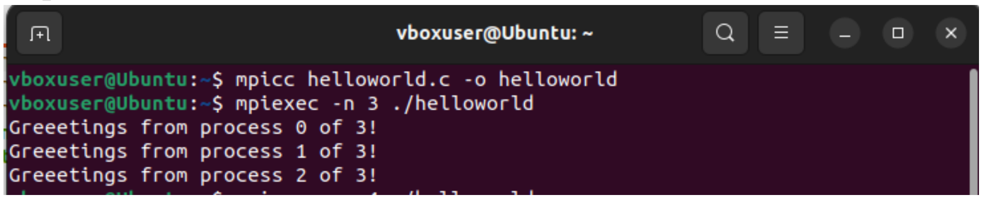
PARALLEL COMPUTING SYSTEM ASSIGNMENT

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;
}
```

Output:



```
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!
```

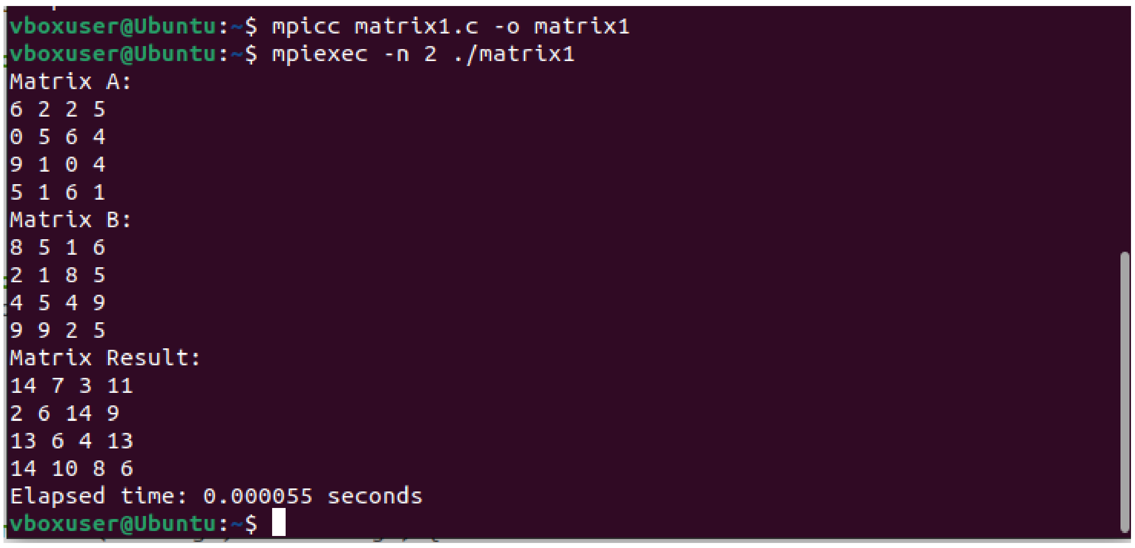
Inference:

The program initiates several MPI processes, where each non-root process transmits a greeting message to the root process (rank 0). The root process, in turn, collects and displays all the received 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;
 }
```

