**PARALLEL COMPUTING SYSTEM**

**ASSIGNMENT-II**

# MPI Programming

Hello world:

## PROGRAM:

#include <stdio.h>

#include <omp.h>

int main() {

#pragma omp parallel

{

int threadID = omp\_get\_thread\_num();

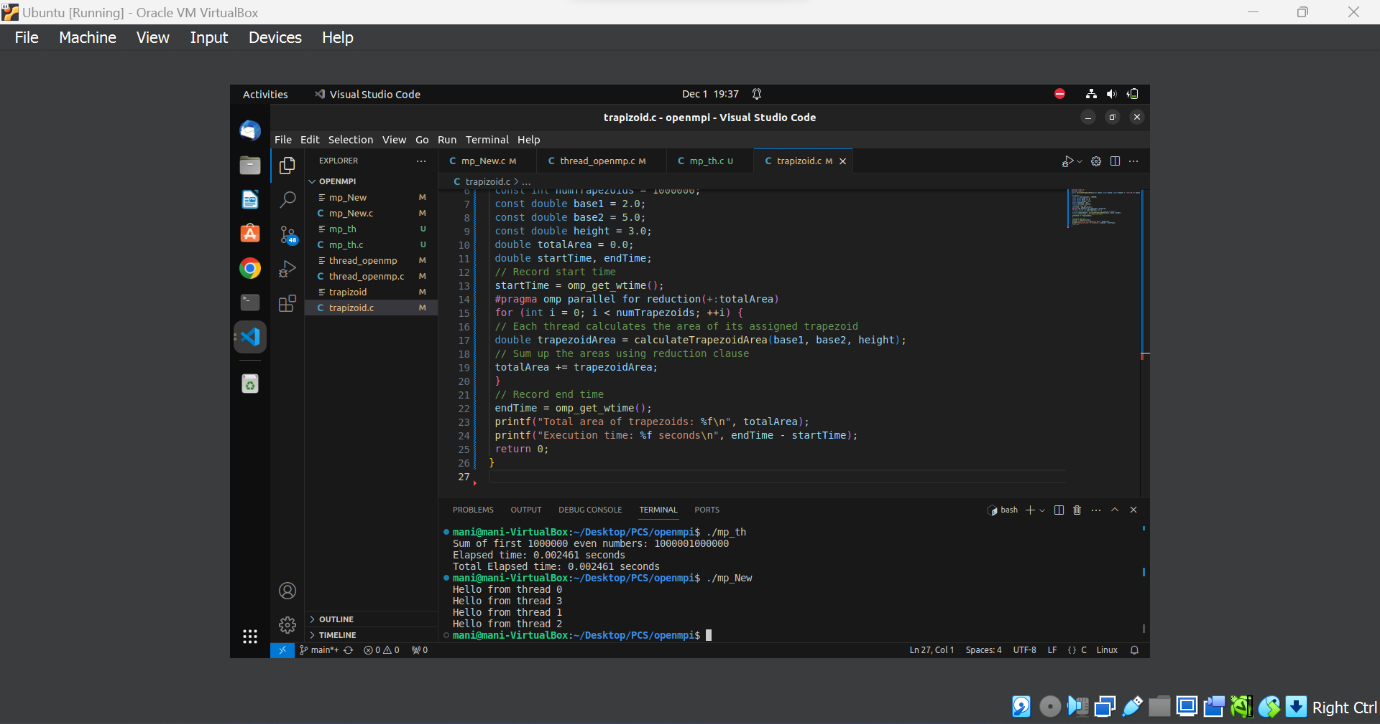
printf("Hello from thread %d\n", threadID);

}

return 0;

}

## OUTPUT:



## INFERENCE:

Presenting the message Salutations. A greeting message is sent to the root process (rank 0) by each non-root MPI process that the application generates. After receiving these messages, the root process prints each greeting.

## MATRIX ADDITION USING MPI SCATTER AND MPI GATHER:

## PROGRAM:

#include <stdio.h>

#include <stdlib.h>

#include <sys/time.h>

#include <mpi.h>

#define MATRIX\_SIZE 4

// Function to generate random values for the matrix

void generateRandomInput(int matrix[MATRIX\_SIZE][MATRIX\_SIZE]) {

for (int i = 0; i < MATRIX\_SIZE; i++) {

for (int j = 0; j < MATRIX\_SIZE; j++) {

matrix[i][j] = rand() % 10; // Generates random values between 0 and 9

}

}

}

// Function for matrix addition

void matrixAddition(int matrix1[MATRIX\_SIZE][MATRIX\_SIZE], int matrix2[MATRIX\_SIZE][MATRIX\_SIZE], int result[MATRIX\_SIZE][MATRIX\_SIZE], int size) {

for (int i = 0; i < size; i++) {

for (int j = 0; j < MATRIX\_SIZE; j++) {

result[i][j] = matrix1[i][j] + matrix2[i][j];

}

}

}

int main(int argc, char\*\* argv) {

int world\_size, my\_rank;

MPI\_Init(&argc, &argv);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &world\_size);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &my\_rank);

int matrix1[MATRIX\_SIZE][MATRIX\_SIZE];

int matrix2[MATRIX\_SIZE][MATRIX\_SIZE];

int local\_matrix1[MATRIX\_SIZE][MATRIX\_SIZE / world\_size];

int local\_matrix2[MATRIX\_SIZE][MATRIX\_SIZE / world\_size];

int local\_result[MATRIX\_SIZE][MATRIX\_SIZE / world\_size];

struct timeval start, end;

long long elapsed\_time;

if (my\_rank == 0) {

generateRandomInput(matrix1); // Generate random input on the root process

generateRandomInput(matrix2); // Generate another random matrix

gettimeofday(&start, NULL); // Start measuring execution time

}

// Scatter matrix1 and matrix2 to all processes

MPI\_Scatter(matrix1, MATRIX\_SIZE \* MATRIX\_SIZE / world\_size, MPI\_INT, local\_matrix1, MATRIX\_SIZE \* MATRIX\_SIZE / world\_size, MPI\_INT, 0, MPI\_COMM\_WORLD);

MPI\_Scatter(matrix2, MATRIX\_SIZE \* MATRIX\_SIZE / world\_size, MPI\_INT, local\_matrix2, MATRIX\_SIZE \* MATRIX\_SIZE / world\_size, MPI\_INT, 0, MPI\_COMM\_WORLD);

// Perform matrix addition locally

matrixAddition(local\_matrix1, local\_matrix2, local\_result, MATRIX\_SIZE / world\_size);

// Gather local results back to the root process using MPI\_Gather

MPI\_Gather(local\_result, MATRIX\_SIZE \* MATRIX\_SIZE / world\_size, MPI\_INT, matrix1, MATRIX\_SIZE \* MATRIX\_SIZE / world\_size, MPI\_INT, 0, MPI\_COMM\_WORLD);

if (my\_rank == 0) {

gettimeofday(&end, NULL); // Stop measuring execution time

elapsed\_time = (end.tv\_sec - start.tv\_sec) \* 1000000 + (end.tv\_usec - start.tv\_usec);

printf("Matrix Addition Result:\n");

for (int i = 0; i < MATRIX\_SIZE; i++) {

for (int j = 0; j < MATRIX\_SIZE; j++) {

printf("%d ", matrix1[i][j]); // Print the result

}

printf("\n");

