

Manthan Gopal Dhole
201080020
Subject :Parallel Computing

Lab Experiment 1

Aim :

Introduction to Cuda

Write a CUDA program for

1. Matrix Addition
2. Matrix Transpose
3. Matrix Multiplication

run it in Jupyter notebook in google colab

Theory :

CUDA is a model created by Nvidia for parallel computing platforms and application programming interfaces. CUDA is the parallel computing architecture of NVIDIA which allows for dramatic increases in computing performance by harnessing the power of the GPU.

Google Colab is a free cloud service and the most important feature able to distinguish Colab from other free cloud services is; Colab offers GPU and is completely free! With Colab you can work on the GPU with CUDA C/C++ for free!

CUDA code will not run on AMD CPU or Intel HD graphics unless you have NVIDIA hardware inside your machine. On Colab you can take advantage of the Nvidia GPU as well as being a fully functional Jupyter Notebook with pre-installed Tensorflow and some other ML/DL tools.

CUDA (Compute Unified Device Architecture) is a parallel computing platform and API created by NVIDIA. CUDA provides a set of functions to execute code on GPUs (Graphical Processing Units) to perform complex computations much faster than a CPU (Central Processing Unit) alone. Some of the main CUDA functions are:

1. `cudaMalloc()` - This function is used to allocate memory on the GPU.
2. `cudaMemcpy()` - This function is used to copy data from the host to the device or vice versa.
3. `cudaFree()` - This function frees memory that has been previously allocated on the GPU.
4. `global` - This keyword is used to specify a function that runs on the GPU and can be called from the host code.
5. `device` - This keyword is used to specify a function that runs on the GPU and can only be called from other GPU functions.
6. `threadIdx` - This variable contains the thread ID within a block.
7. `blockIdx` - This variable contains the block ID within a grid.
8. `blockDim` - This variable contains the dimensions of a block.
9. `gridDim` - This variable contains the dimensions of a grid.

These functions and variables are used to write and execute CUDA code on GPUs.

Link:

https://colab.research.google.com/drive/1swGAydL_inxtdS5BryEjwQ0aAc_kEVIDT?usp=sharing

Code :

Addition of Matrix

```
%%cu
#include <stdio.h>
#include <stdlib.h>

#define N 5
#define BLOCK_DIM 10

__global__ void matrixAdd (int *a, int *b, int *c) {
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;

    int index = col + row * N;

    if (col < N && row < N) {
        c[index] = a[index] + b[index];
    }
}

int main() {
    int a[N][N], b[N][N], c[N][N];
    int *dev_a, *dev_b, *dev_c;

    int size = N * N * sizeof(int);
```

```

    for(int i=0; i<N; i++)
        for (int j=0; j<N; j++){
            a[i][j] = 10;
            b[i][j] = 2;
        }

    cudaMalloc((void**)&dev_a, size);
    cudaMalloc((void**)&dev_b, size);
    cudaMalloc((void**)&dev_c, size);

    cudaMemcpy(dev_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b, b, size, cudaMemcpyHostToDevice);

    dim3 dimBlock(BLOCK_DIM, BLOCK_DIM);
    //dim3 dimGrid((int)ceil(N/dimBlock.x), (int)ceil(N/dimBlock.y));
    dim3 dimGrid((N+dimBlock.x-1)/dimBlock.x,
(N+dimBlock.y-1)/dimBlock.y);
    printf("dimGrid.x = %d, dimGrid.y = %d\n", dimGrid.x, dimGrid.y);
    matrixAdd<<<dimGrid,dimBlock>>>(dev_a,dev_b,dev_c);
    cudaMemcpy(c, dev_c, size, cudaMemcpyDeviceToHost);
    printf("Matrix a is:\n");
    for(int i=0; i<N; i++){
        for (int j=0; j<N; j++){
            printf("%d\t", a[i][j] );
        }
        printf("\n");
    }

    printf("Matrix b is:\n");
    for(int i=0; i<N; i++){
        for (int j=0; j<N; j++){
            printf("%d\t", b[i][j] );
        }
        printf("\n");
    }

    printf("Result after a*b is:\n");
    for(int i=0; i<N; i++){
        for (int j=0; j<N; j++){

```

```

        printf("%d\t", c[i][j] );
    }
    printf("\n");
}

cudaFree(dev_a);
cudaFree(dev_b);
cudaFree(dev_c);
}

```

Output:

Matrix a is:

10	10	10	10	10
10	10	10	10	10
10	10	10	10	10
10	10	10	10	10
10	10	10	10	10

Matrix b is:

2	2	2	2	2
2	2	2	2	2
2	2	2	2	2
2	2	2	2	2
2	2	2	2	2

Result after a*b is:

12	12	12	12	12
12	12	12	12	12
12	12	12	12	12
12	12	12	12	12
12	12	12	12	12

Multiplication of Matrix

```

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

#define N 5
#define BLOCK_DIM 10

__global__ void matrixMult (int *a, int *b, int *c) {
    int col = blockIdx.x * blockDim.x + threadIdx.x;

```

```

int row = blockIdx.y * blockDim.y + threadIdx.y;

if (col < N && row < N) {
    int tempSum=0;

    for(int i=0;i<N;i++){
        tempSum += a[row * N + i] * b[i * N + col];
    }

    c[row * N + col] = tempSum;
}

}

int main() {
    int a[N][N], b[N][N], c[N][N];
    int *dev_a, *dev_b, *dev_c;

    int size = N * N * sizeof(int);

    for(int i=0; i<N; i++)
        for (int j=0; j<N; j++){
            a[i][j] = 3;
            b[i][j] = 2;
        }

    cudaMalloc((void**)&dev_a, size);
    cudaMalloc((void**)&dev_b, size);
    cudaMalloc((void**)&dev_c, size);

    cudaMemcpy(dev_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b, b, size, cudaMemcpyHostToDevice);

    dim3 dimBlock(BLOCK_DIM, BLOCK_DIM);
    //dim3 dimGrid((int)ceil(N/dimBlock.x), (int)ceil(N/dimBlock.y));
    dim3 dimGrid((N+dimBlock.x-1)/dimBlock.x,
(N+dimBlock.y-1)/dimBlock.y);
    printf("dimGrid.x = %d, dimGrid.y = %d\n", dimGrid.x, dimGrid.y);

```

```

matrixMult<<<dimGrid,dimBlock>>>(dev_a,dev_b,dev_c);
cudaMemcpy(c, dev_c, size, cudaMemcpyDeviceToHost);
printf("Matrix a is:\n");
for(int i=0; i<N; i++){
    for (int j=0; j<N; j++){
        printf("%d\t", a[i][j] );
    }
    printf("\n");
}

printf("Matrix b is:\n");
for(int i=0; i<N; i++){
    for (int j=0; j<N; j++){
        printf("%d\t", b[i][j] );
    }
    printf("\n");
}

printf("Result after a*b is:\n");
for(int i=0; i<N; i++){
    for (int j=0; j<N; j++){
        printf("%d\t", c[i][j] );
    }
    printf("\n");
}

cudaFree(dev_a);
cudaFree(dev_b);
cudaFree(dev_c);
}

```

Output:

Matrix a is:

3	3	3	3	3
3	3	3	3	3
3	3	3	3	3
3	3	3	3	3
3	3	3	3	3

Matrix b is:

2	2	2	2	2
2	2	2	2	2
2	2	2	2	2
2	2	2	2	2
2	2	2	2	2

Result after a*b is:

30	30	30	30	30
30	30	30	30	30
30	30	30	30	30
30	30	30	30	30
30	30	30	30	30

.