Output:



```
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:~$
```

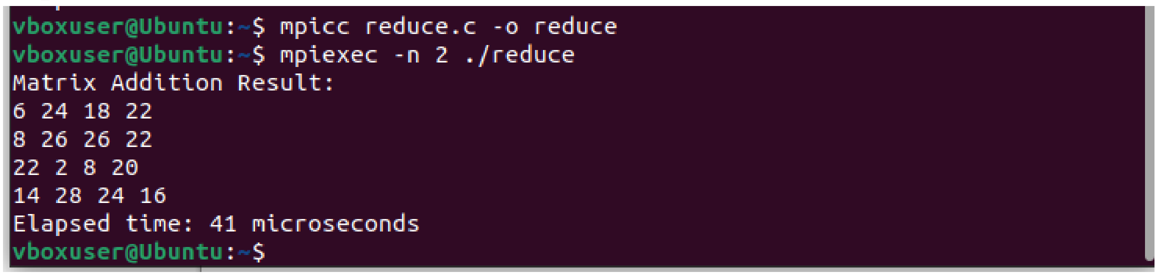
Inference

Enhancing execution efficiency, MPI Scatter and MPI Gather play a pivotal role in reducing the runtime. The code adeptly divides matrices into manageable chunks, distributing them across multiple processes through MPI. This strategic partitioning enables parallel computation, empowering each process to handle its dedicated matrix segment independently. Notably, matrix addition, being embarrassingly parallel, allows for concurrent computation of individual result matrix elements. MPI facilitates this parallelization, efficiently distributing the workload among available processes. The seamless integration of MPI_Scatter and MPI_Gather further streamlines data distribution and consolidation, effectively minimizing communication overhead and contributing to overall performance optimization.

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;
}
```

Output:



```
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

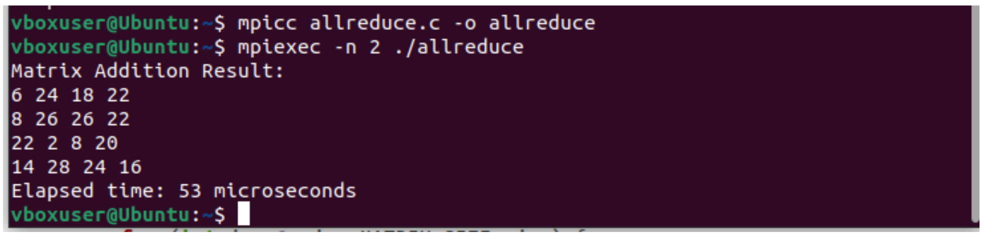
The program showcases parallel matrix addition through MPI, effectively distributing the workload across multiple processes. Utilizing MPI_Bcast ensures the streamlined distribution of matrices to all processes, optimizing efficiency. The incorporation of MPI_Reduce facilitates the gathering and summation of local results on the root process. Additionally, the measurement of elapsed time offers valuable insights into the execution time of the parallelized matrix addition, providing a comprehensive overview of the program's performance.

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;
}
```

Output:



```
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:~$
```

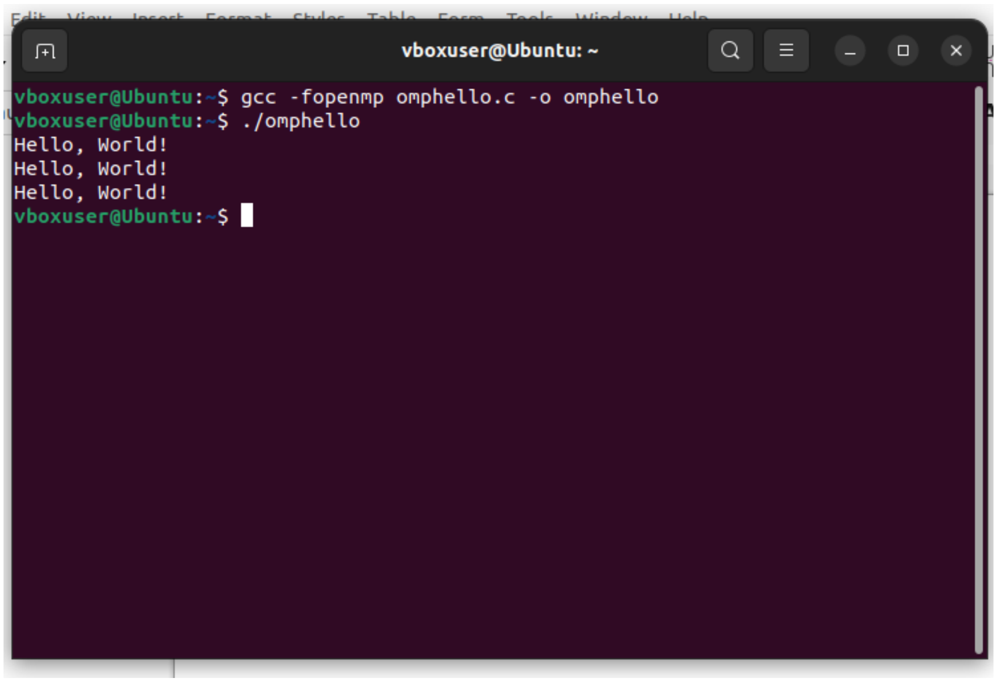
Inference:

MPI Allreduce streamlines the reduction and broadcast steps into a singular collective operation, ensuring that each process promptly receives the final result without necessitating a separate gathering phase. However, it's noteworthy that MPI_Allreduce might incur higher overhead compared to MPI_Reduce due to the increased inter-process communication involved in this combined_operation.

Open MP Programming
Simple Programs

Hello World

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



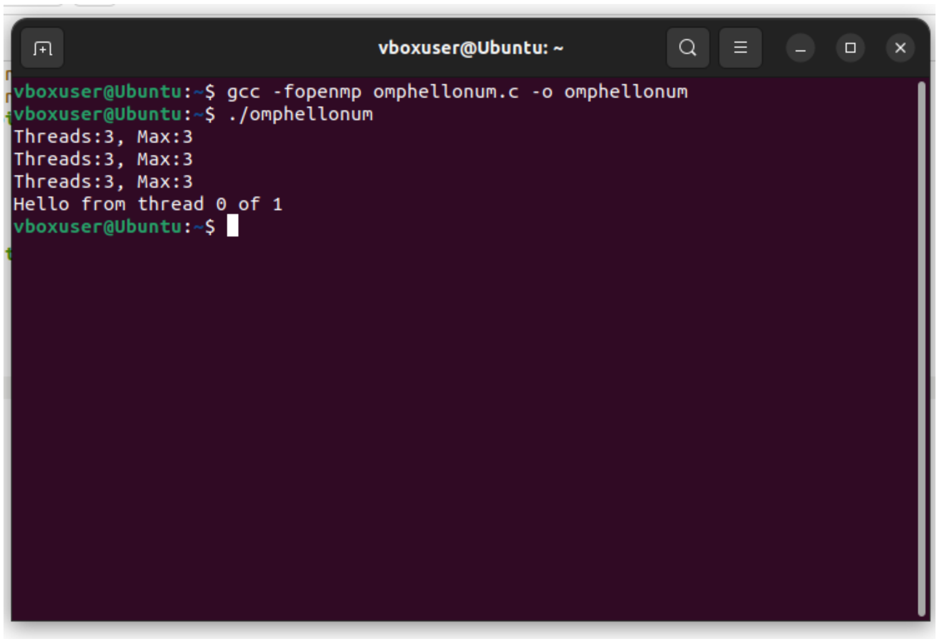
```
vboxuser@Ubuntu:~$ gcc -fopenmp omphello.c -o omphello
vboxuser@Ubuntu:~$ ./omphello
Hello, World!
Hello, World!
Hello, World!
vboxuser@Ubuntu:~$
```

Inference:

By compiling and executing the program with the OpenMP pragma parallel, the display of "Hello World" has been achieved.

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;
}
```