}

printf("Elapsed time: %lld microseconds\n", elapsed\_time); // Print execution time

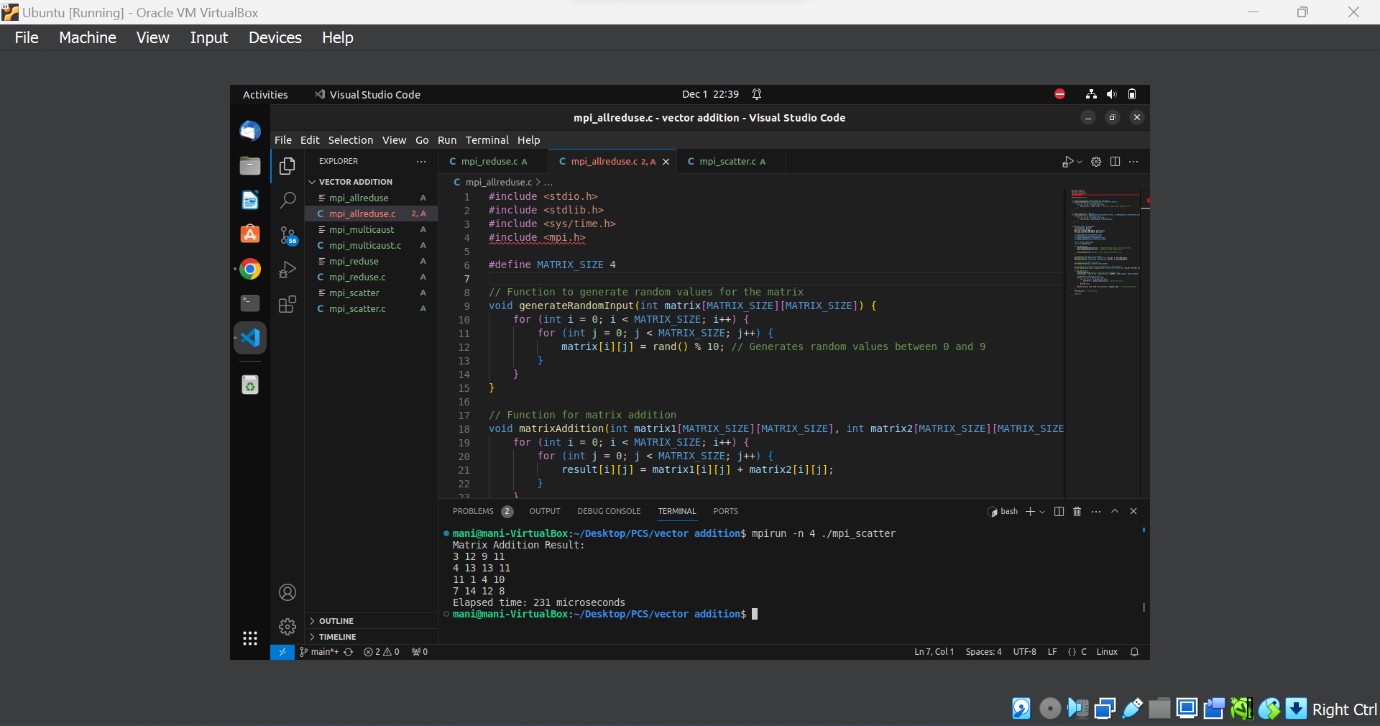
}

MPI\_Finalize(); // Finalize MPI

return 0;

}

## OUTPUT:



## INFERENCE:

* The execution time is faster while using MPI Scatter and MPI Gather.
* The code divides the matrices into chunks and distributes these chunks across multiple processes using MPI. This allows for parallel computation, where each process independently works on its portion of the matrices.
* Matrix addition is an embarrassingly parallel task, meaning that each element of the result matrix can be computed independently. MPI facilitates this parallelization by distributing the workload across available processes.
* MPI\_Scatter and MPI\_Gather efficiently distribute and gather data, minimizing communication overhead.

## MATRIX ADDITION USING MPI REDUCE AND BROADCAST:

## PROGRAM:

#include <stdio.h>

#include <stdlib.h>

#include <sys/time.h>

#include <mpi.h>

#define MATRIX\_SIZE 4

// Function to generate random values for the matrix

void generateRandomInput(int matrix[MATRIX\_SIZE][MATRIX\_SIZE]) {

for (int i = 0; i < MATRIX\_SIZE; i++) {

for (int j = 0; j < MATRIX\_SIZE; j++) {

matrix[i][j] = rand() % 10; // Generates random values between 0 and 9

}

}

}

// Function for matrix addition

void matrixAddition(int matrix1[MATRIX\_SIZE][MATRIX\_SIZE], int matrix2[MATRIX\_SIZE][MATRIX\_SIZE], int result[MATRIX\_SIZE][MATRIX\_SIZE]) {

for (int i = 0; i < MATRIX\_SIZE; i++) {

for (int j = 0; j < MATRIX\_SIZE; j++) {

result[i][j] = matrix1[i][j] + matrix2[i][j];

}

}

}

int main(int argc, char\*\* argv) {

int world\_size, my\_rank;

MPI\_Init(&argc, &argv);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &world\_size);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &my\_rank);

int matrix1[MATRIX\_SIZE][MATRIX\_SIZE];

int matrix2[MATRIX\_SIZE][MATRIX\_SIZE];

int local\_result[MATRIX\_SIZE][MATRIX\_SIZE];

int global\_result[MATRIX\_SIZE][MATRIX\_SIZE];

struct timeval start, end;

long long elapsed\_time;

if (my\_rank == 0) {

generateRandomInput(matrix1); // Generate random input on the root process

generateRandomInput(matrix2); // Generate another random matrix

gettimeofday(&start, NULL); // Start measuring execution time

}

// Broadcast matrices to all processes

MPI\_Bcast(matrix1, MATRIX\_SIZE \* MATRIX\_SIZE, MPI\_INT, 0, MPI\_COMM\_WORLD);

MPI\_Bcast(matrix2, MATRIX\_SIZE \* MATRIX\_SIZE, MPI\_INT, 0, MPI\_COMM\_WORLD);

// Perform matrix addition locally

matrixAddition(matrix1, matrix2, local\_result);

// Sum the local results across all processes using MPI\_Reduce

MPI\_Reduce(local\_result, global\_result, MATRIX\_SIZE \* MATRIX\_SIZE, MPI\_INT, MPI\_SUM, 0, MPI\_COMM\_WORLD);

if (my\_rank == 0) {

gettimeofday(&end, NULL); // Stop measuring execution time

elapsed\_time = (end.tv\_sec - start.tv\_sec) \* 1000000 + (end.tv\_usec - start.tv\_usec);

printf("Matrix Addition Result:\n");

for (int i = 0; i < MATRIX\_SIZE; i++) {

for (int j = 0; j < MATRIX\_SIZE; j++) {

printf("%d ", global\_result[i][j]); // Print the result

}

printf("\n");

}

printf("Elapsed time: %lld microseconds\n", elapsed\_time); // Print execution time

}

MPI\_Finalize(); // Finalize MPI

return 0;

}

## OUTPUT:

## 

## INFERENCE:

* The program demonstrates parallel matrix addition using MPI, distributing the work among multiple processes.
* The use of MPI\_Bcast ensures efficient distribution of matrices to all processes.
* MPI\_Reduce is employed to gather and sum the local results on the root process.
* The elapsed time measurement provides insight into the execution time of the parallelized matrix addition.

