TABLE OF CONTENTS

Ex No: 1 Write a simple Program to demonstrate an OpenMP Fork-Join Parallelism.

AIM:

To write a simple program for demonstration of an OpenMP Fork-Join Parallelism.

ALGORITHM:

Step1: Start
Step 2: Create a program that computes a simple matrix vector multiplication.
Step 3: Input the values for the matrix.
Step 4: Calculate the multiplicative value.
Step 5: Output value.
Step 6: Stop

PROGRAM:

```
#include<stdio.h>
#include <omp.h> int
main(void)
{
printf("Before: total thread number is %d\n", omp_get_num_threads());
#pragmaomp parallel
   {
printf("Thread id is %d\n",omp_get_thread_num());
   }
printf("After: total thread number is %d\n", omp_get_num_threads());return 0;
}
```

OUTPUT:

Input: mat1[3][2] = { {1, 1}, {2, 2}, {3, 3} }
mat2[2][3] = { {1, 1, 1}, {2, 2, 2} }
Output: result[3][3] = { {3, 3, 3}, {6, 6, 6}, {9, 9, 9} }

Result:

Thus, the program has been executed successfully.

Ex No: 2 Simple matrix-vector multiplication using OpenMP directives to make it run in parallel.

AIM:

To create a program that computes a simple matrix-vector multiplicationb=Ax, either in C/C++. Use OpenMP directives to make it run in parallel.

ALGORITHM:

Step 1: Start

Step 2: Creation of program to compute b=Ax

Step 3: Get the input of two matrices

Step 4: Multiply the given matrices

Step 5: Output the resultant matrix

Step 6: Stop

PROGRAM:

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

int main() {
float A[2][2] = {{1,2},{3,4}};
float b[] = {8,10};
float c[2];
int i,j;

// computes A*b #pragmaomp
parallel forfor (i=0; i<2; i++) {
c[i]=0;
for (j=0;j<2;j++) {
c[i]=c[i]+A[i][j]*b[j];
}
}
```

```
// prints result for
(i=0; i<2; i++) {
printf("c[%i]=%f \n",i,c[i]);
}
return 0;
  }
```

OUTPUT:

Input: mat1[3][2] = { {1, 1}, {2, 2}, {3, 3} }
mat2[2][3] = { {1, 1, 1}, {2, 2, 2} }
Output: result[3][3] = { {3, 3, 3}, {6, 6, 6}, {9, 9, 9} }

Result:

Thus, the program has been executed successfully.

Ex No:3 Sum of all the elements in an array using Open MP

AIM:

To create a program that computes the sum of all the elements in an array.

ALGORITHM:

Step 1: Start

Step 2: Creation of a program for computing the sum of all the elements in an array.

Step 3: Input the array elements.

Step4: Process of addition.

Step 5: Print the resultant sum.

Step6: Stop.

PROGRAM:

```
#include<omp.h> #include
<bits/stdc++.h>
usingnamespace std;

intmain(){
        vector<int>arr{3,1,2,5,4,0};
        queue<int> data;
        intarr_sum=accumulate(arr.begin(),arr.end(),0);
        intarr_size=arr.size();
        intnew_data_size, x, y;

          for(inti=0;i<arr_size;i++){
                data.push(arr[i]);
        }
        omp_set_num_threads(ceil(arr_size/2));

        #pragmaomp parallel
                {
                        #pragmaomp critical
                        {
```

```
                                        new_data_size=data.size();
                                        for(int j=1; j<new_data_size; j=j*2){x
                                                =data.front();
                                                data.pop();
                                                y =data.front();
                                                data.pop();
                                                data.push(x+y);
                                        }
                                }

                        }


        cout<<"Array prefix sum:"<<data.front()<<endl;
        if(arr_sum==data.front())
                {cout<<"Correct sum"<<endl;
        }else{  cout<<"Incorrect Answer"<<endl;

        }
        return0;
}
```

OUTPUT:

Array of elements: 1 5 7 9 11

Sum: 33

Result:

Thus, the program has been executed successfully.

Ex No:4 Message-Passing logic using OpenMP.

AIM:

To write a simple program demonstrating Message-Passing logic using OpenMP.

ALGORITHM:

Step 1: Start

Step 2: Creation of simple program Message-Passing logic

Step 3: The message creation for transformation across web.

Step 4:Input the message.

Step 5: Process and print the result.

Step 6:Stop

PROGRAM:

```
#include <omp.h>

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

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

   // Beginning of parallel region

   #pragmaomp parallel
   {

      printf("Hello World... from thread = %d\n",
           omp_get_thread_num());
   }
   // Ending of parallel region

}
```

OUTPUT:

Hello World

Result:

Thus, the program has been executed successfully.

Ex No: 5 Floyd's Algorithm Using OpenMP.

AIM:

To write a program implementing All-Pairs Shortest-Path Problem (Flyod's Algorithm) using OpenMP.

ALGORITHM:

Step 1: Start

Step 2: Get the input of all pairs of co-ordinates

Step 3: Process the path and sort out the shortest path

Step 4:Print the resultant path

Step 5: Stop

PROGRAM:

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

//Define the number of nodes in the graph#define N
1200

//Define minimum function that will be used later on to calcualte minimumvalues between two numbers
#ifndef min
#define min(a,b) (((a) < (b)) ? (a) : (b))#endif

//Define matrix of size N * N to store distances between nodes
//Initialize all distances to zero int
distance_matrix[N][N] = {0};

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

```
{
int nthreads;
int src, dst, middle;
//Initialize the graph with random distancesfor (src =
0; src< N; src++)
{
for (dst = 0; dst< N; dst++)
{
// Distance from node to same node is 0. So, skipping these elementsif(src != dst) {
//Distances are generated to be between 0 and 19
distance_matrix[src][dst] = rand() % 20;
}
}
}
//Define time variable to record start time for execution of programdouble
start_time = omp_get_wtime();
for (middle = 0; middle < N; middle++)
{
int * dm=distance_matrix[middle];for (src
= 0; src< N; src++)
{
int * ds=distance_matrix[src];for
(dst = 0; dst< N; dst++)
{
ds[dst]=min(ds[dst],ds[middle]+dm[dst]);
}
```

```
}
}

double time = omp_get_wtime() - start_time; printf("Total time for
sequential (in sec):%.2f\n", time);

for(nthreads=1; nthreads<= 10; nthreads++) {
//Define different number of threads
omp_set_num_threads(nthreads);

// Define iterator to iterate over distance matrix
//Define time variable to record start time for execution of programdouble
start_time = omp_get_wtime();

/* Taking a node as mediator
check if indirect distance between source and distance via mediatoris less than
direct distance between them */
#pragmaomp parallel shared(distance_matrix)for
(middle = 0; middle < N; middle++)
{
int * dm=distance_matrix[middle];
#pragma omp parallel for private(src, dst) schedule(dynamic)for (src = 0;
src< N; src++)
{
int * ds=distance_matrix[src];for
(dst = 0; dst< N; dst++)
{
ds[dst]=min(ds[dst],ds[middle]+dm[dst]);
}
}
}
```

```
double time = omp_get_wtime() - start_time;
printf("Total time for thread %d (in sec):%.2f\n", nthreads, time);
}
return 0;
  }
```

Input:

The cost matrix of the graph.0 3 6

∞ ∞ ∞ ∞

3 0 2 1 ∞ ∞ ∞

6 2 0 1 4 2 ∞

∞ 1 1 0 2 ∞ 4

∞ ∞ 4 2 0 2 1

∞ ∞ 2 ∞ 2 0 1

∞ ∞ ∞ 4 1 1 0

Output:

Matrix of all pair shortest

path.0 3 4 5 6 7 7

3 0 2 1 3 4 4

4 2 0 1 3 2 3

5 1 1 0 2 3 3

6 3 3 2 0 2 1

7 4 2 3 2 0 1

7 4 3 3 1 1 0

Result:

Thus, the program has been executed successfully.

Ex No:6 Parallel Random Number Generators using Monte CarloMethods

AIM:

To implement a program Parallel Random Number Generators usingMonte Carlo Methods in OpenMP.

ALGORITHM:

Step 1: Start

Step 2: Get the input of random number

Step 3: Process it using Monte Carlo Methods in OpenMPStep 4:

Get the output of estimated value.

Step 5: Stop

PROGRAM:

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

// Function to find estimated
// value of PI using Monte
// Carlo algorithm
void monteCarlo(int N, int K)
{
    // Stores X and Y coordinates
    // of a random point
    double x, y;
    // Stores distance of a random
    // point from origin
    double d;

    // Stores number of points
    // lying inside circleint
    pCircle = 0;

    // Stores number of points
```

```
    // lying inside squareint
    pSquare = 0;

    int i = 0;

// Parallel calculation of random
// points lying inside a circle
#pragma omp parallel firstprivate(x, y, d, i) reduction(+ : pCircle, pSquare)
num_threads(K)
    {
        // Initializes random points
        // with a seed
        srand48((int)time(NULL));

        for (i = 0; i< N; i++) {
            // Finds random X co-ordinatex =
            (double)drand48();

            // Finds random X co-ordinatey =
            (double)drand48();

            // Finds the square of distance
            // of point (x, y) from origind =
            ((x * x) + (y * y));

            // If d is less than or
            // equal to 1if
            (d <= 1) {
                // Increment pCircle by 1
                pCircle++;
            }
            // Increment pSquare by 1
            pSquare++;
        }
    }
    // Stores the estimated value of PI
    double pi = 4.0 * ((double)pCircle / (double)(pSquare));

    // Prints the value in pi
    printf("Final Estimation of Pi = %f\n", pi);
```

```
}

// Driver Codeint
main()
{
    // Input
    int N = 100000;
    int K = 8;
    // Function call
    monteCarlo(N, K);
```