Transpose of Matrix

```
%%cu
#include <stdio.h>
#include <stdlib.h>

#define N 5
#define BLOCK_DIM 10

__global__ void matrixTrans(int *ain, int *aout) {
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;

    if (col < N && row < N) {
        aout[col*N+row]=ain[row*N+col];
    }
}

int main() {
    int a_in[N][N], a_out[N][N];
    int *dev_ain, *dev_aout;
```



```

int size = N * N * sizeof(int);

for(int i=0; i<N; i++)
    for (int j=0; j<N; j++){
        a_in[i][j] = i*i+j;
    }

cudaMalloc((void**) &dev_ain, size);
cudaMalloc((void**) &dev_aout, size);

cudaMemcpy(dev_ain, a_in, size, cudaMemcpyHostToDevice);
cudaMemcpy(dev_aout, a_out, size, cudaMemcpyHostToDevice);

dim3 dimBlock(BLOCK_DIM, BLOCK_DIM);
//dim3 dimGrid((int)ceil(N/dimBlock.x), (int)ceil(N/dimBlock.y));
dim3 dimGrid((N+dimBlock.x-1)/dimBlock.x,
(N+dimBlock.y-1)/dimBlock.y);
printf("dimGrid.x = %d, dimGrid.y = %d\n", dimGrid.x, dimGrid.y);
matrixTrans<<<dimGrid,dimBlock>>>(dev_ain,dev_aout);
cudaMemcpy(a_out, dev_aout, size, cudaMemcpyDeviceToHost);
printf("Input Matrix is:\n");
for(int i=0; i<N; i++){
    for (int j=0; j<N; j++){
        printf("%d\t", a_in[i][j] );
    }
    printf("\n");
}

printf("Transpose Matrix is:\n");
for(int i=0; i<N; i++){
    for (int j=0; j<N; j++){
        printf("%d\t", a_out[i][j] );
    }
    printf("\n");
}
}

```

Output:

Input Matrix is:

1	2	3	4	5
2	3	4	5	6
5	6	7	8	9
10	11	12	13	14
17	18	19	20	21

Transpose Matrix is:

1	2	5	10	17
2	3	6	11	18
3	4	7	12	19
4	5	8	13	20
5	6	9	14	21

Conclusion :

Thus, in this experiment we studied cuda programming and understood how parallel programming works and the importance of the gpu in parallel computing.

We used google colab platform to run programs for matrix addition, transpose and multiplication.

We studied various functions such as cudaMalloc for memory allocation, cudaFree for memory deallocation and the built-in device variables blockDim, blockIdx, and threadIdx used to identify and differentiate GPU threads that execute the kernel in parallel.

Name : Manthan Gopal Dhole
Id : 201080020

TY IT
Parallel Computing

Link : [Click IT](#)

Experiment 2

Aim: Write a code for implementing BFS(Breadth First Search) using CUDA.

Theory :

BFS (Breadth-First Search) is a graph traversal algorithm that visits all the vertices of a graph in breadth-first order. CUDA is a parallel computing platform and API that allows you to run computationally intensive tasks on the GPU (Graphics Processing Unit) rather than the CPU. By using CUDA to implement BFS, you can speed up the computation by utilizing the parallel processing capability of the GPU.

Here's a high-level explanation of how BFS can be implemented using CUDA:

Convert the graph into a data structure that can be processed in parallel on the GPU, such as an adjacency list or matrix.

Allocate GPU memory to store the graph and the distances from the source vertex to all other vertices.

Copy the graph and the distances from the CPU memory to the GPU memory.

Launch a CUDA kernel to perform the BFS. The kernel will be executed in parallel by multiple threads on the GPU.

In the CUDA kernel, each thread processes one vertex at a time. If the distance of the vertex has not been set yet, it is set to the distance of the source vertex plus one. The distance of all the

Name : Manthan Gopal Dhole
Id : 201080020

TY IT
Parallel Computing

neighbors of the vertex are also updated using the atomicMin function, which ensures that only one thread can update the distance of a neighbor at a time.

The BFS continues until all vertices have been processed.

Finally, copy the updated distances from the GPU memory back to the CPU memory.

By using CUDA to parallelize the BFS, you can significantly speed up the computation, especially for large graphs with many vertices and edges.

Code :

```
%%cu
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <cuda.h>
#include <cuda_runtime_api.h>

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

#define NUM_NODES 5

typedef struct
{
    int start;        // Index of first adjacent node in Ea
    int length;       // Number of adjacent nodes
} Node;

__global__ void CUDA_BFS_KERNEL(Node *Va, int *Ea, bool *Fa, bool *Xa, int
*Ca, bool *done)
{
    int id = threadIdx.x + blockIdx.x * blockDim.x;
    if (id > NUM_NODES)
        *done = false;
```

Name : Manthan Gopal Dhole
Id : 201080020

TY IT
Parallel Computing

```
if (Fa[id] == true && Xa[id] == false)
{
    printf("%d ", id); //This printf gives the order of vertices in BFS
    Fa[id] = false;
    Xa[id] = true;
    __syncthreads();
    int k = 0;
    int i;
    int start = Va[id].start;
    int end = start + Va[id].length;
    for (int i = start; i < end; i++)
    {
        int nid = Ea[i];

        if (Xa[nid] == false)
        {
            Ca[nid] = Ca[id] + 1;
            Fa[nid] = true;
            *done = false;
        }

    }

}

// The BFS frontier corresponds to all the nodes being processed at the
current level.

int main()
{
```

Name : Manthan Gopal Dhole
Id : 201080020

TY IT
Parallel Computing

```
Node node[NUM_NODES];

//int edgesSize = 2 * NUM_NODES;
int edges[NUM_NODES];

node[0].start = 0;
node[0].length = 2;

node[1].start = 2;
node[1].length = 1;

node[2].start = 3;
node[2].length = 1;

node[3].start = 4;
node[3].length = 1;

node[4].start = 5;
node[4].length = 0;

edges[0] = 1;
edges[1] = 2;
edges[2] = 4;
edges[3] = 3;
edges[4] = 4;

bool frontier[NUM_NODES] = { false };
bool visited[NUM_NODES] = { false };
int cost[NUM_NODES] = { 0 };

int source = 0;
frontier[source] = true;

Node* Va;
cudaMalloc((void**)&Va, sizeof(Node)*NUM_NODES);
cudaMemcpy(Va, node, sizeof(Node)*NUM_NODES, cudaMemcpyHostToDevice);

int* Ea;
```

