NETAJI SUBHAS UNIVERSITY OF TECHNOLOGY

HIGH PERFORMANCE COMPUTING

Lab practical file

(CACSC20)

SUBMITTED BY: -

Name: *HARSH KUMAR*

Branch: CSAI

Roll no.:2021UCA1829

S NO.	EXPERIMENT
1	Write a program in C to multiply two matrices of size 10000 x 10000 each and find it's execution-time using ""time"" command.
2	Write a parallel program to print “Hello World"" using MP
3	Write a parallel program to find sum of an array using MPI
4	Write a C program for parallel implementation of Matrix Multiplication using MPI
5	Write a C program to implement the Quick Sort Algorithm using MPI.
6	Write a multithreaded program to generate Fibonacci series using pThreads.
7	Write a program to implement Process Synchronization by mutex locks using pThreads.
8	"Write a ""Hello World"" program using OpenMP library also display number of threads created during execution.
9	Write a C program to demonstrate multitask using OpenMP.
10	Write a parallel program to calculate the value of PI/Area of Circle using OpenMP library.
11	Write a C program to demonstrate default, static and dynamic loop scheduling using OpenMP

MPI

MPI, which stands for Message Passing Interface, is a theory for parallel programming. It focuses on dividing a large computational problem into smaller, independent tasks. These tasks are then distributed among multiple processors or computers in a network, allowing them to work on the problem simultaneously.

The core idea behind MPI is communication. Each processor, called a rank in MPI terminology, has its own local memory and can only directly access its own data. To collaborate, ranks need to exchange data by sending and receiving messages. MPI provides a set of functions that define how these messages are structured, sent, received, and synchronized. This allows ranks to work on their assigned tasks, share necessary data with others, and ultimately combine their results to solve the overall problem.

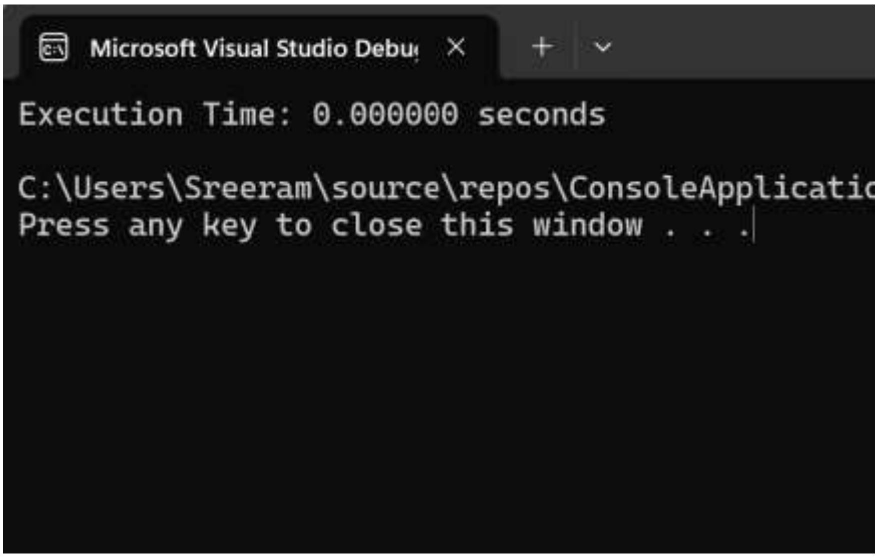
MPI offers several advantages. Firstly, it enables significant speedups by utilizing the combined processing power of multiple machines. Secondly, it promotes modularity by breaking down complex problems into smaller, easier-to-manage tasks. Finally, MPI is portable, meaning programs written with MPI can be run on various computer architectures with minimal changes to the code. This makes it a widely adopted standard for parallel programming in scientific computing and high-performance applications.

PRACTICAL 1

Write a program in C to multiply two matrices of size 10000 x 10000 each and find it's execution-time using "time" command. Try to run this program on two or more machines having different configurations and compare execution-times obtained in each run. Comment on which factors affect the performance of the program

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define SIZE 10

// Function to multiply two matrices
void multiplyMatrices(int firstMatrix[SIZE][SIZE], int secondMatrix[SIZE][SIZE], int
result[SIZE][SIZE]) {
    for (int i = 0; i < SIZE; ++i) {
        for (int j = 0; j < SIZE; ++j) {
            result[i][j] = 0;

            for (int k = 0; k < SIZE; ++k) {
                result[i][j] += firstMatrix[i][k] * secondMatrix[k][j];
            }
        }
    }
}

int main() {
    // Initialize matrices
    int firstMatrix[SIZE][SIZE];
    int secondMatrix[SIZE][SIZE];
    int resultMatrix[SIZE][SIZE];

