Reg No : 20BCE0456
Name : Aditya Krishna
Slot : L15+L16

EX 3 (OPENMP –III)

SCENARIO – I

Write a simple OpenMP program for Matrix Multiplication. It is helpful to understand how threads make use of the cores to enable course-grain parallelism. Note that different threads will write different parts of the result in the array ***a***, so we dont't get any problems during the parallel execution. Note that accesses to matrices ***b and c*** are read-only and do not introduce any problems either.

ALGORITHM:

1. Include necessary header files: `stdio.h`, `stdlib.h`, and `omp.h`.
2. Define a constant `N` to represent the size of the matrices.
3. Declare three pointers `A`, `B`, and `C` of type `double` to represent the matrices.
4. Allocate memory for matrices A, B, and C using `malloc()` function.
5. Initialize matrices A and B with random values using nested for loops.
6. Use `#pragma omp parallel for` directive to parallelize the matrix multiplication. Use `shared(A,B,C)` to declare that matrices A, B, and C are shared among threads and `private(i,j,k)` to declare that loop variables i, j, and k are private to each thread.
7. Compute the product of matrices A and B by using nested for loops and store the result in matrix C.
8. Print the rows of the result matrix C using a for loop.
9. Print the columns of the result matrix C using a for loop.
10. Free the memory allocated for matrices A, B, and C using `free()` function.
11. End the program by returning 0.

SOURCE CODE:

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

#define N 100 // initialized the size of matrices

int main() {
double *A, *B, *C;
```

Reg No : 20BCE0456
Name : Aditya Krishna
Slot : L15+L16

```c
int i, j, k;

// Allocate memory for matrices A, B, and C
A = (double *) malloc(N * N * sizeof(double));
B = (double *) malloc(N * N * sizeof(double));
C = (double *) malloc(N * N * sizeof(double));

// Initialize matrices A and B with random values
for (i = 0; i < N; i++) {
for (j = 0; j < N; j++) {
A[i * N + j] = (double) rand() / RAND_MAX;
B[i * N + j] = (double) rand() / RAND_MAX;
}
}

// Parallelize the matrix multiplication using OpenMP
#pragma omp parallel for shared(A,B,C) private(i,j,k)
for (i = 0; i < N; i++) {
for (j = 0; j < N; j++) {
C[i * N + j] = 0.0;
for (k = 0; k < N; k++) {
C[i * N + j] += A[i * N + k] * B[k * N + j];
}
}
}

// Print the result
printf("Result matrix C (row(s)):\n");
for (j = 0; j < 10; j++) {
printf("%0.2f\n", C[j]);
}
printf("\n");
printf("Result matrix C (columns(s)):\n");
for (i = 0; i < 10; i++) {
printf("%0.2f\n", C[i * N]);
}

// Free memory
free(A);
free(B);
free(C);

return 0;

}
```

Reg No : 20BCE0456
Name : Aditya Krishna
Slot : L15+L16

OUTPUT SCREEN SHOT:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL                bash - lab3

[adityak@20bce0456 lab3]$ gcc -o q1_20bce0456 -fopenmp q1_20bce0456.c
[adityak@20bce0456 lab3]$ ./q1_20bce0456
Result matrix C (row(s)):
24.76
25.37
23.26
25.60
21.85
24.26
25.19
21.93
23.52
24.66

Result matrix C (columns(s)):
24.76
26.97
25.73
26.60
26.79
22.98
26.17
25.63
26.26
25.17
[adityak@20bce0456 lab3]$
```

RESULTS:

The above code performs parallel matrix multiplication using OpenMP. It first allocates memory for matrices A, B, and C and initializes A and B with random values. Then it parallelizes the matrix multiplication using the OpenMP directive `#pragma omp parallel for` and computes the product of matrices A and B by using nested for loops. The result is stored in matrix C. Finally, it prints the rows and columns of the result matrix C and frees the memory allocated for matrices A, B, and C. The code uses shared memory parallelism to distribute the work among multiple threads. The `shared(A,B,C)` clause specifies that matrices A, B, and C are shared among all threads, and the `private(i,j,k)` clause specifies that loop variables i, j, and k are private to each thread.

Reg No : 20BCE0456
Name : Aditya Krishna
Slot : L15+L16

SCENARIO – II

Write a OpenMP program to calculate the value of PI by evaluating the integral 4/(1 + x2) . You can use three pragma block directives to handle the critical section , the sum and the product display.

ALGORITHM:

1. Include necessary header files: stdio.h, omp.h, stdlib.h, and math.h.
2. Define a constant PI to represent the value of PI.
3. Declare variables num_intv, i, sum, x, total_sum, y, partial_sum, sum_thread.
4. Prompt the user to enter the number of intervals and store it in num_intv.
5. Check if num_intv is positive, if not print an error message and exit the program.
6. Initialize sum and sum_thread to 0.
7. Compute the value of y as 1.0 / num_intv.
8. Use #pragma omp parallel for directive to parallelize the loop. Declare x as private to each thread and sum_thread as shared among threads.
9. Within the parallel loop, compute the value of x as y * (i - 0.5), where i is the loop variable.
10. Compute the value of sum_thread as sum_thread + 4.0 / (1 + x * x).
11. Use #pragma omp critical directive to synchronize the threads before updating the value of sum and sum_thread.
12. Compute the value of partial_sum as sum_thread * y.
13. Use #pragma omp critical directive to synchronize the threads before updating the value of sum.
14. Print the value of PI as sum and the error as fabs(sum - PI).
15. End the program.

