

SCENARIO – I

Write a simple OpenMP program to define a parallel region. It should be helpful to understand the purposes of shared variable and local or private variable to be executed in each thread in OpenMP.

Note: A prefix of #pragma omp is used as an OpenMP directive construct to exploit the feature of concurrency.

BRIEF ABOUT YOUR APPROACH:

In the pragma omp block all code that we write is run in parallel by the number of threads we have. Therefore, for a simple execution the execution will happen for n times for n threads. Each thread now works with data. All variables declared outside the parallel block can be accessed by the threads if they are declared as shared. If they are private then new instances are saved with which they are worked with. Variables initialized in the parallel region are by default private and cannot be accessed by outside code during parallel execution. To demonstrate we use a simple code to take two numbers and print hello. Once with the variables being shared and once private.

SOURCE CODE:

(Shared)

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

int main(){

    int a;
    int b;
    int ID;

    scanf("%d",&a);
    scanf("%d",&b);

    omp_set_num_threads(2);

    #pragma omp parallel shared(a,b) private(ID)
    {
        ID = omp_get_thread_num();

        if(ID == 0)
        {
            printf("sum is %d\n",(a+b));
```

Dated : 04-Aug-2018
Assessment No. : 2

Madhur Bhatnagar
16BCB0009

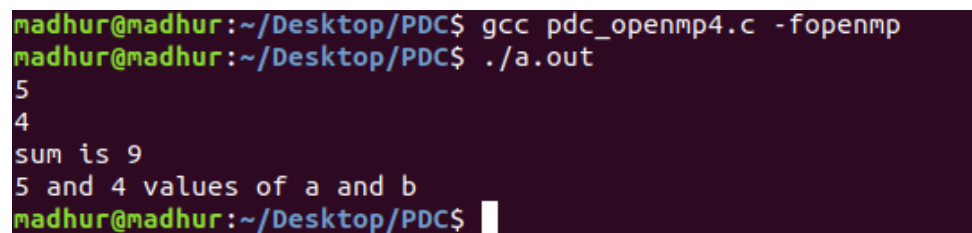
```
    }  
    if(ID == 1)  
    {  
        printf("%d and %d values of a and b\n",a,b);  
    }  
}  
return 0;  
}
```

(Private)

```
#include <stdio.h>  
#include <stdlib.h>  
#include <omp.h>  
  
int main(){  
  
    int a;  
    int b;  
    int ID;  
  
    scanf("%d",&a);  
    scanf("%d",&b);  
  
    omp_set_num_threads(2);  
  
    #pragma omp parallel private(a,b) private(ID)  
    {  
        ID = omp_get_thread_num();  
  
        if(ID == 0)  
        {  
            printf("sum is %d\n",(a+b));  
        }  
        if(ID == 1)  
        {  
            printf("%d and %d values of a and b\n",a,b);  
        }  
    }  
    return 0;  
}
```

EXECUTION:

(Shared)



```
madhur@madhur:~/Desktop/PDC$ gcc pdc_openmp4.c -fopenmp  
madhur@madhur:~/Desktop/PDC$ ./a.out  
5  
4  
sum is 9  
5 and 4 values of a and b  
madhur@madhur:~/Desktop/PDC$
```

(Private)