    // Populate matrices with random values
    for (int i = 0; i < SIZE; ++i) {
        for (int j = 0; j < SIZE; ++j) {
            firstMatrix[i][j] = rand() % 10;
            secondMatrix[i][j] = rand() % 10;
        }
    }

    // Measure execution time
    clock_t start = clock();

    // Multiply matrices
    multiplyMatrices(firstMatrix, secondMatrix, resultMatrix);

    // Calculate execution time
    clock_t end = clock();
    double cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;

    // Print execution time
    printf("Execution Time: %f seconds\n", cpu_time_used);

    return 0;
}
```

Result



PRACTICAL 2

Write a parallel program to print “Hello World" using MPI.

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

int main(int argc, char** argv)
{
    int ierr;

    ierr = MPI_Init(&argc, &argv);
    printf("Hello world\n");

    ierr = MPI_Finalize();
}
```

Result

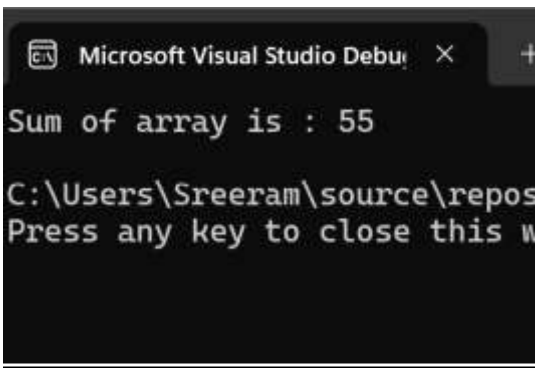
PRACTICAL 3

Write a parallel program to find sum of an array using MPI

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


// size of array
#define n 10

int a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

// Temporary array for slave process
int a2[1000];

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

	int pid, np,
		elements_per_process,
		n_elements_recieved;
	// np -> no. of processes
	// pid -> process id

	MPI_Status status;

	// Creation of parallel processes
	MPI_Init(&argc, &argv);

	// find out process ID,
	// and how many processes were started
	MPI_Comm_rank(MPI_COMM_WORLD, &pid);
	MPI_Comm_size(MPI_COMM_WORLD, &np);

	// master process
	if (pid == 0) {
		int index, i;
		elements_per_process = n / np;

		// check if more than 1 processes are run
		if (np > 1) {
			// distributes the portion of array
			// to child processes to calculate
			// their partial sums
			for (i = 1; i < np - 1; i++) {
				index = i * elements_per_process;

				MPI_Send(&elements_per_process,
					1, MPI_INT, i, 0,
					MPI_COMM_WORLD);
				MPI_Send(&a[index],
					elements_per_process,
					MPI_INT, i, 0,
					MPI_COMM_WORLD);
			}

			// last process adds remaining elements
			index = i * elements_per_process;
			int elements_left = n - index;
```

```
            MPI_Send(&elements_left,
                1, MPI_INT,
                i, 0,
                MPI_COMM_WORLD);
            MPI_Send(&a[index],
                elements_left,
                MPI_INT, i, 0,
                MPI_COMM_WORLD);
        }

        // master process add its own sub array
        int sum = 0;
        for (i = 0; i < elements_per_process; i++)
            sum += a[i];

        // collects partial sums from other processes
        int tmp;
        for (i = 1; i < np; i++) {
            MPI_Recv(&tmp, 1, MPI_INT,
                MPI_ANY_SOURCE, 0,
                MPI_COMM_WORLD,
                &status);
            int sender = status.MPI_SOURCE;

            sum += tmp;
        }

        // prints the final sum of array
        printf("Sum of array is : %d\n", sum);
    }
    // slave processes
    else {
        MPI_Recv(&n_elements_recieved,
            1, MPI_INT, 0, 0,
            MPI_COMM_WORLD,
            &status);

        // stores the received array segment
        // in local array a2
        MPI_Recv(&a2, n_elements_recieved,
            MPI_INT, 0, 0,
            MPI_COMM_WORLD,
            &status);

        // calculates its partial sum
        int partial_sum = 0;
        for (int i = 0; i < n_elements_recieved; i++)
            partial_sum += a2[i];

