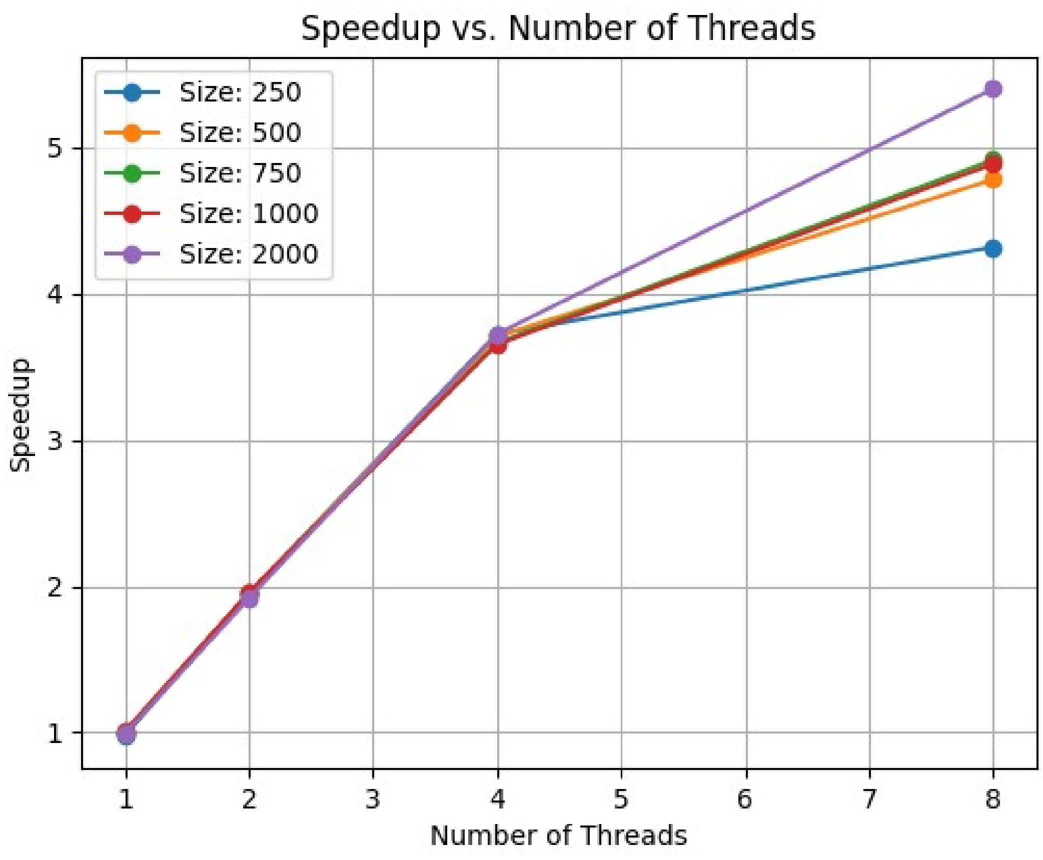


```
(env) nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_2$ python 2_plot.py
```

Analysis



```
MatrixSize,NumThreads,SequentialTime,ParallelTime,Speedup,Efficiency
250,1,0.051939,0.047788,1.086863,1.086863
250,2,0.051939,0.025037,2.074490,1.037245
250,4,0.051939,0.012396,4.189981,1.047495
250,8,0.051939,0.011627,4.467102,0.558388
500,1,0.404120,0.401721,1.005972,1.005972
500,2,0.404120,0.203707,1.983830,0.991915
500,4,0.404120,0.105797,3.819768,0.954942
500,8,0.404120,0.082150,4.919294,0.614912
750,1,1.290771,1.290365,1.000315,1.000315
750,2,1.290771,0.659557,1.957027,0.978514
750,4,1.290771,0.352315,3.663684,0.915921
750,8,1.290771,0.271850,4.748100,0.593513
1000,1,3.843028,3.595140,1.068951,1.068951
1000,2,3.843028,1.762935,2.179903,1.089952
1000,4,3.843028,0.938128,4.096486,1.024121
1000,8,3.843028,0.673337,5.707436,0.713430
2000,1,36.324658,36.073776,1.006955,1.006955
2000,2,36.324658,18.117636,2.004934,1.002467
2000,4,36.324658,9.706792,3.742190,0.935547
2000,8,36.324658,6.707105,5.415848,0.676981
```



This C code performs matrix multiplication using different numbers of threads with OpenMP and measures the performance in terms of execution time, speedup, and efficiency. Here's a brief analysis:

The `matrix_multiply` function multiplies two matrices, `A` and `B`, and stores the result in `C`. The computation is parallelized with OpenMP using the `#pragma omp parallel for` directive, which distributes the outer loop iterations across threads.

The `allocate_matrix` and `free_matrix` functions dynamically allocate and deallocate memory for the matrices. This ensures flexibility in matrix size, which is controlled by the `sizes` array.

The program measures both sequential and parallel execution times for different matrix sizes (250 to 2000) and thread counts (1, 2, 4, 8). `gettimeofday` is used to capture timing, and the results are stored in a CSV file for analysis.

The code calculates the speedup as the ratio of sequential time to parallel time, demonstrating how much faster the parallel version is compared to the sequential version. This data is printed to the console and saved to `**speedup_data.csv**`.

By comparing the performance across varying matrix sizes and thread counts, the program provides insights into how well the matrix multiplication scales with increased parallelism and larger problem sizes.

