

Nithin S
221IT085

IT301 Lab Assignment 1

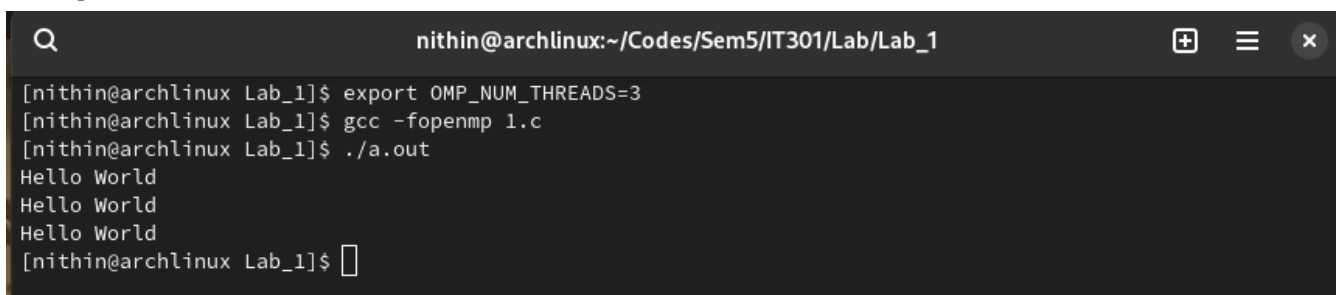
Note: My System has 6 cores , so default no of threads is 12.

Q1. Write an Open MP program to print Hello World by setting the required number of threads using the Environment Variable in the program.

Code :

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

Output :

A terminal window titled 'nithin@archlinux:~/Codes/Sem5/IT301/Lab/Lab_1' showing the execution of an OpenMP program. The user sets the environment variable OMP_NUM_THREADS=3, compiles the program with gcc -fopenmp 1.c, and runs it with ./a.out. The output shows 'Hello World' printed three times, once for each thread.

```
nithin@archlinux:~/Codes/Sem5/IT301/Lab/Lab_1
[nithin@archlinux Lab_1]$ export OMP_NUM_THREADS=3
[nithin@archlinux Lab_1]$ gcc -fopenmp 1.c
[nithin@archlinux Lab_1]$ ./a.out
Hello World
Hello World
Hello World
[nithin@archlinux Lab_1]$
```

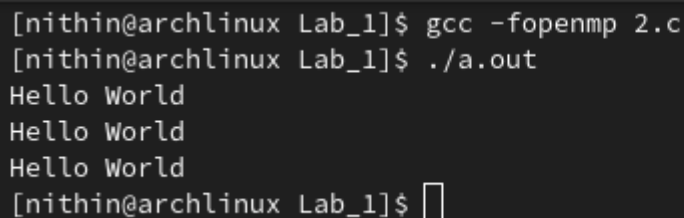
Analysis : This OpenMP code prints "Hello World" multiple times, once for each thread in a parallel region. The number of lines printed depends on the number of threads used, which defaults to the number of available CPU cores.

Q2. Write a simple OpenMP program to print Hello World by adding appropriate clause to the parallel directive in order to set the number of threads

Code :

```
#include<stdio.h>
#include<omp.h>
int main()
{
    #pragma omp parallel num_threads(3)
    printf("Hello World\n");
    return 0;
}
```

Output :

A terminal window with a dark background and light green text. It shows the compilation and execution of a C program. The user is at a prompt in a directory named 'Lab_1' on an 'archlinux' machine. They compile a file '2.c' using 'gcc -fopenmp' and then run the resulting executable 'a.out'. The output shows 'Hello World' printed three times, once on each line, before the prompt returns.

```
[nithin@archlinux Lab_1]$ gcc -fopenmp 2.c
[nithin@archlinux Lab_1]$ ./a.out
Hello World
Hello World
Hello World
[nithin@archlinux Lab_1]$
```

Analysis : This OpenMP code prints "Hello World" three times. The `#pragma omp parallel num_threads(3)` directive creates a parallel region with exactly 3 threads, so the `printf` statement is executed by each of the 3 threads.