        // sends the partial sum to the root process
        MPI_Send(&partial_sum, 1, MPI_INT,
            0, 0, MPI_COMM_WORLD);
    }
    // cleans up all MPI state before exit of process
    MPI_Finalize();
    return 0;
}
```

Result



PRACTICAL 4

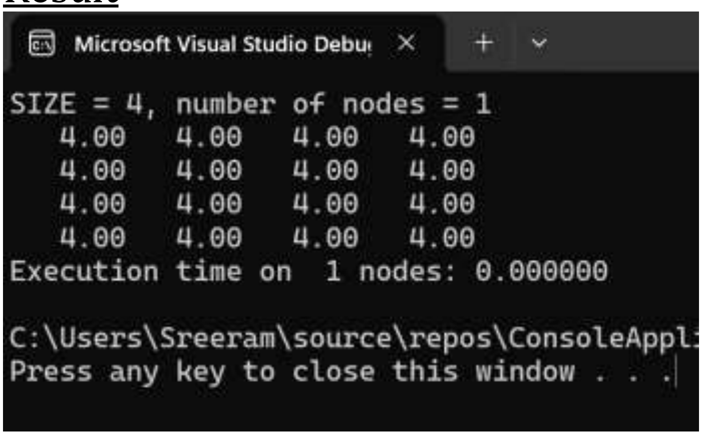
Write a C program for parallel implementation of Matrix Multiplication using MPI.

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

#define SIZE 4
#define FROM_MASTER 1
#define FROM_WORKER 2
#define DEBUG 1

MPI_Status status;

static double a[SIZE][SIZE];
static double b[SIZE][SIZE];
static double c[SIZE][SIZE];

static void init_matrix(void)
{
    int i, j;
    for (i = 0; i < SIZE; i++)
    {
        for (j = 0; j < SIZE; j++) {
            a[i][j] = 1;
            b[i][j] = 1;

        } //end for i
    }  //end for j
} //end init_matrix()

static void print_matrix(void)
{
    int i, j;
    for (i = 0; i < SIZE; i++) {
        for (j = 0; j < SIZE; j++) {
            printf("%7.2f", c[i][j]);
        } //end for i
        printf("\n");
    }    //end for j
}        //end print_matrix

int main(int argc, char** argv)
{
    int myrank, nproc;
    int rows;
    int mtype;
    int dest, src, offseta, offsetb;
    double start_time, end_time;
    int i, j, k, l;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    rows = SIZE / nproc;  //compute the block size
    mtype = FROM_MASTER; //  =1

    if (myrank == 0) {
        /*Initialization*/
        printf("SIZE = %d, number of nodes = %d\n", SIZE, nproc);
```

```
        start_time = MPI_Wtime();

        if (nproc == 1) {
            for (i = 0; i < SIZE; i++) {
                for (j = 0; j < SIZE; j++) {
                    for (k = 0; k < SIZE; k++)
                        c[i][j] = c[i][j] + a[i][k] * b[j][k];
                } //end for i
            }  //end for j
            end_time = MPI_Wtime();
            print_matrix();//---------------------------------
            printf("Execution time on %2d nodes: %f\n", nproc, end_time -
                start_time);
        } // end  if(nproc == 1)

        else {

            for (l = 0; l < nproc; l++) {
                offsetb = rows * l;  //start from (block size * processor id)
                offseta = rows;
                mtype = FROM_MASTER; // tag =1

                for (dest = 1; dest < nproc; dest++) {
                    MPI_Send(&offseta, 1, MPI_INT, dest, mtype,
                        MPI_COMM_WORLD);
                    MPI_Send(&offsetb, 1, MPI_INT, dest, mtype,
                        MPI_COMM_WORLD);
                    MPI_Send(&rows, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
                    MPI_Send(&a[offseta][0], rows * SIZE, MPI_DOUBLE, dest,
                        mtype, MPI_COMM_WORLD);
                    MPI_Send(&b[0][offsetb], rows * SIZE, MPI_DOUBLE, dest,
                        mtype, MPI_COMM_WORLD);

                    offseta += rows;
                    offsetb = (offsetb + rows) % SIZE;

                } // end for dest

                offseta = rows;
                offsetb = rows * l;

