

Lab Exercise No: 02

(1).Study of the following:

- (a). Parallel computing and parallel programming languages
- (b). Introduction to Open MP

(a). Parallel computing

- It involves using several processing components at once to address any issue.
- As each resource that has been used to work on a problem is active at the same time, problems are decomposed into instructions and solved simultaneously.
- Modern computers have evolved parallel processing as an efficient technology to address the demand for increased performance, decreased cost, and precise results in practical applications.
- Today's computers frequently experience concurrent events as a result of multiprogramming, multiprocessing, or multicomputing techniques.
- Software programmes are robust and sophisticated on modern systems.
- We must first comprehend the fundamental evolution of both hardware and software in order to examine the growth of computer performance.
- The process of running numerous processors an application or computation simultaneously is referred to as parallel computing.
- In general, it refers to a type of computing architecture where big issues are divided into separate, smaller, typically related sections that can be processed all at once.
- Multiple CPUs work together to complete it by exchanging information across shared memory, which then combines the findings. It facilitates the execution of complex computations by distributing the enormous problem among multiple processors.
- By boosting the systems' available computational capability, parallel computing also facilitates quicker application processing and task resolution.
- The majority of supercomputers run on parallel computing concepts. In general, parallel processing is employed in operational scenarios that require a lot of computing or processing capacity.

## **Types of Parallelism: -**

- 1. Bit-level parallelism**
- 2. Instruction-level parallelism**
- 3. Task Parallelism**
- 4. Data-level parallelism (DLP)**

### **Bit-level parallelism**

- It is a type of parallel computing that relies on larger and larger processors. It lessens the quantity of instructions the system needs to carry out in order to complete a task on large-scale data.
- Example: Imagine that an 8-bit processor is required to calculate the sum of two 16-bit numbers. It needs two instructions to complete the operation because it must first add the 8 higher-order bits and then sum the 8 lower-order bits. One instruction is all that is needed to complete the operation on a 16-bit processor.
- the type of parallel computing where each work is subject to the word size of the processor. It lessens the quantity of instructions the processor needs to carry out a task on large-sized data.
- It is necessary to divide the process into a number of instructions. An 8-bit CPU, for instance, might be used to perform an operation on 16-bit values.
- The 8 lower-order bits must be operated first, followed by the 8 higher-order bits. Therefore, to complete the operation, two instructions are required. A 16-bit processor has the ability to complete the operation with just one instruction.

### **Instruction-level parallelism**

- For each phase of a clock cycle, a processor can only address fewer than one instruction.
- These instructions can be rearranged and aggregated before being executed concurrently with no impact on the program's outcome. Instruction-level parallelism is the term for this.
- The processor chooses at instruction-level parallelism how many instructions are executed concurrently during a single CPU clock cycle. In instruction-level parallelism, a processor may be able to address that is less than one instruction for each phase of the clock cycle.
- Static parallelism, in which the computer chooses which instructions to execute simultaneously, is the basis of the software approach to instruction-level parallelism.

## **Task Parallelism**

- When a task is divided into smaller tasks, task parallelism assigns a specific amount of time for each subtask to be completed. Concurrently with one another, the processors complete the subtasks.
- Task parallelism is a type of parallelism where jobs are broken down into smaller tasks. Each subtask is then given a time slot for execution. Additionally, processors carry out subtasks concurrently.

## **Data-level parallelism (DLP)**

- Instructions from one stream are applied to many data streams concurrently. Memory bandwidth and irregular data manipulation patterns place restrictions

## **Applications of Parallel Computing**

There are various applications of Parallel Computing, which are as follows:

- One of the primary applications of parallel computing is Databases and Data mining.
- The real-time simulation of systems is another use of parallel computing.
- The technologies, such as Networked videos and Multimedia.
- Science and Engineering.
- Collaborative work environments.
- The concept of parallel computing is used by augmented reality, advanced graphics, and virtual reality.

## **Limitations of Parallel Computing:**

- It handles issues including the challenging synchronisation and communication between several sub-tasks and processes.
- The algorithms need to be handled in a way that allows for parallel processing.
- The algorithms or programmes need to be very cohesive and have little coupling. But developing such programmes is challenging.
- A parallelism-based software can be efficiently created by programmers with higher levels of technical expertise.

## **Advantages of Parallel computing**

Parallel computing advantages are discussed below:

- In parallel computing, more resources are used to complete the task that led to decrease the time and cut possible costs. Also, cheap components are used to construct parallel clusters.
- Comparing with Serial Computing, parallel computing can solve larger problems in a short time.
- For simulating, modeling, and understanding complex, real-world phenomena, parallel computing is much appropriate while comparing with serial computing.
- When the local resources are finite, it can offer benefit you over non-local resources.
- There are multiple problems that are very large and may impractical or impossible to solve them on a single computer; the concept of parallel computing helps to remove these kinds of issues.
- One of the best advantages of parallel computing is that it allows you to do several things in a time by using multiple computing resources.
- Furthermore, parallel computing is suited for hardware as serial computing wastes the potential computing power.

### Disadvantages of Parallel Computing

There are many limitations of parallel computing, which are as follows:

- It addresses Parallel architecture that can be difficult to achieve.
- In the case of clusters, better cooling technologies are needed in parallel computing.
- It requires the managed algorithms, which could be handled in the parallel mechanism.
- The multi-core architectures consume high power consumption.
- The parallel computing system needs low coupling and high cohesion, which is difficult to create.
- The code for a parallelism-based program can be done by the most technically skilled and expert programmers.

### Parallel programming languages

- Languages created specifically for algorithm and application programming on parallel computers are known as parallel programming languages.
- In many application domains, parallel processing offers a huge possibility for creating high performance systems and resolving significant issues.
- Parallel computers with tens to thousands of computational elements have become commercially available over the past few years.

- They are still being acknowledged as effective instruments for engineering, information management, and scientific research. This trend is being driven by parallel programming tools and languages that let parallel computers be helpful for a variety of applications.
- In order to facilitate the creation and development of programmes on parallel computers, numerous models and languages have been developed and put into use.
- The creation of parallel algorithms as a collection of concurrent activities mapped onto various computing units is made possible by parallel programming languages, also known as concurrent languages.
- Depending on the language chosen, there are various ways that two or more actions might cooperate.
- The development of parallel computer programming languages and software tools is crucial for the widespread adoption and effective use of these innovative architectures.
- High-level languages facilitate the use of parallel computers by novice users and reduce the design and execution times of parallel applications.

#### (b). Introduction to Open MP

- An Application Program Interface (API) called OpenMP can be used in C/C++ programmes to directly direct multi-threaded, shared memory parallelism.
- Since the OpenMP instructions are written as pragmas that the compiler interprets, it doesn't interfere with the original serial code.
- The fork-join model of parallel execution is used by OpenMP.
- Every OpenMP application starts with a single master thread that runs sequentially until it encounters a parallel zone, at which point it spawns a group of parallel threads (FORK).
- Only the master thread continues to run sequentially after the team threads synchronise and terminate after finishing the parallel region (JOIN).