Q3. Demonstrate the shared variable concept using a simple OpenMP program.

Code :

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

void main() {
    int var = 98;

    #pragma omp parallel default(none) shared(var)
    {
        printf("Value of var: %d\n", var);
    }
}
```

Output:

[illegible]

Analysis:

This OpenMP code prints the value of var in a parallel region:

`#pragma omp parallel default(none) shared(var):` Creates a parallel region with var shared among all threads. The `default(none)` clause ensures that all variables must be explicitly specified as shared or private.

The `printf` statement prints the value of var from each thread. Since var is shared, all threads see the same value, which is 98.

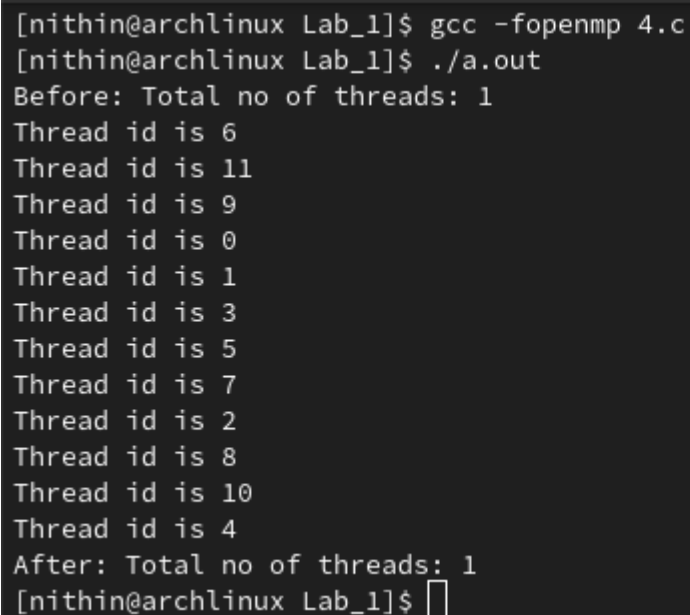
The output will show "Value of var: 98" multiple times, depending on the number of threads which is 12 in my case.

Q4. Demonstrate the fork-join parallel execution model in an OpenMP program.

Code:

```
#include<stdio.h>
#include<omp.h>
int main()
{
    printf("Before: Total no of threads: %d\n",omp_get_num_threads());
    #pragma omp parallel
    {
        printf("Thread id is %d\n",omp_get_thread_num());
    }
    printf("After: Total no of threads: %d\n",omp_get_num_threads());
    return 0;
}
```

Output:



```
[nithin@archlinux Lab_1]$ gcc -fopenmp 4.c
[nithin@archlinux Lab_1]$ ./a.out
Before: Total no of threads: 1
Thread id is 6
Thread id is 11
Thread id is 9
Thread id is 0
Thread id is 1
Thread id is 3
Thread id is 5
Thread id is 7
Thread id is 2
Thread id is 8
Thread id is 10
Thread id is 4
After: Total no of threads: 1
[nithin@archlinux Lab_1]$
```

Analysis:

This OpenMP code prints thread information:

- ``omp_get_num_threads()`` before and after the parallel region will show ``1`` because it's called outside the parallel region.
- Inside the parallel region, ``omp_get_thread_num()`` prints each thread's ID, forking to generate threads.
- After parallel region all threads join together.
- The number of threads used defaults to the number of CPU cores unless set otherwise.

Q5. Write an OpenMP program to extract the thread identifiers from total number of threads using private clause.

Code :

```
#include<stdio.h>
#include<omp.h>
int main()
{
    int tid;
    #pragma omp parallel private(tid)
    {
        tid=omp_get_thread_num();
        printf("Thread no %d from total %d threads\n",tid,omp_get_num_threads());
    }
    return 0;
}
```

Output:

```
[nithin@archlinux Lab_1]$ gcc -fopenmp 5.c
[nithin@archlinux Lab_1]$ ./a.out
Thread no 11 from total 12 threads
Thread no 5 from total 12 threads
Thread no 9 from total 12 threads
Thread no 2 from total 12 threads
Thread no 3 from total 12 threads
Thread no 7 from total 12 threads
Thread no 4 from total 12 threads
Thread no 10 from total 12 threads
Thread no 6 from total 12 threads
Thread no 0 from total 12 threads
Thread no 1 from total 12 threads
Thread no 8 from total 12 threads
[nithin@archlinux Lab_1]$
```

Analysis

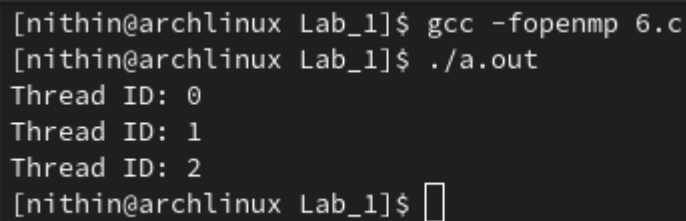
This code uses OpenMP to create a parallel region where each thread prints its thread number (`tid`) and the total number of threads. The `private(tid)` directive ensures that each thread has its own private copy of the `tid` variable, avoiding conflicts. The `omp_get_thread_num()` function retrieves the thread number, while `omp_get_num_threads()` returns the total number of threads in the parallel region.

Q6. Write an OpenMP program to set the number of OpenMP threads and retrieve the thread ID number at runtime.

Code :

```
#include<stdio.h>
#include<omp.h>
int main()
{
    #pragma omp parallel num_threads(3)
    {
        printf("Thread ID: %d\n",omp_get_thread_num());
    }
    return 0;
}
```

Output :



```
[nithin@archlinux Lab_1]$ gcc -fopenmp 6.c
[nithin@archlinux Lab_1]$ ./a.out
Thread ID: 0
Thread ID: 1
Thread ID: 2
[nithin@archlinux Lab_1]$
```

Analysis:

This code uses OpenMP to create a parallel region with exactly 3 threads (`num_threads(3)`). Each thread prints its thread ID using `omp_get_thread_num()`. The output shows the thread ID for each of the three threads executing concurrently.

Q7. Write a simple OpenMP program to demonstrate the parallel loop construct.

Code:

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

int main()
{
    int num_procs = omp_get_num_procs();
    printf("Number of processors available: %d\n", num_procs);

    omp_set_num_threads(4);

    #pragma omp parallel
    {
        #pragma omp for
        for (int i = 0; i < 8; i++) {
            int thread_id = omp_get_thread_num();
            printf("Thread %d: Hello World\n", thread_id);
        }
    }

    printf("Number of threads used: %d\n", omp_get_num_threads());

    if (omp_in_parallel()) {
        printf("Currently in a parallel region.\n");
    } else {
        printf("Not in a parallel region.\n");
    }

    int dynamic_enabled = omp_get_dynamic();
    printf("Dynamic threads enabled: %d\n", dynamic_enabled);

    int nested_enabled = omp_get_nested();
    printf("Nested parallelism enabled: %d\n", nested_enabled);
}
```

```
    return 0;  
}
```

Output:

```
[nithin@archlinux Lab_1]$ gcc -fopenmp 7.c  
[nithin@archlinux Lab_1]$ ./a.out  
Number of processors available: 12  
Thread 0: Hello World  
Thread 0: Hello World  
Thread 3: Hello World  
Thread 3: Hello World  
Thread 2: Hello World  
Thread 2: Hello World  
Thread 1: Hello World  
Thread 1: Hello World  
Number of threads used: 1  
Not in a parallel region.  
Dynamic threads enabled: 0  
Nested parallelism enabled: 0  
[nithin@archlinux Lab_1]$
```

Analysis :

It first retrieves and prints the number of processors available using `omp_get_num_procs()`.