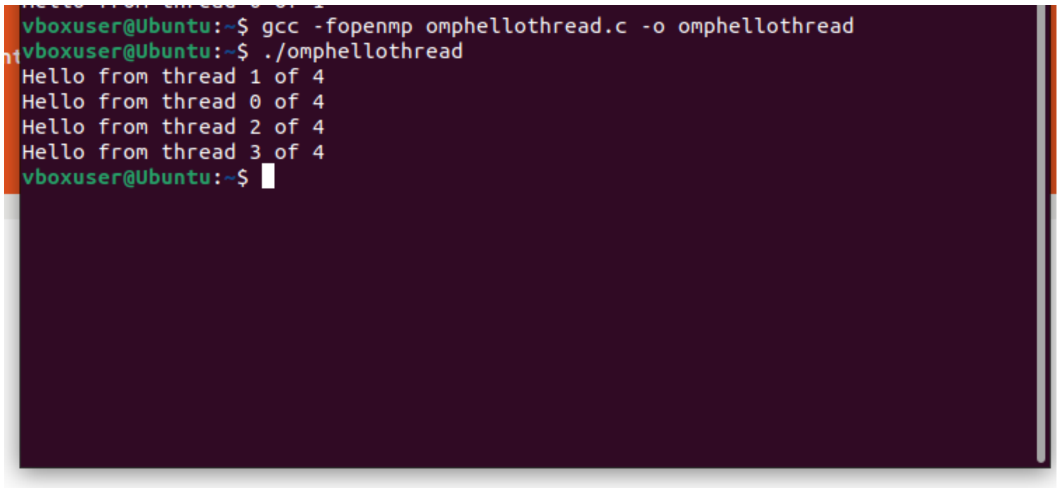
```
vboxuser@Ubuntu:~$ gcc -fopenmp omphellonum.c -o omphellonum
vboxuser@Ubuntu:~$ ./omphellonum
Threads:3, Max:3
Threads:3, Max:3
Threads:3, Max:3
Hello from thread 0 of 1
vboxuser@Ubuntu:~$
```

Inference:

This program utilizes OpenMP to provide insights into the number of threads in a parallel region and subsequently invokes the "say_hello" function within that region to display a "Hello" message from each thread. The #pragma omp parallel directive initiates a team of threads, and the enclosed block is executed by all threads in the team. The printf statement within the parallel region conveys information about the thread count and the maximum thread count, offering valuable details about the parallel execution environment configuration.

