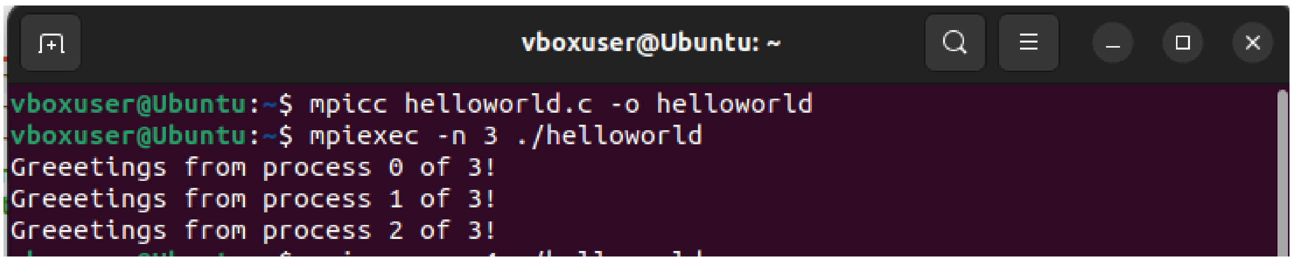
PARALLEL COMPUTING SYSTEM ASSIGNMENT

I.M. MELANSHIA VIOLET
66CG013
917723CS005

MPI Programming

Hello World

```
#include<stdio.h>
#include<string.h>
#include<mpi.h>
const int MAX_STRING=100;
int main(void)
{
        char greeting[MAX_STRING];
        int comm_sz;
        int my_rank;
        MPI_Init(NULL, NULL);
        MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
        MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
        if(my_rank!=0)
        {
                sprintf(greeting, "Greeetings from process %d of %d!", my_rank, comm_sz);
                MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
        }
        else
        {
                printf("Greeetings from process %d of %d!\n", my_rank, comm_sz);
                for(int q=1; q<comm_sz; q++)
                {
                        MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                        printf("%s\n", greeting);
                }
        }
        MPI_Finalize();
        return 0;
}
```



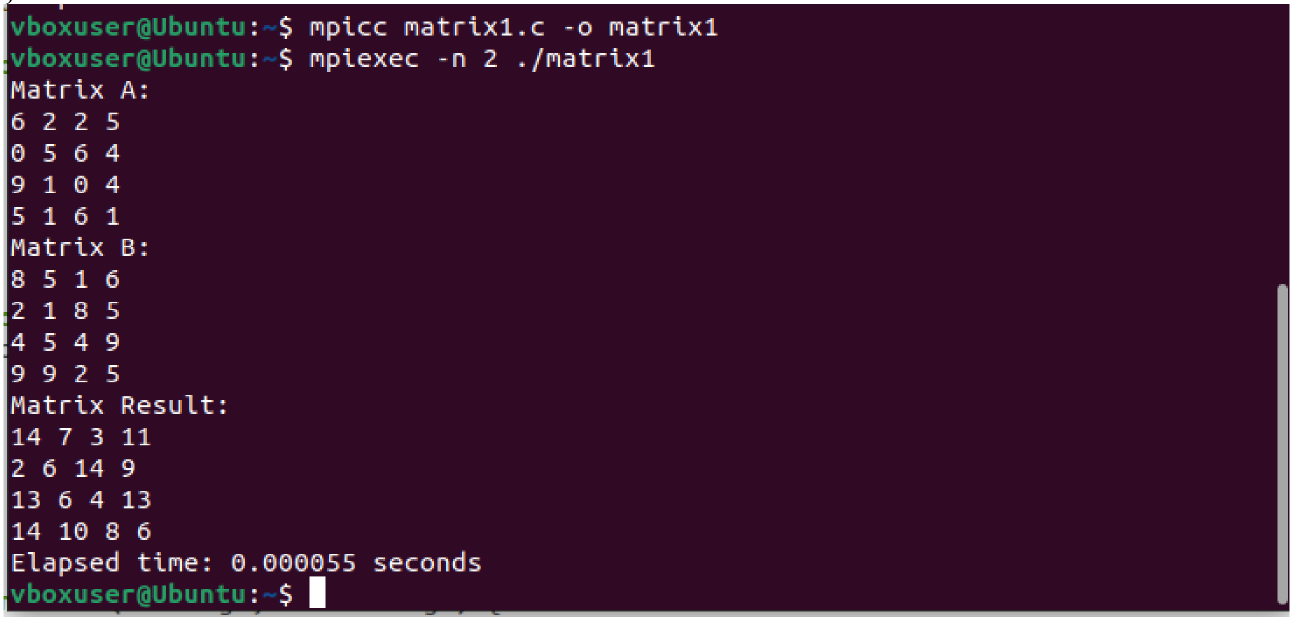
```
vboxuser@Ubuntu:~$ mpicc helloworld.c -o helloworld
vboxuser@Ubuntu:~$ mpiexec -n 3 ./helloworld
Greeetings from process 0 of 3!
Greeetings from process 1 of 3!
Greeetings from process 2 of 3!
```

Inference

Displaying the message Greetings. The program creates multiple MPI processes, and each non-root process sends a greeting message to the root process (rank 0). The root process receives these messages and prints all the greetings.