#### OpenMP core syntax

Most of the constructs in OpenMP are compiler directives.

```
#pragma omp construct [clause [clause]...] □
```

Example

```
#pragma omp parallel num_threads(4)
```

Function prototypes and types in the file:

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

Most OpenMP\* constructs apply to a “structured block”.

- Structured block: a block of one or more statements with one point of entry at the top and one point of exit at the bottom.

- It's OK to have an `exit()` within the structured block.

## Hello World Example

Here is a basic example showing how to parallelize a hello world program. First, the serial version:

```
#include <stdio.h>
int main()
{
    printf( "Hello, World from just me!\n" );
    return 0;
}
```

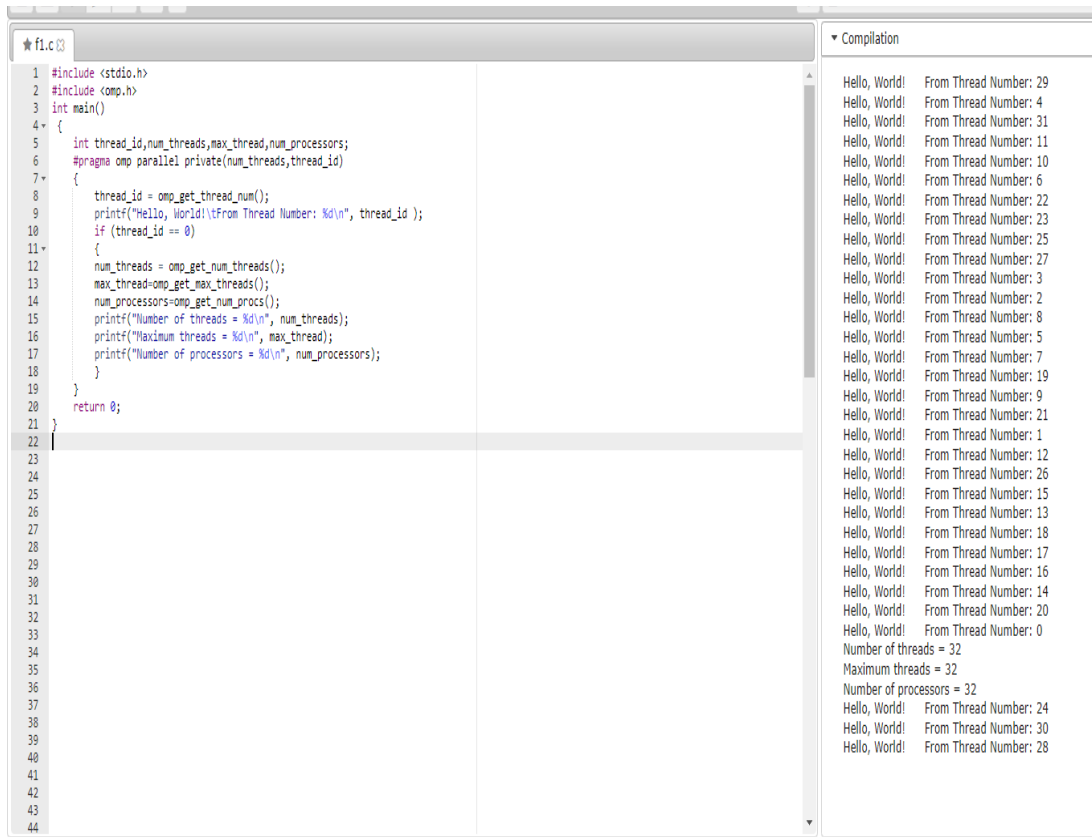
To do this in parallel (have a series of threads print out a “Hello World!” statement), we would do the following:

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

(2) Write a C-program using Open MP to print hello world.

To print the following environment detail:

Number of threads, Thread number, Number of processors and Maximum threads with sample message.



```
1 #include <stdio.h>
2 #include <omp.h>
3 int main()
4 {
5     int thread_id,num_threads,max_thread,num_processors;
6     #pragma omp parallel private(num_threads,thread_id)
7     {
8         thread_id = omp_get_thread_num();
9         printf("Hello, World!\tFrom Thread Number: %d\n", thread_id );
10        if (thread_id == 0)
11        {
12            num_threads = omp_get_num_threads();
13            max_thread=omp_get_max_threads();
14            num_processors=omp_get_num_procs();
15            printf("Number of threads = %d\n", num_threads);
16            printf("Maximum threads = %d\n", max_thread);
17            printf("Number of processors = %d\n", num_processors);
18        }
19    }
20    return 0;
21 }
```

Compilation

Hello, World! From Thread Number: 29  
Hello, World! From Thread Number: 4  
Hello, World! From Thread Number: 31  
Hello, World! From Thread Number: 11  
Hello, World! From Thread Number: 10  
Hello, World! From Thread Number: 6  
Hello, World! From Thread Number: 22  
Hello, World! From Thread Number: 23  
Hello, World! From Thread Number: 25  
Hello, World! From Thread Number: 27  
Hello, World! From Thread Number: 3  
Hello, World! From Thread Number: 2  
Hello, World! From Thread Number: 8  
Hello, World! From Thread Number: 5  
Hello, World! From Thread Number: 7  
Hello, World! From Thread Number: 19  
Hello, World! From Thread Number: 9  
Hello, World! From Thread Number: 21  
Hello, World! From Thread Number: 1  
Hello, World! From Thread Number: 12  
Hello, World! From Thread Number: 26  
Hello, World! From Thread Number: 15  
Hello, World! From Thread Number: 13  
Hello, World! From Thread Number: 18  
Hello, World! From Thread Number: 17  
Hello, World! From Thread Number: 16  
Hello, World! From Thread Number: 14  
Hello, World! From Thread Number: 20  
Hello, World! From Thread Number: 0  
Number of threads = 32  
Maximum threads = 32  
Number of processors = 32  
Hello, World! From Thread Number: 24  
Hello, World! From Thread Number: 30  
Hello, World! From Thread Number: 28

```
#include <stdio.h>
#include <omp.h>
int main()
{
    int thread_id,num_threads,max_thread,num_processors;
    #pragma omp parallel private(num_threads,thread_id)
    {
        thread_id = omp_get_thread_num();
        printf("Hello, World!\tFrom Thread Number: %d\n", thread_id );
        if (thread_id == 0)
        {
            num_threads = omp_get_num_threads();
            max_thread=omp_get_max_threads();
            num_processors=omp_get_num_procs();
            printf("Number of threads = %d\n", num_threads);
            printf("Maximum threads = %d\n", max_thread);
            printf("Number of processors = %d\n", num_processors);
        }
    }
}
```