Displaying the threads within the program or compilation

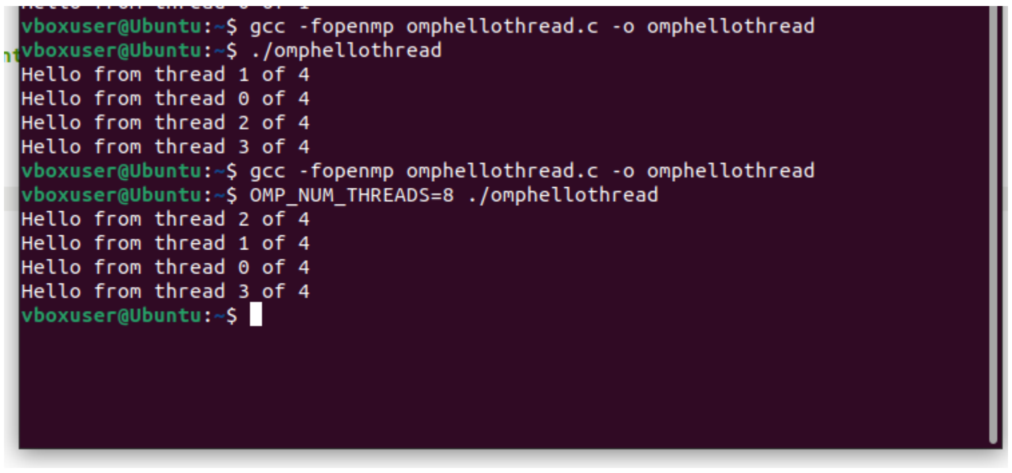
```
#include<stdio.h>
#include<omp.h>
void say_hello(void)
{
        int myrank=omp_get_thread_num();
        int threadcount=omp_get_num_threads();
        printf("Hello from thread %d of %d\n", myrank,threadcount);
}
int main(int argc, char* argv[])
{
        omp_set_num_threads(4);
        #pragma 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:

Utilizing OpenMP, the program employs the omp_get_thread_num() function to obtain the thread number within the team for each thread, and the omp_get_num_threads() function retrieves the total number of threads in the team. Explicitly setting the number of threads to 4, the anticipated output will likely display "Hello" messages from each of the 4 threads. However, the output's order may not be deterministic due to the implementation-dependent scheduling of threads.



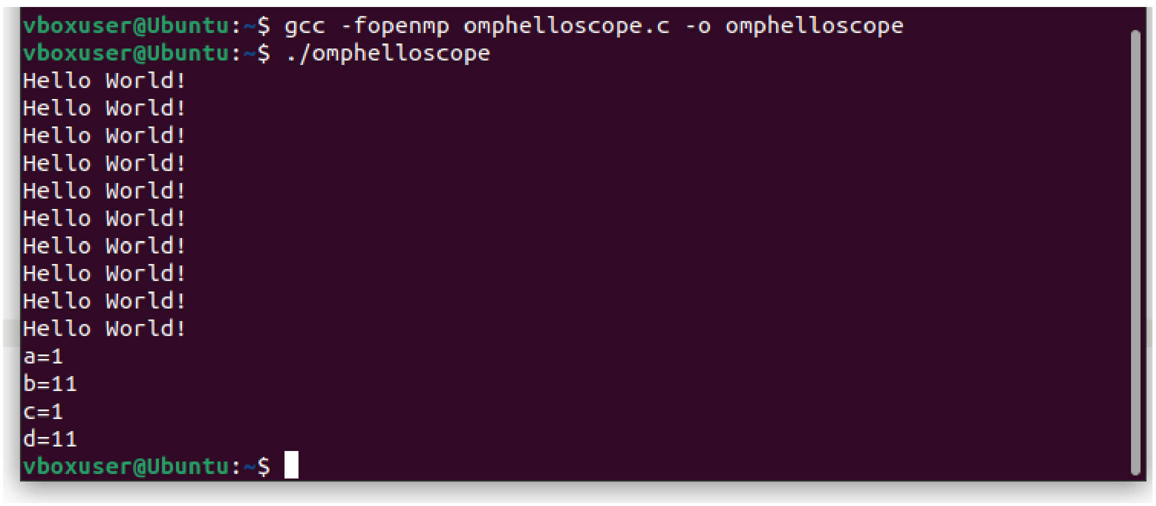
```
vboxuser@Ubuntu:~$ gcc -fopenmp omphellothread.c -o omphellothread
vboxuser@Ubuntu:~$ ./omphellothread
Hello from thread 1 of 4
Hello from thread 0 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4
vboxuser@Ubuntu:~$ gcc -fopenmp omphellothread.c -o omphellothread
vboxuser@Ubuntu:~$ OMP_NUM_THREADS=8 ./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 incorporating "omp_set_num_threads" in the program, we issue a directive to the OpenMP runtime to employ a specific number of threads, allowing the runtime system to endeavor to create and utilize the specified thread count during parallel program execution. In this context, the program employs omp_set_num_threads(4) to programmatically establish the number of threads as 4. It's important to note that this setting within the program is typically viewed as a default or recommendation rather than a strict constraint on the actual number of threads, offering flexibility to the runtime system.