OUTPUT:

Final Estimation of Pi =3.1320757

Result:

Thus, the program has been executed successfully

Ex No:7 MPI Broadcast-Collective communication

AIM:

To write a program to demonstrate MPI-broadcast-and-collective communication in C.

ALGORITHM:

Step 1: Start

Step 2: Get the values for broadcasting.

Step 3: Process using MPI-broadcast-and-collective communication

Step 4:Print the output

Step 5: Stop

PROGRAM:

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

intmain(intargc, char** argv) {int
rank;
intbuf;
MPI_Status status;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if(rank == 0) {
buf = 777;
MPI_Bcast(&buf, 1, MPI_INT, 0, MPI_COMM_WORLD);
      }
 else {
MPI_Recv(&buf, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
printf("rank %d receiving received %d\n", rank, buf);
      }

MPI_Finalize();
return0;
}
```

OUTPUT:

```
>>> ./run.py my_bcast
mpirun -n 2
./my_bcast
Process 0 broadcasting data 100
```

Result:

Thus,the program has been executed successfully

Ex No:8 MPI-scatter-gather-and-all gather in C

AIM:

To write a program to demonstrate MPI-scatter-gather-and-all gather.

ALGORITHM:

Step 1: Start
Step 2: Get an array of random numbers as input.
Step 3: Compute the average of array of numbers.
Step 4: Process and print the result.
Step 5: Stop

PROGRAM:

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

// Creates an array of random numbers. Each number has a value from 0 - 1float
*create_rand_nums(int num_elements) {
float *rand_nums = (float *)malloc(sizeof(float) * num_elements);
assert(rand_nums != NULL);
int i;
for (i = 0; i<num_elements; i++) { rand_nums[i] =
(rand() / (float)RAND_MAX);
}
return rand_nums;
}

// Computes the average of an array of numbers float
compute_avg(float *array, int num_elements) {float sum = 0.f;
```

```
int i;
for (i = 0; i<num_elements; i++) {sum
+= array[i];
}
return sum / num_elements;
}

int main(int argc, char** argv) {if
(argc != 2) {
fprintf(stderr, "Usage: avgnum_elements_per_proc\n");exit(1);
}

int num_elements_per_proc = atoi(argv[1]);
// Seed the random number generator to get different results each time
srand(time(NULL));

MPI_Init(NULL, NULL);

int world_rank; MPI_Comm_rank(MPI_COMM_WORLD,
&world_rank);int world_size;
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

// Create a random array of elements on the root process. Its total
// size will be the number of elements per process times the number
// of processes
float *rand_nums = NULL;if
(world_rank == 0) {
rand_nums = create_rand_nums(num_elements_per_proc * world_size);
}
```

```
// For each process, create a buffer that will hold a subset of the entire
// array
float *sub_rand_nums = (float *)malloc(sizeof(float) *
num_elements_per_proc);
assert(sub_rand_nums != NULL);

// Scatter the random numbers from the root process to all processes in
// the MPI world
MPI_Scatter(rand_nums, num_elements_per_proc, MPI_FLOAT,
sub_rand_nums,
num_elements_per_proc, MPI_FLOAT, 0, MPI_COMM_WORLD);

// Compute the average of your subset
float sub_avg = compute_avg(sub_rand_nums, num_elements_per_proc);

// Gather all partial averages down to all the processes float *sub_avgs
= (float *)malloc(sizeof(float) * world_size);assert(sub_avgs != NULL);
MPI_Allgather(&sub_avg, 1, MPI_FLOAT, sub_avgs, 1, MPI_FLOAT,
MPI_COMM_WORLD);

// Now that we have all of the partial averages, compute the
// total average of all numbers. Since we are assuming each processcomputed
// an average across an equal amount of elements, this computation will
// produce the correct answer.
float avg = compute_avg(sub_avgs, world_size);
printf("Avg of all elements from proc %d is %f\n", world_rank, avg);

// Clean up
if (world_rank == 0) {
free(rand_nums);
}
```

```
free(sub_avgs);
free(sub_rand_nums);

MPI_Barrier(MPI_COMM_WORLD);
MPI_Finalize();
  }
```