Name : Manthan Gopal Dhole
Id : 201080020

TY IT
Parallel Computing

```
cudaMalloc((void**)&Ea, sizeof(Node)*NUM_NODES);
cudaMemcpy(Ea, edges, sizeof(Node)*NUM_NODES, cudaMemcpyHostToDevice);

bool* Fa;
cudaMalloc((void**)&Fa, sizeof(bool)*NUM_NODES);
cudaMemcpy(Fa, frontier, sizeof(bool)*NUM_NODES,
cudaMemcpyHostToDevice);

bool* Xa;
cudaMalloc((void**)&Xa, sizeof(bool)*NUM_NODES);
cudaMemcpy(Xa, visited, sizeof(bool)*NUM_NODES, cudaMemcpyHostToDevice);

int* Ca;
cudaMalloc((void**)&Ca, sizeof(int)*NUM_NODES);
cudaMemcpy(Ca, cost, sizeof(int)*NUM_NODES, cudaMemcpyHostToDevice);


int num_blks = 1;
int threads = 5;


bool done;
bool* d_done;
cudaMalloc((void**)&d_done, sizeof(bool));
printf("\n\n");
int count = 0;


printf("Order: \n\n");
do {
    count++;
    done = true;
    cudaMemcpy(d_done, &done, sizeof(bool), cudaMemcpyHostToDevice);
    CUDA_BFS_KERNEL <<<num_blks, threads >>>(Va, Ea, Fa, Xa, Ca,d_done);
    cudaMemcpy(&done, d_done , sizeof(bool), cudaMemcpyDeviceToHost);

} while (!done);
```

Name : Manthan Gopal Dhole
Id : 201080020

TY IT
Parallel Computing

```
cudaMemcpy(cost, Ca, sizeof(int)*NUM_NODES, cudaMemcpyDeviceToHost);

printf("Number of times the kernel is called : %d \n", count);

printf("\nCost: ");
for (int i = 0; i<NUM_NODES; i++)
    printf(" %d ", cost[i]);
printf("\n");
}
```

Output :

Order:

```
0 1 2 3 4 Number of times the kernel is called : 3
```

```
Cost: 0    1    1    2    2
```

Conclusion:

In conclusion, BFS (Breadth-First Search) is a widely used graph traversal algorithm that visits all the vertices of a graph in breadth-first order. By using CUDA to implement BFS, the computation can be significantly sped up by utilizing the parallel processing capability of the GPU. With CUDA, BFS can be parallelized and executed in parallel by multiple threads on the GPU, which leads to faster computation times, especially for large graphs with many vertices and edges. Implementing BFS using CUDA requires a good understanding of both graph algorithms and parallel computing concepts, but the potential performance gains make it a worthwhile effort.

Name : Manthan Gopal Dhole
Id : 201080020

TY IT
Parallel Computing

Name : Manthan Gopal Dhole
Id : 201080020

TY IT
Parallel Computing

Link : [Click IT](#)

Experiment 2

Aim: Implementation of parallel quicksort [Hyper quick sort]
using CUDA.

Theory :

QuickSort is a popular sorting algorithm used in parallel computing, which follows a divide-and-conquer approach to sort an array. The basic idea is to partition the array into smaller sub-arrays and then sort those sub-arrays.

In parallel computing, the partitioning process can be performed in parallel using multiple threads or processors, making the sorting process faster. The parallelization of quicksort algorithms can be done in two ways: recursive parallelization and iterative parallelization.

In the recursive parallelization method, the partitioning process is performed by dividing the array into smaller sub-arrays recursively until the sub-arrays are small enough to be sorted by a single thread. Each recursive call creates a new thread to perform the partitioning process on the sub-array.

In the iterative parallelization method, the partitioning process is performed iteratively using a single thread that divides the array into smaller sub-arrays. These sub-arrays are pushed onto a stack and then popped off the stack and sorted iteratively until the entire array is sorted.

Name : Manthan Gopal Dhole
Id : 201080020

TY IT
Parallel Computing

In both methods, the partitioning process is the most important part of the quicksort algorithm. It involves selecting a pivot element, and then partitioning the array such that all the elements less than the pivot are on the left side of the pivot, and all the elements greater than the pivot are on the right side of the pivot. This process can be performed in parallel by dividing the array into smaller sub-arrays and performing the partitioning process on each sub-array.

After the partitioning process, the two resulting sub-arrays are sorted recursively or iteratively using quicksort algorithm. Finally, the sorted sub-arrays are merged to get the sorted array.

Parallel implementation of quicksort algorithm is very useful in situations where the size of the array is very large, and sequential sorting algorithm takes too much time.

Code :

```
%%cu
#include <stdio.h>
#include <stdlib.h>

__device__ int d_size;

__global__ void partition (int *arr, int *arr_l, int *arr_h, int n)
{
    int z = blockIdx.x*blockDim.x+threadIdx.x;
    d_size = 0;
    __syncthreads();
    if (z<n)
    {
        int h = arr_h[z];
        int l = arr_l[z];
        int x = arr[h];
        int i = (l - 1);
        int temp;
        for (int j = l; j <= h- 1; j++)
        {
            if (arr[j] <= x)
```

Name : Manthan Gopal Dhole
Id : 201080020

TY IT
Parallel Computing

```
        {
            i++;
            temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
    temp = arr[i+1];
    arr[i+1] = arr[h];
    arr[h] = temp;
    int p = (i + 1);
    if (p-1 > 1)
    {
        int ind = atomicAdd(&d_size, 1);
        arr_l[ind] = 1;
        arr_h[ind] = p-1;
    }
    if ( p+1 < h )
    {
        int ind = atomicAdd(&d_size, 1);
        arr_l[ind] = p+1;
        arr_h[ind] = h;
    }
}

void quickSortIterative (int arr[], int l, int h)
{
    int lstack[ h - l + 1 ], hstack[ h - l + 1];

    int top = -1, *d_d, *d_l, *d_h;

    lstack[ ++top ] = l;
    hstack[ top ] = h;

    cudaMalloc(&d_d, (h-l+1)*sizeof(int));
    cudaMemcpy(d_d, arr, (h-l+1)*sizeof(int), cudaMemcpyHostToDevice);

    cudaMalloc(&d_l, (h-l+1)*sizeof(int));
    cudaMemcpy(d_l, lstack, (h-l+1)*sizeof(int), cudaMemcpyHostToDevice);
```

Name : Manthan Gopal Dhole
Id : 201080020

TY IT
Parallel Computing

```
    cudaMalloc(&d_h, (h-1+1)*sizeof(int));  
    cudaMemcpy(d_h, hstack, (h-1+1)*sizeof(int), cudaMemcpyHostToDevice);  
    int n_t = 1;  
    int n_b = 1;  
    int n_i = 1;  
    while ( n_i > 0 )  
    {  
        partition<<<n_b,n_t>>>( d_d, d_l, d_h, n_i);  
        int answer;  
        cudaMemcpyFromSymbol(&answer, d_size, sizeof(int), 0,  
cudaMemcpyDeviceToHost);  
        if (answer < 1024)  
        {  
            n_t = answer;  
        }  
        else  
        {  
            n_t = 1024;  
            n_b = answer/n_t + (answer%n_t==0?0:1);  
        }  
        n_i = answer;  
        cudaMemcpy(arr, d_d, (h-1+1)*sizeof(int), cudaMemcpyDeviceToHost);  
    }  
}  
  
int main()  
{  
    int arr[10] = {9,8,7,6,5,949,11,1,2,100};  
  
    printf("Initial array: ");  
    for(int i = 0; i < 10; i++){  
        printf("%d " , arr[i]);  
    }  
  
    int n = sizeof( arr ) / sizeof( *arr );  
    quickSortIterative( arr, 0, n - 1 );  
}
```

Name : Manthan Gopal Dhole
Id : 201080020

TY IT
Parallel Computing

```
printf("\nApplying quick sort... \nDisplaying sorted array: ");  
  
for(int i = 0; i < 10; i++){  
    printf("%d " , arr[i]);  
}  
return 0;  
}
```

Output :

Initial array: 9 8 7 6 5 949 11 1 2 100

Applying quick sort...