The speedup generally increases with the number of threads, indicating effective parallelization. This is expected as more threads allow for better distribution of the computational workload.

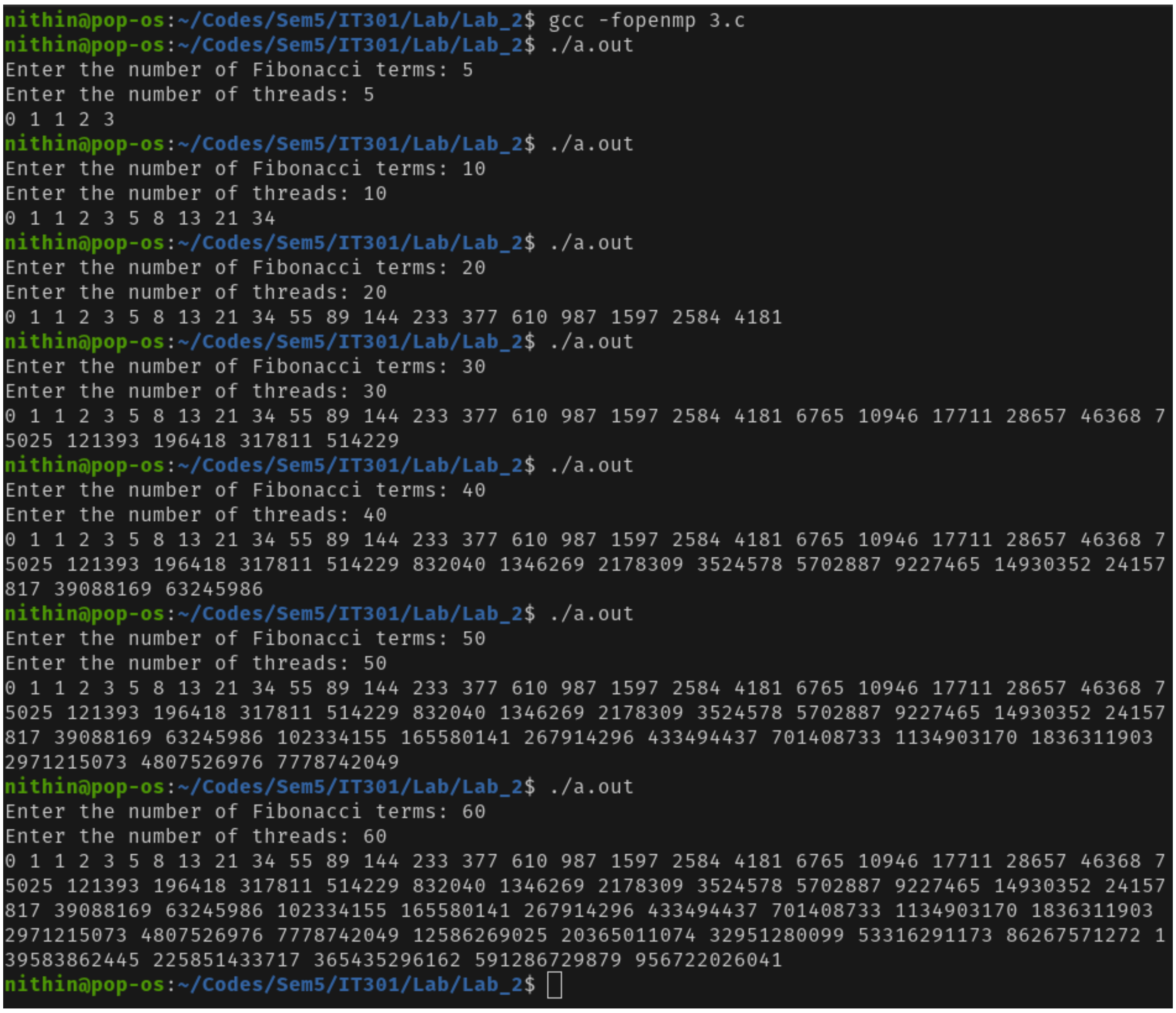
The speedup is more pronounced for larger matrix sizes (e.g., size 2000). This is likely because larger matrices have more computational work, which benefits more from parallel execution.

Q3. Develop a multi-threaded program to generate and print 'n' Fibonacci Series. One thread has to generate the numbers up to the specified limit and another thread has to print them. Ensure proper synchronization. (Note: If your output seems to be OK, try increasing the number of threads and/or n).

Code

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

void generate_fibonacci(long long n, long long fib[]) {
   fib[0] = 0;
   fib[1] = 1;

   for (long long i = 2; i < n; i++) {
      fib[i] = fib[i - 1] + fib[i - 2];
   }
}

void print_fibonacci(long long n, long long fib[]) {
   for (long long i = 0; i < n; i++) {
      printf("%lld ", fib[i]);
   }
   printf("\n");
}

int main() {
   long long n;
   int num_threads;

   printf("Enter the number of Fibonacci terms: ");
   scanf("%lld", &n);
   printf("Enter the number of threads: ");
   scanf("%d", &num_threads);

   omp_set_num_threads(num_threads);

   long long fib[n];