                //--mult the final local and print final global mult
                for (i = 0; i < offseta; i++) {
                    for (j = offsetb; j < offsetb + rows; j++) {
                        for (k = 0; k < SIZE; k++) {
                            c[i][j] = c[i][j] + a[i][k] * b[k][j];
                        }//end for k
                    } //end for j
                }// end for i
                   /*- wait for results from all worker tasks */
                mtype = FROM_WORKER;
                for (src = 1; src < nproc; src++) {
                    MPI_Recv(&offseta, 1, MPI_INT, src, mtype, MPI_COMM_WORLD,
                        &status);
                    MPI_Recv(&offsetb, 1, MPI_INT, src, mtype, MPI_COMM_WORLD,
                        &status);
                    MPI_Recv(&rows, 1, MPI_INT, src, mtype, MPI_COMM_WORLD,
```

```
                    MPI_Recv(&rows, 1, MPI_INT, src, mtype, MPI_COMM_WORLD,
                        &status);
                    for (i = 0; i < rows; i++) {
                        MPI_Recv(&c[offseta + i][offsetb], offseta, MPI_DOUBLE,
                            src, mtype, MPI_COMM_WORLD, &status);
                    } //end for scr
                }//end for i
            } //end for l
            end_time = MPI_Wtime();
            print_matrix();
            printf("Execution time on %2d nodes: %f\n", nproc, end_time -
                start_time);
        }//end else
    } //end if (myrank == 0)

    else {
        /*--------------------------- worker----------------------*/
        if (nproc > 1) {
            for (l = 0; l < nproc; l++) {
                mtype = FROM_MASTER;
                MPI_Recv(&offseta, 1, MPI_INT, 0, mtype, MPI_COMM_WORLD,
                    &status);
                MPI_Recv(&offsetb, 1, MPI_INT, 0, mtype, MPI_COMM_WORLD,
                    &status);
                MPI_Recv(&rows, 1, MPI_INT, 0, mtype, MPI_COMM_WORLD,
                    &status);

                MPI_Recv(&a[offseta][0], rows * SIZE, MPI_DOUBLE, 0, mtype,
                    MPI_COMM_WORLD, &status);
                MPI_Recv(&b[0][offsetb], rows * SIZE, MPI_DOUBLE, 0, mtype,
                    MPI_COMM_WORLD, &status);

                for (i = offseta; i < offseta + rows; i++) {
                    for (j = offsetb; j < offsetb + rows; j++) {
                        for (k = 0; k < SIZE; k++) {
                            c[i][j] = c[i][j] + a[i][k] * b[k][j];
                        } //end for j
                    } //end for i

                } //end for l
                mtype = FROM_WORKER;
                MPI_Send(&offseta, 1, MPI_INT, 0, mtype, MPI_COMM_WORLD);
                MPI_Send(&offsetb, 1, MPI_INT, 0, mtype, MPI_COMM_WORLD);
                MPI_Send(&rows, 1, MPI_INT, 0, mtype, MPI_COMM_WORLD);
                for (i = 0; i < rows; i++) {
                    MPI_Send(&c[offseta + i][offsetb], offseta, MPI_DOUBLE, 0,
                        mtype, MPI_COMM_WORLD);

                } //end for i
            }//end for l

        } //end if (nproc > 1)
    } // end else

    MPI_Finalize();
    return 0;
} //end main()
```

