PARALLEL DISTRIBUTING COMPUTING

***————————————786**
————————————*

Name: Muhammad Sharjeel Akhtar

Roll No: 20p-0101

Assignment No: 01

Submitted To Respected
Sir: Dr Omer Usman Khan

Section: BCS-7F

Task No 1:

1. **Provide a list of run-time routines that are used in OpenMP**

ANSWER: Some common **OpenMP** run-time routines are as follow:

omp_get_num_threads(): Returns the total number of threads in the current team.

omp_get_thread_num(): Returns the thread number of the calling thread.

omp_set_num_threads(): Sets the number of threads to be used for subsequent parallel
regions.

omp_get_num_procs(): Returns the number of processors available.

omp_get_max_threads(): Returns the maximum number of threads that could be used.

2. Why aren’t you seing the Hello World output thread sequence as 0, 1, 2, 3 etc. Why are they disordered?

ANSWER: The order of output from the parallel region is not guaranteed to be in sequence (0, 1,
2, 3, etc.) because the order in which threads execute is not specified in OpenMP. The scheduling of threads is managed by the operating system and the OpenMP runtime. Each thread executes independently, and their order of execution is not deterministic.

3. What happens to the thread_id if you change its scope to before the pragma?

Answer: If we change the scope of **thread_id** to before the **pragma**, it becomes a **shared**
variable
, and **each thread** will have its own **copy**. This may lead to **unexpected behavior** because each

thread will be **modifying** its own **copy** of **thread_id**. It's generally safer to declare thread_id
inside the **parallel region** to ensure each thread has its own **private copy**
.

4. Convert the code to serial code.

ANSWER:

Original Code:

```c
#include <stdio.h>
#include <omp.h>
int main() {
#pragma omp parallel
{ int thread_id = omp_get_thread_num();
printf("Hello, World! This is thread %d\n", thread_id);
}
return 0;
}
```

Serial Code:

```c
#include <stdio.h>

int main(){
    int thread_id=0;

    printf("Hello, World! This is thread %d\n");
    return 0;
}
```

Task No 2:

1. The following code adds two arrays of size 16 together and stores answer in result array.

ANSWER:

Original Code:

Below is the code given in question,

```
#include <stdio.h>
int main() {
int array1[16] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, ..
int array2[16] = {16, ..., 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3,
int result[16];
for (int i = 0; i < 16; i++) {
result[i] = array1[i] + array2[i];
}
for (int i = 0; i < 16; i++) {
printf("%d ", result[i]);
}
printf("\n");

return 0;
}
```

2. Convert it into Parallel, such that only the addition part is parallelized.

ANSWER:

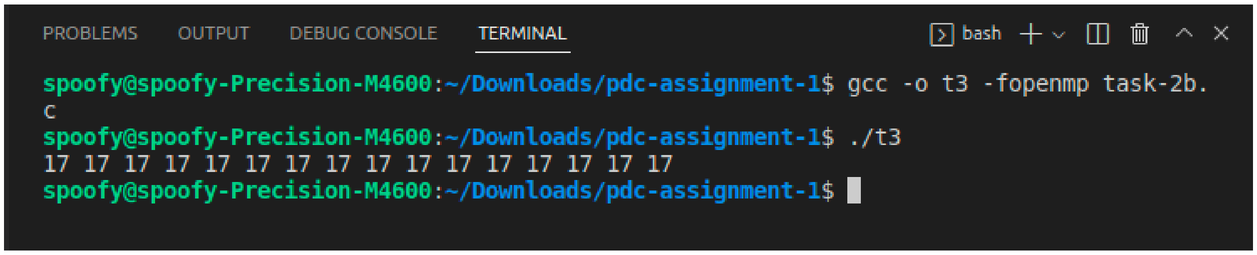
```
#include <stdio.h>
int main(){
    int array1[16]={1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};
    int array2[16]={16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1};
    int result[16];
    // PARALLELIZING THE ADDITION PORTION
    #pragma omp parallel for
    for(int i=0;i<16;i++){
        result[i]=array1[i]+array2[i];
    }
    for (int i=0;i<16;i++){
        printf("%d", result[i]);
    }
    printf("\n");
    return 0;
}
```

3. The display loop at the end displays the result. Modify the code such that this is also parallel, but only thread of id 0 is able to display the entire loop. The others should not
   do anything. When making it parallel, make sure its the old threads and new threads are
   not created. What output do you see?

