(Open Multi-Processing) library, is used for parallel programming in C, C++, and Fortran.
Used:-

Parallelization
Ease of Use
Portability
Scalability:- OpenMP programs can scale from shared-memory systems with a few cores to high-performance computing (HPC) clusters with thousands of cores. The runtime system automatically adjusts the number of threads based on the available resources, making it suitable for a wide range of parallel computing tasks.

To check cores(processor):-

Lscpu

C++

g++ -fopenmp filename.c
./a.out

Or
g++ -fopenmp -o filename filename.c
./filename

To configure Openmp

```
echo |cpp -fopenmp -dM |grep -i open
```

```
#include<stdio.h>
#include<omp.h>//openmp library

int main()
{
	#pragma omp parallel//parallel to different processor
	printf("Welcome to CDAC Bangalore \n");
}
```

If we want to know the which processor it is running then
omp_get_thread_num()

If we want to specific the number of thread
num_thread(6)//any number we can give it does not depend on number of cores

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

int main()
{
	int cores,nthreads,tid,maxt,inpar;
	#pragma omp parallel
	{
		tid = omp_get_thread_num();
		if(tid == 3)
		{
			printf("Thread %d getting
Info..\n",tid);
			cores = omp_get_num_procs();//returns the
number of processor cores available.
			nthreads = omp_get_num_threads();//returns
the number of threads currently in the parallel region.
			maxt = omp_get_max_threads();//returns the
maximum number of threads that can be used in a parallel region.
			inpar = omp_in_parallel();//returns a
nonzero value if the calling thread is executing in parallel, otherwise, it returns 0.

			printf("Number of cores = %d\n", cores);
			printf("Number of threads = %d\n",
nthreads);
			printf("Max threads = %d\n", maxt);
			printf("In Parallel? = %d\n", inpar);
		}
	}
}
```

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

int main()
{
```

```
        int tid,nthreads;
        #pragma omp parallel private(tid,nthreads)
num_threads(4)
        {
                tid = omp_get_thread_num();
                printf("Hello World in to %d\n" ,tid);

                if (tid == 1)
                {
                        nthreads = omp_get_num_threads();
                        printf("Number of threads %d
\n",nthreads);
                }
        }
}
```

The `private(tid, nthreads)` clause ensures that each thread has its own private copy of the variables `tid` and `nthreads`, so modifications made to these variables within the parallel region do not affect their values outside the region

If we not private

all threads will access and modify the same memory locations for these variables, which can lead to race conditions and incorrect behavior.

```
//#include<stdlib.h>
#include<omp.h>
#define CHUNKSIZE 7
#define N 30
int main()
{
    int nthreads,tid,i,chunk;
    float a[N],b[N],c[N];
    for(i=0;i<N;i++)
        a[i] = b[i] = i *2.0;
    chunk = CHUNKSIZE;
    #pragma omp parallel shared(a,b,c,nthreads,chunk) private(i,tid) num_threads(4)
    {
        tid = omp_get_thread_num();
```

```
            printf("hello World from thread = %d\n", tid);
            if(tid == 0)
            {
                  nthreads = omp_get_num_threads();
                  printf("Number of threads = %d\n", nthreads);

            }
            #pragma omp for schedule(static,chunk)

            for(i=0;i<N;i++)
            {
                  c[i] = a[i] + b[i];
                  printf("Thread %d: c[%d] = %f\n",tid,i,c[i]);
            }

      }
}
```

Difference between Shared & Private Variable

- Shared Variables: Shared variables are visible and accessible to all threads within the parallel region. Changes made by one thread affect the values seen by other threads.
- Private Variables: Private variables have separate instances for each thread. Changes made by one thread do not affect the values of private variables in other threads.

shared

- Usage: `shared(variable1, variable2, ...)`
- Purpose: Specifies that the listed variables are shared among all threads in the parallel region.
- Behavior: All threads have read and write access to the same memory location for shared variables. Any modification to a shared variable by one thread is visible to all other threads.
- Typical Use Cases:
    - Variables that are read and modified by multiple threads within the parallel region.
    - Data that needs to be accessed or modified by all threads in the parallel region.