## MATRIX ADDITION USING MPI ALLREDUCE AND BROADCAST:

## PROGRAM:

#include <stdio.h>

#include <stdlib.h>

#include <sys/time.h>

#include <mpi.h>

#define MATRIX\_SIZE 4

// Function to generate random values for the matrix

void generateRandomInput(int matrix[MATRIX\_SIZE][MATRIX\_SIZE]) {

for (int i = 0; i < MATRIX\_SIZE; i++) {

for (int j = 0; j < MATRIX\_SIZE; j++) {

matrix[i][j] = rand() % 10; // Generates random values between 0 and 9

}

}

}

// Function for matrix addition

void matrixAddition(int matrix1[MATRIX\_SIZE][MATRIX\_SIZE], int matrix2[MATRIX\_SIZE][MATRIX\_SIZE], int result[MATRIX\_SIZE][MATRIX\_SIZE]) {

for (int i = 0; i < MATRIX\_SIZE; i++) {

for (int j = 0; j < MATRIX\_SIZE; j++) {

result[i][j] = matrix1[i][j] + matrix2[i][j];

}

}

}

int main(int argc, char\*\* argv) {

int world\_size, my\_rank;

MPI\_Init(&argc, &argv);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &world\_size);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &my\_rank);

int matrix1[MATRIX\_SIZE][MATRIX\_SIZE];

int matrix2[MATRIX\_SIZE][MATRIX\_SIZE];

int local\_result[MATRIX\_SIZE][MATRIX\_SIZE];

int global\_result[MATRIX\_SIZE][MATRIX\_SIZE];

struct timeval start, end;

long long elapsed\_time;

if (my\_rank == 0) {

generateRandomInput(matrix1); // Generate random input on the root process

generateRandomInput(matrix2); // Generate another random matrix

gettimeofday(&start, NULL); // Start measuring execution time

}

// Broadcast matrices to all processes

MPI\_Bcast(matrix1, MATRIX\_SIZE \* MATRIX\_SIZE, MPI\_INT, 0, MPI\_COMM\_WORLD);

MPI\_Bcast(matrix2, MATRIX\_SIZE \* MATRIX\_SIZE, MPI\_INT, 0, MPI\_COMM\_WORLD);

// Perform matrix addition locally

matrixAddition(matrix1, matrix2, local\_result);

// Sum the local results across all processes using MPI\_Allreduce

MPI\_Allreduce(local\_result, global\_result, MATRIX\_SIZE \* MATRIX\_SIZE, MPI\_INT, MPI\_SUM, MPI\_COMM\_WORLD);

if (my\_rank == 0) {

gettimeofday(&end, NULL); // Stop measuring execution time

elapsed\_time = (end.tv\_sec - start.tv\_sec) \* 1000000 + (end.tv\_usec - start.tv\_usec);

printf("Matrix Addition Result:\n");

for (int i = 0; i < MATRIX\_SIZE; i++) {

for (int j = 0; j < MATRIX\_SIZE; j++) {

printf("%d ", global\_result[i][j]); // Print the result

}

printf("\n");

}

printf("Elapsed time: %lld microseconds\n", elapsed\_time); // Print execution time

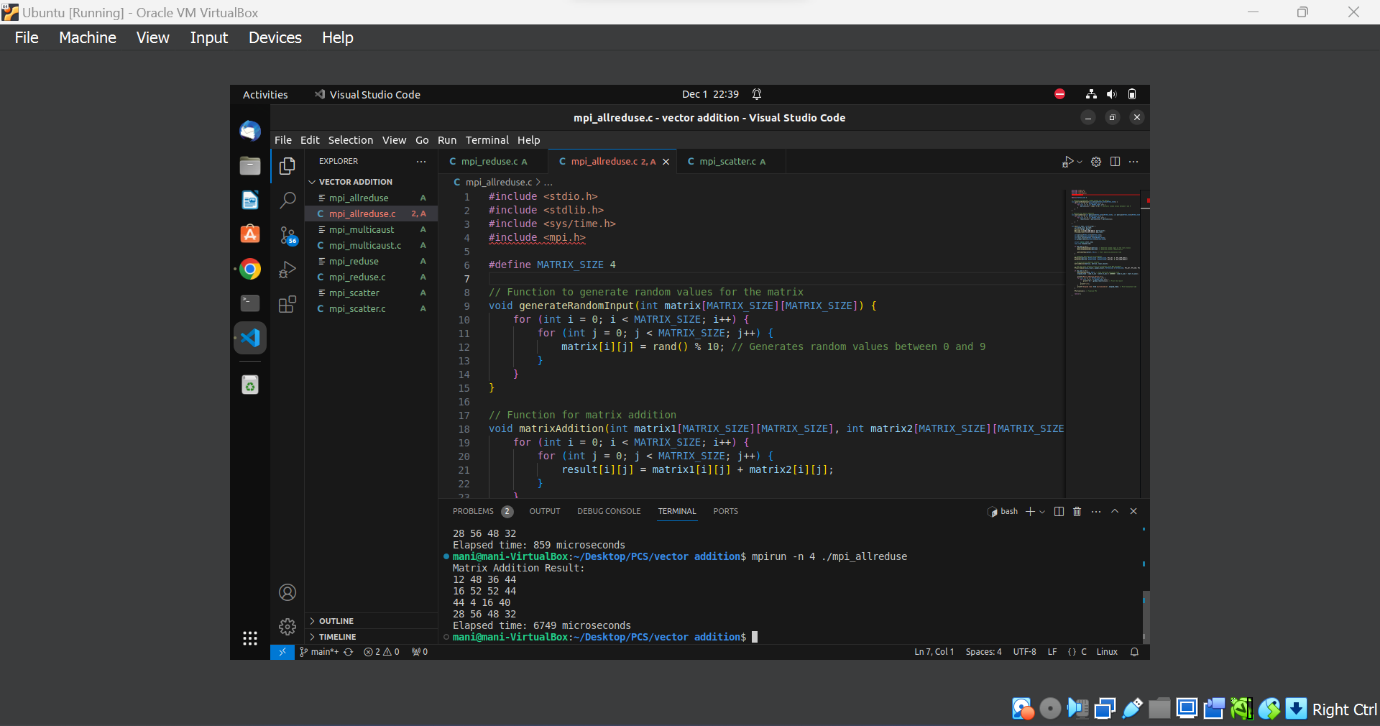
}

MPI\_Finalize(); // Finalize MPI

return 0;

}

## OUTPUT:



## INFERENCE:

* MPI All reduce combines the reduction and broadcast steps in a single collective operation.
* Every process receives the final result directly after the operation, eliminating the need for a separate gathering step.
* MPI\_Allreduce might have a higher overhead than MPI\_Reduce because it involves more communication between processes.