ANSWER:

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

int main() {
    int array1[16] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};
    int array2[16] = {16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1};
    int result[16];

    // PARALLELIZING THE ADDITION PORTION
    #pragma omp parallel for
    for(int i = 0; i < 16; i++) {
        result[i] = array1[i] + array2[i];
    }

    // Only master thread (thread with ID 0) displays the resul
    #pragma omp master
    {
        for (int i = 0; i < 16; i++) {
            printf("%d ", result[i]);
        }
        printf("\n");
    }

    return 0;
}
```

OUTPUT:



PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
spoofy@spoofy-Precision-M4600:~/Downloads/pdc-assignment-1$ gcc -o t3 -fopenmp task-2b.
c
spoofy@spoofy-Precision-M4600:~/Downloads/pdc-assignment-1$ ./t3
17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17
spoofy@spoofy-Precision-M4600:~/Downloads/pdc-assignment-1$ 
```

Task No 3:

1. Modify the code of Task 2 and do the job in half of the threads.

ANSWER:

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

int main() {
    int array1[16] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};
    int array2[16] = {16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1};
    int result[16];

    // Getting the total number of threads
    int total_threads=omp_get_max_threads();

    // we have to use only half of threads
    int num_threads_to_use=total_threads/2;

    // PARALLELIZING THE ADDITION PORTION
    #pragma omp parallel for num_threads(num_threads_to_use)
    for(int i = 0; i < 16; i++) {
        result[i] = array1[i] + array2[i];
    }

    // Display the result
    for(int i=0;i<16;i++){
        printf("%d",result[i]);
        printf("\n");
    }

    return 0;
}
```

Task No 4:

1. The following code adds the contents of the array array1 and array2.

ANSWER:

```
#include <stdio.h>
int main() {
int array1[16] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
int array2[16] = {16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4,
int result1 = 0, result2 = 0;
for (int i = 0; i < 16; i++) {result1 += array1[i];
}
if (result1 > 10) {
result2 = result1;
for (int i = 0; i < 16; i++) {
result2 += array2[i];
}
}
printf("%d\n", result2);
return 0;
}
```

2. Convert it into Parallel using 16 threads.

ANSWER:

```
#include <stdio.h>
int main() {
int array1[16] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
int array2[16] = {16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4,
int result1 = 0, result2 = 0;

#pragma omp parallel num_threads(16)
{
    #pragma omp for reduction(+:result1)
    for(int i=0;i<16;i++){
        result1+=array1[i];
    }
    #pragma omp sections
    {
        #pragma omp section
        {
            if (result1>10){
                #pragma omp parallel for reduction(+:result2)
                for (int i=0;i<16;i++){
                    result2+=array2[i];
                }
            }
        }
    }
}
printf("%d\n", result2);

return 0;
}
```

3. Try removing the reduction() clause and add #pragma omp atomic just beore the +=. What is the effect on result? Explain.

ANSWER:

```
#include <stdio.h>
int main() {
int array1[16] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
int array2[16] = {16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4,
int result1 = 0, result2 = 0;

