LABORATORY MANUAL

LABORATORY PRACTICE - I

BE-COMP

SEMESTER-I

TEACHING SCHEME	EXAMINATION SCHEME	
Lectures: 3 Hrs/Week	Theory:	100 Marks
Practical: 4 Hrs/Week	Practical:	50 Marks
	TW:	50 Marks

DEPARTMENT OF COMPUTER ENGINEERING

DHOLE PATIL COLLEGE OF ENGINEERING, PUNE

2018-2019

Sr. No.	Name of Experiment
	Group A
1	a) Implement Parallel Reduction using Min, Max, Sum and Average operations. b) Write a CUDA program that, given an N-element vector,find- □ The maximum element in the vector □ The minimum element in the vector □ The arithmetic mean of the vector □ The standard deviation of the values in the vector Test for input N and generate a randomized vector V of length N (N should be large). The program should generate output as the two computed maximum values as well as the time taken to find each value.
2	**Vector and Matrix Operations-** Design parallel algorithm to 1. Add two large vectors 2. Multiply Vector and Matrix 3. Multiply two N × N arrays using n2 processors
3	**Parallel Sorting Algorithms-** For Bubble Sort and Merger Sort, based on existing sequential algorithms, design and implement parallel algorithm utilizing all resources available.
4	**Parallel Search Algorithm-** Design and implement parallel algorithm utilizing all resources available. for □ **Binary Search for Sorted Array** □ **Depth-First Search** (tree or an undirected graph) **OR** □ **Breadth-First Search** (tree or an undirected graph) **OR** □ **Best-First Search that** (traversal of graph to reach a target in the shortest possible path)
	Group B
5	Implement Tic-Tac-Toe using A* algorithm
6	Implement 3 missionaries and 3 cannibals problem depicting appropriate graph. Use A* algorithm.
7	Solve 8-puzzle problem using A* algorithm. Assume any initial configuration and define goal configuration clearly
8	Use Heuristic Search Techniques to Implement Hill-Climbing Algorithm.
	Group C
9	Download the Iris flower dataset or any other dataset into a DataFrame. (eg https://archive.ics.uci.edu/ml/datasets/Iris) Use Python/R and Perform following – How many features are there and what are their types (e.g., numeric, nominal)? □ Compute and display summary statistics for each feature available in the dataset. (eg. minimum value, maximum value, mean, range, standard deviation, variance and percentiles □ Data Visualization-Create a histogram for each feature in the

	dataset to illustrate the feature distributions. Plot each histogram. ☐ Create a boxplot for each feature in the dataset. All of the boxplots should be combined into a single plot. Compare distributions and identify outliers.
10	Download Pima Indians Diabetes dataset. Use Naive Bayes" Algorithm for classification ☐ Load the data from CSV file and split it into training and test datasets. ☐ Summarize the properties in the training dataset so that we can calculate probabilities and make predictions. ☐ Classify samples from a test dataset and a summarized training dataset.
11	Trip History Analysis: Use trip history dataset that is from a bike sharing service in the United States. The data is provided quarter-wise from 2010 (Q4) onwards. Each file has 7 columns. Predict the class of user. Sample Test data set available here https://www.capitalbikeshare.com/trip-history-data
12	Twitter Data Analysis: Use Twitter data for sentiment analysis. The dataset is 3MB in size and has 31,962 tweets. **Identify the tweets which are hate tweets and which are not.** Sample Test data set available here https://datahack.analyticsvidhya.com/contest/practice-problem-twitter-sentiment-analysis/

GROUP A: ASSIGNMENTS

Experiment No: 01a

Aim: Implement parallel reduction using Min, Max, Sum and Average Operations.

- Minimum
- Maximum
- Sum
- Average

Objective: To study and implementation of directive based parallel programming model.

Outcome: Students will be understand the implementation of sequential program augmente with compiler directives to specify parallelism.

Pre-requisites:

64-bit Open source Linux or its derivative

Programming Languages: C/C++

Theory:

OpenMP:

OpenMP is a set of C/C++ programs (or FORTRAN equivalents) which provide the programmer a high-level front-end interface which get translated as calls to threads (or other similar entities). The key phrase here is "higher-level"; the goal is to better enable the programmer to "think parallel," alleviating him/her of the burden and distraction of dealing with setting up and coordinating threads. For example, the OpenMP directive. OpenMP Core Syntax:

Most of the constructs in OpenMP are compiler directives:

#pragma omp construct [clause [clause]...]