```
madhur@madhur:~/Desktop/PDC$ gcc pdc_openmp4.c -fopenmp
madhur@madhur:~/Desktop/PDC$ ./a.out
5
4
sum is 0
0 and 0 values of a and b
madhur@madhur:~/Desktop/PDC$
```

RESULTS:

Hence it can be seen that when the variables are declared as shared then the values are accessed but when the variables are declared as private then the variables are initialized to zero since these are new instances unique to the threads.

SCENARIO – II

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.

BRIEF ABOUT YOUR APPROACH:

Using the Library classes available in openmp we can find out the environment conditions of the compiler when in use.

SOURCE CODE:

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

int main(){

    omp_set_num_threads(5);

    printf("\nThe number of threads received is %d",omp_get_num_threads());
    printf("\nThe number of processors outside parallel is: %d",omp_get_num_procs());

    #pragma omp parallel
    {
        int ID = omp_get_thread_num();

        if(ID == 0){
            printf("\nThe number of threads in parallel region: %d",omp_get_num_threads());
            printf("\nThe number of processors inside parallel is: %d",omp_get_num_procs());
        }

        if(ID == 1){
            printf("\nThe dynamic runtime adjustment is %s",omp_get_dynamic() ? "true" : "false");
        }
        if(ID == 2){
            printf("\nThe nested parallelism adjustment is %s",omp_get_nested() ? "true" : "false");
        }

    }

    printf("\n");

    return 0;
}
```

EXECUTION:

```
madhur@madhur:~/Desktop/PDC$ gcc pdc_openmp5.c -fopenmp
madhur@madhur:~/Desktop/PDC$ ./a.out

The number of threads received is 1
The number of processors outside parallel is: 4
The dynamic runtime adjustment is false
The nested parallelism adjustment is false
The number of threads in parallel region: 5
The number of processors inside parallel is: 4
madhur@madhur:~/Desktop/PDC$
```

RESULTS:

Hence on running the code we see the environment conditions of the compiler in use.

SCENARIO – III

By using the private() and shared() directives, you can specify variables within the parallel region as being shared, i.e. visible and accessible by all threads simultaneously, or private, i.e. private to each thread, meaning each thread will have its own local copy.

Write a simple OpenMP program that uses loop iteration counters in OpenMP loop. It constructs the for loop to be privately accessed variable (in case of executing the i variable).

It is helpful to understand how threads make use of the cores to enable course-grain parallelism.

Hint: Output Execution

```
Thread 4 performed 125 iterations of the loop.
Thread 7 performed 125 iterations of the loop.
Thread 2 performed 125 iterations of the loop.
Thread 6 performed 125 iterations of the loop.
Thread 5 performed 125 iterations of the loop.
Thread 0 performed 125 iterations of the loop.
Thread 3 performed 125 iterations of the loop.
Thread 1 performed 125 iterations of the loop.
```

BRIEF ABOUT YOUR APPROACH:

Shared variables are all declared variables outside the block declared as parallel, private variables are those where we need every thread to have a unique instance of the declared variable. Variables declared within the parallel scope are by default private and hence can work independantly

SOURCE CODE:

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

int add(int a,int b)
{
    return (a+b);
}

int sub(int a, int b)
{
    return (a-b);
}

int mul(int a, int b)
{
    return (a*b);
}

int main()
{
    int a;
    int b;
    int ID;

    printf("Enter value of a: ");
    scanf("%d",&a);
    printf("Enter value of b: ");
    scanf("%d",&b);
    printf("\n");

    omp_set_num_threads(3);

    #pragma omp parallel shared(a,b) private(ID)
    {
        ID = omp_get_thread_num();

        if(ID == 0)
        {
            printf("%d thread gave value %d on addition\n",ID,add(a,b));
        }
        if(ID == 1)
        {
            printf("%d thread gave value %d on subtraction\n",ID,sub(a,b));
        }
        if(ID == 2)
        {
            printf("%d thread gave value %d on multiplication\n",ID,mul(a,b));
        }
    }

    return 0;
}
```

EXECUTION:

```
madhur@madhur:~/Desktop/PDC$ gcc pdc_openmp6.c -fopenmp
madhur@madhur:~/Desktop/PDC$ ./a.out
Enter value of a: 150
Enter value of b: 200

0 thread gave value 350 on addition
2 thread gave value 30000 on multiplication
1 thread gave value -50 on subtraction
madhur@madhur:~/Desktop/PDC$
```

RESULTS:

Therefore, we see the 3 threads run concurrently with the same shared values of a and b but all have different ID values. This is because the values a and b are declared as shared while the ID value is declared as private.