   #pragma omp parallel sections
   {
      #pragma omp section
      {
            generate_fibonacci(n, fib);
        }
        #pragma omp section
        {
            #pragma omp critical
            {
                print_fibonacci(n, fib);
            }
        }
    }

    return 0;
}
```

Output



```
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_2$ gcc -fopenmp 3.c
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_2$ ./a.out
Enter the number of Fibonacci terms: 5
Enter the number of threads: 5
0 1 1 2 3
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_2$ ./a.out
Enter the number of Fibonacci terms: 10
Enter the number of threads: 10
0 1 1 2 3 5 8 13 21 34
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_2$ ./a.out
Enter the number of Fibonacci terms: 20
Enter the number of threads: 20
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_2$ ./a.out
Enter the number of Fibonacci terms: 30
Enter the number of threads: 30
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657 46368 7
5025 121393 196418 317811 514229
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_2$ ./a.out
Enter the number of Fibonacci terms: 40
Enter the number of threads: 40
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657 46368 7
5025 121393 196418 317811 514229 832040 1346269 2178309 3524578 5702887 9227465 14930352 24157
817 39088169 63245986
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_2$ ./a.out
Enter the number of Fibonacci terms: 50
Enter the number of threads: 50
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657 46368 7
5025 121393 196418 317811 514229 832040 1346269 2178309 3524578 5702887 9227465 14930352 24157
817 39088169 63245986 102334155 165580141 267914296 433494437 701408733 1134903170 1836311903
2971215073 4807526976 7778742049
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_2$ ./a.out
Enter the number of Fibonacci terms: 60
Enter the number of threads: 60
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657 46368 7
5025 121393 196418 317811 514229 832040 1346269 2178309 3524578 5702887 9227465 14930352 24157
817 39088169 63245986 102334155 165580141 267914296 433494437 701408733 1134903170 1836311903
2971215073 4807526976 7778742049 12586269025 20365011074 32951280099 53316291173 86267571272 1
39583862445 225851433717 365435296162 591286729879 956722026041
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_2$
```

Analysis

The generate_fibonacci function calculates the first n Fibonacci numbers and stores them in the fib array.

The #pragma omp parallel sections directive is used to parallelize two tasks: generating and printing the Fibonacci sequence. However, these tasks are not truly parallel because generating the sequence (generate_fibonacci) must complete before printing (print_fibonacci).

The #pragma omp critical directive is used to ensure that only one thread accesses the printing section at a time. This is crucial to prevent race conditions, but in this case, it also highlights a design issue—printing the sequence while generating it is redundant since the sequence must be fully generated first.

The user specifies the number of threads, but the parallelization is minimal and doesn't significantly benefit from multiple threads due to the sequential dependency between generation and printing.

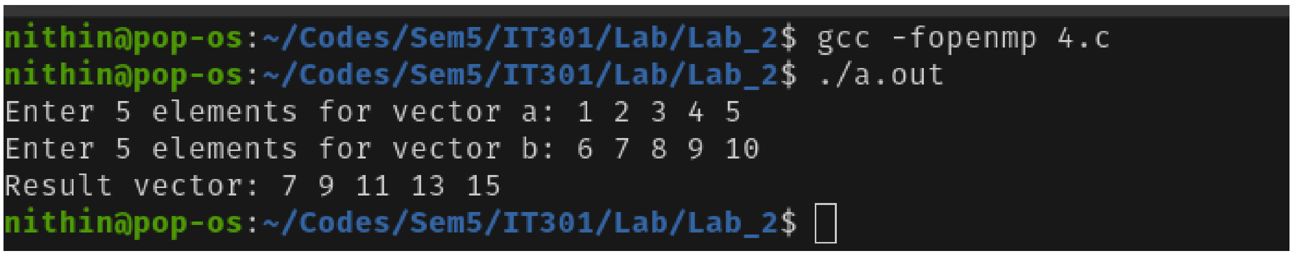
Q4. Write an OpenMP program to perform vector addition of two one-dimensional arrays of size 5 using 5 threads.

Code

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

#define SIZE 5

int main() {
   int i;
   int a[SIZE];
   int b[SIZE];
   int c[SIZE];


   printf("Enter %d elements for vector a: ", SIZE);
   for (i = 0; i < SIZE; i++) {
      scanf("%d", &a[i]);
   }


   printf("Enter %d elements for vector b: ", SIZE);
```

```
    for (i = 0; i < SIZE; i++) {
        scanf("%d", &b[i]);
    }

    omp_set_num_threads(SIZE);


    #pragma omp parallel for
    for (i = 0; i < SIZE; i++) {
        c[i] = a[i] + b[i];
    }


    printf("Result vector: ");
    for (i = 0; i < SIZE; i++) {
        printf("%d ", c[i]);
    }
    printf("\n");

    return 0;
}
```

Output



```
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_2$ gcc -fopenmp 4.c
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_2$ ./a.out
Enter 5 elements for vector a: 1 2 3 4 5
Enter 5 elements for vector b: 6 7 8 9 10
Result vector: 7 9 11 13 15
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_2$
```

Analysis

This C code performs a simple parallel addition of two vectors using OpenMP. The program first prompts the user to input elements for two vectors, a and b, each of size 5.

After the user inputs the values, the code utilizes OpenMP's #pragma omp parallel for directive to parallelize the addition of corresponding elements from the two vectors.

Each thread handles the addition of one pair of elements, and the results are stored in a third vector, c. Finally, the resulting vector c is printed to the console.

The use of OpenMP in this context, while basic, effectively demonstrates parallel processing for simple operations, where each addition is independent and can be easily distributed across multiple threads. This approach speeds up the execution by leveraging multiple threads, especially for larger datasets, though in this small example with only five elements, the performance gain might not be noticeable.