# OPENMP PROGRAMMING

## SIMPLE PROGRAM:

## HELLO WORLD:

## PROGRAM:

#include <stdio.h>

#include <omp.h>

int main() {

#pragma omp parallel

{

int threadID = omp\_get\_thread\_num();

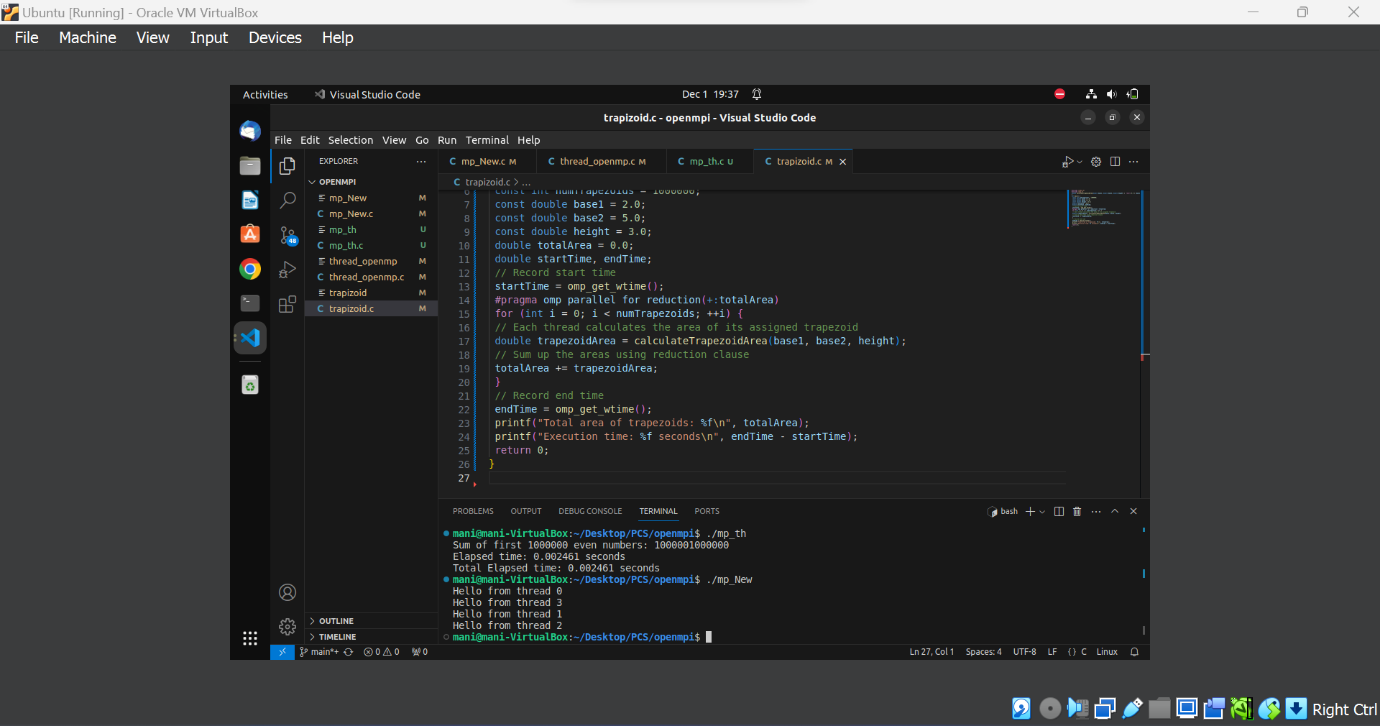
printf("Hello from thread %d\n", threadID);

}

return 0;

}

## OUTPUT:



## INFERENCE:

Using the Open MP pragma parallel, Hello World has been displayed by compiling and executing the program.

## DISPLAYING THE MAXIMUM NUMBER OF THREADS:

#include<stdio.h>

#include<omp.h>

void say\_hello(void)

{

int myrank=omp\_get\_thread\_num();

int threadcount=omp\_get\_num\_threads();

printf("Hello from thread %d of %d\n", myrank,threadcount);

}

int main(void)

{

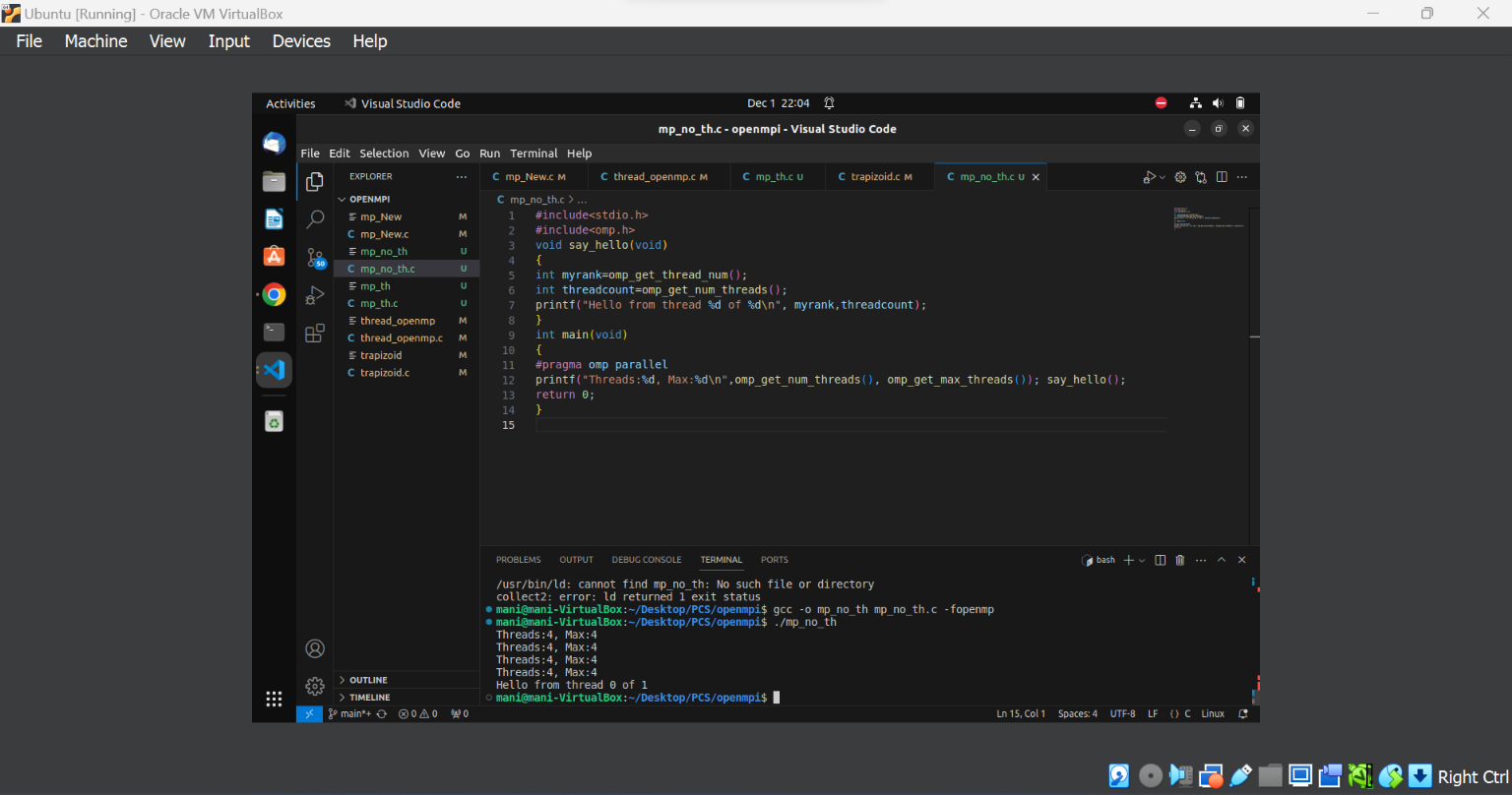
#pragma omp parallel

printf("Threads:%d, Max:%d\n",omp\_get\_num\_threads(), omp\_get\_max\_threads()); say\_hello();

return 0;