Displaying sorted array: 1 2 5 6 7 8 9 11 100 949

```
Initial array: 9 8 7 6 5 949 11 1 2 100  
Applying quick sort...  
Displaying sorted array: 1 2 5 6 7 8 9 11 100 949
```

Explanation :

This code is an implementation of the quicksort algorithm using CUDA, a parallel computing platform and programming model developed by NVIDIA. The quicksort algorithm is used to sort an array of integers in ascending order.

The main function initializes an array of 10 integers and calls the quickSortIterative function to sort it. The quickSortIterative function takes three arguments: the array to be sorted, the lower index of the array, and the higher index of the array. It first initializes three arrays: lstack, hstack, and d_d, where lstack and hstack are used to store the lower and higher indexes of the partitions and d_d is a device (GPU) pointer that points to the array to be sorted in the device memory.

The function then starts a loop to iteratively sort the array using the quicksort algorithm. In each iteration, it calls the partition function which partitions the array based on a pivot

Name : Manthan Gopal Dhole
Id : 201080020

TY IT
Parallel Computing

element and returns the lower and higher indexes of the two partitions. The partition function is executed in parallel on the device by multiple threads organized in blocks. The number of threads and blocks is determined dynamically based on the size of the current partition. The atomicAdd function is used to increment the d_size variable, which is shared by all threads, in order to dynamically determine the size of the next partitions.

After the partition function completes, the size of the next partitions is obtained by copying the value of d_size from the device to the host. If the size is less than or equal to 1024, the number of threads in the next partition is set to the size. Otherwise, the number of threads is set to 1024 and the number of blocks is calculated to cover the entire partition. The loop continues until all partitions have been sorted.

Finally, the sorted array is copied back to the host memory and printed.

Conclusion:

CUDA programming model is well-suited for hyper quicksort due to its ability to efficiently handle large amounts of data and its support for parallelism at the thread and block levels. The performance of hyper quicksort in parallel computing depends on various factors, including the size of the input array, the number of processing units available, and the quality of the load balancing scheme. Generally, hyper quicksort can achieve significant speedup and scalability when executed on a large number of processing units. However, care must be taken to ensure that the load is well-balanced across all the processing units to avoid performance degradation.

Name : Manthan Dhole

Id : 201080020

Subject : Parallel Computing

Experiment 4

AIM:

Implementation using OpenMP.

i. Fork Join model, ii. Producer

Consumer problem, iii. Matrix

Multiplication, iv. find prime
number,

v. Largest Element in an array and vi.

Pi calculation

THEORY:

What is OpenMP?

OpenMP is a standard parallel programming API for shared memory environments, written in C, C++, or FORTRAN. It consists of a set of compiler directives with a “lightweight” syntax, library routines, and environment variables that influence run-time behavior. OpenMP is governed by the OpenMP Architecture Review Board (or OpenMP ARB), and is defined by several hardware and software vendors.

OpenMP behavior is directly dependent on the OpenMP implementation. Capabilities of this implementation can enable the programmer to separate the program into serial and parallel regions rather than just concurrently running threads, hides stack management, and provides synchronization of constructs. That being said, OpenMP will not guarantee speedup, parallelize dependencies, or prevent data racing. Data racing, keeping track of dependencies, and working towards a speedup are all up to the programmer.

Why do we use OpenMP?

OpenMP has received considerable attention in the past decade and is considered by many to be an ideal solution for parallel programming because it has unique advantages as a mainstream directive-based programming model.

First of all, OpenMP provides a cross-platform, cross-compiler solution. It supports lots of platforms such as Linux, macOS, and Windows. Mainstream compilers including GCC, LLVM/Clang, Intel Fortran, and C/C++ compilers provide OpenMP good support. Also, with the rapid development of OpenMP, many researchers and computer vendors are constantly exploring how to optimize the execution efficiency of OpenMP programs and continue to propose improvements for existing compilers or develop new compilers. What’s more. OpenMP

Name : Manthan Dhole

Id : 201080020

Subject : Parallel Computing

is a standard specification, and all compilers that support it implement the same set of standards, and there are no portability issues.

Secondly, using OpenMP can be very convenient and flexible to modify the number of threads. To solve the scalability problem of the number of CPU cores. In the multi-core era, the number of threads needs to change according to the number of CPU cores. OpenMP has irreplaceable advantages in this regard.

Thirdly, using OpenMP to create threads is considered to be convenient and relatively easy because it does not require an entry function, the code within the same function can be decomposed into multiple threads for execution, and a for loop can be decomposed into multiple threads for execution. If OpenMP is not used, when the operating system API creates a thread, the code in a function needs to be manually disassembled into multiple thread entry functions. To sum up, OpenMP has irreplaceable advantages in parallel programming. More and more new directives are being added to achieve more functions, and they are playing an important role on many different platforms

NOTEBOOK LINK:

https://colab.research.google.com/drive/1l69Ap2SycBoVskpegLON_Mb1lVmLk6qx#scrollTo=h5ePDwBGDZsN

CODE:

i. Fork Join model,

```
code1 = """ // OpenMP program to print
Hello World
// using C language
// OpenMP header
#include <omp.h>
#include <stdio.h> #include
<stdlib.h> int main(int argc,
char* argv[]) {
    // Beginning of parallel region
    #pragma omp parallel { printf("Hello
World... from thread = %d \\n",
omp_get_thread_num()); }
    // Ending of parallel region
} """ text_file =
open("code1.c", "w")
text_file.write(code1)
text_file.close() %env
OMP_NUM_THREADS=3
```

Name : Manthan Dhole

Id : 201080020

Subject : Parallel Computing

```
!gcc -o code1 -fopenmp code1.c
!./code1
```

ii. Producer Consumer problem,

```
code2 = ""
```

```
//Code to implement and solve producer consumer problem
//Using OpenMP

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

#define BUFFER_SIZE 10

int buffer[BUFFER_SIZE];
int count = 0; int in =
0; int out = 0;

omp_lock_t lock;

void producer() {    int item;    while (1) {        item
= rand() % 100;        omp_set_lock(&lock);        if
(count < BUFFER_SIZE) {            buffer[in] = item;
in = (in + 1) % BUFFER_SIZE;            count++;
printf(" Producer produced item %d\\n", item);        }
omp_unset_lock(&lock);        #pragma omp flush
sleep(1);    } }
```

```
void consumer() {
int item;
while (1) {
omp_set_lock(&lock);
if
(count > 0) {
item =
buffer[out];
out = (out + 1)
% BUFFER_SIZE;
```

Name : Manthan Dhole

Id : 201080020

Subject : Parallel Computing

```
count--;
printf("Consumer
consumed item
%d\\n", item);
}
omp_unset_lock(&
lock);
#pragma omp
flush
sleep(1);    } }
```

```
int main() {    omp_init_lock(&lock);
#pragma omp parallel num_threads(2)
{        int tid =
omp_get_thread_num();        if (tid
== 0) {        producer();
} else {        consumer();
}    }    omp_destroy_lock(&lock);
return 0; }
```

```
""" text_file = open("code2.c",
"w") text_file.write(code2)
text_file.close() %env
OMP_NUM_THREADS=3
!gcc -o code2 -fopenmp code2.c
!./code2
```

iii. Matrix Multiplication,

code3="""

```
#include <pthread.h>
```

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

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

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

```
#include <sys/time.h>
```

```
#define N 5
```