Matrix addition using MPI Scatter and MPI Gather

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <mpi.h>

#define MATRIX_SIZE 4

void initializeMatrix(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;  // Initialize with random values (modify as needed)
        }
    }
}

void matrixAddition(int local_matrixA[MATRIX_SIZE][MATRIX_SIZE], int
local_matrixB[MATRIX_SIZE][MATRIX_SIZE], int
local_result[MATRIX_SIZE][MATRIX_SIZE]) {
    for (int i = 0; i < MATRIX_SIZE; i++) {
        for (int j = 0; j < MATRIX_SIZE; j++) {
            local_result[i][j] = local_matrixA[i][j] + local_matrixB[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);

    srand(time(NULL));  // Seed for random number generation

    int matrixA[MATRIX_SIZE][MATRIX_SIZE];
    int matrixB[MATRIX_SIZE][MATRIX_SIZE];
    int local_matrixA[MATRIX_SIZE][MATRIX_SIZE];
    int local_matrixB[MATRIX_SIZE][MATRIX_SIZE];
    int local_result[MATRIX_SIZE][MATRIX_SIZE];

    if (my_rank == 0) {
        // Initialize matrices with random values
        initializeMatrix(matrixA);
        initializeMatrix(matrixB);
    }

    double start_time, end_time;

    if (my_rank == 0) {
        start_time = MPI_Wtime(); // Start measuring execution time
    }

    MPI_Scatter(matrixA, MATRIX_SIZE * MATRIX_SIZE / world_size, MPI_INT,
local_matrixA,
                MATRIX_SIZE * MATRIX_SIZE / world_size, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Scatter(matrixB, MATRIX_SIZE * MATRIX_SIZE / world_size, MPI_INT, local_matrixB,
                MATRIX_SIZE * MATRIX_SIZE / world_size, MPI_INT, 0, MPI_COMM_WORLD);

    matrixAddition(local_matrixA, local_matrixB, local_result);

    int (*final_result)[MATRIX_SIZE] = NULL;

    if (my_rank == 0) {
        final_result = (int (*)[MATRIX_SIZE])malloc(MATRIX_SIZE * MATRIX_SIZE *
sizeof(int));
    }

    MPI_Gather(local_result, MATRIX_SIZE * MATRIX_SIZE / world_size, MPI_INT,
final_result,
                MATRIX_SIZE * MATRIX_SIZE / world_size, MPI_INT, 0, MPI_COMM_WORLD);

    if (my_rank == 0) {
        end_time = MPI_Wtime(); // Stop measuring execution time
        printf("Matrix A:\n");
        for (int i = 0; i < MATRIX_SIZE; i++) {
            for (int j = 0; j < MATRIX_SIZE; j++) {
                printf("%d ", matrixA[i][j]);
            }
            printf("\n");
        }

        printf("Matrix B:\n");
        for (int i = 0; i < MATRIX_SIZE; i++) {
            for (int j = 0; j < MATRIX_SIZE; j++) {
                printf("%d ", matrixB[i][j]);
            }
            printf("\n");
        }

        printf("Matrix Result:\n");
        for (int i = 0; i < MATRIX_SIZE; i++) {
            for (int j = 0; j < MATRIX_SIZE; j++) {
                printf("%d ", final_result[i][j]);
            }
            printf("\n");
        }

        printf("Elapsed time: %f seconds\n", end_time - start_time);

        free(final_result);
    }
    MPI_Finalize();

    return 0;
}
```



```
vboxuser@Ubuntu:~$ mpicc matrix1.c -o matrix1
vboxuser@Ubuntu:~$ mpiexec -n 2 ./matrix1
Matrix A:
6 2 2 5
0 5 6 4
9 1 0 4
5 1 6 1
Matrix B:
8 5 1 6
2 1 8 5
4 5 4 9
9 9 2 5
Matrix Result:
14 7 3 11
2 6 14 9
13 6 4 13
14 10 8 6
Elapsed time: 0.000055 seconds
vboxuser@Ubuntu:~$
```

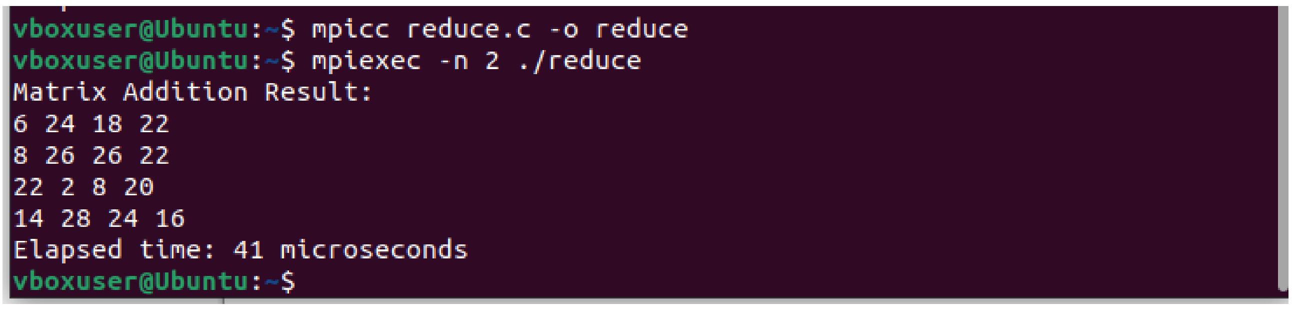
Inference

- 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