}

## OUTPUT:



## INFERENCE:

* This program uses OpenMP to print information about the number of threads in the parallel region and then calls a function “say\_hello” from within the parallel region to print a "Hello" message from each thread.
* The **#pragma omp parallel** directive creates a team of threads and the enclosed block is executed by all the threads in the team.
* The printf statement within the parallel region prints information about the number of threads and the maximum number of threads. This is useful for understanding the configuration of the parallel execution environment.
* The say\_hello function is called from within the parallel region, demonstrating the parallel execution of the function by multiple threads. Each thread prints its rank and the total number of threads.

## DISPLAYING THE THREADS WITHIN THE PROGRAM OR COMPILATION:

## PROGRAM:

#include<stdio.h>

#include<omp.h>

void say\_hello(void)

{

int myrank=omp\_get\_thread\_num();

int threadcount=omp\_get\_num\_threads();

printf("Hello from thread %d of %d\n", myrank,threadcount);

}

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

{

omp\_set\_num\_threads(4);

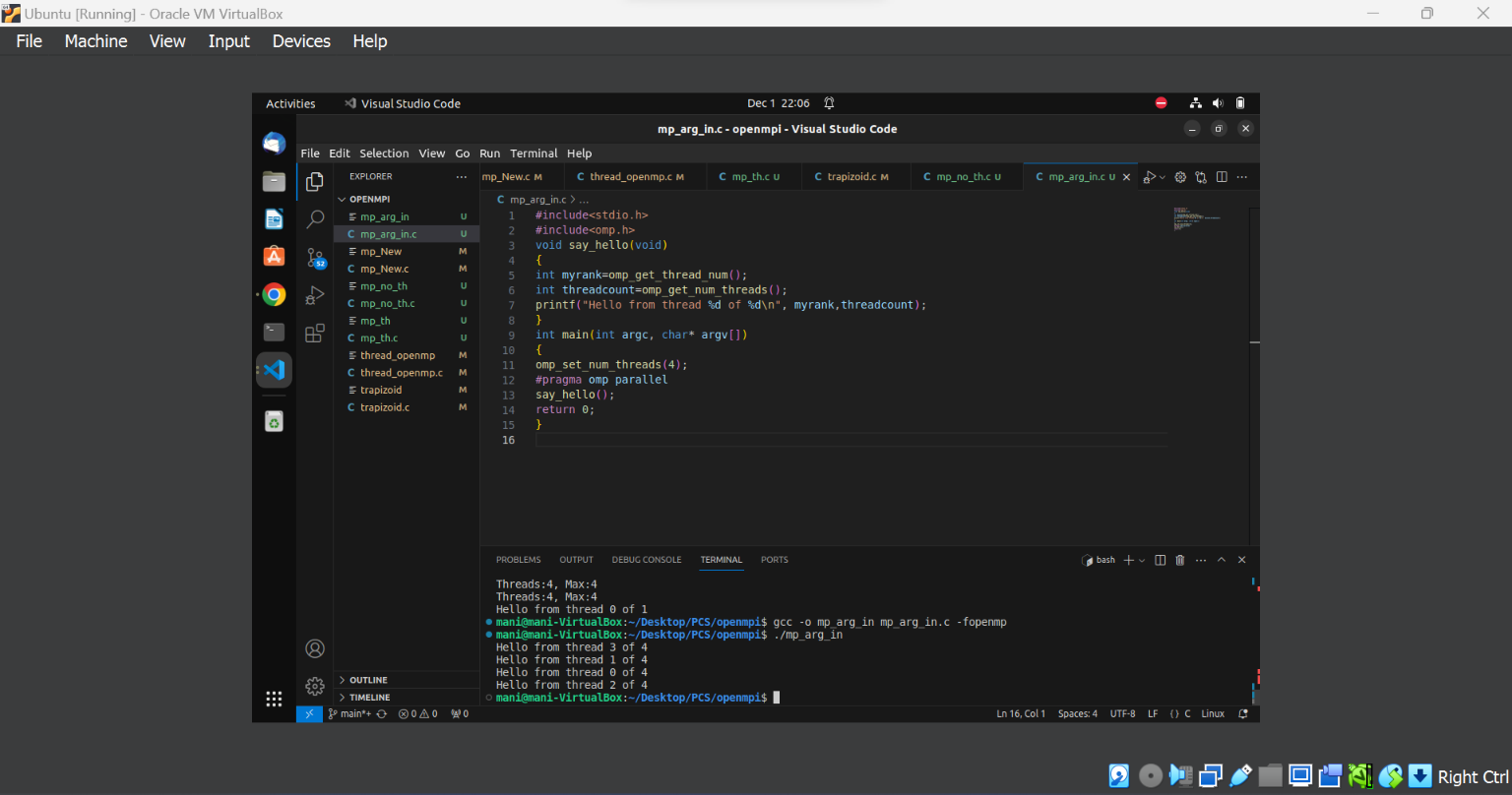
#pragma omp parallel

say\_hello();

return 0;

}

## OUTPUT:



## INFERENCE:

* The omp\_get\_thread\_num() function retrieves the thread number within the team for each thread.
* The omp\_get\_num\_threads() function retrieves the total number of threads in the team.
* Since the number of threads is set to 4 explicitly, the output will likely show "Hello" messages from each of the 4 threads.
* The output might not be deterministic in terms of the order in which the threads print their messages, as the scheduling of threads is implementation-dependent.

## INFERENCE:

* When we set the number of threads using “omp\_set\_num\_threads” in the program, we are providing a directive to the OpenMP runtime to use a specific number of threads. The runtime system then attempts to create and use the specified number of threads during the parallel execution of the program.
* In the program, we use omp\_set\_num\_threads(4) to programmatically set the number of threads to 4.
* However, the setting within the program is generally considered a default or a recommendation. It does not necessarily impose a strict constraint on the number of threads.