Example

```
#pragma omp parallel num_threads(4)
```

Function prototypes and types in the file:

#include

<omp.h>

Most OpenMP constructs apply to a "structured block"

Structured block:

a block of one or more statements surrounded by "{ }", with one point of entry at the top and one point of exit at the bottom.

Following is the sample code which illustrates max operator usage in OpenMP :

```
#include <stdio.h>
#include <omp.h>
int main()
{
   double arr[10];
   omp_set_num_threads(4);
   double max_val=0.0;
   int i;
   for( i=0; i<10; i++)
      arr[i] = 2.0 + i;
   #pragma omp parallel for reduction(max : max_val)
   for( i=0;i<10; i++)
   {
      printf("thread id = %d and i = %d", omp_get_thread_num(),i);
      if(arr[i] > max_val)
      {
         max_val = arr[i];
      }
   }
   printf("\nmax_val = %f", max_val);
}
```

Following is the sample code which illustrates min operator usage in OpenMP :

```
#include <stdio.h>
#include <omp.h>
int main()
{
   double arr[10];
   omp_set_num_threads(4);
   double min_val=0.0;
      int i;
      for( i=0; i<10; i++)
```

```
            arr[i] = 2.0 + i;
          #pragma omp parallel for reduction(min : min_val)
    for( i=0;i<10; i++)
    {
      printf("thread id = %d and i = %d", omp_get_thread_num(), i);
      if(arr[i] < min_val)
      {
        min_val = arr[i];
      }
    }
    printf("\nmin_val = %f", min_val);
}
```

Following is the sample code which illustrates sum operation usage in OpenMP :

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

int main (int argc, char *argv[])
{
int  i, n;
float a[100], b[100], sum;
/* Some initializations */
n = 100;
for (i=0; i < n; i++)
 a[i] = b[i] = i * 1.0;
sum = 0.0;

#pragma omp parallel for reduction(+:sum)
 for (i=0; i < n; i++)
   sum = sum + (a[i] * b[i]);
printf("   Sum = %f\n",sum);
}
```

Following is the sample code which illustrates sum operation usage in OpenMP :

