**Assignment no:- 2**

**Title**: - Write a parallel program (using OpenMp) to add two large vectors.

**Objective**: - 1. To understand vector operations.

          2. To implement parallel Algorithm to perform vector operations.

**Outcome**: - Understand vector addition using parallel algorithm.

**Software** **required**: - C++ ,OpenMp.

**Theory**: -

OpenMP on shared memory multicore machines creates *threads* that execute concurrently OpenMP (Open Multi-Processing) provides constructs (API) to support parallel programming in C++, C, and Fortran on Linux, MacOS, and Windows.

A sequential code is transformed to a parallel one by adding pragmas, so if a compiler does not support OpenMP, the pragmas are skipped and the output is a sequential program.

OpenMP 4.0 added constructs for the vectorization, offloading, and extended tasks.

OpenMP is used in software like Blender, fftw, OpenBLAS, and eigen to accelerate computations.

It is relatively easy to use in scientific applications.

**Vector Addition in C using OpenMp: -**

#include <stdlib.h>

#include <stdio.h>

#include <omp.h>      *//OpenMP*   
   
*// Very small values for this simple illustrative example* 

#define ARRAY\_SIZE 8     *//Size of arrays whose elements will be added together.* 

#define NUM\_THREADS 4    *//Number of threads to use for vector addition.*   
   
*/\**   
*\*  Classic vector addition using openMP default data decomposition.*   
*\**   
*\*  Compile using gcc like this:*   
*\*  gcc -o va-omp-simple VA-OMP-simple.c -fopenmp*   
*\**   
*\*  Execute:*   
*\*  ./va-omp-simple*   
*\*/* 

int main (int argc, char \*argv[])    
{ 

*// elements of arrays a and b will be added*   
*// and placed in array c*   
int \* a;   
int \* b;    
int \* c;   
           
        int n = ARRAY\_SIZE;                 *// number of array elements*

int n\_per\_thread;                   *// elements per thread*

int total\_threads = NUM\_THREADS;    *// number of threads to use* 

int i;       *// loop index*   
           
        *// allocate spce for the arrays* 

a = (int \*) malloc(**sizeof**(int)\*n);

b = (int \*) malloc(**sizeof**(int)\*n); 

c = (int \*) malloc(**sizeof**(int)\*n);   
 

*// initialize arrays a and b with consecutive integer values*   
*// as a simple*

**for**(i=0; i<n; i++) {   
            a[i] = i;   
        }   
        **for**(i=0; i<n; i++) {   
            b[i] = i;   
        }      
           
*// Additional work to set the number of threads.*   
*// We hard-code to 4 for illustration purposes only.*

omp\_set\_num\_threads(total\_threads);   
   
*// determine how many elements each process will work on* 

n\_per\_thread = n/total\_threads;

   
        *// Compute the vector addition*   
*// Here is where the 4 threads are specifically 'forked' to*   
*// execute in parallel. This is directed by the pragma and*   
*// thread forking is compiled into the resulting exacutable.*   
*// Here we use a 'static schedule' so each thread works on*   
*// a 2-element chunk of the original 8-element arrays.*

#pragma omp parallel for shared(a, b, c) private(i) schedule(static, n\_per\_thread) 

**for**(i=0; i<n; i++)

{   
c[i] = a[i]+b[i];   
*// Which thread am I? Show who works on what for this samll example*

printf("Thread %d works on element%d**\n**", omp\_get\_thread\_num(), i); 

}   
   
*// Check for correctness (only plausible for small vector size)*   
*// A test we would eventually leave out*   
printf("i**\t**a[i]**\t**+**\t**b[i]**\t**=**\t**c[i]**\n**");

        **for**(i=0; i<n; i++) {

printf("%d**\t**%d**\t\t**%d**\t\t**%d**\n**", i, a[i], b[i], c[i]); 

       }   
   
        *// clean up memory*

        free(a);

free(b);

free(c); 

**return** 0;   
}

Conclusion:-

The vector addtion using openmp is performed successfully.