Result

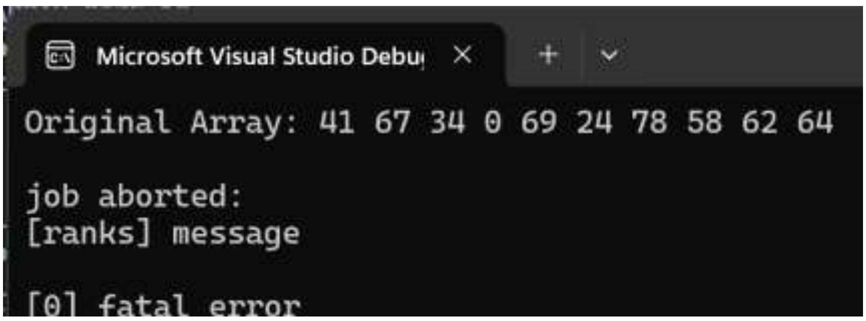


PRACTICAL 5

Write a C program to implement the Quick Sort Algorithm using MPI.

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#define ARRAY_SIZE 10
void swap(int* a, int* b) {
        int temp = *a;
        *a = *b;
        *b = temp;
}
int partition(int array[], int low, int high) {
        int pivot = array[high];
        int i = low - 1;
        for (int j = low; j < high; j++) {
                if (array[j] < pivot) {
                        i++;
                                swap(&array[i], &array[j]);
                }
        }
        swap(&array[i + 1], &array[high]);
        return i + 1;
}
void quickSort(int array[], int low, int high) {
        if (low < high) {
                int pi = partition(array, low, high);
                quickSort(array, low, pi - 1);
                quickSort(array, pi + 1, high);
        }
}
int main(int argc, char* argv[]) {
        int rank, size;
        int array[ARRAY_SIZE];
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
                if (rank == 0) {
                        printf("Original Array: ");
                        for (int i = 0; i < ARRAY_SIZE; i++) {
                                array[i] = rand() % 100;
                                printf("%d ", array[i]);
                        }
                        printf("\n");
                }
        MPI_Bcast(array, ARRAY_SIZE, MPI_INT, 0, MPI_COMM_WORLD);
        int chunk_size = ARRAY_SIZE / size;
        int start = rank * chunk_size;
        int end = (rank == size - 1) ? ARRAY_SIZE - 1 : start +
                chunk_size - 1;
        quickSort(array, start, end);
        MPI_Gather(array + start, chunk_size, MPI_INT, array,
                chunk_size, MPI_INT, 0, MPI_COMM_WORLD);
        if (rank == 0) {
                quickSort(array, 0, ARRAY_SIZE - 1);

                        printf("Sorted Array: ");
                for (int i = 0; i < ARRAY_SIZE; i++) {
                        printf("%d ", array[i]);
                }
                printf("\n");
        }
        MPI_Finalize();
        return 0;
}
```

Result



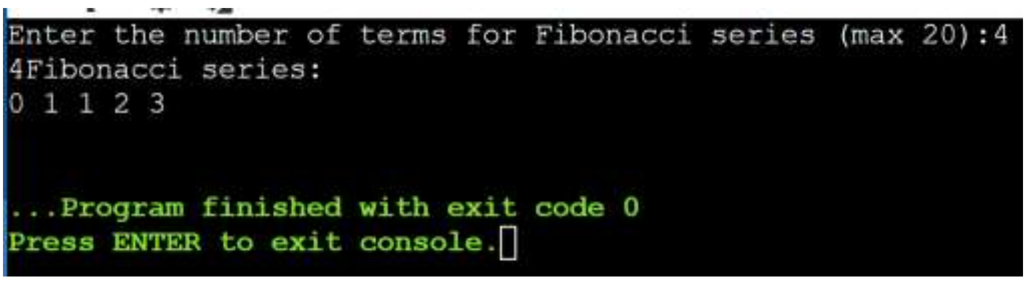
PRACTICAL 6

Write a multithreaded program to generate Fibonacci series using pThreads.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define MAX_TERM 20
// Function to calculate Fibonacci series
void* fibonacci(void* arg) {
	int* arr = (int*)arg;
	int n = arr[0];
	int* result = (int*)malloc((n + 1) * sizeof(int));
	result[0] = 0;
	result[1] = 1;
	for (int i = 2; i <= n; i++) {
		result[i] = result[i - 1] + result[i - 2];
	}
	pthread_exit(result);
}
int main() {
	int n;
	printf("Enter the number of terms for Fibonacci series (max %d):", MAX_TERM);
	scanf("%d", &n);
	if (n > MAX_TERM || n <= 0) {
		printf("Invalid input. Please enter a positive integer less than or
equal to % d.\n", MAX_TERM);
		return 1;
	}
	pthread_t thread;
	int arg[1];
	arg[0] = n;
	// Create thread
	if (pthread_create(&thread, NULL, fibonacci, arg) != 0) {
		fprintf(stderr, "Error creating thread.\n");
		return 1;
	}
	int* result;
	// Wait for thread to finish
	if (pthread_join(thread, (void**)&result) != 0) {
		fprintf(stderr, "Error joining thread.\n");
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define MAX_TERM 20
		// Function to calculate Fibonacci series
		void* fibonacci(void* arg) {
			int* arr = (int*)arg;
			int n = arr[0];
			int* result = (int*)malloc((n + 1) * sizeof(int));
			result[0] = 0;
			result[1] = 1;
			for (int i = 2; i <= n; i++) {
				result[i] = result[i - 1] + result[i - 2];
			}
			pthread_exit(result);
		}
		int main() {
			int n;
			printf("Enter the number of terms for Fibonacci series (max
%d):", MAX_TERM);
			scanf("%d", &n);
```

```
                    if (n > MAX_TERM || n <= 0) {
                          printf("Invalid input. Please enter a positive integer
less than or equal to % d.\n", MAX_TERM);
                          return 1;
                    }
                    pthread_t thread;
                    int arg[1];
                    arg[0] = n;
                    // Create thread
                    if (pthread_create(&thread, NULL, fibonacci, arg) != 0) {
                          fprintf(stderr, "Error creating thread.\n");
                          return 1;
                    }
                    int* result;
                    // Wait for thread to finish
                    if (pthread_join(thread, (void**)&result) != 0) {
                          fprintf(stderr, "Error joining thread.\n");
                          return 1;
                    }
                    // Display Fibonacci series
                    printf("Fibonacci series:\n");
                    for (int i = 0; i <= n; i++) {
                          printf("%d ", result[i]);
                    }
                    printf("\n");
                    // Free memory
                    free(result);
                    return 0;
              }
      }
      // Display Fibonacci series
      printf("Fibonacci series:\n");
      for (int i = 0; i <= n; i++) {
            printf("%d ", result[i]);
      }
      printf("\n");
      // Free memory
      free(result);
      return 0;
}
```