#pragma omp parallel num_threads(16)
{
    #pragma omp for
    for(int i=0;i<16;i++){
                #pragma omp atomic
        result1+=array1[i];
    }
    #pragma omp sections
    {
        #pragma omp section
        {
            if (result1>10){
                #pragma omp parallel for
                for (int i=0;i<16;i++){
                                        #pragma omp atomic
                    result2+=array2[i];
                }
            }
        }
    }
}
printf("%d\n", result2);
return 0;
}
```

EFFECT OF RESULT:

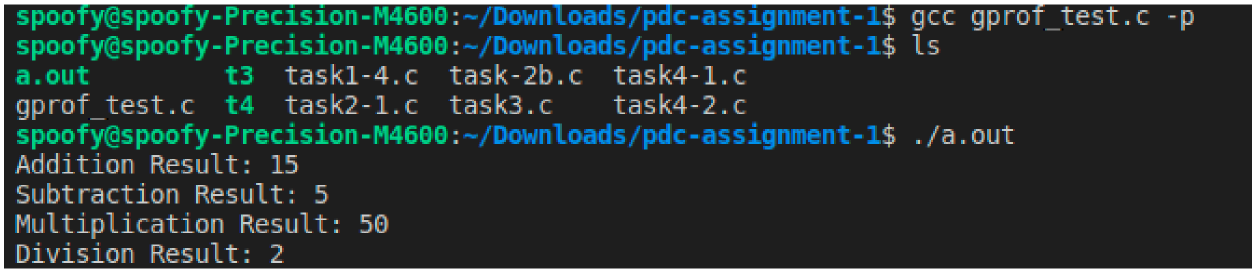
The **#pragma omp atomic** directive **ensures** that the specified **operation** is executed **atomically**, **avoiding race conditions** that may occur in **parallel regions**. However, using atomic operations can introduce contention, and in some cases, it might lead to decreased performance compared to using a reduction clause. In this specific code, since the updates to **result1** and **result2** are performed atomically, the final result should still be correct. However, **the performance** characteristics may vary depending on the specifics of the system and workload.

Task No 5

ANSWER:

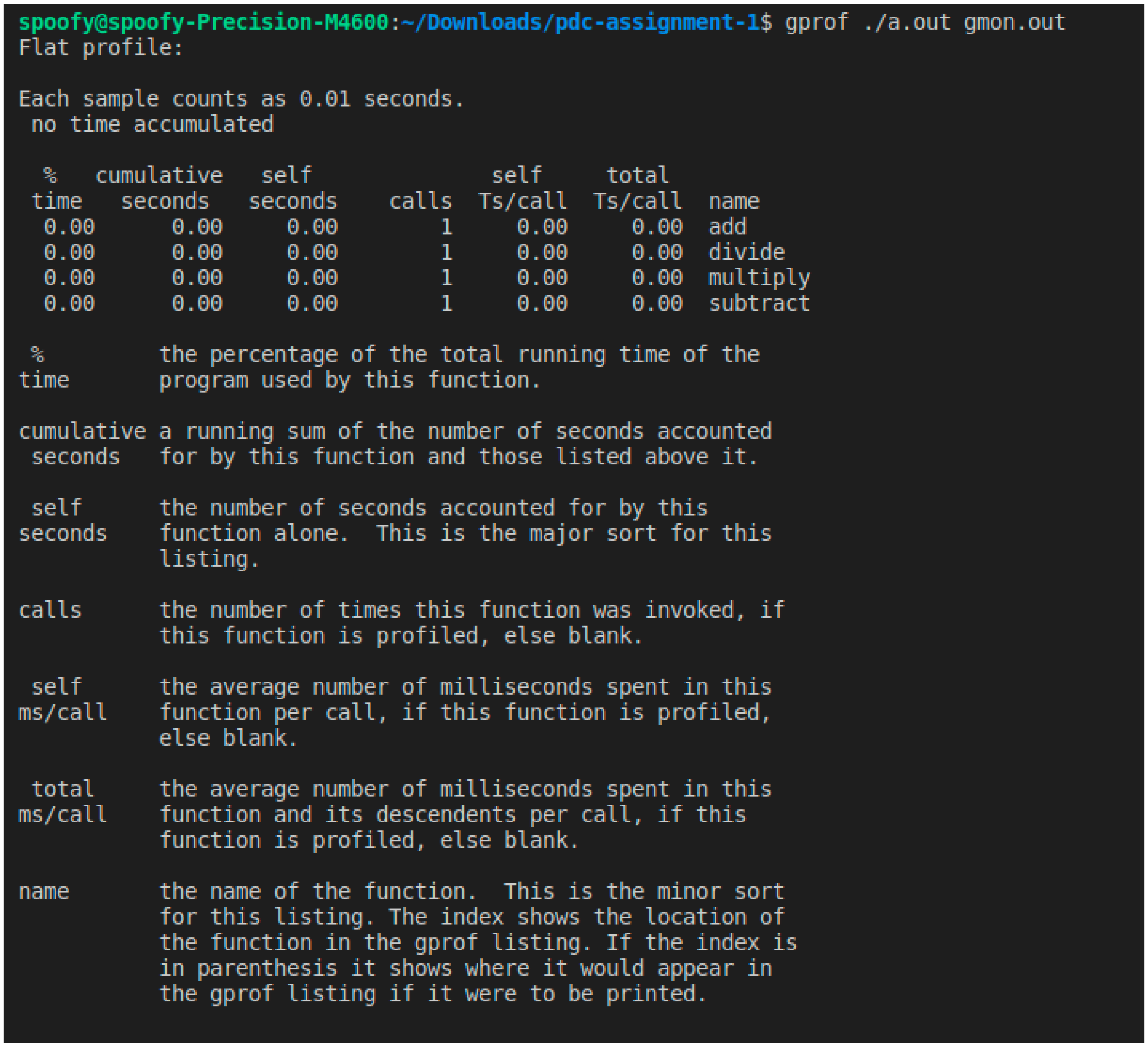
```
#include <stdio.h>