private

- Usage: `private(variable1, variable2, ...)`
- Purpose: Specifies that each thread should have its own private copy of the listed variables.
- Behavior: Each thread gets its own separate memory location for private variables. Modifications to private variables by one thread do not affect the values of those variables in other threads.
- Typical Use Cases:
    - Loop counters or temporary variables used within parallel loops.
    - Variables that store intermediate results or temporary calculations.
    - Thread-specific data or control variables.

```
#include<stdio.h>
#include<stdlib.h>
#include<omp.h>
#define N 30
int main(int argc, char *argv[])
{
	int nthreads,tid,i,chunk;
	float a[N],b[N],c[N],d[N];
	for(i=0;i<N;i++)
	{
		a[i] = i * 1.5;
		b[i] = i + 22.35;
		c[i] = d[i] = 0.0;
	}
	#pragma omp parallel shared(a,b,c,nthreads) private(i,tid)
	{
		tid = omp_get_thread_num();

		if(tid == 0)
		{
			nthreads = omp_get_num_threads();
			printf("Number of threads = %d\n", nthreads);

		}
//a sections construct where each section is executed independently and
asynchronously.
		#pragma omp sections nowait
		{
```

```
            #pragma omp section
            {
                printf("Thread %d doing section 1\n",tid);
                for(i=0;i<N;i++)
                {
                    c[i] = a[i] +b[i];
                    printf("Thread %d: c[%d] = %f\n",tid,i,c[i]);
                }
            }

            #pragma omp section
            {
                printf("Thread %d doing section 2\n",tid);
                for(i=0;i<N;i++)
                {
                    c[i] = a[i] * b[i];
                    printf("Thread %d: c[%d] = %f\n",tid,i,c[i]);
                }
            }
        }
```

```
#include<stdio.h>
#include<stdlib.h>
#include<omp.h>
#define N 30
int main(int argc, char *argv[])
{
    int nthreads,tid,i,chunk;
    float a[N],b[N],c[N],d[N];
    for(i=0;i<N;i++)
    {
        a[i] = i * 1.5;
        b[i] = i + 22.35;
        c[i] = d[i] = 0.0;
    }
    #pragma omp parallel shared(a,b,c,nthreads) private(i,tid)
    {
        tid = omp_get_thread_num();

        if(tid == 0)
        {
```

```
                nthreads = omp_get_num_threads();
                printf("Number of threads = %d\n", nthreads);

          }
/*#pragma omp single directive ensures that the following block of code is executed
by only one thread (not necessarily the master thread).*/
          #pragma omp single
                {
                      printf("Thread %d doing section 1\n",tid);
                      for(i=0;i<N;i++)
                      {
                            c[i] = a[i] + b[i];
                            printf("Thread %d: c[%d] = %f\n",tid,i,c[i]);
                      }
                }
     }
}
```

```
#include <stdio.h>
#include <omp.h>
#include <stdlib.h>
#include <sys/time.h>
#define MAXIMUM 655368

/* Main Program */
main(int argc,char **argv)
{
   int *array, i, Noofelements, cur_max, current_value, Noofthreads;
   struct timeval  TimeValue_Start;
   struct timezone TimeZone_Start;
   struct timeval  TimeValue_Final;
   struct timezone TimeZone_Final;
   long   time_start, time_end;
   double time_overhead_serial, time_overhead_parallel;

   printf("\n\t\t----------------------------------------------------------------");
   printf("\n\t\t Centre for Development of Advanced Computing (C-DAC)");
   printf("\n\t\t----------------------------------------------------------------");
   printf("\n\t\t Objective : Finding Maximum element of an Array using  ");
   printf("\n\t\t OpenMP Parallel for directive and Critical Section  ");
```

```
printf("\n\t\t.....\n");

/* Checking for command line arguments */
if( argc != 3 )
   {
     printf("\t\t Very Few Arguments\n ");
     printf("\t\t Syntax : exec <Threads> <No. of elements> \n");
     exit(-1);
   }

   Noofthreads=atoi(argv[1]);
   Noofelements=atoi(argv[2]);

if (Noofelements <= 0) {
   printf("\n\t\t The array elements cannot be stored\n");
   exit(1);
}
   printf("\n\t\t Threads                    : %d ",Noofthreads);
   printf("\n\t\t Number of elements in Array : %d \n ",Noofelements);
/* Dynamic Memory Allocation */
array = (int *) malloc(sizeof(int) * Noofelements);



/* Allocating Random Number Values To The Elements Of An Array */
srand(MAXIMUM);
for (i = 0; i < Noofelements; i++)
   array[i] = rand();

if (Noofelements == 1) {
   printf("\n\t\t The Largest Number In The Array is %d", array[0]);
   exit(1);
}

cur_max = 0;
   gettimeofday(&TimeValue_Start, &TimeZone_Start);

   /* Set the No. of threads */
omp_set_num_threads(Noofthreads);

/* OpenMP Parallel For Directive And Critical Section : Fork a team of threads */
#pragma omp parallel for
for (i = 0; i < Noofelements; i = i + 1) {
   if (array[i] > cur_max)
```

```
   #pragma omp critical
          if (array[i] > cur_max)
             cur_max = array[i];
   } /* End of the parallel section */


    gettimeofday(&TimeValue_Final, &TimeZone_Final);



      /* calculate the timing for the computation */

      time_start = TimeValue_Start.tv_sec * 1000000 + TimeValue_Start.tv_usec;
      time_end = TimeValue_Final.tv_sec * 1000000 + TimeValue_Final.tv_usec;
      time_overhead_parallel = (time_end - time_start)/1000000.0;



   /* Serial Calculation */

    gettimeofday(&TimeValue_Final, &TimeZone_Final);

   current_value = array[0];
   for (i = 1; i < Noofelements; i++)
      if (array[i] > current_value)
         current_value = array[i];

    gettimeofday(&TimeValue_Final, &TimeZone_Final);

      time_start = TimeValue_Start.tv_sec * 1000000 + TimeValue_Start.tv_usec;
      time_end = TimeValue_Final.tv_sec * 1000000 + TimeValue_Final.tv_usec;
      time_overhead_serial = (time_end - time_start)/1000000.0;


 /* Checking For Output Validity */

   if (current_value == cur_max)
      printf("\n\t\t The Max Value Is Same From Serial And Parallel OpenMP
Directive\n");
   else {
      printf("\n\t\t The Max Value Is Not Same In Serial And Parallel OpenMP
Directive\n");
      exit(-1);
   }
```

```
   /* Freeing Allocated Memory */

   printf("\n");
   free(array);
   printf("\n\t\t The Largest Number In The Given Array Is %d\n", cur_max);
      printf("\n\t\t Time in Seconds (T) for Serial       : %lf Seconds
\n",time_overhead_serial);
      printf("\n\t\t Time in Seconds (T) for Parallel     : %lf Seconds
\n",time_overhead_parallel);
      printf("\n\t\t.....\n");
}
```