Name : Manthan Dhole

Id : 201080020

Subject : Parallel Computing

```
int A[N][N];
int B[N][N];
int C[N][N];

int main() {    int i,j,k;    struct timeval
tv1, tv2;    struct timezone tz;    double
elapsed;
omp_set_num_threads(omp_get_num_procs());
for (i= 0; i< N; i++)        for (j= 0; j<
N; j++)    {
        A[i][j] = 2;
        B[i][j] = 2;    }
printf("Matrix A:\\n");    for (i=
0; i< N; i++)        {        for (j=
0; j< N; j++)            {
printf("%d\\t",A[i][j]);            }
printf("\\n");        }
printf("\\nMatrix B:\\n");    for
(i= 0; i< N; i++)        {        for
(j= 0; j< N; j++)            {
printf("%d\\t",B[i][j]);            }
printf("\\n");        }
gettimeofday(&tv1, &tz);
#pragma omp parallel for
private(i,j,k) shared(A,B,C)
for (i = 0; i < N; ++i) {
for (j = 0; j < N; ++j) {
for (k = 0; k < N; ++k) {
C[i][j] += A[i][k] * B[k][j];
        }
    }
}

    gettimeofday(&tv2, &tz);    elapsed = (double)
(tv2.tv_sec-tv1.tv_sec) + (double) (tv2.tv_usec-tv1.tv_usec) * 1.e-6;
printf("\\nelapsed time = %f seconds.\\n", elapsed);
```

Name : Manthan Dhole

Id : 201080020

Subject : Parallel Computing

```
printf("\n\nThe product:\n");
for (i= 0; i< N; i++)    {
for (j= 0; j< N; j++)    {
printf("%d\t",C[i][j]);    }
printf("\n");    }
} """ text_file =
open("code3.c", "w")
text_file.write(code3)
text_file.close() %env
OMP_NUM_THREADS=3
!gcc -o code3 -fopenmp code3.c
!./code3
```

iv. find prime number, code4="""

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

int is_prime(int n) {    if (n <= 1)
{    return 0;    }    for (int
i = 2; i * i <= n; i++) {    if
(n % i == 0) {    return 0;
}    }    return 1; }

int main() {    #pragma omp parallel
for    for (int i = 2; i <= 100; i++) {
if (is_prime(i)) {
printf("%d is prime\n", i);    }
}    return 0; } """ text_file =
open("code4.c", "w")
text_file.write(code4)
text_file.close() %env
OMP_NUM_THREADS=3
!gcc -o code4 -fopenmp code4.c
!./code4
```

Name : Manthan Dhole

Id : 201080020

Subject : Parallel Computing

v. Largest Element in an array and

```
code5=""          #include
<stdio.h>
#include <stdlib.h>
#include <omp.h>

#define ARRAY_SIZE 10

int main() {      int i;
int largest = 0;   int
array[ARRAY_SIZE];

    // initialize the array with random values    for (i
= 0; i < ARRAY_SIZE; i++) {        array[i] = rand() %
10000;        printf("%d\\t",array[i]);    }
printf("\\n");    // find the largest element in the
array using OpenMP    #pragma omp parallel for
reduction(max:largest)    for (i = 0; i < ARRAY_SIZE;
i++) {        if (array[i] > largest) {
largest = array[i];        }    }

    printf("The largest element in the array is %d\\n", largest);

    return 0; } "" text_file =
open("code5.c", "w")
text_file.write(code5)
text_file.close() %env
OMP_NUM_THREADS=3
!gcc -o code5 -fopenmp code5.c
!./code5
```

vi. Pi calculation

```
code6="" #include
<stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>
```

Name : Manthan Dhole

Id : 201080020

Subject : Parallel Computing

```
int main() {    int n = 10000;    // number
of iterations    double x, y, pi;    int
count = 0;

    // set random seed
srand(time(NULL));

    #pragma omp parallel for private(x, y) reduction(+:count)
for (int i = 0; i < n; i++) {        // generate random
point (x, y) in [0, 1] x [0, 1]        x = (double)rand() /
RAND_MAX;        y = (double)rand() / RAND_MAX;

        // test if point is inside unit circle
if (x * x + y * y <= 1) {
count++;        }
    }

    // calculate pi    pi
= 4.0 * count / n;

    printf("pi = %f\\n", pi);

    return 0;
}

""" text_file = open("code6.c",
"w") text_file.write(code6)
text_file.close() %env
OMP_NUM_THREADS=3
!gcc -o code6 -fopenmp code6.c
!./code6
```

OUTPUT:

i. Fork Join model,

```
env: OMP_NUM_THREADS=3
Hello World... from thread = 0
Hello World... from thread = 2
Hello World... from thread = 1
```

Name : Manthan Dhole

Id : 201080020

Subject : Parallel Computing

ii. Producer Consumer problem,

```
code2.c: In function 'producer':
code2.c:32:9: warning: implicit declaration of function 'sleep' [-Wimplicit-function-declaration]
   32 |         sleep(1);
       |         ^~~~~
   Producer produced item 83
   Consumer consumed item 83
   Producer produced item 86
   Consumer consumed item 86
   Producer produced item 77
   Consumer consumed item 77
   Producer produced item 15
   Producer produced item 93
   Consumer consumed item 15
   Producer produced item 35
   Consumer consumed item 93
   Consumer consumed item 35
   Producer produced item 86
   Consumer consumed item 86
   Producer produced item 92
   Consumer consumed item 92
   Producer produced item 49
   Consumer consumed item 49
   Producer produced item 21
   Consumer consumed item 21
   Producer produced item 62
   Consumer consumed item 62
   Producer produced item 27
   Consumer consumed item 27
   Producer produced item 90
^C
```

iii. Matrix Multiplication,

Name : Manthan Dhole

Id : 201080020

Subject : Parallel Computing

```
Matrix A:
2      2      2      2      2
2      2      2      2      2
2      2      2      2      2
2      2      2      2      2
2      2      2      2      2
```

```
Matrix B:
2      2      2      2      2
2      2      2      2      2
2      2      2      2      2
2      2      2      2      2
2      2      2      2      2
```

```
elapsed time = 0.000049 seconds.
```

```
The product:
20      20      20      20      20
20      20      20      20      20
20      20      20      20      20
20      20      20      20      20
20      20      20      20      20
```

iv. find prime number,

Name : Manthan Dhole

Id : 201080020

Subject : Parallel Computing

```
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime
71 is prime
73 is prime
79 is prime
83 is prime
89 is prime
97 is prime
37 is prime
41 is prime
43 is prime
47 is prime
53 is prime
59 is prime
61 is prime
67 is prime
```

v. Largest Element in an array and

```
9383  886  2777  6915  7793  8335  5386  492  6649  1421
The largest element in the array is 9383
```

vi. Pi calculation