## SCOPE OF VARIABLES

#include<stdio.h>

int main(void)

{

int a=1, b=1, c=1, d=1;

#pragma omp parallel num\_threads(10) \

private(a) shared(b) firstprivate(c)

{

printf("Hello World!\n");

a++;

b++;

c++;

d++;

}

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

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

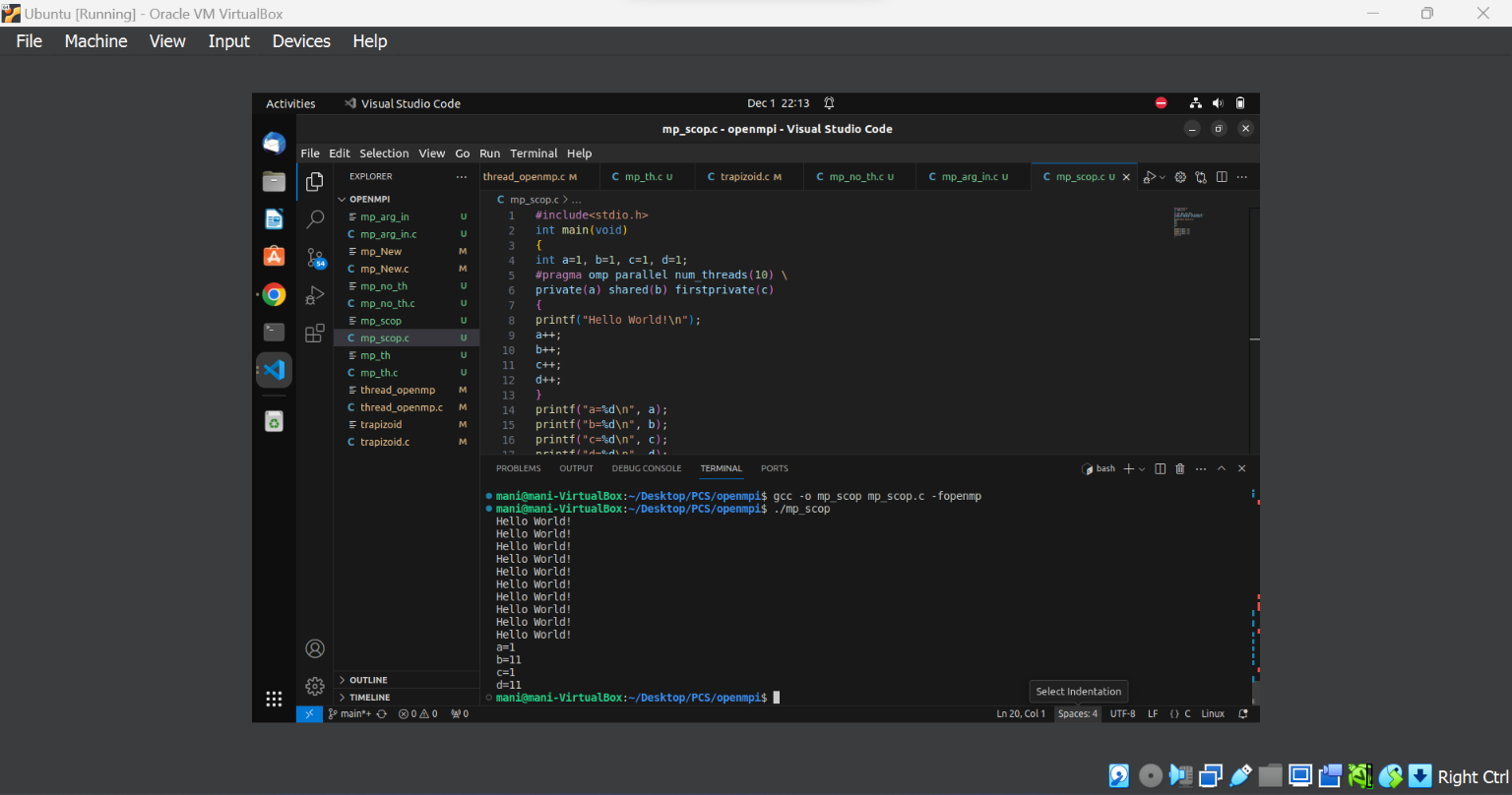
printf("c=%d\n", c);

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

return 0;

}

## OUTPUT:



## INFERENCE:

* Each thread in the parallel region will execute the "Hello World!" print statement. Since there are 10 threads **(num\_threads(10)),** we will see 10 "Hello World!" messages.
* The value of **a** outside the parallel region remains 1 because it was private to each thread inside the parallel region.
* The final value of **b** is the sum of the increments made by all threads (1 increment per thread\*10 threads).
* The value of **c** outside the parallel region remains 1 because it was firstprivate to each thread inside the parallel region.
* The final value of **d** is the sum of the increments made by all threads (1 increment per thread\*10 threads).

## ATOMIC, CRITICAL:

#include <stdio.h>

#include <omp.h>

int main() {

const int numIterations = 1000000;

int sharedVar = 0;

#pragma omp parallel for

for (int i = 0; i < numIterations; ++i) {

#pragma omp atomic

sharedVar++; // Atomic operation to increment sharedVar safely

// Use of 'if' construct to conditionally increment sharedVar

#pragma omp critical

if (i % 2 == 0)

sharedVar++;