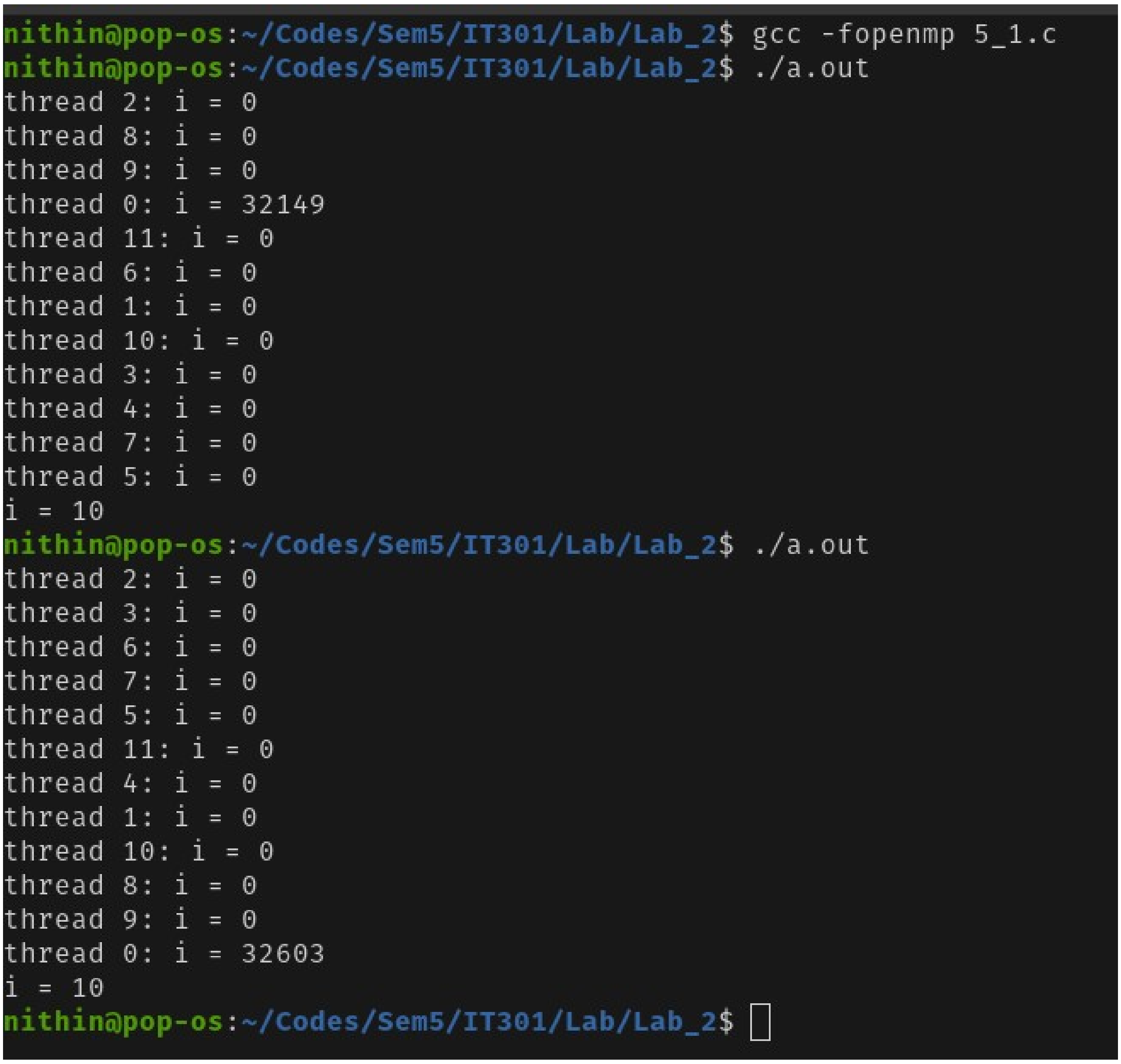
Q5. Write an OpenMP program to understand and analyze the concepts of private(), firstprivate().

(i) First execute the program by declaring the variable as private(), note down the results and write your observation.

Code

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

int main (void)
{
   int i = 10;

   #pragma omp parallel private(i)
   {
      printf("thread %d: i = %d\n", omp_get_thread_num(), i);
      i = 1000 + omp_get_thread_num();
   }

   printf("i = %d\n", i);

   return 0;
}
```

Output (Running the same program two times)