```
pi = 3.134400
```

CONCLUSION:

In this lab we learned about openMP, and used many openMP inbuilt functions in order to include parallel processing into the program we are writing. We learned the format in which the code is written so that we can fork it and run it in parallel. We also learned how to declare the number of threads and the fork join model. We also revised some other concepts like consumer producer problem.

Name : Manthan Gopal Dhole
Id : 201080020

TY IT
Parallel Computing

Link : [Click IT](#)

Experiment 5

Aim: Implement DIJKSTRA'S ALGORITHM using OpenMP

Theory :

Dijkstra's algorithm is a popular shortest path algorithm used to find the shortest path between a source vertex and all other vertices in a weighted graph. OpenMP is a parallel programming model used to achieve parallelism in C/C++ programs.

The following are the steps involved in implementing Dijkstra's algorithm using OpenMP:

Initialize the distance array and visited array: In this step, we initialize the distance array to infinity and the visited array to zero for all vertices in the graph. This is done using an OpenMP parallel loop.

Set the distance to the starting node as 0: We set the distance to the starting node as 0.

Find the shortest path for all vertices: In this step, we find the shortest path for all vertices in the graph. We use an OpenMP parallel loop to iterate over all vertices in the graph. For each vertex, we find the vertex with the minimum distance value and mark it as visited. Then, we update the distance values of its adjacent vertices. This step is repeated until all vertices are visited.

Print the distances of all vertices from the starting vertex: In this step, we print the distances of all vertices from the starting vertex.

Dijkstra's algorithm is a popular shortest path algorithm that can be used to find the shortest path between two vertices in a graph. OpenMP is a programming model that allows for parallel programming in shared-memory architectures.

Name : Manthan Gopal Dhole
Id : 201080020

TY IT
Parallel Computing

When implementing Dijkstra's algorithm using OpenMP, the algorithm can be parallelized by dividing the work among multiple threads. This can be achieved by partitioning the graph into smaller subgraphs, and assigning each subgraph to a separate thread.

One way to do this is to use a priority queue to keep track of the vertices that need to be visited, with the vertex with the lowest distance being the next to be visited. Each thread can take vertices from the priority queue in parallel, and update their distances to neighboring vertices. This can be done by assigning each thread a portion of the graph and having it work on its own subgraph in parallel.

To ensure correctness, synchronization mechanisms such as locks or atomic operations may be necessary to avoid data races and ensure that only one thread updates a vertex's distance at a time.

Overall, using OpenMP can greatly speed up Dijkstra's algorithm, especially for large graphs, by taking advantage of multiple processors in a shared-memory system.

Code : [Link](#)

```
code =""  
#include <stdio.h>  
#include <stdlib.h>  
#include <omp.h>  
  
#define INF 999999  
  
void dijkstra(int** graph, int n, int start)  
{  
    int* dist = (int*)malloc(n * sizeof(int));  
    int* visited = (int*)malloc(n * sizeof(int));  
  
    // initialize the distance array and visited array  
    #pragma omp parallel for  
    for (int i = 0; i < n; i++) {  
        dist[i] = INF;  
        visited[i] = 0;  
    }  
}
```

Name : Manthan Gopal Dhole
Id : 201080020

TY IT
Parallel Computing

```
// set the distance to the starting node as 0
dist[start] = 0;

// find shortest path for all vertices
#pragma omp parallel for
for (int count = 0; count < n - 1; count++) {
    // find the vertex with minimum distance value
    int min_dist = INF;
    int u;
    #pragma omp parallel for
    for (int v = 0; v < n; v++) {
        if (!visited[v] && dist[v] <= min_dist) {
            min_dist = dist[v];
            u = v;
        }
    }

    // mark the selected vertex as visited
    visited[u] = 1;

    // update the distance values of the adjacent vertices of the
selected vertex
    #pragma omp parallel for
    for (int v = 0; v < n; v++) {
        if (!visited[v] && graph[u][v] && dist[u] != INF && dist[u] +
graph[u][v] < dist[v]) {
            dist[v] = dist[u] + graph[u][v];
        }
    }
}

// print the distances of all vertices from the starting vertex
printf("Vertex \\t Distance from Start\\n");
for (int i = 0; i < n; i++) {
    printf("%d \\t\\t %d\\n", i, dist[i]);
}

free(dist);
```

Name : Manthan Gopal Dhole
Id : 201080020

TY IT
Parallel Computing

```
        free(visited);
    }

int main()
{
    int n, start;
    printf("Enter the number of vertices: ");
    scanf("%d", &n);

    int** graph = (int**)malloc(n * sizeof(int*));
    for (int i = 0; i < n; i++) {
        graph[i] = (int*)malloc(n * sizeof(int));
    }

    printf("Enter the adjacency matrix:\\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%d", &graph[i][j]);
        }
    }

    printf("Enter the starting vertex: ");
    scanf("%d", &start);

    dijkstra(graph, n, start);

    for (int i = 0; i < n; i++) {
        free(graph[i]);
    }
    free(graph);

    return 0;
}

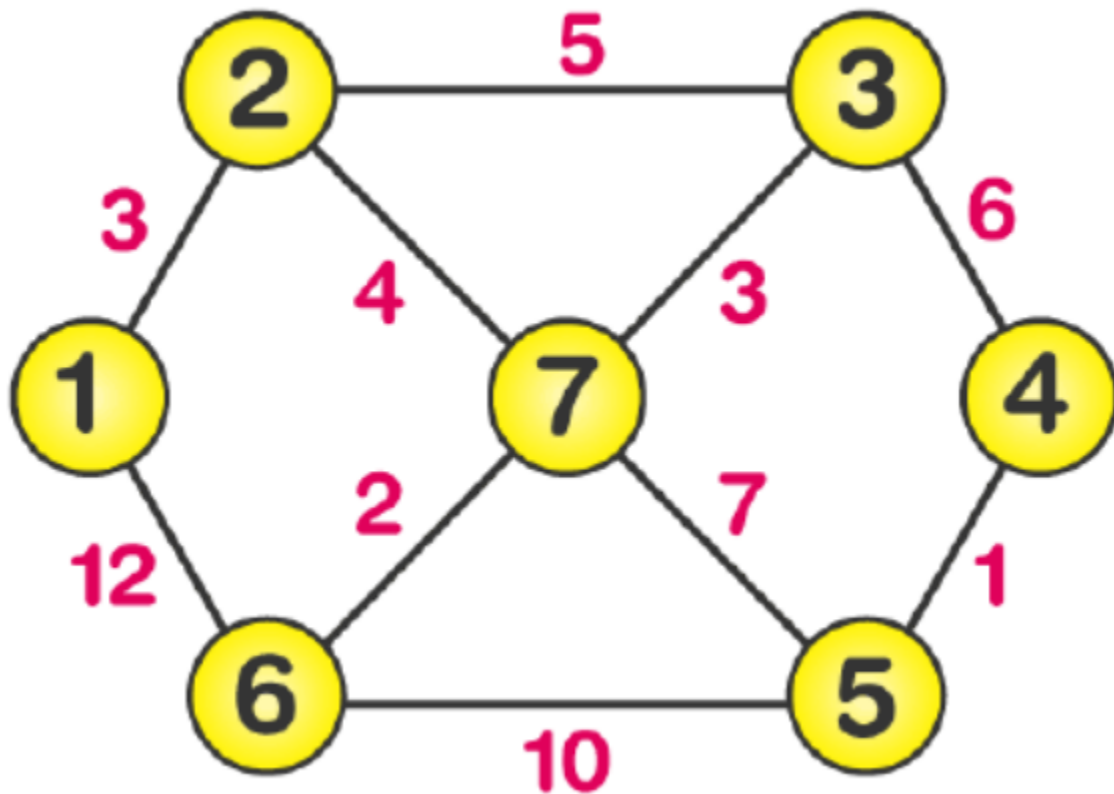
"""
text_file=open("Dijkstra.c","w")
text_file.write(code)
text_file.close()
!gcc -o Dijkstra -fopenmp Dijkstra.c
```

Name : Manthan Gopal Dhole
Id : 201080020

TY IT
Parallel Computing