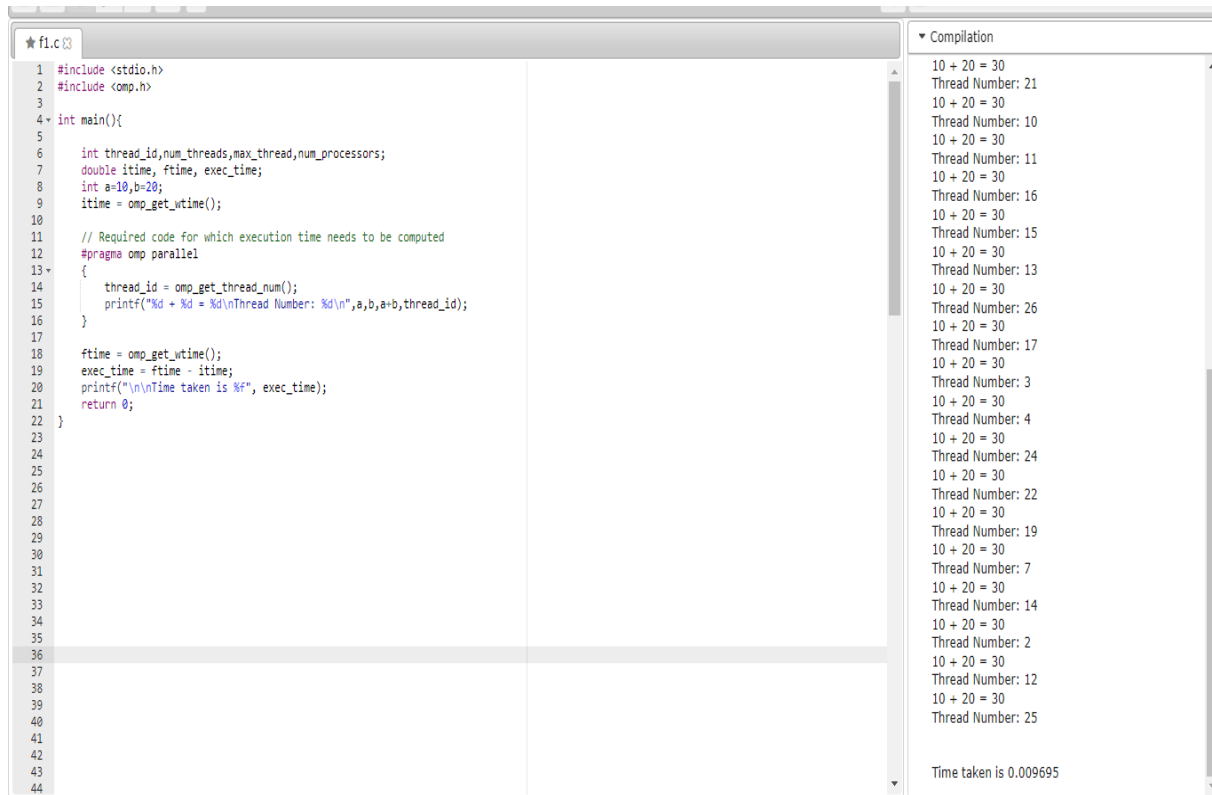
```
    }  
    return 0;  
}
```

OUTPUT : -

```
Hello, World!   From Thread Number: 29  
Hello, World!   From Thread Number: 4  
Hello, World!   From Thread Number: 31  
Hello, World!   From Thread Number: 11  
Hello, World!   From Thread Number: 10  
Hello, World!   From Thread Number: 6  
Hello, World!   From Thread Number: 22  
Hello, World!   From Thread Number: 23  
Hello, World!   From Thread Number: 25  
Hello, World!   From Thread Number: 27  
Hello, World!   From Thread Number: 3  
Hello, World!   From Thread Number: 2  
Hello, World!   From Thread Number: 8  
Hello, World!   From Thread Number: 5  
Hello, World!   From Thread Number: 7  
Hello, World!   From Thread Number: 19  
Hello, World!   From Thread Number: 9  
Hello, World!   From Thread Number: 21  
Hello, World!   From Thread Number: 1  
Hello, World!   From Thread Number: 12  
Hello, World!   From Thread Number: 26  
Hello, World!   From Thread Number: 15  
Hello, World!   From Thread Number: 13  
Hello, World!   From Thread Number: 18  
Hello, World!   From Thread Number: 17  
Hello, World!   From Thread Number: 16  
Hello, World!   From Thread Number: 14  
Hello, World!   From Thread Number: 20  
Hello, World!   From Thread Number: 0  
Number of threads = 32  
Maximum threads = 32  
Number of processors = 32  
Hello, World!   From Thread Number: 24  
Hello, World!   From Thread Number: 30  
Hello, World!   From Thread Number: 28
```

(3) Write a c-program using open MP to perform the arithmetic operations and logical operations between two integers using multiple threads and measure the time?





```
1 #include <stdio.h>
2 #include <omp.h>
3
4 int main(){
5
6     int thread_id,num_threads,max_thread,num_processors;
7     double itime, ftime, exec_time;
8     int a=10,b=20;
9     itime = omp_get_wtime();
10
11     // Required code for which execution time needs to be computed
12     #pragma omp parallel
13     {
14         thread_id = omp_get_thread_num();
15         printf("%d + %d = %d\nThread Number: %d\n",a,b,a+b,thread_id);
16     }
17
18     ftime = omp_get_wtime();
19     exec_time = ftime - itime;
20     printf("\n\nTime taken is %f", exec_time);
21     return 0;
22 }
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
```

Compilation

10 + 20 = 30  
Thread Number: 21  
10 + 20 = 30  
Thread Number: 10  
10 + 20 = 30  
Thread Number: 11  
10 + 20 = 30  
Thread Number: 16  
10 + 20 = 30  
Thread Number: 15  
10 + 20 = 30  
Thread Number: 13  
10 + 20 = 30  
Thread Number: 26  
10 + 20 = 30  
Thread Number: 17  
10 + 20 = 30  
Thread Number: 3  
10 + 20 = 30  
Thread Number: 4  
10 + 20 = 30  
Thread Number: 24  
10 + 20 = 30  
Thread Number: 22  
10 + 20 = 30  
Thread Number: 19  
10 + 20 = 30  
Thread Number: 7  
10 + 20 = 30  
Thread Number: 14  
10 + 20 = 30  
Thread Number: 2  
10 + 20 = 30  
Thread Number: 12  
10 + 20 = 30  
Thread Number: 25

