PARALLEL COMPUTING SYSTEM ASSIGNMENT

A.RAKSHANA MALYA

66CG001

917723CS009

Open MP Programming

Hello world program:

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

OUTPUT:

EXPLANATION:

This C program uses OpenMP (Open Multi-Processing) to parallelize the "Hello, World!" print statement. Here's a breakdown:

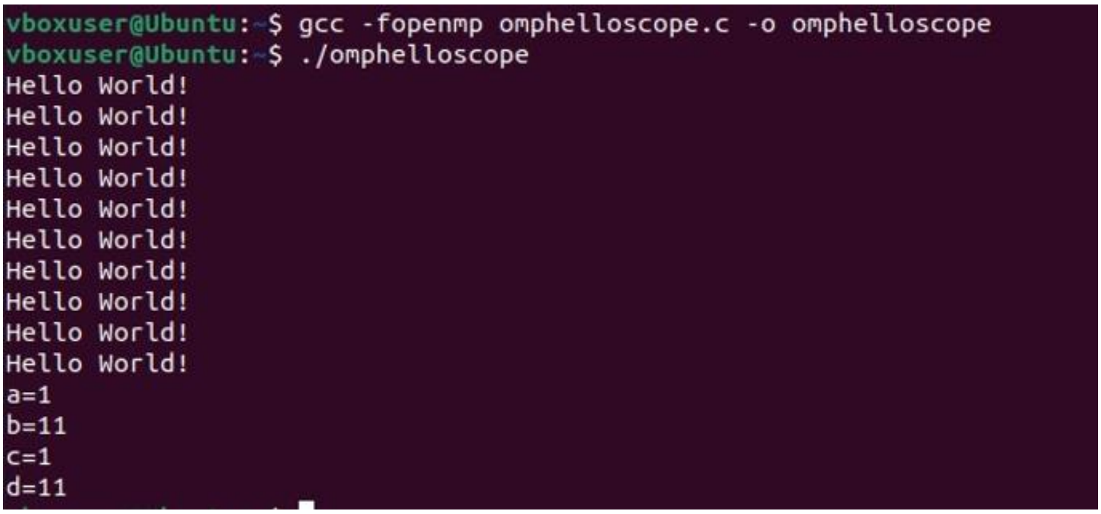
- **#pragma omp parallel**: This directive tells the compiler to create a parallel region. The code within this region will be executed by multiple threads concurrently.
- **{ printf("Hello, World!\n"); }:** Inside the parallel region, each thread executes the printf statement, printing "Hello, World!".
- **return 0;**: The program returns 0, indicating successful execution.

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:



EXPLANATION:

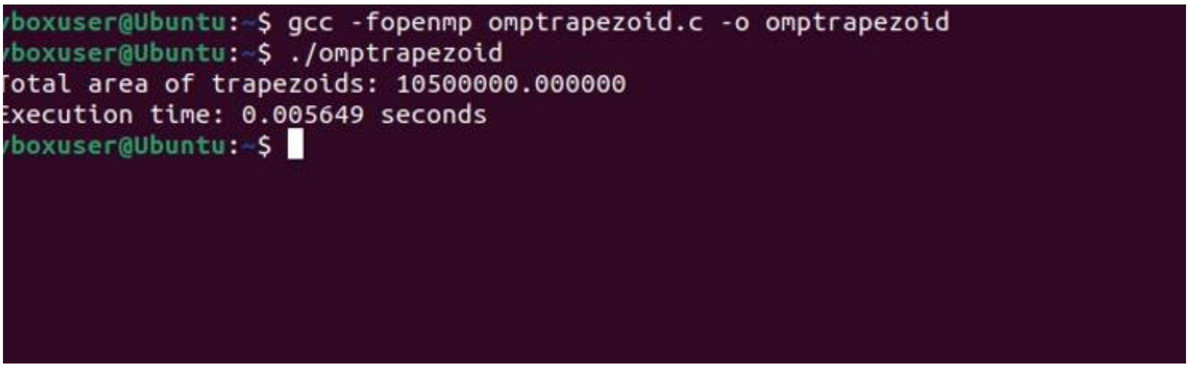
This C program demonstrates OpenMP parallelization. Inside a parallel region with 10 threads, variables a are private, b is shared, and c is firstprivate. Each thread increments private variables. After the parallel region, it prints the final values of a, b (common to all threads), c (initial private value), and d (unchanged). The output reflects parallel thread behavior on shared and private variables.