```
#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[MA-
TRIX_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;
}
```



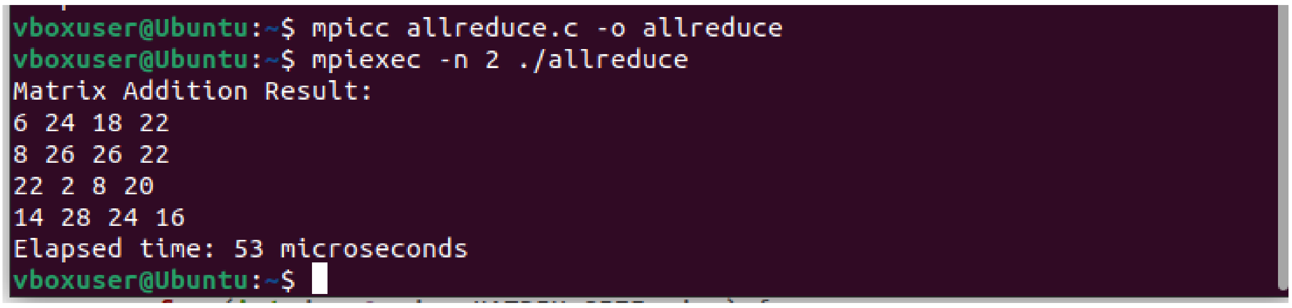
```
vboxuser@Ubuntu:~$ mpicc reduce.c -o reduce
vboxuser@Ubuntu:~$ mpiexec -n 2 ./reduce
Matrix Addition Result:
6 24 18 22
8 26 26 22
22 2 8 20
14 28 24 16
Elapsed time: 41 microseconds
vboxuser@Ubuntu:~$
```

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

```
#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;
}
```



```
vboxuser@Ubuntu:~$ mpicc allreduce.c -o allreduce
vboxuser@Ubuntu:~$ mpiexec -n 2 ./allreduce
Matrix Addition Result:
6 24 18 22
8 26 26 22
22 2 8 20
14 28 24 16
Elapsed time: 53 microseconds
vboxuser@Ubuntu:~$
```

Inference

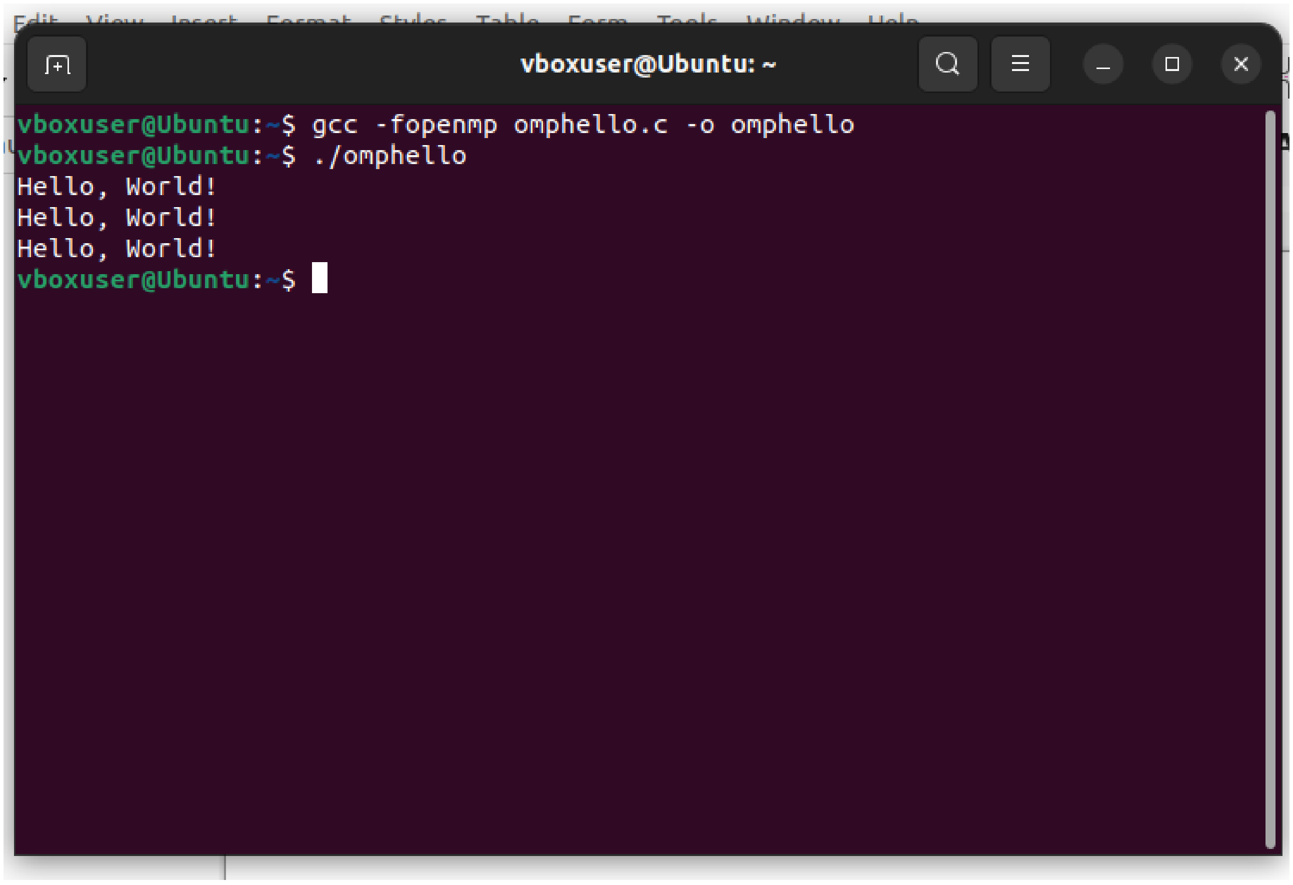
- MPI All reduce combines the reduction and broadcast steps in a single collective operation.
- Every process receives the final result directly after the operation, eliminating the need for a separate gathering step.
- MPI_Allreduce might have a higher overhead than MPI_Reduce because it involves more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

Open MP Programming

Simple Programs

Hello World