Result



PRACTICAL 7

Write a program to implement Process Synchronization by mutex locks using pThreads.

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

// Shared variable to be protected by the mutex
int counter = 0;

// Mutex to synchronize access to the shared variable
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void* increment_counter(void* arg) {
    for (int i = 0; i < 100000; i++) {
        // Lock the mutex before accessing the shared variable
        pthread_mutex_lock(&mutex);
        counter++;
        // Unlock the mutex after accessing the shared variable
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}

int main(int argc, char* argv) {
    int num_threads = 4; // Number of threads to create

    // Create threads
    pthread_t threads[num_threads];
    for (int i = 0; i < num_threads; i++) {
        if (pthread_create(&threads[i], NULL, increment_counter, NULL) != 0) {
            perror("Failed to create thread");
            return 1;
        }
    }

    // Wait for all threads to finish
    for (int i = 0; i < num_threads; i++) {
        if (pthread_join(threads[i], NULL) != 0) {
            perror("Failed to join thread");
            return 1;
        }
    }

    printf("Final counter value: %d\n", counter);

    // Destroy the mutex
    pthread_mutex_destroy(&mutex);

    return 0;
}
```

Result

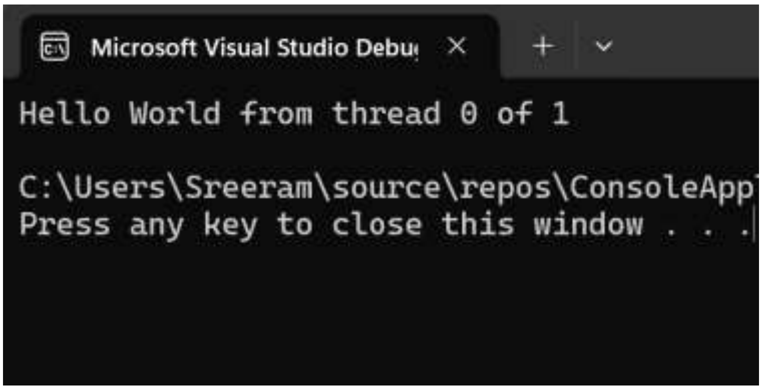
```
Final counter value: 400000


...Program finished with exit code 0
Press ENTER to exit console.
```

PRACTICAL 8

Write a "Hello World" program using OpenMP library also display number of threads created during execution.

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

int main() {
    int num_threads;

#pragma omp parallel private(num_threads)
    {
        // Get the number of threads within the parallel region
        num_threads = omp_get_num_threads();

        // Print "Hello World" from each thread
        printf("Hello World from thread %d of %d\n", omp_get_thread_num(),
num_threads);
    }

    return 0;
}
```

Result

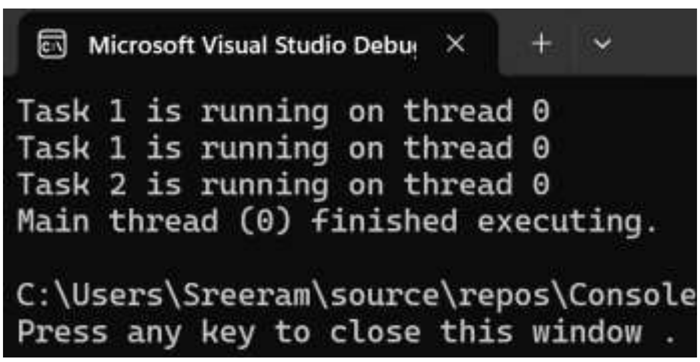


PRACTICAL 9

Write a C program to demonstrate multitask using OpenMP.