```
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_2$ gcc -fopenmp 5_1.c
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_2$ ./a.out
thread 2: i = 0
thread 8: i = 0
thread 9: i = 0
thread 0: i = 32149
thread 11: i = 0
thread 6: i = 0
thread 1: i = 0
thread 10: i = 0
thread 3: i = 0
thread 4: i = 0
thread 7: i = 0
thread 5: i = 0
i = 10
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_2$ ./a.out
thread 2: i = 0
thread 3: i = 0
thread 6: i = 0
thread 7: i = 0
thread 5: i = 0
thread 11: i = 0
thread 4: i = 0
thread 1: i = 0
thread 10: i = 0
thread 8: i = 0
thread 9: i = 0
thread 0: i = 32603
i = 10
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_2$
```

Analysis

private variables are not initialised, i.e. they start with random values like any other local automatic variable (and they are often implemented using automatic variables on the stack of each thread).

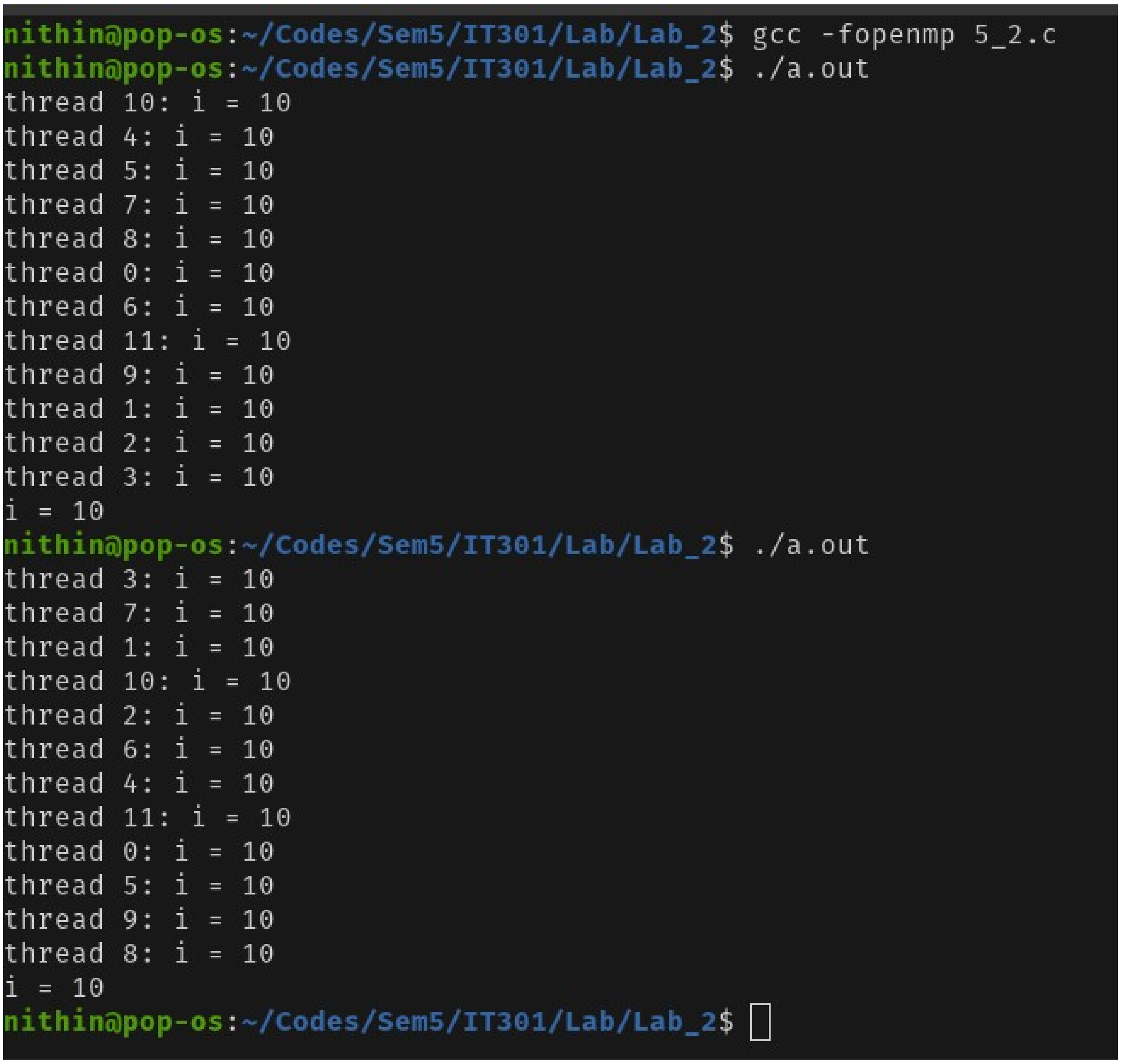
This clearly demonstrates that the value of i is random (not initialised) inside the parallel region and that any modifications to it are not visible after the parallel region (i.e. the variable keeps its value from before entering the region).

(ii) Execute the same program with firstprivate(), record the results and write your observation.

Code

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

int main (void)
{
   int i = 10;

   #pragma omp parallel firstprivate(i)
   {
      printf("thread %d: i = %d\n", omp_get_thread_num(), i);
      i = 1000 + omp_get_thread_num();
   }

   printf("i = %d\n", i);

   return 0;
}
```

Output (Running the same program two times)



```
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_2$ gcc -fopenmp 5_2.c
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_2$ ./a.out
thread 10: i = 10
thread 4: i = 10
thread 5: i = 10
thread 7: i = 10
thread 8: i = 10
thread 0: i = 10
thread 6: i = 10
thread 11: i = 10
thread 9: i = 10
thread 1: i = 10
thread 2: i = 10
thread 3: i = 10
i = 10
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_2$ ./a.out
thread 3: i = 10
thread 7: i = 10
thread 1: i = 10
thread 10: i = 10
thread 2: i = 10
thread 6: i = 10
thread 4: i = 10
thread 11: i = 10
thread 0: i = 10
thread 5: i = 10
thread 9: i = 10
thread 8: i = 10
i = 10
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_2$ 
```

Analysis

If i is made firstprivate, then it is initialised with the value that it has before the parallel region. Still modifications to the value of i inside the parallel region are not visible after it.