```
#include<stdio.h>
int main(void)
{
        #pragma omp parallel
        {
                printf("Hello, World!\n");
        }
        return 0;
}
```



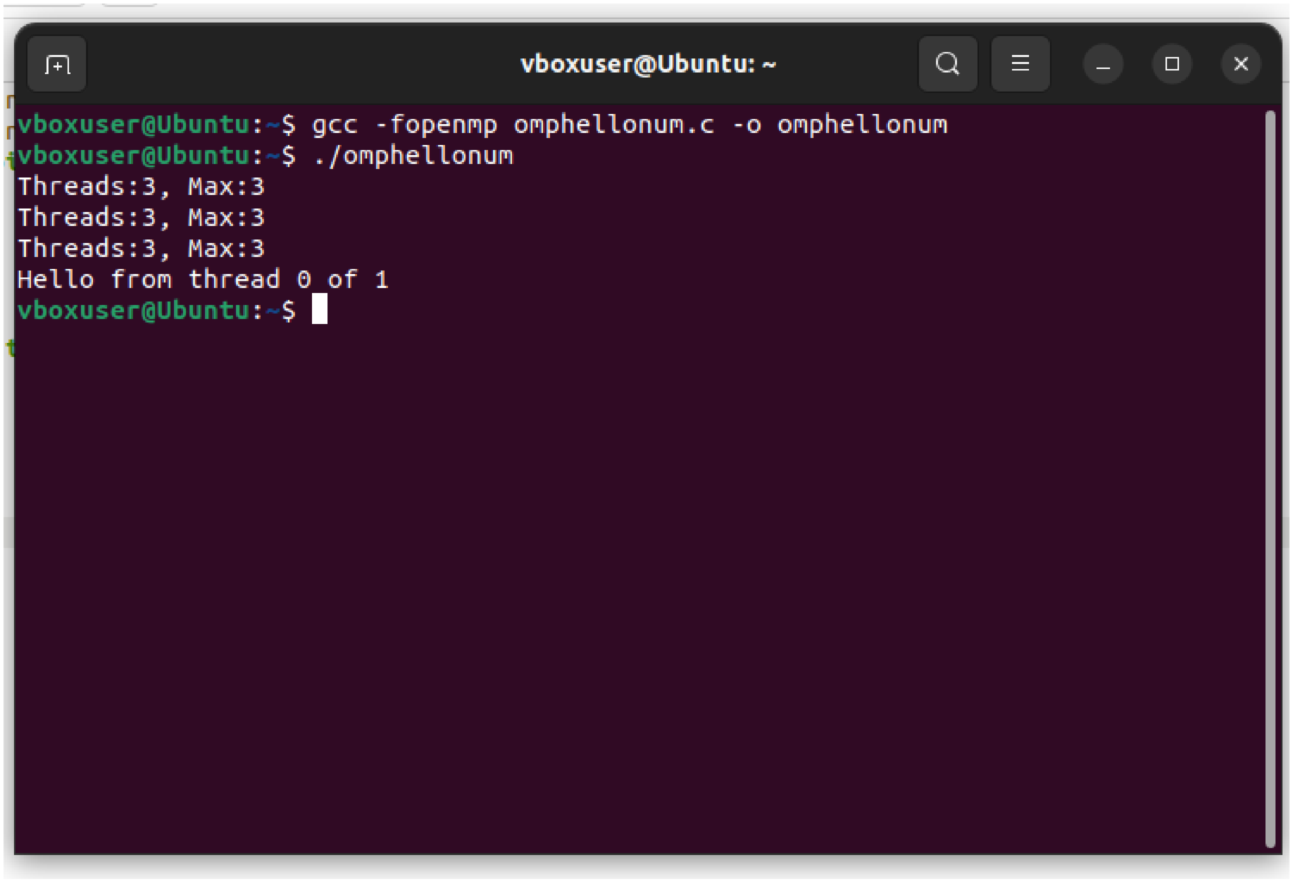
```
vboxuser@Ubuntu:~$ gcc -fopenmp omphello.c -o omphello
vboxuser@Ubuntu:~$ ./omphello
Hello, World!
Hello, World!
Hello, World!
vboxuser@Ubuntu:~$
```

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;
}
```



```
vboxuser@Ubuntu:~$ gcc -fopenmp omphellonum.c -o omphellonum
vboxuser@Ubuntu:~$ ./omphellonum
Threads:3, Max:3
Threads:3, Max:3
Threads:3, Max:3
Hello from thread 0 of 1
vboxuser@Ubuntu:~$
```

