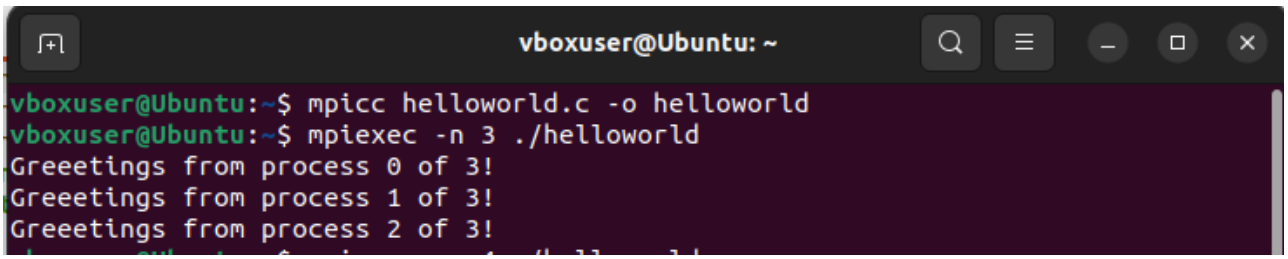
MPI Programming

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,
        else                                    MPI_COMM_WORLD);
        {


                                                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;
}
```



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

.Using MPI Scatter and MPI Gather makes the program run faster. The code breaks down matrices into parts and spreads them among different processes using MPI. This lets multiple processes work on their own parts of the matrices at the same time. Matrix addition is a task where each result matrix element can be calculated independently. MPI helps with this by spreading the work across processes. MPI_Scatter and MPI_Gather effectively share and collect data, reducing the need for a lot of communication.

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 shows how to add matrices in parallel using MPI, dividing the work between different processes. MPI_Bcast is used to efficiently share matrices with all processes. MPI_Reduce is used to bring together and add up the