It sets the number of threads to 4 with `omp_set_num_threads(4)`.

Inside the parallel region, a loop is divided among the threads using `#pragma omp for`, with each thread printing its ID and "Hello World."

After the parallel region, it prints the number of threads used and checks if the code is currently in a parallel region using `omp_in_parallel()`.

Finally, it checks and prints the status of dynamic thread adjustment and nested parallelism using `omp_get_dynamic()` and `omp_get_nested()`.

Q8. Write a simple OpenMP program to illustrate the usage of shared clause.

Code:

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

int main() {
    int sharedVar=0;
    #pragma omp parallel shared(sharedVar)
    {
        #pragma omp critical
        {
            printf("Thread no %d : sharedVar = %d\n",omp_get_thread_num(),sharedVar++);
        }
    }

    return 0;
}
```

Output:

```
[nithin@archlinux Lab_1]$ gcc -fopenmp 8.c
[nithin@archlinux Lab_1]$ ./a.out
Thread no 9 : sharedVar = 0
Thread no 11 : sharedVar = 1
Thread no 1 : sharedVar = 2
Thread no 3 : sharedVar = 3
Thread no 2 : sharedVar = 4
Thread no 6 : sharedVar = 5
Thread no 10 : sharedVar = 6
Thread no 4 : sharedVar = 7
Thread no 7 : sharedVar = 8
Thread no 8 : sharedVar = 9
Thread no 0 : sharedVar = 10
Thread no 5 : sharedVar = 11
[nithin@archlinux Lab_1]$
```

Analysis:

This code demonstrates the use of OpenMP's `#pragma omp critical` to safely update and print a shared variable (`sharedVar`) in a parallel region. The `sharedVar` is accessible by all threads, but the `#pragma omp critical` section ensures that only one thread at a time can increment and print the variable, preventing race conditions. Each thread prints its ID and the current value of `sharedVar`, then increments it.

Q9. Demonstrate the use of private clause in a simple OpenMP program.

Code:

```
#include<stdio.h>
#include<omp.h>
int main()
{
    int tid;
    int pVar;
    #pragma omp parallel private(tid,pVar)
    {
        tid=omp_get_thread_num();
        pVar=0;
        if(tid==0){
            pVar++;
        }
        printf("Thread no %d: pVar is %d\n",tid,pVar);
    }
    return 0;
}
```

Output:

```
[nithin@archlinux Lab_1]$ gcc -fopenmp 9.c
[nithin@archlinux Lab_1]$ ./a.out
Thread no 4: pVar is 0
Thread no 8: pVar is 0
Thread no 11: pVar is 0
Thread no 7: pVar is 0
Thread no 6: pVar is 0
Thread no 2: pVar is 0
Thread no 9: pVar is 0
Thread no 1: pVar is 0
Thread no 5: pVar is 0
Thread no 10: pVar is 0
Thread no 3: pVar is 0
Thread no 0: pVar is 1
[nithin@archlinux Lab_1]$
```

Analysis:

This code uses OpenMP to create a parallel region where each thread has its own private copies of `tid` and `pVar` due to the `private(tid, pVar)` directive. Each thread initializes `pVar` to 0 and retrieves its thread ID using `omp_get_thread_num()`. If the thread ID is 0, `pVar` is incremented by 1. Then, each thread prints its ID and the value of `pVar`. Since `pVar` is private, the value is 1 only for thread 0 and remains 0 for all other threads.

Q10. Write a simple OpenMP program that uses if clause to get the desired output.

Case 1 :

Code:

```
#include<stdio.h>
#include<omp.h>
int main()
{
    int isParallel=0;
    #pragma omp parallel if(isParallel==1) num_threads(4)
    {
        printf("Hello\n");
    }
    return 0;
}
```

Output:

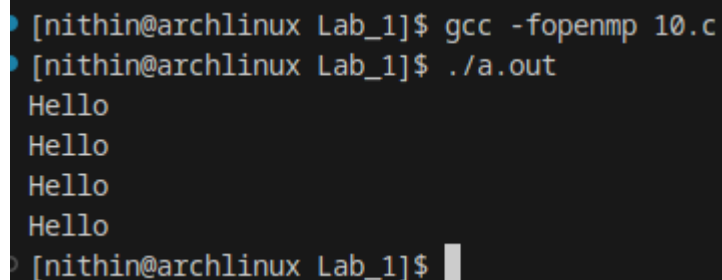
```
[nithin@archlinux Lab_1]$ gcc -fopenmp 10.c
[nithin@archlinux Lab_1]$ ./a.out
Hello
[nithin@archlinux Lab_1]$
```


Case 2:

Code:

```
#include<stdio.h>
#include<omp.h>
int main()
{
    int isParallel=1;
    #pragma omp parallel if(isParallel==1) num_threads(4)
    {
        printf("Hello\n");
    }
    return 0;
}
```

Output:



```
[nithin@archlinux Lab_1]$ gcc -fopenmp 10.c
[nithin@archlinux Lab_1]$ ./a.out
Hello
Hello
Hello
Hello
[nithin@archlinux Lab_1]$
```

Analysis:

This code uses OpenMP to conditionally create a parallel region based on the value of `isParallel`. Here's how the code behaves in two cases:

****Case 1: `isParallel = 0`****

- The `if(isParallel==1)` condition is false, so the `#pragma omp parallel` directive is ignored.
- The code runs sequentially in a single thread, printing "Hello" only once.

****Case 2: `isParallel = 1`****

- The `if(isParallel==1)` condition is true, so a parallel region is created with 4 threads (`num_threads(4)`).

- Each of the 4 threads prints "Hello," resulting in "Hello" being printed four times, each potentially by a different thread.

In summary:

- ****Case 1****: The code runs sequentially and prints "Hello" once.

- ****Case 2****: The code runs in parallel with 4 threads, each printing "Hello."

Q11. Write a simple OpenMP program to demonstrate the parallel loop construct

Code

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

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

```
int main() {  
    int i;  
    int sum = 0;  
    omp_set_num_threads(4);
```

```
    #pragma omp parallel  
    {
```

```
        #pragma omp for reduction(+:sum)  
        for (i = 1; i <= 10; i++) {  
            sum += i;
```

```
            printf("Thread %d: i = %d\n", omp_get_thread_num(), i);  
        }  
    }
```

```
    printf("Total sum: %d\n", sum);
```

```
    return 0;
```

```
}
```

Output

```
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_2$ gcc 6.c -fopenmp
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_2$ ./a.out
Thread 0: i = 1
Thread 2: i = 7
Thread 2: i = 8
Thread 1: i = 4
Thread 1: i = 5
Thread 1: i = 6
Thread 3: i = 9
Thread 3: i = 10
Thread 0: i = 2
Thread 0: i = 3
Total sum: 55
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_2$
```

Analysis

This C code calculates the sum of integers from 1 to 10 using OpenMP to parallelize the summation process. The program sets up 4 threads with `omp_set_num_threads(4)` and uses the `#pragma omp parallel` directive to initiate parallel execution.

Within this parallel region, the `#pragma omp for reduction(+:sum)` directive is employed to divide the loop iterations among the threads, ensuring that each thread processes a portion of the summation. The `reduction(+:sum)` clause is critical as it combines the partial sums computed by each thread into the final sum variable, avoiding race conditions.

The program also prints which thread processes each value of `i`, showcasing the work distribution. Finally, the total sum is printed. This approach demonstrates efficient parallel computation with OpenMP, where the reduction operation safely aggregates results from multiple threads.