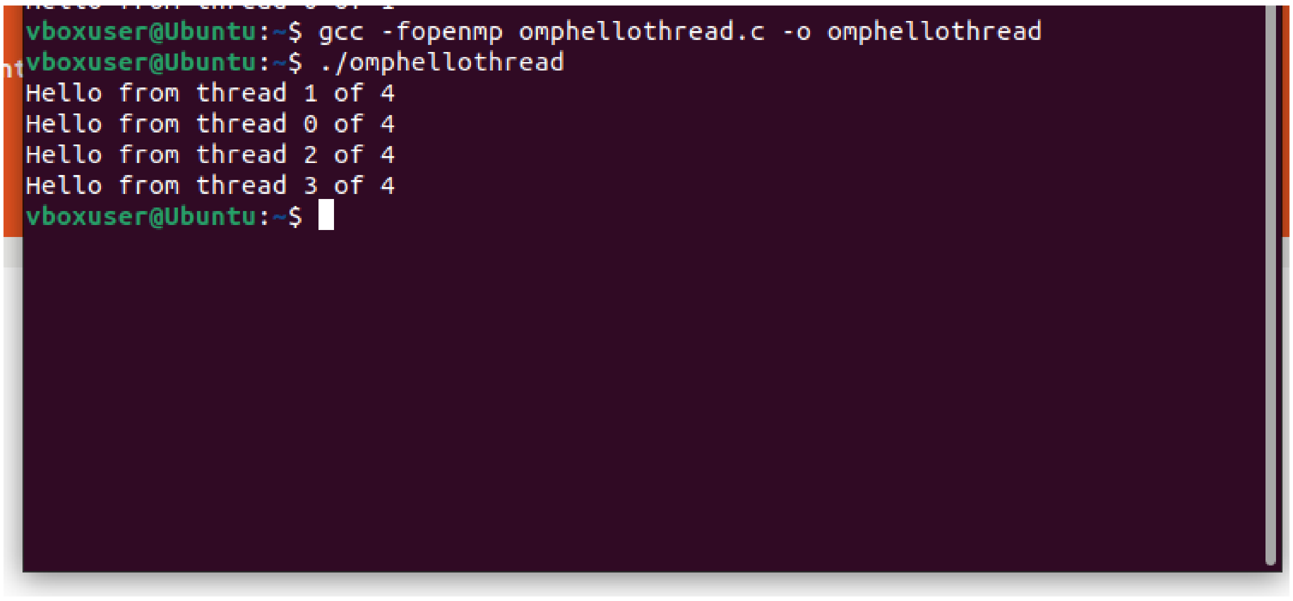
Inference

- 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

```
#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;
}
```

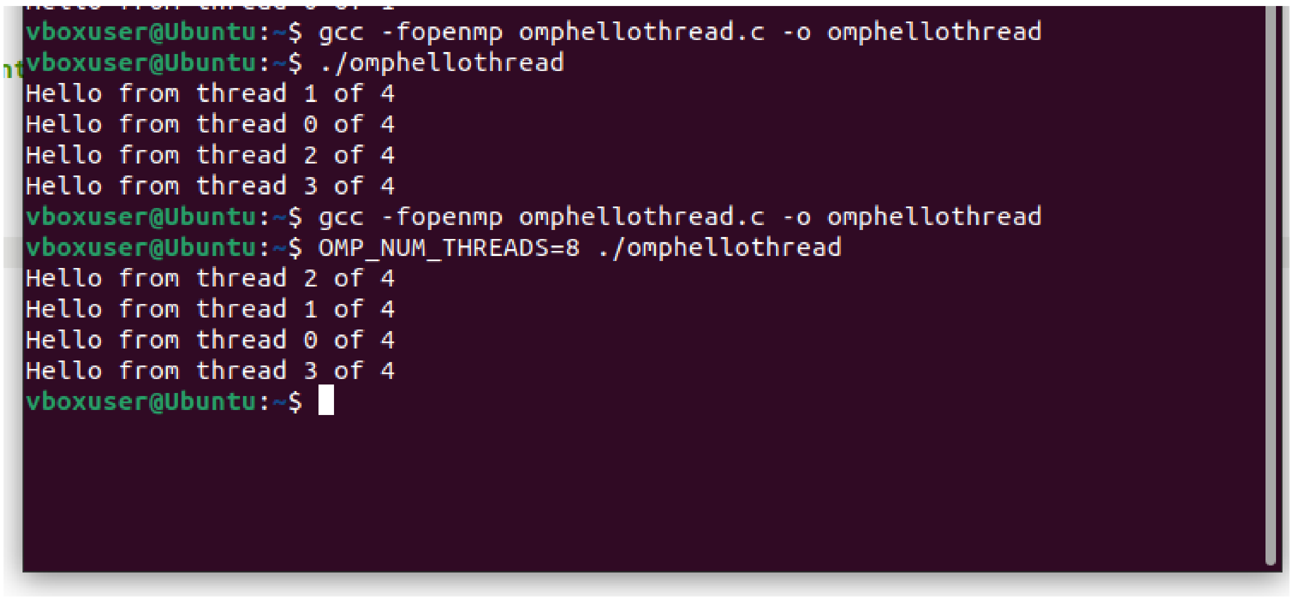


```
vboxuser@Ubuntu:~$ gcc -fopenmp omphellothread.c -o omphellothread
vboxuser@Ubuntu:~$ ./omphellothread
Hello from thread 1 of 4
Hello from thread 0 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4
vboxuser@Ubuntu:~$
```

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.