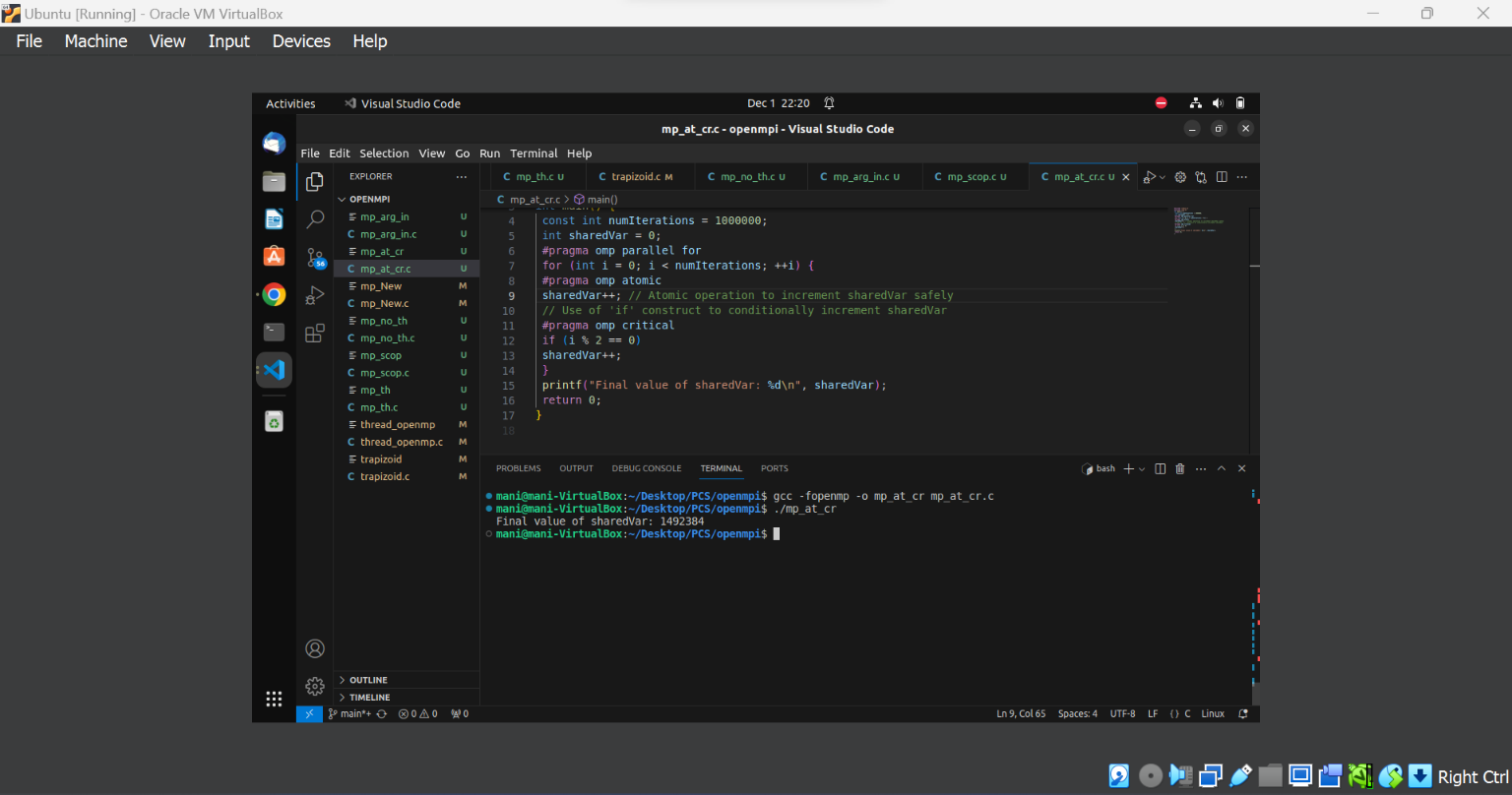
}

printf("Final value of sharedVar: %d\n", sharedVar);

return 0;

}

## OUTPUT:



## INFERENCE:

* #pragma omp atomic is used to ensure the increment operation on sharedVar is atomic, preventing data races when multiple threads concurrently update the variable.
* #pragma omp critical is used to create a critical section for the conditional increment, ensuring that only one thread at a time executes the code inside the critical section, avoiding potential race conditions.

## AREA OF A TRAPEZOID

## PROGRAM:

#include <stdio.h>

#include <omp.h>

double calculateTrapezoidArea(double base1, double base2, double height) { return 0.5 \* (base1 + base2) \* height;

}

int main() {

const int numTrapezoids = 1000000;

const double base1 = 2.0;

const double base2 = 5.0;

const double height = 3.0;

double totalArea = 0.0;

double startTime, endTime;

// Record start time

startTime = omp\_get\_wtime();

#pragma omp parallel for reduction(+:totalArea)

for (int i = 0; i < numTrapezoids; ++i) {

// Each thread calculates the area of its assigned trapezoid

double trapezoidArea = calculateTrapezoidArea(base1, base2, height);

// Sum up the areas using reduction clause

totalArea += trapezoidArea;

}

// Record end time

endTime = omp\_get\_wtime();

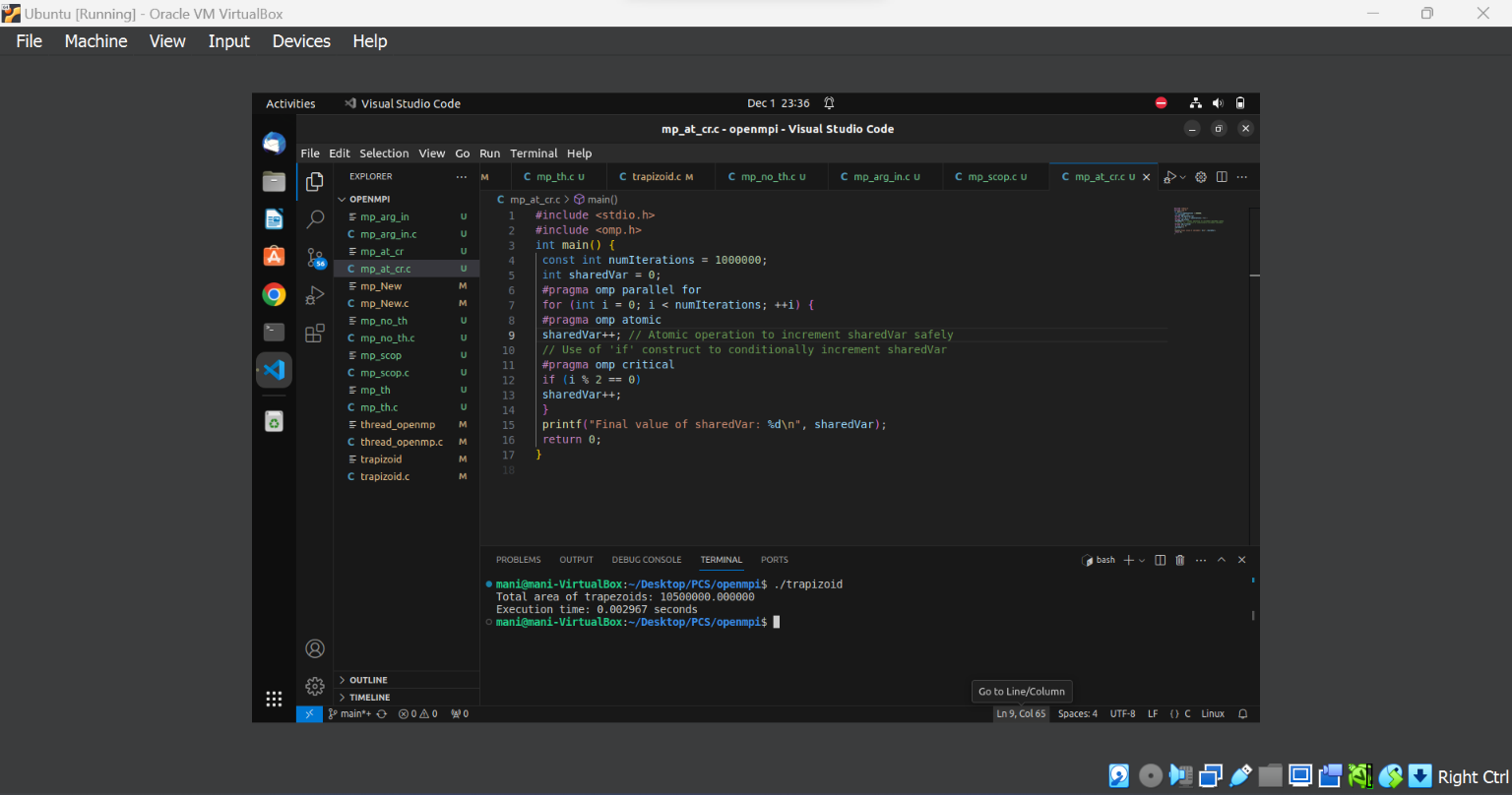
printf("Total area of trapezoids: %f\n", totalArea);

printf("Execution time: %f seconds\n", endTime - startTime);

return 0;

}

## OUTPUT:



## INFERENCE:

* Calculate the total area of a large number of trapezoids in parallel.
* The #pragma omp parallel for reduction(+:totalArea) directive is used to parallelize the loop, dividing the iterations among multiple threads.
* The reduction(+:totalArea) clause specifies that each thread should have its private copy of totalArea, and the final result should be obtained by summing up these private copies.
* The reduction(+:totalArea) clause ensures that the partial results from each thread are correctly combined using the addition (+) reduction operation.
* The use of reduction(+:totalArea) is essential to prevent race conditions and ensure the correctness of the final result.
* Without the reduction clause, multiple threads updating the shared totalArea simultaneously would lead to data races and incorrect results.