Time taken is 0.009695

The screenshot shows a web browser window with a C compiler interface. The left pane displays a C program named `f1.c` that uses OpenMP for parallel execution. The program calculates the sum of two numbers, `num1` and `num2`, using a parallel region with a private thread. The right pane shows the compilation output, which includes the OpenMP version, the number of threads, and the total time taken by each thread to complete the calculation.

```
1 #include <omp.h>
2 #include <stdio.h>
3 void main(void)
4 {
5     double start_time, end_time, total_time;
6     start_time = omp_get_wtime();
7     int tid;
8
9     int num1 = 20;
10    int num2 = 10;
11    #pragma omp parallel private(tid)
12
13
14    {
15        tid = omp_get_thread_num();
16
17        if(tid==0)
18        {
19            printf("id = %d \n", num1, num2, num1+num2, tid);
20        }
21        else if(tid==1)
22        {
23            printf("id = %d \n", num1, num2, num1+num2, tid);
24        }
25        else if(tid==2)
26        {
27            printf("id = %d \n", num1, num2, num1+num2, tid);
28        }
29        else
30        {
31            printf("id = %d \n", num1, num2, num1+num2, tid);
32        }
33    }
34    end_time = omp_get_wtime();
35    total_time = end_time - start_time;
36    printf("Total Time taken by thread %d is %f\n", tid, total_time);
37 }
38
39
40
```

Compilation Output:

```
20 / 10 = 2    from thread number 27
Total Time taken by thread 27 is 0.004750

20 / 10 = 2    from thread number 19
Total Time taken by thread 19 is 0.004890

20 / 10 = 2    from thread number 26
Total Time taken by thread 26 is 0.004950

20 / 10 = 2    from thread number 31
20 / 10 = 2    from thread number 28
Total Time taken by thread 28 is 0.005054

20 / 10 = 2    from thread number 23
Total Time taken by thread 23 is 0.005112

20 / 10 = 2    from thread number 7
20 / 10 = 2    from thread number 6
Total Time taken by thread 6 is 0.005199

20 / 10 = 2    from thread number 9
Total Time taken by thread 9 is 0.005257

20 - 10 = 10   from thread number 1
Total Time taken by thread 1 is 0.005356

20 * 10 = 200  from thread number 2
Total Time taken by thread 2 is 0.005433

20 / 10 = 2    from thread number 16
Total Time taken by thread 16 is 0.005536

20 / 10 = 2    from thread number 10
Total Time taken by thread 10 is 0.005593

20 + 10 = 30   from thread number 0
Total Time taken by thread 0 is 0.005636

20 / 10 = 2    from thread number 21
Total Time taken by thread 21 is 0.005712

20 / 10 = 2    from thread number 18
Total Time taken by thread 18 is 0.005793

20 / 10 = 2    from thread number 14
Total Time taken by thread 14 is 0.005842

20 / 10 = 2    from thread number 5
Total Time taken by thread 5 is 0.005890

20 / 10 = 2    from thread number 25
Total Time taken by thread 25 is 0.008716

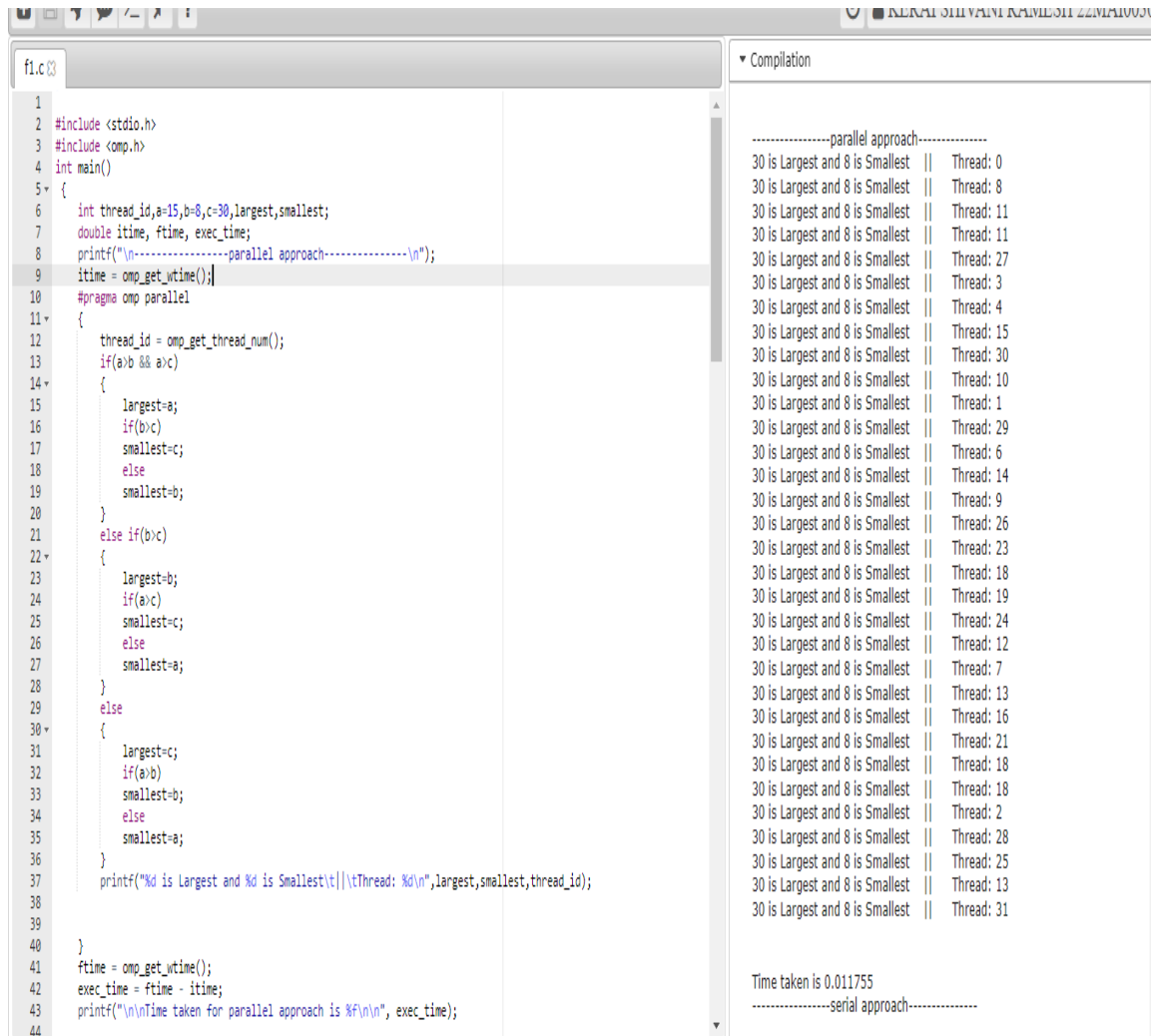
20 / 10 = 2    from thread number 11
Total Time taken by thread 11 is 0.008782

20 / 10 = 2    from thread number 8
```

(4) Write a c-program using open MP to find the largest and smallest among

three numbers using thread approach.