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;
}
```



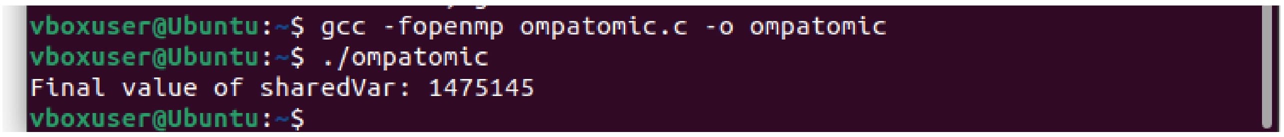
```
vboxuser@Ubuntu:~$ gcc -fopenmp omphelloscope.c -o omphelloscope
vboxuser@Ubuntu:~$ ./omphelloscope
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
a=1
b=11
c=1
d=11
vboxuser@Ubuntu:~$ 
```

Inference:

Within the parallel region, each of the 10 threads executes the "Hello World!" print statement, resulting in 10 "Hello World!" messages. Notably, the value of variable 'a' outside the parallel region remains 1 as it was private to each thread within the parallel scope. Conversely, the final value of variable 'b' is the cumulative sum of increments made by all threads, with each thread contributing one increment. Furthermore, the value of variable 'c' outside the parallel region remains 1 since it was firstprivate to each thread within the parallel context. Finally, the ultimate value of variable 'd' corresponds to the summation of increments made by all 10 threads, each contributing one increment.

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 criticalif (i % 2 == 0)
            sharedVar++;
    }

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

    return 0;
}
```



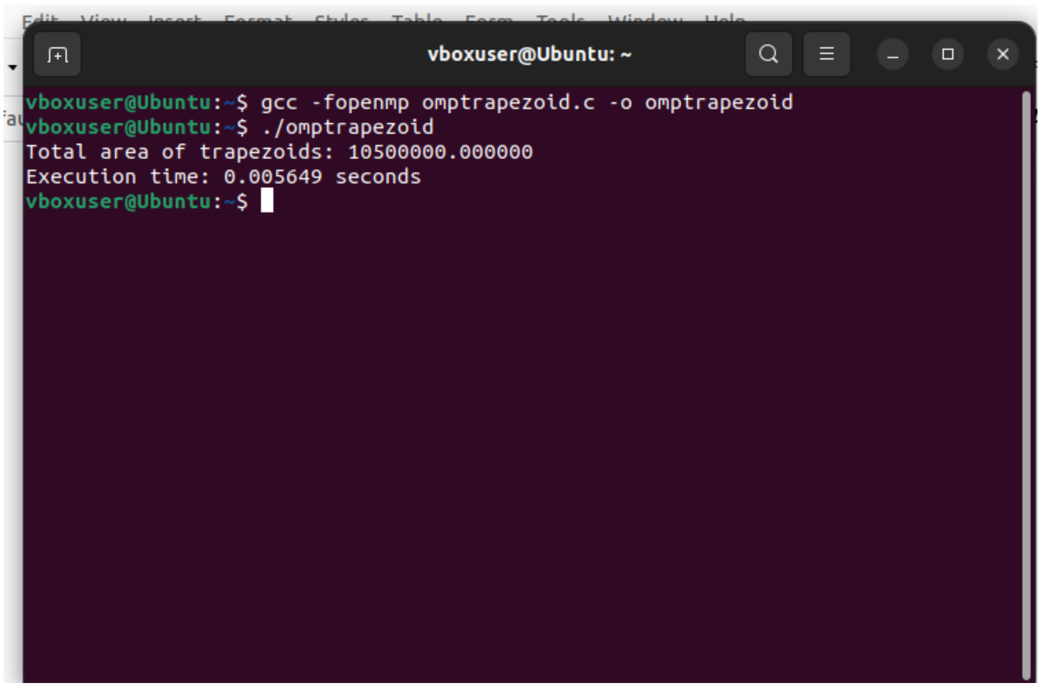
```
vboxuser@Ubuntu:~$ gcc -fopenmp ompatomic.c -o ompatomic
vboxuser@Ubuntu:~$ ./ompatomic
Final value of sharedVar: 1475145
vboxuser@Ubuntu:~$
```

Inference:

The program employs "#pragma omp atomic" to safeguard the increment operation on 'sharedVar,' ensuring atomicity and preventing data races when multiple threads concurrently update the variable. Additionally, "#pragma omp critical" establishes a critical section for the conditional increment, ensuring that only one thread at a time executes the code within the critical section. This measure mitigates potential race conditions, enhancing the thread-safe execution of the program.