individual results on the main process. Measuring the time it takes gives us an idea of how long it takes to do the parallel 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
            a.   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

.Using MPI Scatter and MPI Gather makes the program run faster. The code breaks down matrices into parts and spreads them among different processes using MPI. This lets multiple processes work on their own parts of the matrices at the same time. Matrix addition is a task where each result matrix element can be calculated independently. MPI helps with this by spreading the work across processes. MPI_Scatter and MPI_Gather effectively share and collect data, reducing the need for a lot of communication.

Open MP Programming Simple Programs

Hello World

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

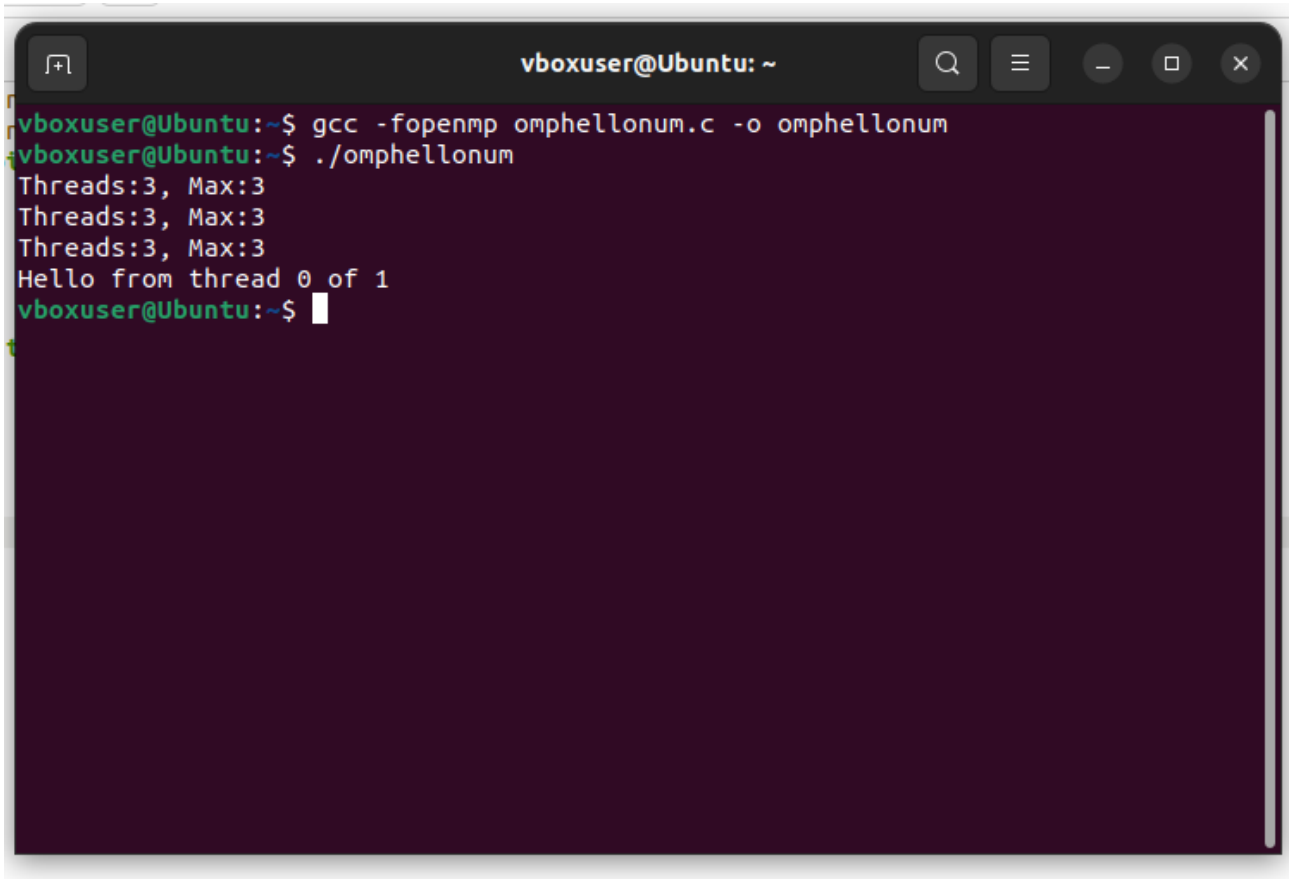
```
vboxuser@Ubuntu: ~
vboxuser@Ubuntu:~$ gcc -fopenmp omphello.c -o omphello
vboxuser@Ubuntu:~$ ./omphello
Hello, World!
Hello, World!
Hello, World!
vboxuser@Ubuntu:~$
```

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;
}
```