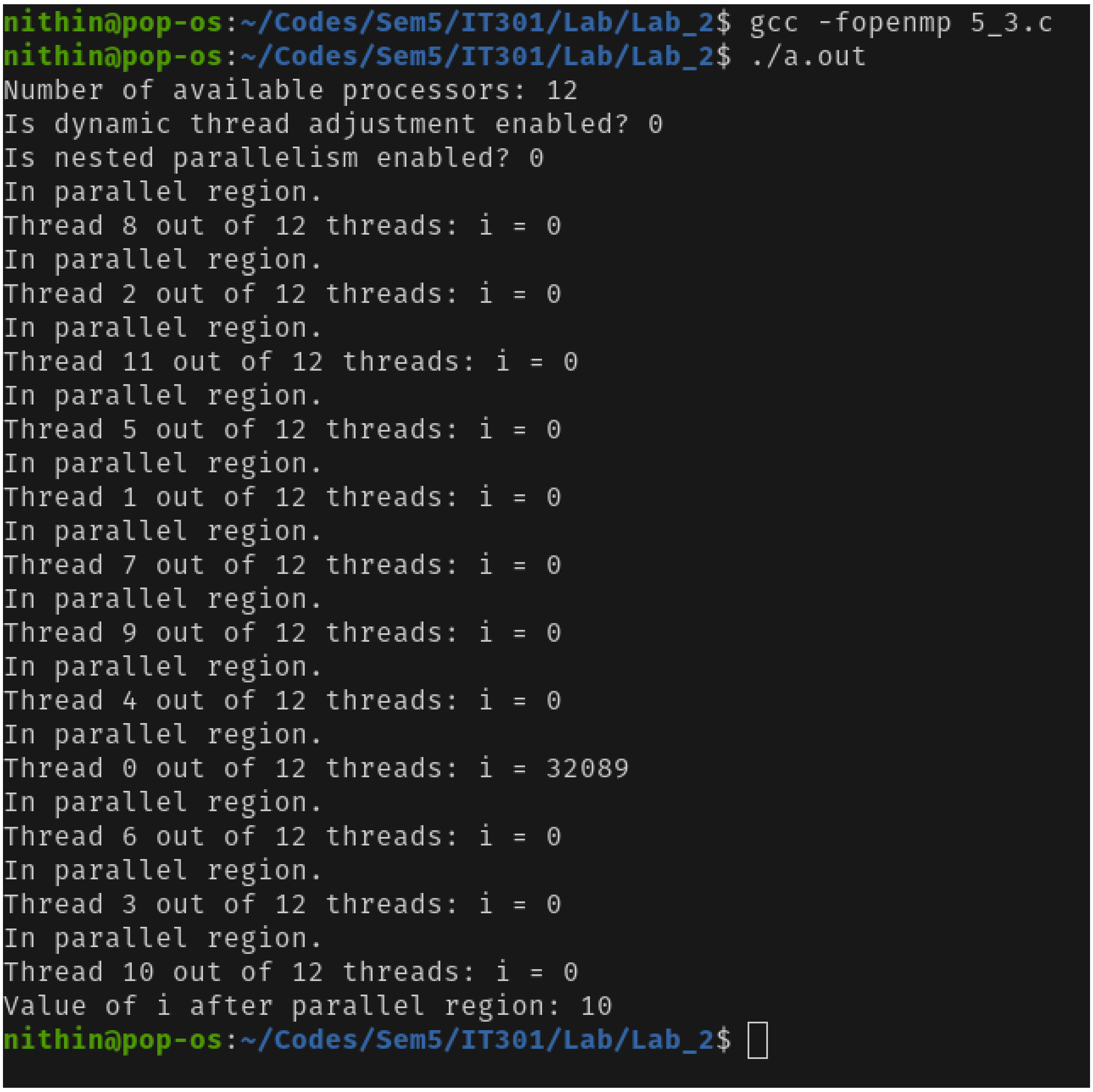
(iii,iv, v, vi, vii,viii,ix) To examine the above scenario, the functions such as omp_get_num_procs()

Case 1 – private()

Code

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

int main(void) {

   int i = 10;
   int num_procs = omp_get_num_procs();
   printf("Number of available processors: %d\n", num_procs);
   omp_set_num_threads(num_procs);
   int is_dynamic = omp_get_dynamic();
   printf("Is dynamic thread adjustment enabled? %d\n", is_dynamic);
   int is_nested = omp_get_nested();
   printf("Is nested parallelism enabled? %d\n", is_nested);
   #pragma omp parallel private(i)
   {
      if (omp_in_parallel()) {
         printf("In parallel region.\n");
      }
      int thread_num = omp_get_thread_num();
      int num_threads = omp_get_num_threads();
      printf("Thread %d out of %d threads: i = %d\n", thread_num,
num_threads, i);
      i = 1000 + thread_num;
   }
   printf("Value of i after parallel region: %d\n", i);