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

void task1() {
    printf("Task 1 is running on thread %d\n", omp_get_thread_num());
    for (int i = 0; i < 1000000; i++);  // Simulate some work
}

void task2() {
    printf("Task 2 is running on thread %d\n", omp_get_thread_num());
    for (int i = 0; i < 2000000; i++);  // Simulate some work
}

int main() {
    int num_threads = omp_get_max_threads();

#pragma omp parallel num_threads(num_threads)
    {
        int thread_id = omp_get_thread_num();

        // Assign tasks based on thread ID (optional)
        if (thread_id % 2 == 0) {
            task1();
        }
        else {
            task2();
        }

        // Alternatively, use sections for more granular control
#pragma omp sections nowait
        {
#pragma omp section
            {
                task1();
            }
#pragma omp section
            {
                task2();
            }
        }
    }

    printf("Main thread (%d) finished executing.\n", omp_get_thread_num());

    return 0;
}
```

Result

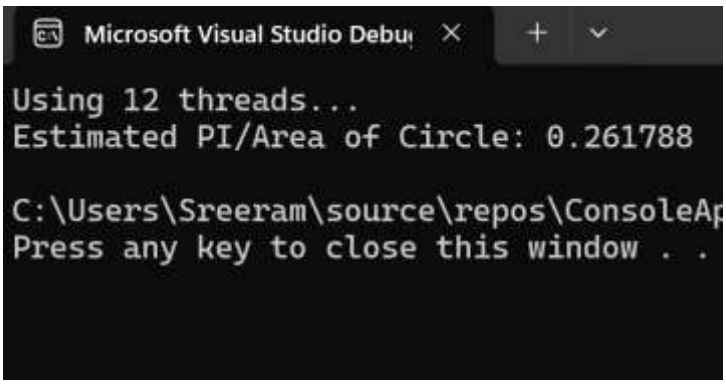


PRACTICAL 10

Write a parallel program to calculate the value of PI/Area of Circle using OpenMP library.

```
#include <stdio.h>
#include <omp.h>
#include <time.h>
#include <math.h>
#include <cstdlib>

#define DARTS 1000000  // Number of darts to throw

double calculate_pi(long num_threads) {
    long num_in_circle = 0;
    double x, y;
    unsigned int seed;  // Use unsigned int for seed

#pragma omp parallel private(x, y, seed) reduction(+:num_in_circle)
num_threads(num_threads)
    {
        seed = time(NULL) + omp_get_thread_num(); // Seed based on time and thread
ID
        srand(seed);
        for (long dart_index = 0; dart_index < DARTS / num_threads; dart_index++) {
            // Generate random coordinates within the square [-1, 1] x [-1, 1]
            x = (double)rand() / RAND_MAX * 2.0 - 1.0;
            y = (double)rand() / RAND_MAX * 2.0 - 1.0;

            // Check if the point falls within the circle of radius 1
            if (sqrt(x * x + y * y) <= 1.0) {
                num_in_circle++;
            }
        }
    }

    // Calculate PI based on the number of darts inside the circle
    return 4.0 * (double)num_in_circle / (DARTS);
}

int main() {
    int num_threads = omp_get_max_threads();
    double pi;

    printf("Using %d threads...\n", num_threads);
    pi = calculate_pi(num_threads);
    printf("Estimated PI/Area of Circle: %.6f\n", pi);