OUTPUT:

```
>>> ./run.py avg
/home/kendall/bin/mpirun -n 4 ./avg 100Avg of
all elements is 0.478699
Avg computed across original data is 0.478699
```

Result:

Thus, the program has been executed successfully

Ex No:9 MPI-send-and-receive in C.

AIM:

To write a program to demonstrate MPI-send-and-receive in C.

ALGORITHM:

Step 1: Start

Step 2: Create a program to demonstrate MPI-send-and-receive.

Step 3:Input the message to send and receive.

Step 4: Process the message and print the output message.

Step 5: Stop

PROGRAM:

```
intmain(intargc, char ** argv)
{
int * array;
int tag=1; int
size; int
rank;
MPI_Status status;
MPI_Init (&argc,&argv);
MPI_Comm_size (MPI_COMM_WORLD,&size);
MPI_Comm_rank (MPI_COMM_WORLD,&rank);

if (rank == 0)
   {
array = malloc (10 * sizeof(int)); // Array of 10 elementsif(!array)
// error checking
      {
MPI_Abort (MPI_COMM_WORLD,1);
      }
MPI_Send(&array,10,MPI_INT,1,tag,MPI_COMM_WORLD);
   }

if (rank == 1)
   {
MPI_Recv (&array,10,MPI_INT,0,tag,MPI_COMM_WORLD,&status);
// more code here
   }

MPI_Finalize();
```

OUTPUT:

```
>>> ./run.py send_recv
mpirun –n/2 ./send_recv
Process 1 received number -1 from process 0
```

Result:

Thus, the program has been executed successfully.

Ex No: 10 Parallel rank-with-MPI in C

AIM:

To write a program for demonstrating performing-parallel-rank-with-MPI in C.

ALGORITHM:

Step 1: Start

Step 2: Create a program to demonstrate parallel-rank-with-MPI

Step 3:Input the message to send and receive.

Step 4: Process the message and print the output message.

Step 5: Stop

PROGRAM:

```
#include <stdio.h> #include
<stdlib.h> #include <mpi.h>
#include "tmpi_rank.h"
#include <time.h>

int main(int argc, char** argv) {
MPI_Init(NULL, NULL);

int world_rank; MPI_Comm_rank(MPI_COMM_WORLD,
&world_rank);int world_size;
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

// Seed the random number generator to get different results each time
srand(time(NULL) * world_rank);

float rand_num = rand() / (float)RAND_MAX;
int rank;
TMPI_Rank(&rand_num, &rank, MPI_FLOAT, MPI_COMM_WORLD); printf("Rank for %f
on process %d - %d\n", rand_num, world_rank, rank);

MPI_Barrier(MPI_COMM_WORLD);
MPI_Finalize();
  }
```

OUTPUT:

```
>>> ./run.py random_rank Mpirun –n
4 ./random_rank 100
Rank for 0.242578 on process 0 – 0
Rank for 0.894732 on process 1 – 3
Rank for 0.789463 on process 2 – 2
Rank for 0.684195 on process 3 – 1
```

Result:

Thus, the program has been executed successfully.