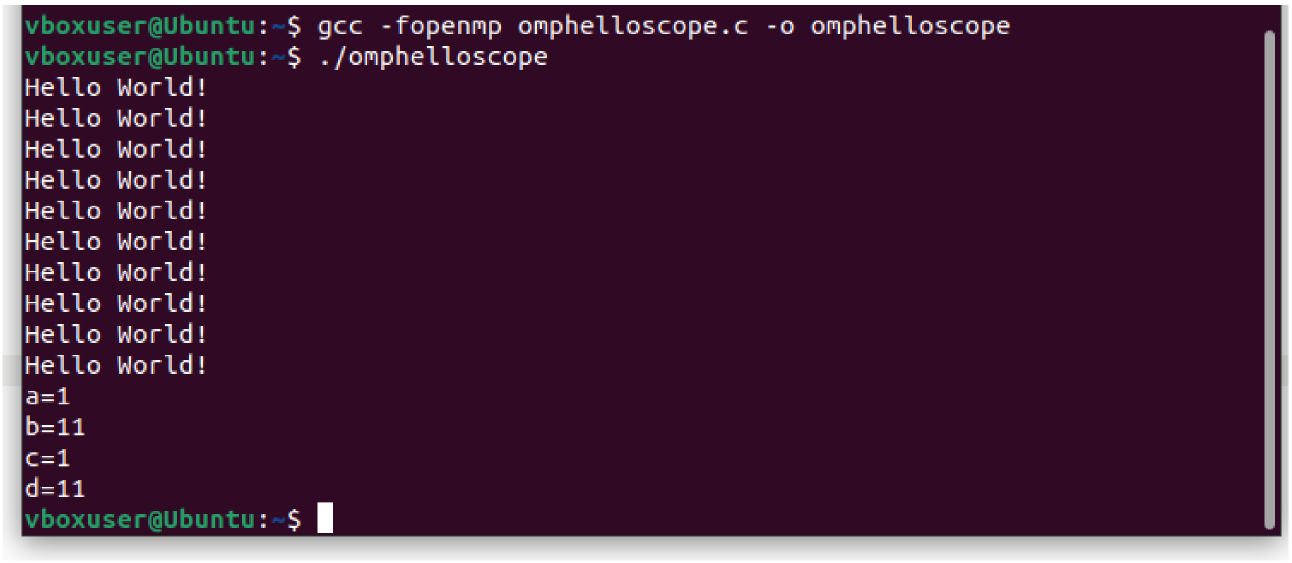
```
vboxuser@Ubuntu:~$ gcc -fopenmp omphellothread.c -o omphellothread
vboxuser@Ubuntu:~$ ./omphellothread
Hello from thread 1 of 4
Hello from thread 0 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4
vboxuser@Ubuntu:~$ gcc -fopenmp omphellothread.c -o omphellothread
vboxuser@Ubuntu:~$ OMP_NUM_THREADS=8 ./omphellothread
Hello from thread 2 of 4
Hello from thread 1 of 4
Hello from thread 0 of 4
Hello from thread 3 of 4
vboxuser@Ubuntu:~$
```

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;
}
```



```
vboxuser@Ubuntu:~$ gcc -fopenmp omphelloscope.c -o omphelloscope
vboxuser@Ubuntu:~$ ./omphelloscope
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
a=1
b=11
c=1
d=11
vboxuser@Ubuntu:~$
```

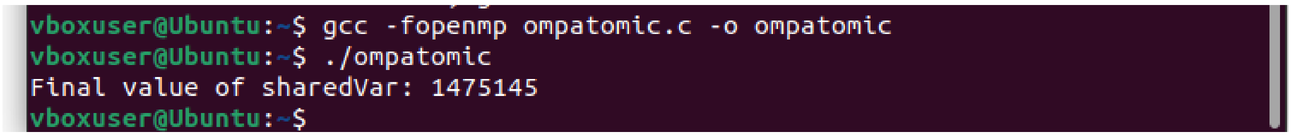
Inference

- 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;
}
```



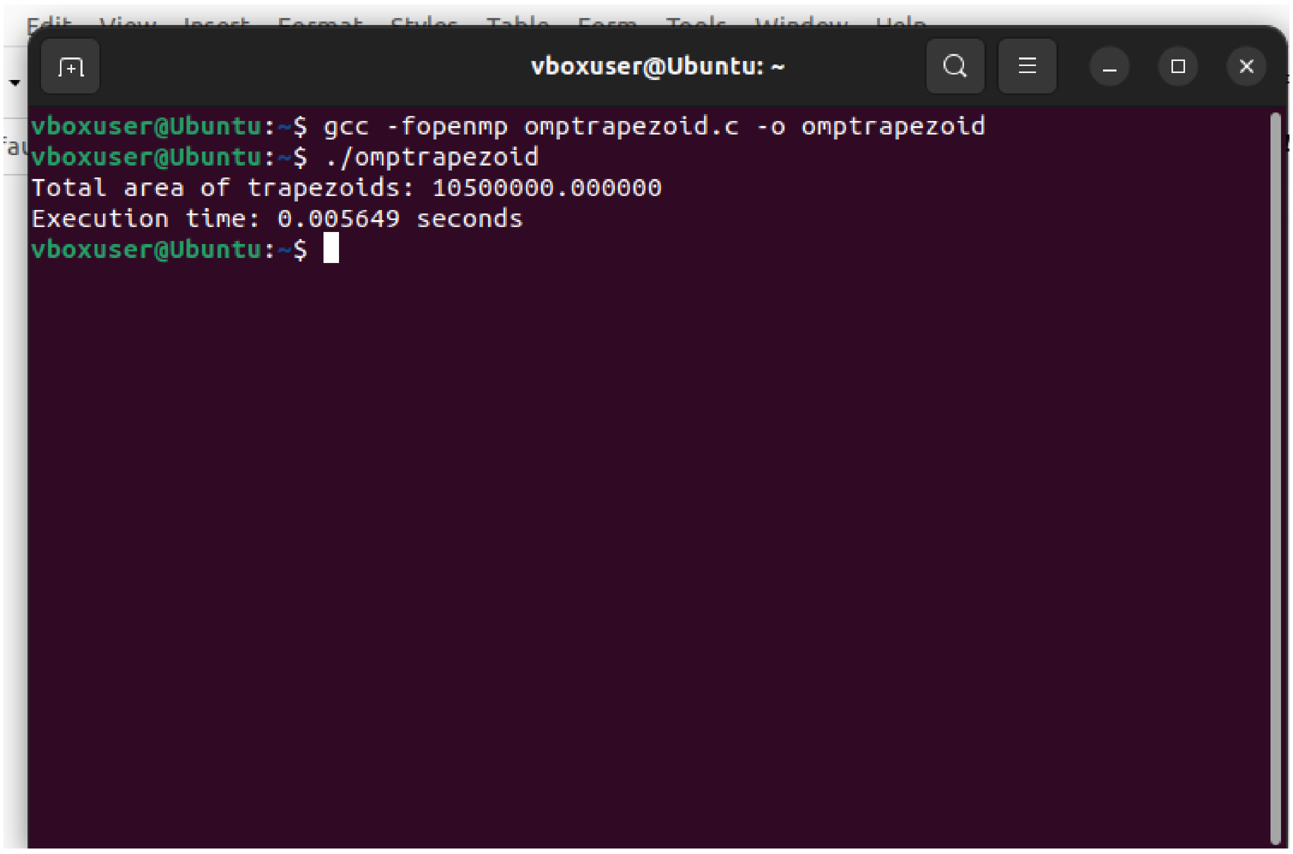
```
vboxuser@Ubuntu:~$ gcc -fopenmp ompatomic.c -o ompatomic
vboxuser@Ubuntu:~$ ./ompatomic
Final value of sharedVar: 1475145
vboxuser@Ubuntu:~$
```