    return 0;
}
```

Result

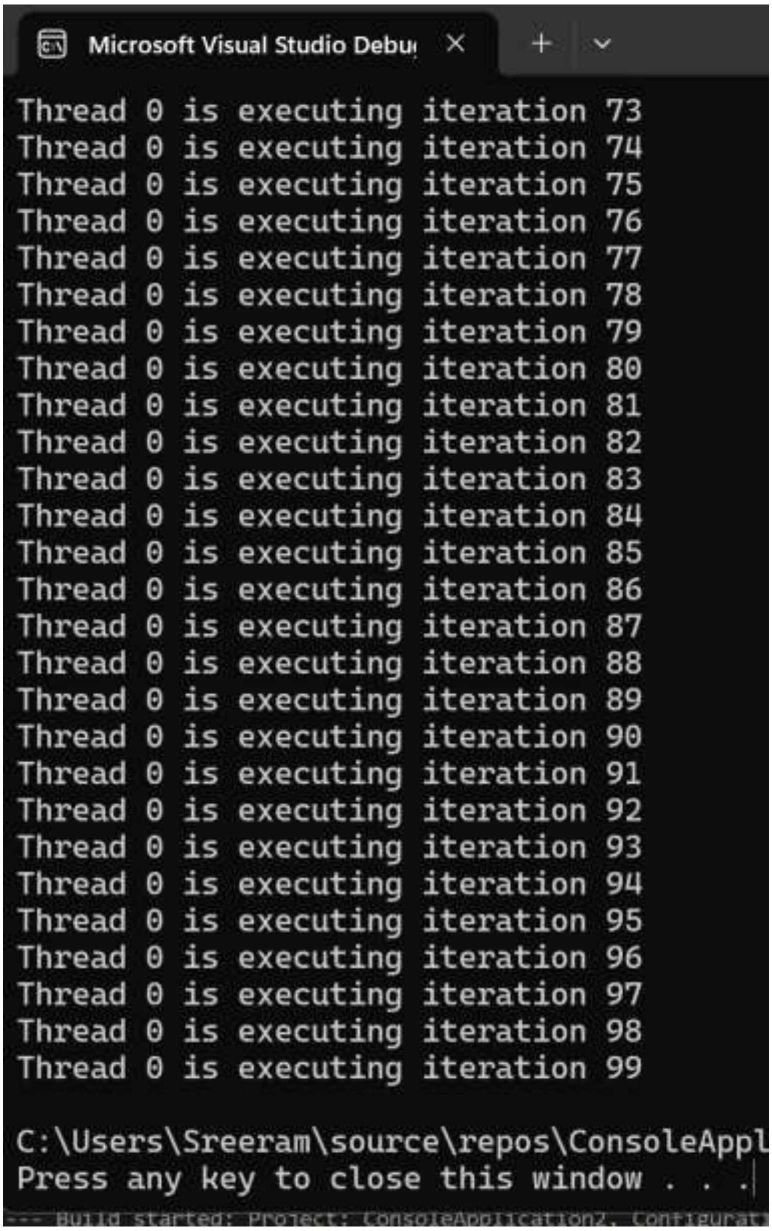


PRACTICAL 11

Write a C program to demonstrate default, static and dynamic loop scheduling using OpenMP.

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

#define N 100  // Number of iterations

void print_schedule(const char* schedule_type) {
    printf("Using %s loop scheduling...\n", schedule_type);
}

void default_schedule() {
    print_schedule("default");

#pragma omp parallel for
    for (int i = 0; i < N; i++) {
        printf("Thread %d is executing iteration %d\n", omp_get_thread_num(), i);
    }
}

void static_schedule(int chunk_size) {
    print_schedule("static");

#pragma omp parallel for schedule(static, chunk_size)
    for (int i = 0; i < N; i++) {
        printf("Thread %d is executing iteration %d\n", omp_get_thread_num(), i);
    }
}

void dynamic_schedule(int chunk_size) {
    print_schedule("dynamic");

#pragma omp parallel for schedule(dynamic, chunk_size)
    for (int i = 0; i < N; i++) {
        printf("Thread %d is executing iteration %d\n", omp_get_thread_num(), i);
    }
}

int main() {
    int num_threads = omp_get_max_threads();
    printf("Number of threads: %d\n", num_threads);

    default_schedule();
    static_schedule(4);  // Adjust chunk size as needed
    dynamic_schedule(8);  // Adjust chunk size as needed

    return 0;
}
```

<u>Result</u>


Microsoft Visual Studio Debu
Thread 0 is executing iteration 73
Thread 0 is executing iteration 74
Thread 0 is executing iteration 75
Thread 0 is executing iteration 76
Thread 0 is executing iteration 77
Thread 0 is executing iteration 78
Thread 0 is executing iteration 79
Thread 0 is executing iteration 80
Thread 0 is executing iteration 81
Thread 0 is executing iteration 82
Thread 0 is executing iteration 83
Thread 0 is executing iteration 84
Thread 0 is executing iteration 85
Thread 0 is executing iteration 86
Thread 0 is executing iteration 87
Thread 0 is executing iteration 88
Thread 0 is executing iteration 89
Thread 0 is executing iteration 90
Thread 0 is executing iteration 91
Thread 0 is executing iteration 92
Thread 0 is executing iteration 93
Thread 0 is executing iteration 94
Thread 0 is executing iteration 95
Thread 0 is executing iteration 96
Thread 0 is executing iteration 97
Thread 0 is executing iteration 98
Thread 0 is executing iteration 99
C:\Users\Sreeram\source\repos\ConsoleAppl
Press any key to close this window . . .