   return 0;
}
```

Output



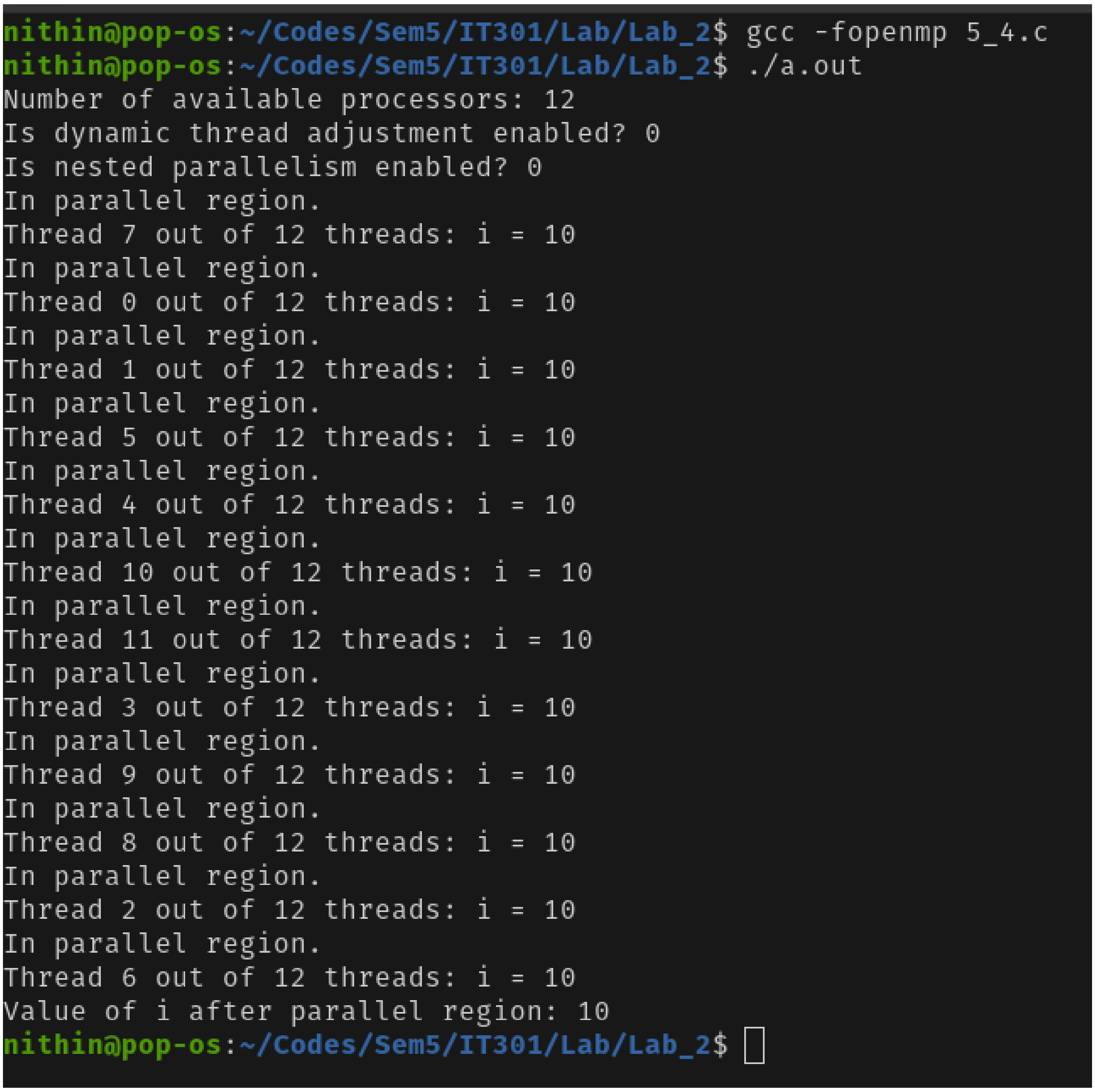
```
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_2$ gcc -fopenmp 5_3.c
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_2$ ./a.out
Number of available processors: 12
Is dynamic thread adjustment enabled? 0
Is nested parallelism enabled? 0
In parallel region.
Thread 8 out of 12 threads: i = 0
In parallel region.
Thread 2 out of 12 threads: i = 0
In parallel region.
Thread 11 out of 12 threads: i = 0
In parallel region.
Thread 5 out of 12 threads: i = 0
In parallel region.
Thread 1 out of 12 threads: i = 0
In parallel region.
Thread 7 out of 12 threads: i = 0
In parallel region.
Thread 9 out of 12 threads: i = 0
In parallel region.
Thread 4 out of 12 threads: i = 0
In parallel region.
Thread 0 out of 12 threads: i = 32089
In parallel region.
Thread 6 out of 12 threads: i = 0
In parallel region.
Thread 3 out of 12 threads: i = 0
In parallel region.
Thread 10 out of 12 threads: i = 0
Value of i after parallel region: 10
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_2$ 
```

Case 2 – firstprivate()

Code

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

int main(void) {

   int i = 10;
   int num_procs = omp_get_num_procs();
   printf("Number of available processors: %d\n", num_procs);
   omp_set_num_threads(num_procs);
   int is_dynamic = omp_get_dynamic();
   printf("Is dynamic thread adjustment enabled? %d\n", is_dynamic);
   int is_nested = omp_get_nested();
   printf("Is nested parallelism enabled? %d\n", is_nested);
   #pragma omp parallel firstprivate(i)
   {
      if (omp_in_parallel()) {
         printf("In parallel region.\n");
      }
      int thread_num = omp_get_thread_num();
      int num_threads = omp_get_num_threads();
      printf("Thread %d out of %d threads: i = %d\n", thread_num,
num_threads, i);
      i = 1000 + thread_num;
   }
   printf("Value of i after parallel region: %d\n", i);
   return 0;
}
```