SOURCE CODE:

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

Reg No : 20BCE0456
Name : Aditya Krishna
Slot : L15+L16

```c
#include<math.h>

#define PI 3.1415926538837211

int main()
{

int num_intv, i;

float sum, x, total_sum, y, partial_sum, sum_thread;

printf("Enter the number of intervals\n");

scanf("%d", &num_intv);
if (num_intv <= 0)//constraint/condition
{

printf("No. of intervals should be positive integer!!!");

exit(1);//exit

}

sum = 0.0;
y = 1.0 / num_intv;

#pragma omp parallel for private(x) shared(sum_thread)
for (i = 1; i < num_intv + 1; i = i + 1)
{
x = y * (i - 0.5);

//OpenMP Critical Section Directive
#pragma omp critical
sum_thread = sum_thread + 4.0 / (1 + x * x);
}
partial_sum = sum_thread * y;

//OpenMP Critical Section Directive
#pragma omp critical
sum = sum + partial_sum;
printf("The value of PI: %f \nError: %1.16f\n", sum, fabs(sum - PI));
}
```

Reg No : 20BCE0456
Name : Aditya Krishna
Slot : L15+L16

OUTPUT :

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL                bash - lab3

[adityak@20bce0456 lab3]$ gcc -o q2_20bce0456 -fopenmp q2_20bce0456.c
[adityak@20bce0456 lab3]$ ./q2_20bce0456
Enter the number of intervals
2
The value of PI: 3.162353
Error: 0.0207603849041207
[adityak@20bce0456 lab3]$
```

RESULTS:

The code calculates an approximation of the value of PI using the given formula. It prompts the user to enter the number of intervals, and then parallelizes a loop using OpenMP to calculate the sum of terms in the formula. It uses critical sections to synchronize the threads and update the sum of the terms. Finally, it prints the approximation of PI and the error between the approximation and the actual value of PI.

SCENARIO – III

Write a simple OpenMP program that uses some OpenMP API functions to extract information about the environment. It should be helpful to understand the language / compiler features of OpenMP runtime library.

To examine the above scenario, the functions such as omp_get_num_procs(), *omp_set_num_threads(), omp_get_num_threads(), omp_in_parallel(), omp_get_dynamic()* and *omp_get_nested()* are listed and the explanation is given below to explore the concept practically.

omp_set_num_threads() takes an integer argument and requests that the Operating System provide that number of threads in subsequent parallel regions.
omp_get_num_threads() (integer function) returns the actual number of threads in the current team of threads.
omp_get_thread_num() (integer function) returns the ID of a thread, where the ID ranges from 0 to the number of threads minus 1. The thread with the ID of 0 is the master thread.
omp_get_num_procs() returns the number of processors that are available when the function is called.
omp_get_dynamic() returns a value that indicates if the number of threads available in subsequent parallel region can be adjusted by the run time.
omp_get_nested( ) returns a value that indicates if nested parallelism is enabled.

ALGORITHM:

Reg No : 20BCE0456
Name : Aditya Krishna
Slot : L15+L16

1. Get the number of processors available using the OpenMP function omp_get_num_procs().

2. Print the number of processors.

3. Set the number of threads to be used in parallel using omp_set_num_threads().

4. Get the actual number of threads to be used in parallel using omp_get_num_threads() and print it.

5. Enter a parallel region using #pragma omp parallel private(thread_num).

6. Get the thread number for each thread using omp_get_thread_num() and print it.

7. Check if currently in a parallel region using omp_in_parallel(), and print the result.

8. Exit the parallel region using the end of the parallel block.

9. Get the dynamic adjustment status of threads using omp_get_dynamic() and print the result.

10. Get the status of nested parallelism using omp_get_nested() and print the result.

11. Return 0 to terminate the program.

SOURCE CODE:

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

int main() {
int num_procs, num_threads, thread_num, is_dynamic, is_nested;
num_procs = omp_get_num_procs();// get number of processors
printf("Number of processors: %d\n", num_procs);
omp_set_num_threads(4);// set number of threads
num_threads = omp_get_num_threads();//get actual number of threads
printf("Number of threads: %d\n", num_threads);

#pragma omp parallel private(thread_num)
{
thread_num = omp_get_thread_num();//get thread number
printf("Thread number: %d\n", thread_num);
if (omp_in_parallel()) //check in parallel region
{
printf("\nIn parallel region\n");
} else {
```

Reg No : 20BCE0456
Name : Aditya Krishna
Slot : L15+L16

```c
printf("Not in parallel region\n");
}
}
// check if dynamic adjustment of threads is allowed
is_dynamic = omp_get_dynamic();
printf("Dynamic adjustment of threads: %d\n", is_dynamic);
// check if nested parallelism is enabled
is_nested = omp_get_nested();
printf("Nested parallelism enabled: %d\n", is_nested);
return 0;
}
```

OUTPUT :

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL bash - lab3

```
[adityak@20bce0456 lab3]$ gcc -o q3_20bce0456 -fopenmp q3_20bce0456.c
[adityak@20bce0456 lab3]$ ./q3_20bce0456
Number of processors: 8
Number of threads: 1
Thread number: 2

In parallel region
Thread number: 3

In parallel region
Thread number: 0

In parallel region
Thread number: 1

In parallel region
Dynamic adjustment of threads: 0
Nested parallelism enabled: 0
[adityak@20bce0456 lab3]$
```

RESULTS:

The above code uses OpenMP library functions to get information about the number of processors, threads, and their status. It sets the number of threads to 4 and prints the thread number for each thread. It checks if it is in a parallel region or not, gets the status of dynamic adjustment of threads, and whether nested parallelism is enabled or not.