[Analyse the time between serial and parallel approach]



```
1
2 #include <stdio.h>
3 #include <omp.h>
4 int main()
5 {
6     int thread_id,a=15,b=8,c=30,largest,smallest;
7     double itime, ftime, exec_time;
8     printf("\n-----parallel approach-----\n");
9     itime = omp_get_wtime();
10    #pragma omp parallel
11    {
12        thread_id = omp_get_thread_num();
13        if(a>b && a>c)
14        {
15            largest=a;
16            if(b>c)
17                smallest=c;
18            else
19                smallest=b;
20        }
21        else if(b>c)
22        {
23            largest=b;
24            if(a>c)
25                smallest=c;
26            else
27                smallest=a;
28        }
29        else
30        {
31            largest=c;
32            if(a>b)
33                smallest=b;
34            else
35                smallest=a;
36        }
37        printf("%d is Largest and %d is Smallest\t\t\tThread: %d\n",largest,smallest,thread_id);
38    }
39
40    ftime = omp_get_wtime();
41    exec_time = ftime - itime;
42    printf("\nTime taken for parallel approach is %f\n\n", exec_time);
43
44
```

Compilation

-----parallel approach-----

Output	Thread
30 is Largest and 8 is Smallest	0
30 is Largest and 8 is Smallest	8
30 is Largest and 8 is Smallest	11
30 is Largest and 8 is Smallest	11
30 is Largest and 8 is Smallest	27
30 is Largest and 8 is Smallest	3
30 is Largest and 8 is Smallest	4
30 is Largest and 8 is Smallest	15
30 is Largest and 8 is Smallest	30
30 is Largest and 8 is Smallest	10
30 is Largest and 8 is Smallest	1
30 is Largest and 8 is Smallest	29
30 is Largest and 8 is Smallest	6
30 is Largest and 8 is Smallest	14
30 is Largest and 8 is Smallest	9
30 is Largest and 8 is Smallest	26
30 is Largest and 8 is Smallest	23
30 is Largest and 8 is Smallest	18
30 is Largest and 8 is Smallest	19
30 is Largest and 8 is Smallest	24
30 is Largest and 8 is Smallest	12
30 is Largest and 8 is Smallest	7
30 is Largest and 8 is Smallest	13
30 is Largest and 8 is Smallest	16
30 is Largest and 8 is Smallest	21
30 is Largest and 8 is Smallest	18
30 is Largest and 8 is Smallest	18
30 is Largest and 8 is Smallest	2
30 is Largest and 8 is Smallest	28
30 is Largest and 8 is Smallest	25
30 is Largest and 8 is Smallest	13
30 is Largest and 8 is Smallest	31

Time taken is 0.011755

-----serial approach-----

```
1 #include<stdio.h>
2 #include<time.h>
3 #include <unistd.h>
4 void largest(int num1,int num2,int num3);
5 void smallest(int num1,int num2,int num3);
6
7 void main(void)
8 {
9     time_t start = time(NULL);
10    printf("Start time is : %ld\n",start);
11    int num1 = 11;
12    int num2 = 39; int num3 = 89;
13    smallest(num1,num2,num3);
14    time_t end = time(NULL);
15    printf("End time is : %ld\n",end);
16    printf("The elapsed time is %ld seconds\n", (end - start));
17 }
18 void largest(int num1,int num2,int num3)
19 {
20     int largest;
21     if (num1 >= num2) {
22         largest = num1>num3?num1:num3;
23         printf("%d is the largest number\n", largest);
24     }
25     else {
26         largest = num2>num3?num2:num3;
27         printf("%d is the largest number\n", largest);
28     }
29 }
30 void smallest(int num1,int num2,int num3)
31 {
32     int smallest;
33     if (num1 <= num2) {
34         smallest = num1<num3?num1:num3;
35         printf("%d is the smallest number\n", smallest);
36     }
37     else if(num2<=num1) {
38         smallest = num2<num3?num2:num3;
39         printf("%d is the smallest number\n", smallest);
40     }
41 }
42
43
44
45
```

Compilation

Start time is : 1666963124  
11 is the smallest number  
End time is : 1666963124  
The elapsed time is 0 seconds

```
33         smallest=b;
34     else
35         smallest=a;
36 }
37 printf("%d is Largest and %d is Smallest\t\t\tThread: %d\n",largest,smallest,thread_id);
38
39
40 }
41 ftime = omp_get_wtime();
42 exec_time = ftime - itime;
43 printf("\n\nTime taken for parallel approach is %f\n\n", exec_time);
44
45 printf("\n\n-----serial approach-----\n\n");
46 itime = omp_get_wtime();
47 if(a>b && a>c)
48 {
49     largest=a;
50     if(b>c)
51         smallest=c;
52     else
53         smallest=b;
54 }
55 else if(b>c)
56 {
57     largest=b;
58     if(a>c)
59         smallest=c;
60     else
61         smallest=a;
62 }
63 else
64 {
65     largest=c;
66     if(a>b)
67         smallest=b;
68     else
69         smallest=a;
70 }
71 printf("\n%d is Largest and %d is Smallest\t\t\tThread: %d\n",largest,smallest,omp_get_thread_num());
72 ftime = omp_get_wtime();
73 exec_time = ftime - itime;
74 printf("\n\nTime taken for serial approach is %f", exec_time);
75 return 0;
76
```

Compilation

30 is Largest and 8 is Smallest || Thread: 11  
30 is Largest and 8 is Smallest || Thread: 11  
30 is Largest and 8 is Smallest || Thread: 27  
30 is Largest and 8 is Smallest || Thread: 3  
30 is Largest and 8 is Smallest || Thread: 4  
30 is Largest and 8 is Smallest || Thread: 15  
30 is Largest and 8 is Smallest || Thread: 30  
30 is Largest and 8 is Smallest || Thread: 10  
30 is Largest and 8 is Smallest || Thread: 1  
30 is Largest and 8 is Smallest || Thread: 29  
30 is Largest and 8 is Smallest || Thread: 6  
30 is Largest and 8 is Smallest || Thread: 9  
30 is Largest and 8 is Smallest || Thread: 26  
30 is Largest and 8 is Smallest || Thread: 23  
30 is Largest and 8 is Smallest || Thread: 18  
30 is Largest and 8 is Smallest || Thread: 19  
30 is Largest and 8 is Smallest || Thread: 24  
30 is Largest and 8 is Smallest || Thread: 12  
30 is Largest and 8 is Smallest || Thread: 7  
30 is Largest and 8 is Smallest || Thread: 13  
30 is Largest and 8 is Smallest || Thread: 16  
30 is Largest and 8 is Smallest || Thread: 21  
30 is Largest and 8 is Smallest || Thread: 18  
30 is Largest and 8 is Smallest || Thread: 13  
30 is Largest and 8 is Smallest || Thread: 2  
30 is Largest and 8 is Smallest || Thread: 28  
30 is Largest and 8 is Smallest || Thread: 25  
30 is Largest and 8 is Smallest || Thread: 13  
30 is Largest and 8 is Smallest || Thread: 31