## SUM OF EVEN NUMBERS USING OPENMP:

## PROGRAM:

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

#define NUM\_THREADS 4

int main() {

int n = 100; // Replace with the desired number of even elements

int numbers[n];

// Initialize array with first n even numbers

for (int i = 0; i < n; i++) {

numbers[i] = 2 \* (i + 1); // Generate even numbers (2, 4, 6, ...)

}

long long totalSum = 0;

#pragma omp parallel num\_threads(NUM\_THREADS)

{

int threadID = omp\_get\_thread\_num();

int chunkSize = n / NUM\_THREADS;

int remainder = n % NUM\_THREADS;

int start = threadID \* chunkSize;

int end = (threadID == NUM\_THREADS - 1) ? (threadID + 1) \* chunkSize + remainder : (threadID + 1) \* chunkSize;

long long localSum = 0;

// Calculate local sum for each thread

for (int i = start; i < end; i++) {

localSum += numbers[i];

}

// Use critical section to update totalSum

#pragma omp critical

{

totalSum += localSum;

}

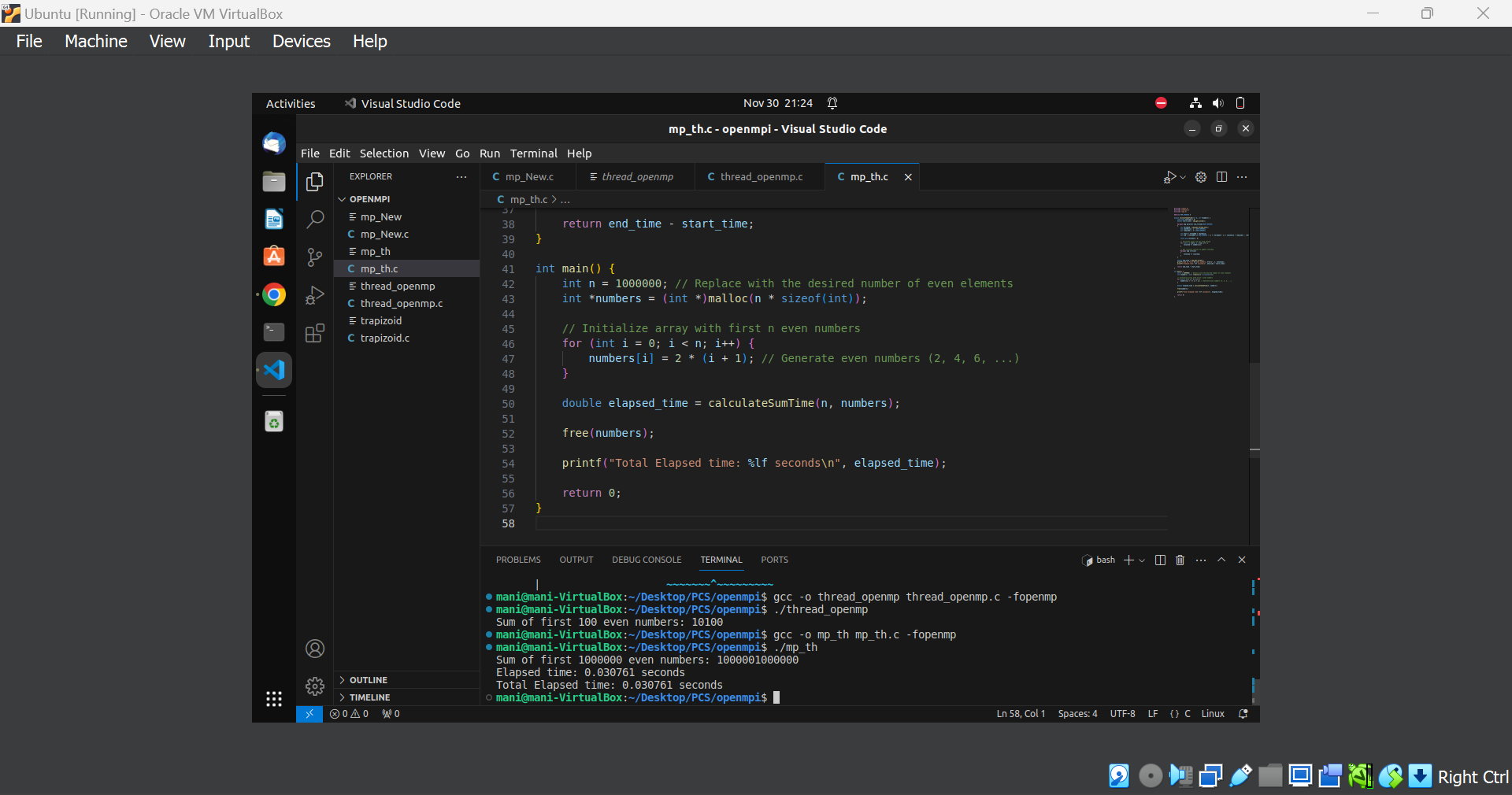
}

printf("Sum of first %d even numbers: %lld\n", n, totalSum);

return 0;

}

## OUTPUT:



## INFERENCE:

The choice between using threads and OpenMP for a sum of even numbers program depends on various factors, including the complexity of the program, ease of implementation, and specific requirements. Let's discuss the differences and considerations for both approaches.

## PROGRAM WITH THREADS:

IMPLEMENTATION:

- In a threaded program, you typically create a number of threads, and each thread is responsible for computing the sum of even numbers within a specific range.

- You may use synchronization mechanisms, such as mutexes or barriers, to coordinate access to shared resources like the sum variable.

## PROS:

1. Fine-Grained Control: Threads provide fine-grained control over the parallelization process, allowing you to explicitly manage thread creation, synchronization, and data sharing.

2. Versatility: Threads can be used for a wide range of parallel programming scenarios beyond simple loop parallelism.

CONS:

1. Complexity: Implementing and managing threads can be more complex than high-level constructs like OpenMP, especially for simple tasks.

2. Error-Prone: Manually managing synchronization can lead to errors, such as race conditions or deadlocks.

PROGRAM WITH OPENMP:

## IMPLEMENTATION:

- In an OpenMP program, you annotate the parallelizable loop with directives, and the compiler automatically generates the necessary code for parallel execution.

- The `#pragma omp parallel for` directive can be used to parallelize the loop.

PROS:

1. Ease of Use: OpenMP simplifies parallel programming by providing high-level directives that automatically handle many aspects of parallelization.

2. Readability: OpenMP code is often more concise and readable, making it easier to understand and maintain.

## CONS:

1. Less Control: OpenMP abstracts away many low-level details, providing less control compared to manual thread management.

2. Limited to Loop Parallelism: While OpenMP is great for loop parallelism, it may not be as flexible for more complex parallelization scenarios.

## EFFICIENCY CONSIDERATIONS:

Workload Size: For small workloads, the overhead of managing threads or OpenMP parallelism may outweigh the benefits. It's crucial to consider the size of the problem to determine if parallelization is worthwhile.

Ease of Use: If simplicity and ease of implementation are essential, OpenMP is often a better choice due to its high-level directives.

Scalability: For large-scale parallelism, threads may offer more fine-grained control and scalability, especially when dealing with complex algorithms.