Private Clause: Specifies that each thread should have its private copy of the listed variables.
Shared Clause: Declares variables to be shared among all threads in a parallel region. Shared variables are accessible and modifiable by all threads.
Firstprivate Clause: Combines the behavior of private and shared. Each thread gets its private copy of the variable initialized with the original value from the master thread.

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;
}
```

OUTPUT:



EXPLANATION:

This C program uses OpenMP to parallelize the calculation of trapezoid areas. The **#pragma omp parallel** for distributes the iterations of the loop across threads. The **reduction(+:totalArea)** clause ensures a private copy of **totalArea** for each thread, and the results are summed up at the end, improving performance. The program measures and prints the total area of trapezoids and the execution time.

QUICK SORT:

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

// Function to swap two elements
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Partition function to select a pivot and partition the array
int partition(int arr[], int low, int high) {
    int pivot = arr[high]; // Choose the pivot as the last element
    int i = (low - 1); // Initialize the index of the smaller element

    #pragma omp parallel for
    for (int j = low; j <= high - 1; j++) {
        // If the current element is smaller than or equal to the pivot
        if (arr[j] <= pivot) {
            #pragma omp critical
            {
                i++; // Increment index of the smaller element
                swap(&arr[i], &arr[j]);
            }
        }
    }

    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

// Parallel quicksort function using OpenMP
```

```c
void parallelQuickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);

        #pragma omp task
        parallelQuickSort(arr, low, pi - 1);

        #pragma omp task
        parallelQuickSort(arr, pi + 1, high);
    }
}

int main() {
    int n = 1000000; // Adjust the array size as needed
    int* arr_serial = (int*)malloc(n * sizeof(int));
    int* arr_parallel = (int*)malloc(n * sizeof(int));

    // Initialize arrays with random values
    srand(time(NULL));
    for (int i = 0; i < n; i++) {
        arr_serial[i] = arr_parallel[i] = rand() % 1000;
    }

    printf("Array size: %d\n", n);

    // Serial quicksort
    clock_t start_time = clock();
    parallelQuickSort(arr_serial, 0, n - 1);
    clock_t end_time = clock();
    double serial_execution_time = (double)(end_time - start_time) / CLOCKS_PER_SEC;
    printf("Serial quicksort execution time: %f seconds\n", serial_execution_time);

    // Parallel quicksort using OpenMP
    start_time = clock();
    #pragma omp parallel
    {
        #pragma omp single
        parallelQuickSort(arr_parallel, 0, n - 1);
    }
    end_time = clock();
    double parallel_execution_time = (double)(end_time - start_time) / CLOCKS_PER_SEC;
    printf("Parallel quicksort execution time: %f seconds\n", parallel_execution_time);

    // Check if the arrays are sorted correctly
```

```
    for (int i = 0; i < n - 1; i++) {
        if (arr_serial[i] > arr_serial[i + 1] || arr_parallel[i] > arr_parallel[i + 1]) {
            printf("Sorting Successful!");
            break;
        }
    }

    free(arr_serial);
    free(arr_parallel);
    return 0;
}
```

OUTPUT:

```
Success #stdin #stdout 1.45s 8892KB
Array size: 1000000
Serial quicksort execution time: 0.722543 seconds
Parallel quicksort execution time: 0.712033 seconds
```

EXPLANATION:

This C program implements serial and parallel quicksort using OpenMP. It initializes arrays with random values, measures the execution time of both serial and parallel quicksort, and checks if the arrays are sorted correctly. The parallel version employs OpenMP tasks for parallelization. The program demonstrates the performance improvement achieved through parallel sorting in a shared-memory parallel computing environment, as seen in the reduced execution time compared to the serial version.