Time taken is 0.011755  
-----serial approach-----  
30 is Largest and 8 is Smallest || Thread: 0  
Time taken is 0.000009

```
#include <stdio.h>
#include <omp.h>
int main()
{
    int thread_id,a=15,b=8,c=30,largest,smallest;
    double itime, ftime, exec_time;
    printf("\n-----parallel approach-----\n");
    itime = omp_get_wtime();
    #pragma omp parallel
    {
        thread_id = omp_get_thread_num();
        if(a>b && a>c)
        {
            largest=a;
            if(b>c)
                smallest=c;
            else
                smallest=b;
        }
        else if(b>c)
        {
            largest=b;
            if(a>c)
                smallest=c;
            else
                smallest=a;
        }
        else
        {
            largest=c;
            if(a>b)
                smallest=b;
            else
                smallest=a;
        }
        printf("%d is Largest and %d is Smallest\t| |\tThread:
%d\n",largest,smallest,thread_id);
    }
    ftime = omp_get_wtime();
    exec_time = ftime - itime;
    printf("\n\nTime taken for parallel approach is %f\n\n", exec_time);

    printf("\n-----serial approach-----\n");
```

```
    itime = omp_get_wtime();
    if(a>b && a>c)
    {
        largest=a;
        if(b>c)
            smallest=c;
        else
            smallest=b;
    }
    else if(b>c) {
        largest=b;
        if(a>c)
            smallest=c;
        else
            smallest=a;    }
    else {
        largest=c;
        if(a>b)
            smallest=b;
        else
            smallest=a;
    }
    printf("\n%d is Largest and %d is Smallest\t| |\tThread:
%d\n",largest,smallest,omp_get_thread_num());
    ftime = omp_get_wtime();
    exec_time = ftime - itime;
    printf("\n\nTime taken for serial approach is %f", exec_time);
    return 0;
}
```

### OUTPUT:-

```
-----parallel approach-----
30 is Largest and 8 is Smallest ||      Thread: 0
30 is Largest and 8 is Smallest ||      Thread: 8
30 is Largest and 8 is Smallest ||      Thread: 11
```

```
30 is Largest and 8 is Smallest || Thread: 11
30 is Largest and 8 is Smallest || Thread: 27
30 is Largest and 8 is Smallest || Thread: 3
30 is Largest and 8 is Smallest || Thread: 4
30 is Largest and 8 is Smallest || Thread: 15
30 is Largest and 8 is Smallest || Thread: 30
30 is Largest and 8 is Smallest || Thread: 10
30 is Largest and 8 is Smallest || Thread: 1
30 is Largest and 8 is Smallest || Thread: 29
30 is Largest and 8 is Smallest || Thread: 6
30 is Largest and 8 is Smallest || Thread: 14
30 is Largest and 8 is Smallest || Thread: 9
30 is Largest and 8 is Smallest || Thread: 26
30 is Largest and 8 is Smallest || Thread: 23
30 is Largest and 8 is Smallest || Thread: 18
30 is Largest and 8 is Smallest || Thread: 19
30 is Largest and 8 is Smallest || Thread: 24
30 is Largest and 8 is Smallest || Thread: 12
30 is Largest and 8 is Smallest || Thread: 7
30 is Largest and 8 is Smallest || Thread: 13
30 is Largest and 8 is Smallest || Thread: 16
30 is Largest and 8 is Smallest || Thread: 21
30 is Largest and 8 is Smallest || Thread: 18
30 is Largest and 8 is Smallest || Thread: 18
30 is Largest and 8 is Smallest || Thread: 2
30 is Largest and 8 is Smallest || Thread: 28
30 is Largest and 8 is Smallest || Thread: 25
30 is Largest and 8 is Smallest || Thread: 13
30 is Largest and 8 is Smallest || Thread: 31
Time taken is 0.011755
-----serial approach-----
30 is Largest and 8 is Smallest || Thread: 0

Time taken is 0.000009
```

(5) Write a C-program using Open MP to demonstrate the shared, private, first private, last private and thread private concepts.

The image shows a web-based IDE interface with a C code editor on the left and a compilation output window on the right. The code is a C program using OpenMP for parallelization. It defines several variables: `nPrivate`, `nFirstPrivate`, `nLastPrivate`, and `nShared`. The program uses `#pragma omp parallel` to create a parallel region and `#pragma omp for` to parallelize a loop. The output window shows the compilation warnings and the execution output, which includes the values of the variables at different points in the program.

```
1 #include <assert.h>
2 #include <stdio.h>
3 #include <omp.h>
4 #define NUM_THREADS 4
5 #define SLEEP_THREAD 1
6 #define NUM_LOOPS 2
7 #define MAX_TYPES {
8     ThreadPrivate,
9     Private,
10    FirstPrivate,
11    LastPrivate,
12    Shared,
13    MAX_TYPES
14 }
15 int nSave[NUM_THREADS][MAX_TYPES][NUM_LOOPS] = {{0}};
16 int nThreadPrivate;
17 #pragma omp threadprivate(nThreadPrivate)
18 #pragma warning(disable:4700)
19 int main() {
20     int nPrivate = NUM_THREADS;
21     int nFirstPrivate = NUM_THREADS;
22     int nLastPrivate = NUM_THREADS;
23     int nShared = NUM_THREADS;
24     int nKnt = 0;
25     int i;
26     int j;
27     int nLoop = 0;
28     nThreadPrivate = NUM_THREADS;
29     printf("These are the variables before entry "
30           "into the parallel region.\n");
31     printf("nThreadPrivate = %d\n", nThreadPrivate);
32     printf("nPrivate = %d\n", nPrivate);
33     printf("nFirstPrivate = %d\n", nFirstPrivate);
34     printf("nLastPrivate = %d\n", nLastPrivate);
35     printf("nShared = %d\n", nShared);
36     omp_set_num_threads(NUM_THREADS);
37
38     #pragma omp parallel copyin(nThreadPrivate) private(nPrivate) shared(nShared) firstprivate(nFirstPrivate)
39     {
40         #pragma omp for schedule(static) lastprivate(nLastPrivate)
41         for (i = 0; i < NUM_THREADS; ++i) {
42             for (j = 0; j < NUM_LOOPS; ++j) {
43                 int nThread = omp_get_thread_num();
44                 assert(nThread < NUM_THREADS);
45                 if (nThread == SLEEP_THREAD)
46                     //Sleep(100);
47                 nSave[nThread][ThreadPrivate][j] = nThreadPrivate;
48                 nSave[nThread][Private][j] = nPrivate;
49                 nSave[nThread][Shared][j] = nShared;
50                 nSave[nThread][FirstPrivate][j] = nFirstPrivate;
51                 nSave[nThread][LastPrivate][j] = nLastPrivate;
52                 nThreadPrivate = nThread;
53                 nPrivate = nThread;
54                 nShared = nThread;
55                 nLastPrivate = nThread;
56                 --nFirstPrivate;
57             }
58         }
59
60         for (i = 0; i < NUM_LOOPS; ++i) {
61             for (j = 0; j < NUM_THREADS; ++j) {
62                 printf("These are the variables at entry of "
63                       "loop %d of thread %d.\n", i + 1, j);
64                 printf("nThreadPrivate = %d\n",
65                       nSave[j][ThreadPrivate][i]);
66             }
67         }
68     }
```