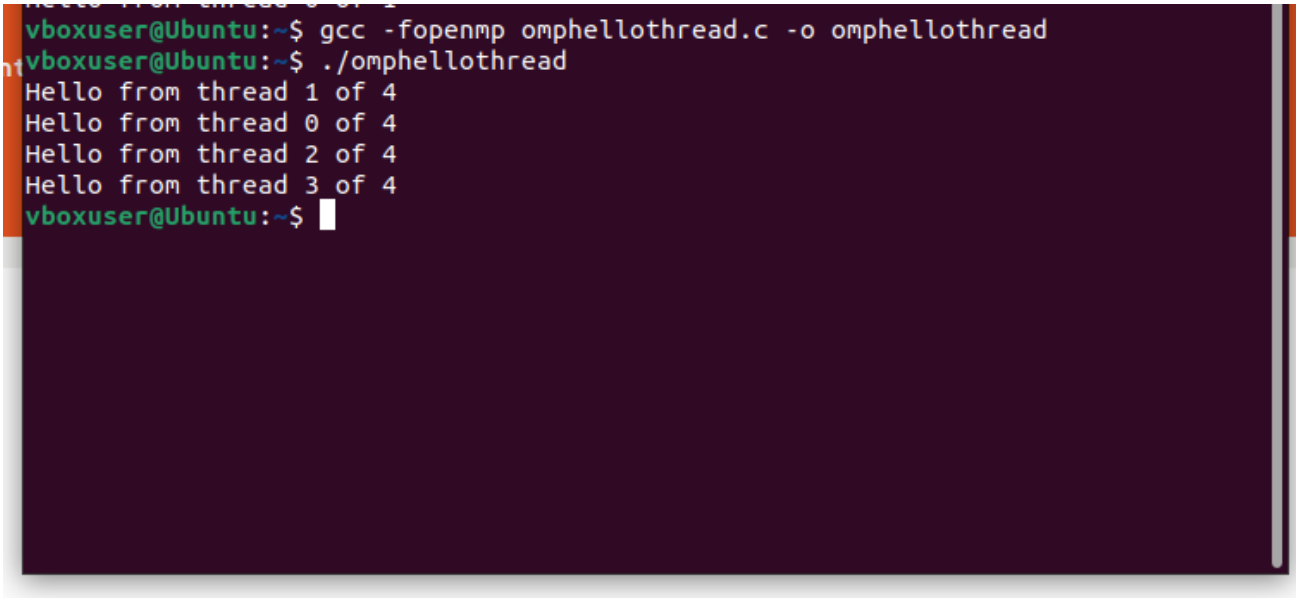
Inference

- this program uses OpenMP to do two things. First, it tells us how many threads are working together, and then it makes each thread say "Hello" using a function called "say_hello."
- The #pragma omp parallel part brings all the threads together to work as a team, and they all do what's inside the curly braces.

- The printf part inside the team of threads tells us how many threads are there and the most threads that can be there. This helps us understand how things are set up for the team.
- Then, the program asks each thread to say "Hello" by using the say_hello function. Each thread also says its number and how many threads are there in total.

Displaying the threads within the program or compilation

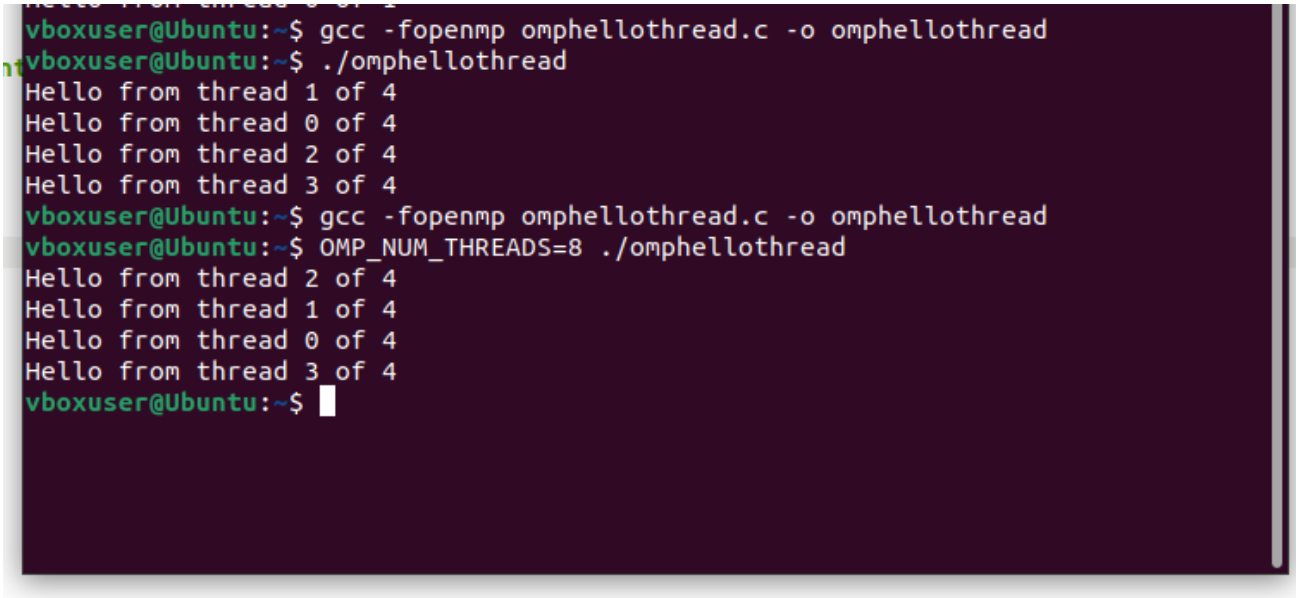
```
#include<stdio.h>
#include<omp.h>
void say_hello(void)
{
        int myrank=omp_get_thread_num();
        int threadcount=omp_get_num_threads();
        printf("Hello from thread %d of %d\n", myrank,threadcount);
}
int main(int argc, char* argv[])
{
        omp_set_num_threads(4);
        #pragma omp parallel
        say_hello();
        return 0;
}
```