```
!./Dijkstra
```

Input Graph :



```
0 3 0 0 0 12 0
3 0 5 0 0 0 4
0 5 0 6 0 0 4
0 0 6 0 1 0 0
0 0 0 1 0 10 7
12 0 0 0 10 0 2
0 4 3 0 7 2 0
```

Name : Manthan Gopal Dhole
Id : 201080020

TY IT
Parallel Computing

Output :

```
!./Dijkstra
Enter the number of vertices: 7
Enter the adjacency matrix:
0 3 0 0 0 12 0 3 0 5 0 0 0 4 0 5 0 6 0 0 4 0 0 6 0 1 0 0 0 0 0 1 0 10 7 12 0 0 0 10 0 2 0 4 3 0 7 2 0
Enter the starting vertex: 0
Vertex    Distance from Start
0          0
1          3
2          8
3         14
4         14
5          9
6          7
```

Conclusion:

Overall, using OpenMP can greatly speed up Dijkstra's algorithm, especially for large graphs, by taking advantage of multiple processors in a shared-memory system.

Name : Manthan Gopal Dhole
Id : 201080020

TY IT
Parallel Computing

Link : [Click IT](#)

Experiment 6

Aim: Implement Parallel Traveling Salesman Problem using OpenMP.

Theory :

The Traveling Salesman Problem (TSP) is a well-known optimization problem in which a salesman has to visit a given set of cities exactly once and return to his starting city, while minimizing the total distance traveled. The Parallel TSP (PTSP) is the TSP problem solved in parallel by dividing the search space among different threads.

OpenMP is a popular parallel programming model that allows for easy implementation of parallel algorithms on shared-memory architectures. OpenMP uses a fork-join model, in which a team of threads is created and work is divided among them.

To implement PTSP using OpenMP, we can follow these steps:

Create a list of cities and their coordinates.

Divide the list of cities among the available threads.

Each thread will independently search for the shortest path that visits all the cities assigned to it.

Merge the solutions of all the threads to find the overall shortest path.

The Traveling Salesman Problem (TSP) is a well-known problem in computer science and operations research. It involves finding the shortest possible route that visits a set of cities and returns to the starting city. The problem becomes more complicated when there are multiple salesmen involved, and each salesman has to visit a subset of the cities.

Name : Manthan Gopal Dhole
Id : 201080020

TY IT
Parallel Computing

This problem is known as the Parallel Traveling Salesman Problem (PTSP). In the PTSP, a set of salesmen is given, and each salesman has to visit a subset of the cities. The goal is to find the shortest possible route for each salesman such that all the cities are visited exactly once.

OpenMP is a popular framework for parallel programming in shared memory architectures. It provides a set of directives and libraries that allow developers to parallelize their code easily. In the context of the PTSP, OpenMP can be used to parallelize the computation of the shortest route for each salesman.

The basic idea is to divide the set of salesmen into multiple subsets and assign each subset to a different thread. Each thread will then compute the shortest route for the salesmen in its subset. Once all the threads have finished, the computed routes can be combined to obtain the final solution.

The parallelization of the PTSP using OpenMP can be done in several ways. One approach is to use a shared memory model, where all the threads have access to a shared memory pool. The shared memory can be used to store the distance matrix, the current solution, and any intermediate results. The threads can then synchronize their access to the shared memory to avoid race conditions.

Another approach is to use a distributed memory model, where each thread has its own memory space. In this approach, the distance matrix is divided into smaller sub-matrices, and each thread is assigned a subset of the sub-matrices to compute. Once all the threads have finished, the results can be combined to obtain the final solution.

Code :

```
code = ""  
#include <stdio.h>  
#include <stdlib.h>  
#include <limits.h>  
#include <omp.h>  
  
#define MAX_N 16  
  
int n;  
int dist[MAX_N][MAX_N];
```

Name : Manthan Gopal Dhole
Id : 201080020

TY IT
Parallel Computing

```
int visited[MAX_N];
int curr_path[MAX_N];
int min_path[MAX_N];
int min_cost = INT_MAX;

void tsp(int curr_cost, int curr_pos, int level)
{
    if (level == n) {
        // Visited all cities
        curr_cost += dist[curr_pos][0];
        if (curr_cost < min_cost) {
            min_cost = curr_cost;
            #pragma omp critical
            {
                for (int i = 0; i < n; i++) {
                    min_path[i] = curr_path[i];
                }
            }
        }
        return;
    }
    for (int i = 1; i < n; i++) {
        if (!visited[i]) {
            visited[i] = 1;
            curr_path[level] = i;
            tsp(curr_cost + dist[curr_pos][i], i, level + 1);
            visited[i] = 0;
        }
    }
}

int main()
{
    printf("Enter the number of cities: ");
    scanf("%d", &n);

    printf("Enter the distances between the cities:\\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
```

Name : Manthan Gopal Dhole
Id : 201080020

TY IT
Parallel Computing

```
        scanf("%d", &dist[i][j]);  
    }  
}  
  
visited[0] = 1;  
curr_path[0] = 0;  
tsp(0, 0, 1);  
  
printf("Path: %d", min_path[0] + 1);  
for (int i = 1; i < n; i++) {  
    printf("->%d", min_path[i] + 1);  
}  
printf("->%d\\n", min_path[0] + 1);  
printf("Minimum Cost/Minimum weight Hamiltonian Cycle: %d\\n",  
min_cost);  
  
return 0;  
}  
  
"""
```

Output :