Compilation output:

```
f1.c: In function 'main':
f1.c:87:30: warning: format '%d' expects argument of type 'int', but ar
printf("    nShared = %d (The value assigned ", from the delayed
    ^
    %s
f1.c:87:11: warning: too many arguments for format [-Wformat-extra-
printf("    nShared = %d (The value assigned ", from the delayed

These are the variables before entry into the parallel region.
nThreadPrivate = 4
nPrivate = 4
nFirstPrivate = 4
nLastPrivate = 4
nShared = 4

These are the variables at entry of loop 1 of thread 0.
nThreadPrivate = 0
nPrivate = 0
nFirstPrivate = 4
nLastPrivate = 0
nShared = 4

These are the variables at entry of loop 1 of thread 1.
nThreadPrivate = 4
nPrivate = 0
nFirstPrivate = 4
nLastPrivate = 0
nShared = 4

These are the variables at entry of loop 1 of thread 2.
nThreadPrivate = 0
nPrivate = 0
nFirstPrivate = 4
nLastPrivate = 0
nShared = 4

These are the variables at entry of loop 1 of thread 3.
nThreadPrivate = 0
nPrivate = 0
nFirstPrivate = 4
nLastPrivate = 0
nShared = 2

These are the variables at entry of loop 2 of thread 0.
nThreadPrivate = 0
nPrivate = 0
nFirstPrivate = 3
nLastPrivate = 0
nShared = 0

These are the variables at entry of loop 2 of thread 1.
nThreadPrivate = 1
nPrivate = 1
nFirstPrivate = 3
```



```
f1.c
1
2 #include <stdio.h>
3 #include <omp.h>
4 int sum(int a,int b)
5 {
6     return a+b;
7 }
8 int sub(int a,int b)
9 {
10    return a-b;
11 }
12 int mul(int a,int b)
13 {
14    return a*b;
15 }
16 float div(int a,int b)
17 {
18    return (float)a/b;
19 }
20
21 int main()
22 {
23     int thread_id,i=10,a=20,b=5;
24     printf("\n-----shared-----\n");
25
26     #pragma omp parallel shared(i)
27     {
28         thread_id = omp_get_thread_num();
29         printf("%d + %d = %d\t||\tThread: %d\n",a,b,sum(a,b),thread_id);
30     }
31     printf("\n-----private-----\n");
32
33     #pragma omp parallel private(i)
34     {
35         thread_id = omp_get_thread_num();
36         printf("%d - %d = %d\t||\tThread: %d\n",a,b,sub(a,b),thread_id);
37     }
38     printf("\n-----firstprivate-----\n");
39
40     #pragma omp parallel firstprivate(i)
41     {
42         thread_id = omp_get_thread_num();
43     }
44 }
```

Compilation

```
-----shared-----
20 + 5 = 25 || Thread: 1
20 + 5 = 25 || Thread: 25
20 + 5 = 25 || Thread: 22
20 + 5 = 25 || Thread: 5
20 + 5 = 25 || Thread: 18
20 + 5 = 25 || Thread: 9
20 + 5 = 25 || Thread: 28
20 + 5 = 25 || Thread: 29
20 + 5 = 25 || Thread: 26
20 + 5 = 25 || Thread: 8
20 + 5 = 25 || Thread: 27
20 + 5 = 25 || Thread: 31
20 + 5 = 25 || Thread: 20
20 + 5 = 25 || Thread: 17
20 + 5 = 25 || Thread: 30
20 + 5 = 25 || Thread: 13
20 + 5 = 25 || Thread: 16
20 + 5 = 25 || Thread: 11
20 + 5 = 25 || Thread: 24
20 + 5 = 25 || Thread: 2
20 + 5 = 25 || Thread: 4
20 + 5 = 25 || Thread: 0
20 + 5 = 25 || Thread: 19
20 + 5 = 25 || Thread: 23
20 + 5 = 25 || Thread: 15
20 + 5 = 25 || Thread: 15
20 + 5 = 25 || Thread: 12
20 + 5 = 25 || Thread: 3
20 + 5 = 25 || Thread: 14
20 + 5 = 25 || Thread: 18
20 + 5 = 25 || Thread: 10
20 + 5 = 25 || Thread: 21

-----private-----
20 - 5 = 15 || Thread: 14
20 - 5 = 15 || Thread: 12
20 - 5 = 15 || Thread: 3
```

```
f1.c
15 }
16 float div(int a,int b)
17 {
18    return (float)a/b;
19 }
20
21 int main()
22 {
23     int thread_id,i=10,a=20,b=5;
24     printf("\n-----shared-----\n");
25
26     #pragma omp parallel shared(i)
27     {
28         thread_id = omp_get_thread_num();
29         printf("%d + %d = %d\t||\tThread: %d\n",a,b,sum(a,b),thread_id);
30     }
31     printf("\n-----private-----\n");
32
33     #pragma omp parallel private(i)
34     {
35         thread_id = omp_get_thread_num();
36         printf("%d - %d = %d\t||\tThread: %d\n",a,b,sub(a,b),thread_id);
37     }
38     printf("\n-----firstprivate-----\n");
39
40     #pragma omp parallel firstprivate(i)
41     {
42         thread_id = omp_get_thread_num();
43         printf("%d * %d = %d\t||\tThread: %d\n",a,b,mul(a,b),thread_id);
44     }
45     printf("\n-----lastprivate-----\n");
46
47     #pragma omp lastprivate(i)
48     {
49         thread_id = omp_get_thread_num();
50         printf("%d / %d = %f\t||\tThread: %d\n",a,b,div(a,b),thread_id);
51     }
52     return 0;
53 }
54
55
56
57
58 }
```