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;
}
```

```
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

- In the part where many things happen at the same time, each of the 10 threads will say "Hello World!" because we have 10 threads (num_threads(10)).
-
- The number 'a' outside this part stays 1 because each thread has its own 'a' inside this part.
-
- The last number 'b' is the total of all the times each thread added 1 (10 threads, 1 increment each).
-
- The number 'c' outside this part stays 1 because each thread has its own 'c' inside this part.
-
- The last number 'd' is the total of all the times each thread added 1 (10 threads, 1 increment each).

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

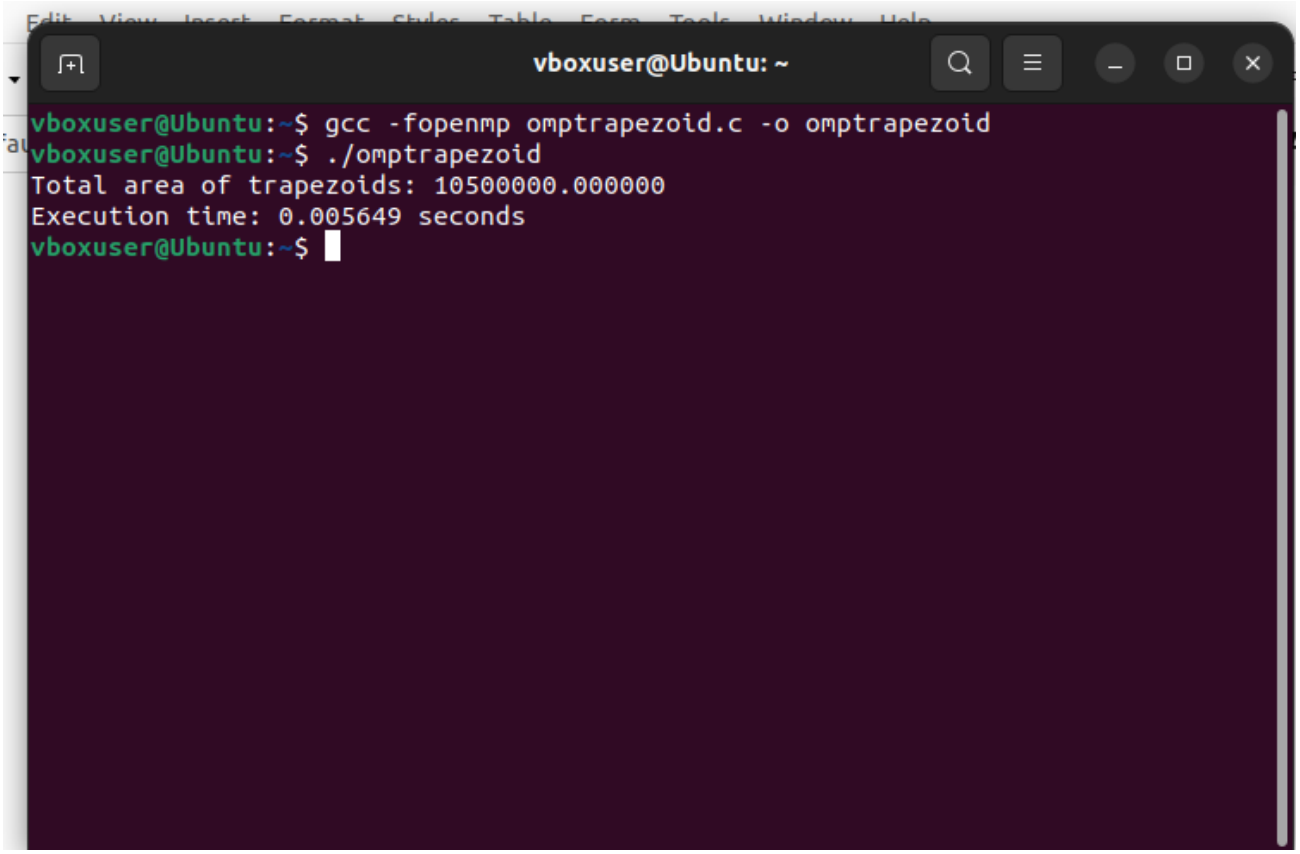
We use "#pragma omp atomic" to make sure that when many threads are trying to add something to "sharedVar" at the same time, it happens one after the other. This helps to avoid mistakes that can happen when lots of threads are changing the variable at the exact same time.