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

```
#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;
}
```



```
vboxuser@Ubuntu:~$ gcc -fopenmp omptrapezoid.c -o omptrapezoid
vboxuser@Ubuntu:~$ ./omptrapezoid
Total area of trapezoids: 10500000.000000
Execution time: 0.005649 seconds
vboxuser@Ubuntu:~$
```

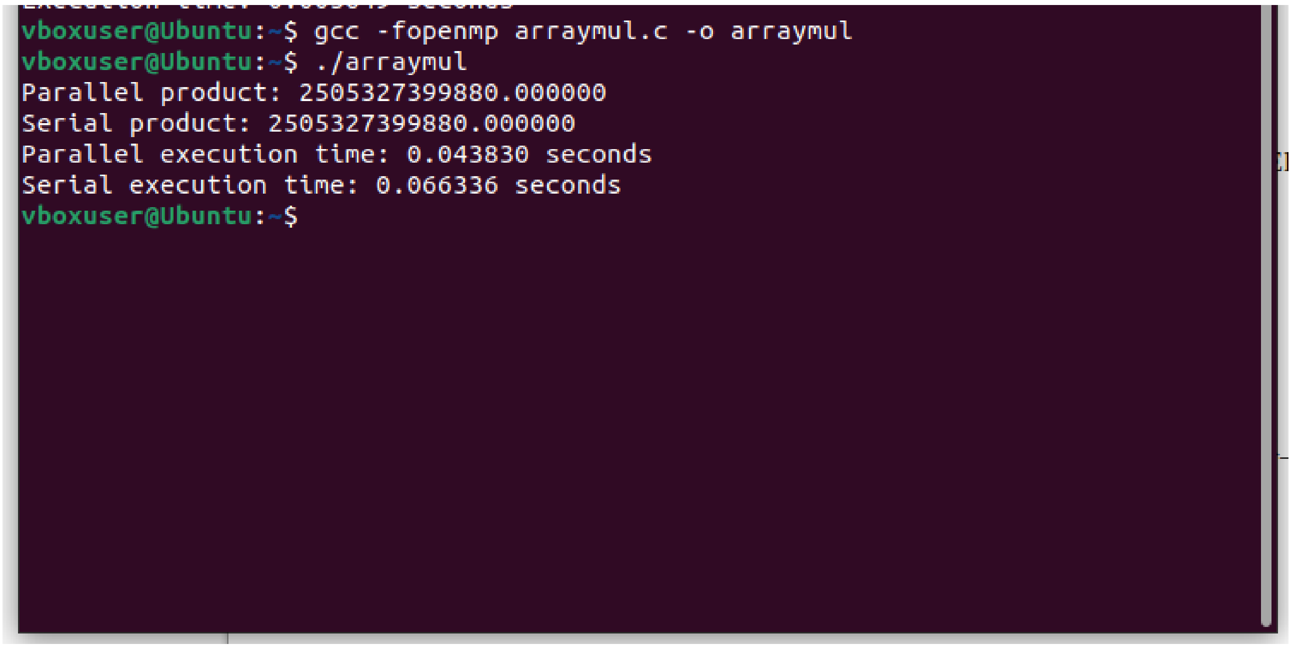
Inference

- 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.

Multiplication of array size with random positive numbers

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>
#define ARRAY_SIZE 10000000 // 100 Million
double array1[ARRAY_SIZE];
double array2[ARRAY_SIZE];
double product_parallel = 0.0;
double product_serial = 0.0;
void initialize_arrays() {
   for (int i = 0; i < ARRAY_SIZE; i++) {
      array1[i] = (double)(rand() % 1000 + 1); // Random positive numbers
      array2[i] = (double)(rand() % 1000 + 1);
   }
}
// Parallel Calculation
void calculate_product_parallel() {
   #pragma omp parallel for reduction(+:product_parallel)
   for (int i = 0; i < ARRAY_SIZE; i++) {
      product_parallel += array1[i] * array2[i];
   }
}
// Serial Calculation
double calculate_product_serial() {
   double local_product = 0.0;
   for (int i = 0; i < ARRAY_SIZE; i++) {
      local_product += array1[i] * array2[i];
   }
   return local_product;
}
int main() {
   srand(time(NULL));
   // Initialize arrays with random values
   initialize_arrays();
   clock_t start_time, end_time;
   // Measure execution time for parallel calculation
   start_time = clock();
   // Calculate the product in parallel
   calculate_product_parallel();
   end_time = clock();
   double parallel_execution_time = (double)(end_time - start_time) / CLOCKS_PER_SEC;
   // Measure execution time for serial calculation
   start_time = clock();
   // Calculate the product in serial
   product_serial = calculate_product_serial();
   end_time = clock();
   double serial_execution_time = (double)(end_time - start_time) / CLOCKS_PER_SEC;

   printf("Parallel product: %lf\n", product_parallel);
   printf("Serial product: %lf\n", product_serial);
```