Output



```
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_2$ gcc -fopenmp 5_4.c
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_2$ ./a.out
Number of available processors: 12
Is dynamic thread adjustment enabled? 0
Is nested parallelism enabled? 0
In parallel region.
Thread 7 out of 12 threads: i = 10
In parallel region.
Thread 0 out of 12 threads: i = 10
In parallel region.
Thread 1 out of 12 threads: i = 10
In parallel region.
Thread 5 out of 12 threads: i = 10
In parallel region.
Thread 4 out of 12 threads: i = 10
In parallel region.
Thread 10 out of 12 threads: i = 10
In parallel region.
Thread 11 out of 12 threads: i = 10
In parallel region.
Thread 3 out of 12 threads: i = 10
In parallel region.
Thread 9 out of 12 threads: i = 10
In parallel region.
Thread 8 out of 12 threads: i = 10
In parallel region.
Thread 2 out of 12 threads: i = 10
In parallel region.
Thread 6 out of 12 threads: i = 10
Value of i after parallel region: 10
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_2$ 
```

Analysis

omp_get_num_procs(): Returns the number of processors available to the program. This can help in determining the optimal number of threads to use.
omp_set_num_threads(int n): Sets the number of threads to be used in the next parallel region to n. In this example, it's set to the number of available processors.
omp_get_num_threads(): Returns the number of threads currently being used in the parallel region.

omp_in_parallel(): Returns a non-zero value if the code is currently executing within a parallel region, and 0 otherwise. This is useful to check if you're inside a parallel region.

omp_get_dynamic(): Checks if dynamic adjustment of the number of threads is enabled. If dynamic adjustment is enabled, OpenMP can change the number of threads during execution.

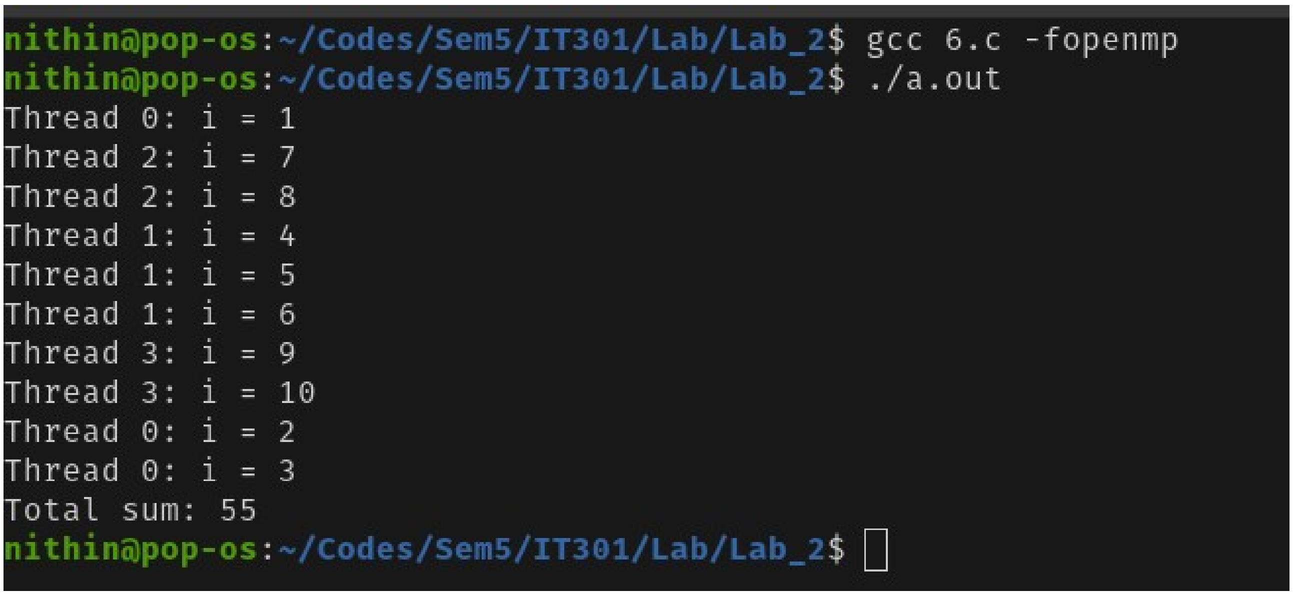
omp_get_nested(): Checks if nested parallelism is enabled, which allows parallel regions to be nested inside other parallel regions.

Q6. Write a simple OpenMP program to demonstrate the parallel loop construct

Code

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

int main() {
   int i;
   int sum = 0;
   omp_set_num_threads(4);


   #pragma omp parallel
   {

      #pragma omp for reduction(+:sum)
      for (i = 1; i <= 10; i++) {
         sum += i;

         printf("Thread %d: i = %d\n", omp_get_thread_num(), i);
      }
   }

   printf("Total sum: %d\n", sum);

   return 0;
}
```

Output



```
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_2$ gcc 6.c -fopenmp
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_2$ ./a.out
Thread 0: i = 1
Thread 2: i = 7
Thread 2: i = 8
Thread 1: i = 4
Thread 1: i = 5
Thread 1: i = 6
Thread 3: i = 9
Thread 3: i = 10
Thread 0: i = 2
Thread 0: i = 3
Total sum: 55
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_2$ 
```

Analysis

This C code calculates the sum of integers from 1 to 10 using OpenMP to parallelize the summation process. The program sets up 4 threads with omp_set_num_threads(4) and uses the #pragma omp parallel directive to initiate parallel execution.

Within this parallel region, the #pragma omp for reduction(+:sum) directive is employed to divide the loop iterations among the threads, ensuring that each thread processes a portion of the summation. The reduction(+:sum) clause is critical as it combines the partial sums computed by each thread into the final sum variable, avoiding race conditions.

The program also prints which thread processes each value of i, showcasing the work distribution. Finally, the total sum is printed. This approach demonstrates efficient parallel computation with OpenMP, where the reduction operation safely aggregates results from multiple threads.