For "#pragma omp critical," it's like saying, "Hey, only one thread can do this part at a time." This way, we prevent any problems that might occur if more than one thread tries to do it simultaneously.

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;
}
```



Inference

- We want to find the total area of many trapezoids all at once, so we use "#pragma omp parallel for reduction(+:totalArea)" to make this happen faster. This line tells the computer to split the work between different threads.
- With "reduction(+:totalArea)," each thread gets its own private copy of totalArea, and in the end, we add up all these private copies to get the final result.
- This is important because without "reduction(+:totalArea)," if many threads try to update the shared totalArea at the same time, it could cause problems, and we might get the wrong answer. So, by using this, we make sure everything is correct.
-

Image Processing

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

#define IMAGE_WIDTH 100
#define IMAGE_HEIGHT 100
#define NUM_THREADS 4

typedef struct {
   int width;
   int height;
   unsigned char* data;
} Image;

Image* createImage(int width, int height) {
   Image* img = (Image*)malloc(sizeof(Image));
   img->width = width;
   img->height = height;
   img->data = (unsigned char*)malloc(width * height * sizeof(unsigned char));
   return img;
}

// Function to free memory for an image
void freeImage(Image* img) {
   free(img->data);
   free(img);
}

void doublePixelValues(Image* img) {
   #pragma omp parallel for
   for (int i = 0; i < img->width * img->height; i++) {
      img->data[i] *= 2;
   }
}

int main() {
   Image* inputImage = createImage(IMAGE_WIDTH, IMAGE_HEIGHT);

     for (int i = 0; i < IMAGE_WIDTH * IMAGE_HEIGHT; i++) {
      inputImage->data[i] = rand() % 256;
   }

   clock_t start_time = clock();
    doublePixelValues(inputImage);

   clock_t end_time = clock();
```

Inference

It's like having a canvas of 100 pixels in width and 100 pixels in height. Each pixel can have a brightness level, and the image is initially filled with random brightness values between 0 and 255.

Next, the program goes through each pixel in the image and makes it brighter. The brightness levels are multiplied by 2. This makes the entire image appear brighter because each pixel is now twice as bright as before.

The #pragma omp parallel for part is like calling in a team of workers to help with the task. Instead of one worker going through all the pixels, the work is divided among multiple workers (threads). Each worker takes care of a portion of the image, making the process faster.

Finally, the code frees up the memory used for the image. It's like cleaning up after finishing the jo

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.