void add(int a, int b) {
    int result = a + b;
    printf("Addition Result: %d\n", result);
}

void subtract(int a, int b) {
    int result = a - b;
    printf("Subtraction Result: %d\n", result);
}

void multiply(int a, int b) {
    int result = a * b;
    printf("Multiplication Result: %d\n", result);
}

void divide(int a, int b) {
    if (b != 0) {
        int result = a / b;
        printf("Division Result: %d\n", result);
    } else {
        printf("Cannot divide by zero!\n");
    }
}

int main() {
    int num1 = 10, num2 = 5;

    add(num1, num2);
    subtract(num1, num2);
    multiply(num1, num2);
    divide(num1, num2);

    return 0;
}
```

BASIC RUN:



```
spoofy@spoofy-Precision-M4600:~/Downloads/pdc-assignment-1$ gcc gprof_test.c -p
spoofy@spoofy-Precision-M4600:~/Downloads/pdc-assignment-1$ ls
a.out         t3  task1-4.c  task-2b.c  task4-1.c
gprof_test.c  t4  task2-1.c  task3.c    task4-2.c
spoofy@spoofy-Precision-M4600:~/Downloads/pdc-assignment-1$ ./a.out
Addition Result: 15
Subtraction Result: 5
Multiplication Result: 50
Division Result: 2
```

GPROF RUN AND REPRESENTATION:



```
spoofy@spoofy-Precision-M4600:~/Downloads/pdc-assignment-1$ gprof ./a.out gmon.out
Flat profile:

Each sample counts as 0.01 seconds.
 no time accumulated

  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00        1     0.00     0.00  add
  0.00      0.00     0.00        1     0.00     0.00  divide
  0.00      0.00     0.00        1     0.00     0.00  multiply
  0.00      0.00     0.00        1     0.00     0.00  subtract

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
           else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
           function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
           the function in the gprof listing. If the index is
           in parenthesis it shows where it would appear in
           the gprof listing if it were to be printed.
```