**Class:** Final Year (Computer Science and Engineering)

**Year:** 2022-23 **Semester:**1

**Course:** High Performance Computing Lab

**Practical No. 2**

**Exam Seat No:**

2019BTECS00005 – Ashish Sutar

**Title of practical:**

Study and implementation of firstprivate and shared variables

**Problem Statement 1:**

Implement a parallel code for vector scalar addition

**Screenshot 1:**

#include <stdlib.h>   //malloc and free

#include <stdio.h>    //printf

#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.

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

{

    // elements of arrays a and b will be added

    // and placed in array c

    int \* a;

    int  b=2;

    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 example

        for(i=0; i<n; i++) {

            a[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;

        // 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);

        }

    printf("\n array 1 a[i]   - ");

        for(i=0; i<n; i++) {

        printf("%d ",a[i]);

        }

        printf("\n  b -  %d ",b);

    printf("\n res array c[i] - ");

        for(i=0; i<n; i++) {

        printf("%d ",c[i]);

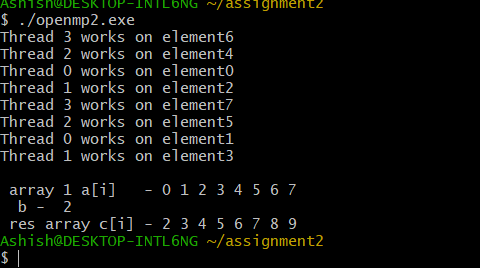
        }

        free(a);   free(c);

    return 0;

}

**Screenshot 2:**

****

**Information 1:**

The clause private (variable list) indicates that the set of variables specified is local to each thread – i.e., each thread has its own copy of each variable in the list. The clause firstprivate (variable list) is similar to the private clause, except the values of variables on entering the threads are initialized to corresponding values before the parallel directive. The clause shared (variable list) indicates that all variables in the list are shared across all the threads, i.e., there is only one copy.

**Problem Statement 2:**

Implement a parallel code for vector vector addition

**Screenshot 3:**

#include <stdlib.h>   //malloc and free

#include <stdio.h>    //printf

#include <omp.h>      //OpenMP

#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.

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

{

    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 example

        for(i=0; i<n; i++) {

            a[i] = i;

        }

        for(i=0; i<n; i++) {

            b[i] = i;

        }

    omp\_set\_num\_threads(total\_threads);

    n\_per\_thread = n/total\_threads;

    #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);

    }

    // printf("\n array 1 a[i]   - ");

    //     for(i=0; i<n; i++) {

    //  printf("%d ",a[i]);

    //     }

    // printf("\n array 2 b[i]   - ");

    //     for(i=0; i<n; i++) {

    //  printf("%d ",b[i]);

    //     }

    printf("\n res array c[i] - ");

        for(i=0; i<n; i++) {

        printf("%d ",c[i]);

        }

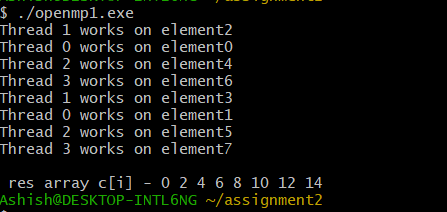
        // clean up memory

        free(a);  free(b); free(c);

    return 0;

}

**Screenshot 4:**

****

**Information 2:**

The schedule clause of the for directive deals with the assignment of iterations to threads. The general form of the schedule directive is schedule(scheduling\_class[, parameter]). OpenMP supports four scheduling classes: static, dynamic, guided, and runtime. The general form of the static scheduling class is schedule(static[, chunk-size]). This technique splits the iteration space into equal chunks of size chunk-size and assigns them to threads in a round-robin fashion. OpenMP has a dynamic scheduling class. The general form of this class is schedule(dynamic[, chunk-size]). The iteration space is partitioned into chunks given by chunk-size. However, these are assigned to threads as they become idle.

**Github Link:**