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;
}
```



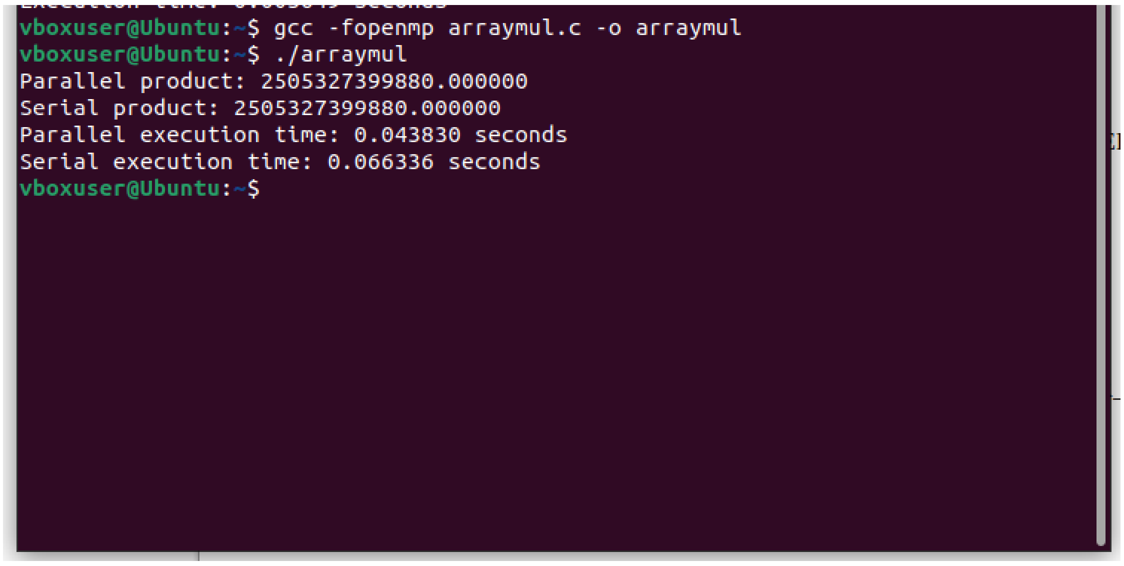
```
vboxuser@Ubuntu:~$ gcc -fopenmp omptrapezoid.c -o omptrapezoid
vboxuser@Ubuntu:~$ ./omptrapezoid
Total area of trapezoids: 10500000.000000
Execution time: 0.005649 seconds
vboxuser@Ubuntu:~$ 
```

Inference

In the task of calculating the total area of numerous trapezoids in parallel, the program employs the "#pragma omp parallel for reduction(+:totalArea)" directive to parallelize the loop, distributing the iterations among multiple threads. This directive, augmented by the "reduction(+:totalArea)" clause, designates that each thread possesses a private copy of 'totalArea,' with the final result computed by summing up these private copies. This approach guarantees correct aggregation of partial results from each thread, utilizing the addition (+) reduction operation. The inclusion of the "reduction(+:totalArea)" clause is imperative to prevent race conditions, ensuring the accuracy of the final result. Without this clause, simultaneous updates by multiple threads on the shared 'totalArea' could lead to data races and consequently yield 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:

The program undertakes the computation of the dot product for two arrays, implementing both parallel and serial approaches, subsequently measuring and comparing their execution times. Remarkably, the execution time in OpenMP for array multiplication with randomly generated positive numbers proves notably efficient compared to the parallelization program utilizing threads. OpenMP, leveraging a directive-based methodology, provides a more accessible approach to parallel programming, notably streamlining the process through compiler directives like #pragma omp, thereby reducing the need for manual thread creation and synchronization. Its implicit parallelism allows developers to articulate parallelism without explicitly managing threads. The parallel calculation of the dot product in OpenMP is achieved using the #pragma omp parallel for reduction(+:product_parallel) directive, wherein each thread computes a segment of the dot product, and the reduction clause ensures accurate_summation.

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.