Compilation

```
20 - 5 = 15 || Thread: 16

-----firstprivate-----
20 * 5 = 100 || Thread: 25
20 * 5 = 100 || Thread: 1
20 * 5 = 100 || Thread: 12
20 * 5 = 100 || Thread: 7
20 * 5 = 100 || Thread: 13
20 * 5 = 100 || Thread: 30
20 * 5 = 100 || Thread: 4
20 * 5 = 100 || Thread: 3
20 * 5 = 100 || Thread: 6
20 * 5 = 100 || Thread: 24
20 * 5 = 100 || Thread: 5
20 * 5 = 100 || Thread: 17
20 * 5 = 100 || Thread: 16
20 * 5 = 100 || Thread: 11
20 * 5 = 100 || Thread: 31
20 * 5 = 100 || Thread: 26
20 * 5 = 100 || Thread: 22
20 * 5 = 100 || Thread: 23
20 * 5 = 100 || Thread: 2
20 * 5 = 100 || Thread: 12
20 * 5 = 100 || Thread: 10
20 * 5 = 100 || Thread: 6
20 * 5 = 100 || Thread: 0
20 * 5 = 100 || Thread: 20
20 * 5 = 100 || Thread: 20
20 * 5 = 100 || Thread: 21
20 * 5 = 100 || Thread: 18
20 * 5 = 100 || Thread: 27
20 * 5 = 100 || Thread: 14
20 * 5 = 100 || Thread: 19
20 * 5 = 100 || Thread: 9
20 * 5 = 100 || Thread: 28

-----lastprivate-----
20 / 5 = 4.000000 || Thread: 0
```

```
#include <stdio.h>
#include <omp.h>
int sum(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;
}
float div(int a,int b)
{
    return (float)a/b;
}
int main()
{
    int thread_id,i=10,a=20,b=5;
    printf("\n-----shared-----\n");

    #pragma omp parallel shared(i)
    {
        thread_id = omp_get_thread_num();
        printf("%d + %d = %d\t | |\tThread: %d\n",a,b,sum(a,b),thread_id);
    }
    printf("\n-----private-----\n");

    #pragma omp parallel private(i)
    {
        thread_id = omp_get_thread_num();
        printf("%d - %d = %d\t | |\tThread: %d\n",a,b,sub(a,b),thread_id);
    }
    printf("\n-----firstprivate-----\n");

    #pragma omp parallel firstprivate(i)
    {
        thread_id = omp_get_thread_num();
```

```
    printf("%d * %d = %d\t | | \tThread: %d\n",a,b,mul(a,b),thread_id);  
  
}  
printf("\n-----lastprivate-----\n");  
  
#pragma omp lastprivate(i)  
{  
    thread_id = omp_get_thread_num();  
    printf("%d / %d = %f\t | | \tThread: %d\n",a,b,div(a,b),thread_id);  
  
}  
return 0;  
}
```

### OUTPUT :-

```
-----shared-----  
20 + 5 = 25 || Thread: 1  
20 + 5 = 25 || Thread: 25  
20 + 5 = 25 || Thread: 22  
20 + 5 = 25 || Thread: 5  
20 + 5 = 25 || Thread: 18  
20 + 5 = 25 || Thread: 9  
20 + 5 = 25 || Thread: 28  
20 + 5 = 25 || Thread: 29  
20 + 5 = 25 || Thread: 26  
20 + 5 = 25 || Thread: 8  
20 + 5 = 25 || Thread: 27  
20 + 5 = 25 || Thread: 31  
20 + 5 = 25 || Thread: 20  
20 + 5 = 25 || Thread: 17  
20 + 5 = 25 || Thread: 30  
20 + 5 = 25 || Thread: 13  
20 + 5 = 25 || Thread: 16  
20 + 5 = 25 || Thread: 11  
20 + 5 = 25 || Thread: 24  
20 + 5 = 25 || Thread: 2  
20 + 5 = 25 || Thread: 4  
20 + 5 = 25 || Thread: 0  
20 + 5 = 25 || Thread: 19  
20 + 5 = 25 || Thread: 23  
20 + 5 = 25 || Thread: 15  
20 + 5 = 25 || Thread: 15  
20 + 5 = 25 || Thread: 12  
20 + 5 = 25 || Thread: 3  
20 + 5 = 25 || Thread: 14  
20 + 5 = 25 || Thread: 18  
20 + 5 = 25 || Thread: 10  
20 + 5 = 25 || Thread: 21
```

-----private-----		
20 - 5 = 15		Thread: 14
20 - 5 = 15		Thread: 12
20 - 5 = 15		Thread: 3
20 - 5 = 15		Thread: 26
20 - 5 = 15		Thread: 25
20 - 5 = 15		Thread: 0
20 - 5 = 15		Thread: 23
20 - 5 = 15		Thread: 15
20 - 5 = 15		Thread: 24
20 - 5 = 15		Thread: 9
20 - 5 = 15		Thread: 10
20 - 5 = 15		Thread: 4
20 - 5 = 15		Thread: 21
20 - 5 = 15		Thread: 19
20 - 5 = 15		Thread: 11
20 - 5 = 15		Thread: 27
20 - 5 = 15		Thread: 2
20 - 5 = 15		Thread: 31
20 - 5 = 15		Thread: 5
20 - 5 = 15		Thread: 17
20 - 5 = 15		Thread: 7
20 - 5 = 15		Thread: 7
20 - 5 = 15		Thread: 8
20 - 5 = 15		Thread: 29
20 - 5 = 15		Thread: 30
20 - 5 = 15		Thread: 22
20 - 5 = 15		Thread: 1
20 - 5 = 15		Thread: 28
20 - 5 = 15		Thread: 20
20 - 5 = 15		Thread: 6
20 - 5 = 15		Thread: 16
20 - 5 = 15		Thread: 16
-----firstprivate-----		
20 * 5 = 100		Thread: 25
20 * 5 = 100		Thread: 1
20 * 5 = 100		Thread: 12
20 * 5 = 100		Thread: 7
20 * 5 = 100		Thread: 13
20 * 5 = 100		Thread: 30
20 * 5 = 100		Thread: 4
20 * 5 = 100		Thread: 3
20 * 5 = 100		Thread: 6
20 * 5 = 100		Thread: 24
20 * 5 = 100		Thread: 5
20 * 5 = 100		Thread: 17
20 * 5 = 100		Thread: 16
20 * 5 = 100		Thread: 11
20 * 5 = 100		Thread: 31
20 * 5 = 100		Thread: 26
20 * 5 = 100		Thread: 22
20 * 5 = 100		Thread: 23
20 * 5 = 100		Thread: 2
20 * 5 = 100		Thread: 12
20 * 5 = 100		Thread: 10
20 * 5 = 100		Thread: 6
20 * 5 = 100		Thread: 0

```
20 * 5 = 100  ||      Thread: 20
20 * 5 = 100  ||      Thread: 20
20 * 5 = 100  ||      Thread: 21
20 * 5 = 100  ||      Thread: 18
20 * 5 = 100  ||      Thread: 27
20 * 5 = 100  ||      Thread: 14
20 * 5 = 100  ||      Thread: 19
20 * 5 = 100  ||      Thread: 9
20 * 5 = 100  ||      Thread: 28

-----lastprivate-----
20 / 5 = 4.000000  ||      Thread: 0
```