```
    printf("Parallel execution time: %lf seconds\n", parallel_execution_time);
    printf("Serial execution time: %lf seconds\n", serial_execution_time);

    return 0;
}
```



```
vboxuser@Ubuntu:~$ gcc -fopenmp arraymul.c -o arraymul
vboxuser@Ubuntu:~$ ./arraymul
Parallel product: 2505327399880.000000
Serial product: 2505327399880.000000
Parallel execution time: 0.043830 seconds
Serial execution time: 0.066336 seconds
vboxuser@Ubuntu:~$
```

Inference

- Calculate the dot product of two arrays in both parallel and serial fashion.
- Measure and compare the execution times for parallel and serial calculations.
- The execution time in Open MP for the multiplication of array size using random positive numbers is much efficient compared to the parallelisation program using threads.
- OpenMP provides a higher-level, directive-based approach to parallel programming, making it more accessible and easier to implement parallelization.
- OpenMP uses compiler directives (e.g., #pragma omp) to specify parallel regions, reducing the need for manual thread creation and synchronization.
- The implicit parallelism in OpenMP allows developers to express parallelism without explicitly managing threads.
- The dot product is calculated in parallel using OpenMP with the #pragma omp parallel for reduction(+:product_parallel) directive.
- Each thread calculates a portion of the dot product, and the reduction clause ensures the correct summation.
- The program aims to showcase the efficiency gains achieved through parallelization using OpenMP for a dot product calculation. The parallel version is expected to demonstrate faster execution, especially for large array sizes, leveraging multiple threads for parallel computation.

CUDA Programming

In CUDA Programming which would we prefer either block or thread?

The choice between using more threads or more blocks depends on the nature of the algorithm and the characteristics of the problem trying to solve.

Thread:

- A thread is the smallest unit of execution in a CUDA program.
- Threads are organized into blocks, and each thread has a unique identifier called a thread ID.
- Threads within the same block can cooperate and communicate through shared memory.
- Threads are suitable for tasks that can be parallelized at a fine-grained level.

Block:

- A block is a group of threads that can be scheduled and executed together on a streaming multiprocessor (SM) on the GPU.
- Threads within the same block can synchronize and communicate through shared memory.
- Blocks are suitable for tasks that can be parallelized at a coarser level.

Difference between block and thread in working architecture

Thread:

- Basic Unit of Execution: A thread is the smallest unit of execution in a CUDA program. Each thread represents a single instance of the code that will be executed in parallel.
- Thread ID: Each thread within a GPU has a unique identifier known as a thread ID. This ID is often used to determine the data or task that a specific thread will operate on.
- Parallel Execution: Threads are designed to execute code concurrently, allowing for parallel processing of data.
- Threads within a block are executed concurrently on the GPU. The GPU's architecture is designed to efficiently handle a large number of threads running in parallel.

Block:

- Group of Threads: A block is a collection of threads that can be scheduled and executed together on a streaming multiprocessor (SM) of the GPU.
- Shared Memory: Threads within the same block can share data through shared memory, allowing for efficient communication and collaboration between threads in the same block.
- Scheduling Unit: The block is the unit that is scheduled on an SM, and the threads within a block are scheduled to run on the available processing cores within that SM.
- Blocks are scheduled to run on streaming multiprocessors (SMs). The SMs execute the blocks in a way that optimizes resource utilization and throughput.

Which is best according to performance metrics either thread or block?

Thread-Level Parallelism (TLP):
Advantages:

- Fine-grained parallelism.
- Well-suited for data-parallel tasks where individual elements can be processed independently.

Considerations:

- Large numbers of threads can lead to better utilization of GPU resources.

Block-Level Parallelism (BLP):
Advantages:

- Coarser parallelism.
- Threads within a block can cooperate and share data through shared memory.

- Well-suited for tasks where collaboration between threads is essential.

Considerations:

- Limited shared memory per block may need to be efficiently utilized.
- Synchronization and coordination between threads in a block can be important.

The best configuration is problem-dependent, and achieving optimal performance often requires a balance between thread-level and block-level parallelism, efficient use of shared memory, and careful consideration of memory access patterns.

Provide the applications which are best in thread and block

Thread-Level Parallelism (TLP):
Data-Parallel Tasks:
Example Applications:

- Image processing (e.g., pixel-level operations).
- Signal processing (e.g., per-sample operations).
- Matrix operations (e.g., element-wise operations).

Parallelism at Fine Granularity:
Example Applications:

- Parallel reduction tasks (e.g., summing elements of an array).
- Element-wise operations on large arrays.
- Monte Carlo simulations.

Block-Level Parallelism (BLP):
Cooperative Tasks:
Example Applications:

- Parallel reduction within a block where threads need to cooperate.
- Histogram computation within a block.
- Parallel reduction followed by block-wise results.

Shared Memory Communication:
Example Applications:

- Stencil-based computations where neighboring elements' values are needed.
- Parallel reduction with intermediate results stored in shared memory.