```
!./Travelling_salesman  
Enter the number of cities: 3  
Enter the distances between the cities:  
3  
5  
2  
55  
6  
2  
66  
2  
4  
Path: 1->3->2->1  
Minimum Cost/Minimum weight Hamiltonian Cycle: 59
```

Name : Manthan Gopal Dhole
Id : 201080020

TY IT
Parallel Computing

Conclusion:

In summary, OpenMP can be used to parallelize the PTSP by dividing the set of salesmen into multiple subsets and assigning each subset to a different thread. The threads can then compute the shortest route for the salesmen in their subset and synchronize their results to obtain the final solution. The choice of shared or distributed memory model depends on the specific requirements of the problem and the available resources.

Name: Manthan Gopal Dhole

ID : 201080020

Sub : Parallel Computing

Parallel Computing experiment 7

Aim:

To implement following using MPI.

- i. calculating Rank and Number of processor.**
- ii. Pi calculation.**
- iii. advanced MPI program that has a total number of 4 processes, where the process with rank = 0 should send VJTI letter to all the processes using MPI_Scatter call.**
- iv. find the maximum value in array of six integers with 6 processes, and print the result in root process using MPI_Reduce call.**
- v. Ring topology.**

Theory:

MPI

MPI (Message Passing Interface) is a standardized communication protocol that allows multiple processes running on different computing nodes to exchange messages and data. It is commonly used in parallel and distributed computing to implement communication between processes, especially in high-performance computing (HPC) applications.

MPI provides a set of functions and data types that enable processes to send and receive messages with each other, synchronize their activities, and coordinate their computations. It supports various communication modes, such as point-to-point, collective, and one-sided communication, and can be used with different programming languages, including C, C++, Fortran, and Python.

MPI has several implementations available, including MPICH, Open MPI, and Intel MPI, and is widely used in scientific, engineering, and data analysis applications that require parallel processing on multiple processors or computing clusters.

i. calculating Rank and Number of processor.

In MPI, each process is assigned a unique identifier called rank, which ranges from 0 to the total number of processes running in the MPI job. The rank is used to distinguish one process from another and to determine which process should receive or send messages.

To obtain the rank of the current process and the total number of processes in the MPI job, you can use the `MPI_Comm_rank()` and `MPI_Comm_size()` functions, respectively.

ii. Pi calculation.

Calculating Pi using MPI is a common parallel programming exercise that demonstrates the use of MPI for distributing computation across multiple processes.

Our program calculates an approximation of pi using the Monte Carlo method. The program distributes the computation of the sum of a series across multiple MPI processes and then combines the partial results to obtain the final value of pi.

iii. advanced MPI program that has a total number of 4 processes, where the process with rank = 0 should send VJTI letter to all the processes using `MPI_Scatter` call.

In Our program, the message "VJTI" is stored in the msg array in process 0. The MPI_Scatter call is used to send a part of this message to each of the other processes. The message is split into 4 character chunks and each process receives one chunk. The message is received into the same msg array to simplify the code, but in a real program you would probably use a separate receive buffer.

When each process receives its part of the message, it prints a message to the console indicating the rank of the process and the part of the message it received.

iv. find the maximum value in array of six integers with 6 processes, and print the result in root process using MPI_Reduce call.

In our program, each process initializes a local variable local_max with the value of the corresponding element in the array. The MPI_Reduce call is used to find the maximum value across all processes using the MPI_MAX operation. The result is stored in the max_value variable on the root process. Finally, the root process prints the maximum value.

Note that in a real program, the array would probably be distributed among the processes using MPI_Scatter or a similar function, but for simplicity we've just used a hardcoded array here.

v. Ring topology.

In this program, each process sends a message to its right neighbor and receives a message from its left neighbor. The rank of each neighbor is calculated using the modulo operator (%) to ensure that the indices wrap around in a circular fashion. The MPI_Send and MPI_Recv functions are used for communication, with the appropriate ranks of the neighbors passed as arguments. Finally, each process prints the message it received.

Note that in a real program, you would typically use a loop to pass the message around the ring multiple times, or to pass different messages with different tags or data types.

Code link:

<https://colab.research.google.com/drive/1J3EFKJCQBw6DFLLeguWPpRbkNnbLrgEH?usp=sharing>

Code and Output:

i. calculating Rank and Number of processor.

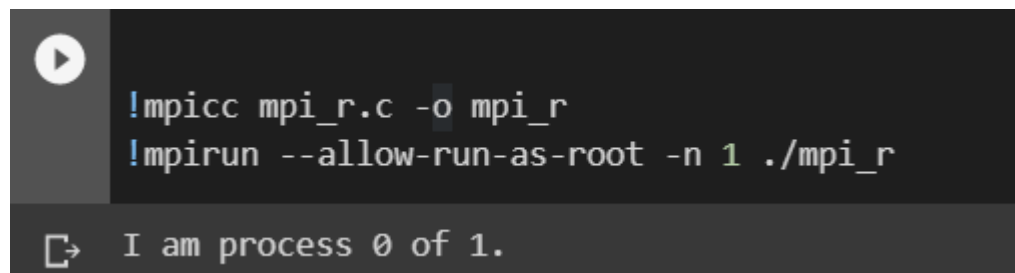
```
%%writefile mpi_r.c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
    int rank, size;

    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    printf("I am process %d of %d.\n", rank, size);

    MPI_Finalize();
    return 0;
}
```



```
!mpicc mpi_r.c -o mpi_r
!mpirun --allow-run-as-root -n 1 ./mpi_r

I am process 0 of 1.
```

ii. Pi calculation.

```
%%writefile mpi_pi.c
#include <mpi.h>
#include <stdio.h>
#include <math.h>

int main(int argc, char** argv) {
    int rank, size;
```

```

long long int n = 1000000;
double x, pi, sum = 0.0;
int i;

MPI_Init(NULL, NULL);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

for (i = rank; i < n; i += size) {
    x = (double)(i + 0.5) / (double)n;
    sum += 4.0 / (1.0 + x * x);
}

double local_pi = sum / n;
MPI_Reduce(&local_pi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD)
;

if (rank == 0) {
    printf("pi is approximately %.16f.\n", pi);
}

MPI_Finalize();
return 0;
}

```

```

[ ] !mpicc mpi_pi.c -o mpi_pi
    !mpirun --allow-run-as-root -np 1 ./mpi_pi

```

```

pi is approximately 3.1415926535897643.

```

iii. advanced MPI program that has a total number of 4 processes, where the process with rank = 0 should send VJTI letter to all the processes using MPI_Scatter call.

```

%%writefile mpi_vjti.c
#include <mpi.h>
#include <stdio.h>
#include <string.h>

#define MESSAGE_SIZE 5

```

```

int main(int argc, char** argv) {
    int rank, size;
    char message[MESSAGE_SIZE + 1];

    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (rank == 0) {
        strcpy(message, "VJTI");
    }

    MPI_Scatter(message, MESSAGE_SIZE, MPI_CHAR, message, MESSAGE_SIZE, MPI_COMM_WORLD);

    printf("Process %d received message: %s\n", rank, message);

    MPI_Finalize();
    return 0;
}

```

```

!mpicc mpi_vjti.c -o mpi_vjti
!mpirun --allow-run-as-root -np 1 ./mpi_vjti

Process 0 received message: VJTI

```

iv. find the maximum value in array of six integers with 6 processes, and print the result in root process using MPI_Reduce call.

```

%%writefile mpi_maximum.c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
    int rank, size, max_value = 0;
    const int root = 0;
    int arr[6] = {5, 10, 7, 3, 8, 1};
    int local_max = arr[rank];

    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

```

```

MPI_Comm_size(MPI_COMM_WORLD, &size);

MPI_Reduce(&local_max, &max_value, 1, MPI_INT, MPI_MAX, root, MPI_COMM_WORLD);

if (rank == root) {
    printf("The maximum value is %d.\n", max_value);
}

MPI_Finalize();
return 0;
}

```



```

!mpicc mpi_maximum.c -o mpi_maximum
!mpirun --allow-run-as-root -np 1 ./mpi_maximum

```



The maximum value in the array is: 1

v. Ring topology.

```

%%writefile mpi_ring.c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
    int rank, size;
    int message_in, message_out;

    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (rank == 0) {
        message_out = 1;
        MPI_Send(&message_out, 1, MPI_INT, (rank + 1) % size, 0, MPI_COMM_WORLD);
        MPI_Recv(&message_in, 1, MPI_INT, size - 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process %d received message %d from process %d.\n", rank, message_in, size - 1);
    } else {

```

```
MPI_Recv(&message_in, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD, MPI_
STATUS_IGNORE);
printf("Process %d received message %d from process %d.\n", rank, m
essage_in, rank - 1);
if (rank == size - 1) {
    message_out = 0;
} else {
    message_out = message_in + 1;
}
MPI_Send(&message_out, 1, MPI_INT, (rank + 1) % size, 0, MPI_COMM_W
ORLD);
}

MPI_Finalize();
return 0;
}
```



```
!mpicc mpi_ring.c -o mpi_ring
!mpirun --allow-run-as-root -np 1 ./mpi_ring
```



```
Process 0 received message 1 from process 0.
```

Conclusion:

We have Implemented some operations in MPI. In this experiment, we have learned deeply about the Message Passing Interface. MPI programs typically divide the problem into smaller subproblems, which are solved concurrently by different processes.