```
#include<iostream>
#include<omp.h>//header file for openmp
using namespace std;
main()
{
	int arr[5]={1,2,3,4,5},i;
	float avg=0;
	#pragma omp parallel
	{
		int id=omp_get_thread_num();//id will tell us which thread is running the addition
		#pragma omp for //used for running the loop parallelly
		for(i=0;i<5;i++)
		{
			avg+=arr[i];//summation
			cout<<"for i= "<<i<<" thread "<<id<<" is executing"<<endl;
		}
	}
	avg/=5;
	cout<<"Output "<<avg<<endl;
}
```

Conclusion: We have implemented parallel reduction using Min, Max, Sum and Average Operations.

Experiment No: 01 b

Aim: To write a CUDA program that, given an N-element vector, find.

- Minimum element in vector
- Maximum element in vector
- Arithmetic mean of the vector
- Standard deviation of the values in the vector

Objective: To study and implement the operations on vector, generate o/p as two computed max values as well as time taken to find each value.

Outcome: Students will be understand the implementation of operations on vector, generate o/p as two computed max with respect to time.

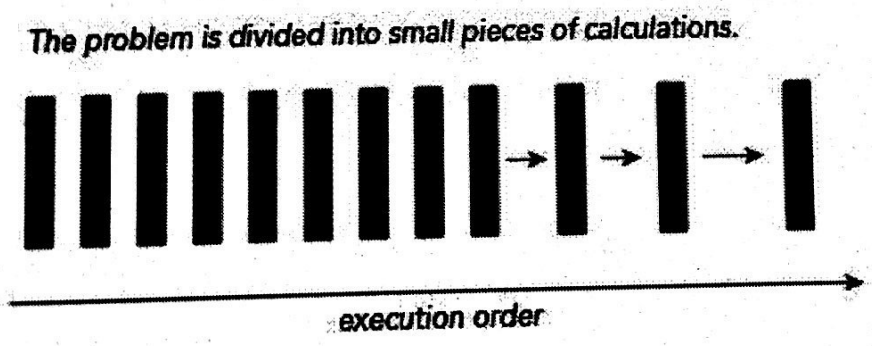
Pre-requisites:

64-bit Open source Linux or its derivative
Programming Languages: C/C++,CUDA

Theory:

Sequential Programming:

When solving a problem with a computer program, it is natural to divide the problem into discrete series of calculations; each calculation performs a specified task, as shown in following Figure .Such a pro-gram is called a sequential program.



Parallel Programming:

There are two fundamental types of parallelism in applications:

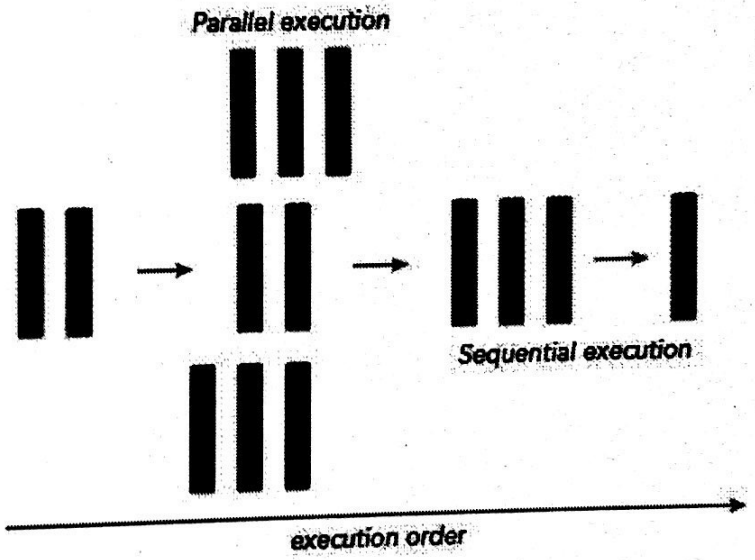
➢ Task parallelism

➢ Data parallelism

Task parallelism arises when there are many tasks or functions that can be operated independently and largely in parallel. Task parallelism focuses on distributing functions across multiple cores.

Data parallelism arises when there are many data items that can be operated on at the same time.

Data parallelism focuses on distributing the data across multiple cores.



CUDA :

CUDA programming is especially well-suited to address problems that can be expressed as data- parallel computations. Any applications that process large data sets can use a data-parallel model to speed up the computations. Data-parallel processing maps data elements to parallel threads.

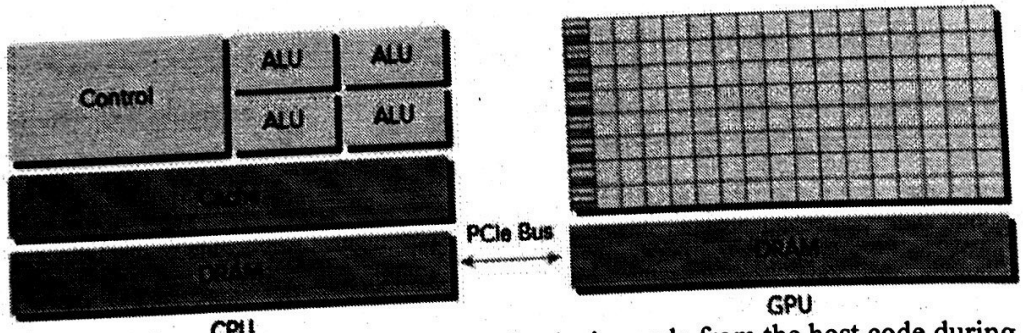
The first step in designing a data parallel program is to partition data across threads, with each thread working on a portion of the data.

CUDA Architecture:

A heterogeneous application consists of two parts:

- Host code
- Device code

Host code runs on CPUs and device code runs on GPUs. An application executing on a heterogeneous platform is typically initialized by the CPU. The CPU code is responsible for managing the environment, code, and data for the device before loading compute-intensive tasks on the device. With computational intensive applications, program sections often exhibit a rich amount of data parallelism. GPUs are used to accelerate the execution of this portion of data parallelism. When a hardware component that is physically separate from the CPU is used to accelerate computationally intensive sections of an application, it is referred to as a hardware accelerator. GPUs are arguably the most common example of a hardware accelerator. GPUs must operate in conjunction with a CPU-based host through a PCI-Express bus, as shown in Figure.



NVIDIA's CUDA nvcc compiler separates the device code from the host code during the compilation process. The device code is written using CUDA C extended with keywords for labeling data-parallel functions, called kernels . The device code is further compiled by

Nvcc . During the link stage, CUDA runtime libraries are added for kernel procedure calls and explicit GPU device manipulation. Further kernel function, named helloFromGPU, to print the string of "Hello World from GPU!" as follows:

```
__global__ void helloFromGPU(void)
{
printf("Hello World from GPU!\n");
}
```

The qualifier __global__tells the compiler that the function will be called from the CPU and executed on the GPU. Launch the kernel function with the following code:

```
helloFromGPU <<<1,10>>>();
```

Triple angle brackets mark a call from the host thread to the code on the device side. A kernel is executed by an array of threads and all threads run the same code. The parameters within the triple angle brackets are the execution configuration, which specifies how many threads will execute the kernel. In this example, you will run 10 GPU threads.

A typical processing flow of a CUDA program follows this pattern:

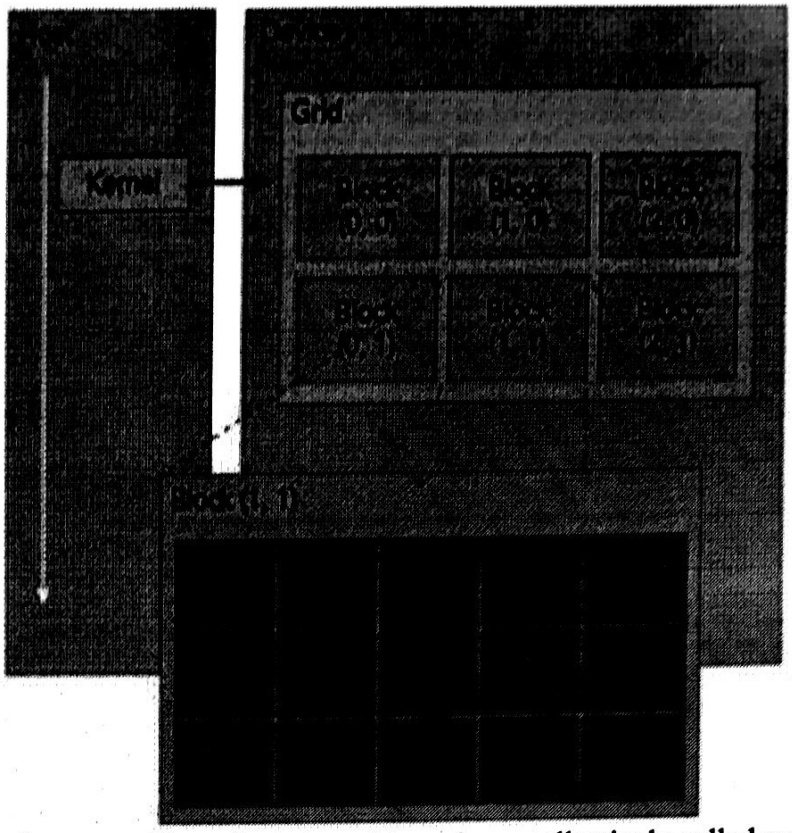
1. Copy data from CPU memory to GPU memory.
2. Invoke kernels to operate on the data stored in GPU memory.
3. Copy data back from GPU memory to CPU memory

Table lists the standard C functions and their corresponding CUDA C functions for memor operations. Host and Device Memory Functions are follows.

STANDARD C FUNCTIONS	CUDA C FUNCTIONS
malloc	cudaMalloc
memcpy	cudaMemcpy
memset	cudaMemset
free	cudaFree

Organizing Threads:

When a kernel function is launched from the host side, execution is moved to a device wher a large number of threads are generated and each thread executes the statements specified b the kernel function. The two-level thread hierarchy decomposed into blocks of threads an grids of blocks, as shown in following figure:



All threads spawned by a single kernel launch are collectively called a grid . All threads in a grid
share the same global memory space. A grid is made up of many thread blocks. A thread block is a group of threads that can cooperate with each other using:

➢ Block-local synchronization

➢ Block-local shared memory

Threads from different blocks cannot cooperate.
Threads rely on the following two unique coordinates to distinguish themselves from each other:

➢ blockIdx (block index within a grid)

➢ threadIdx (thread index within a block)

These variables appear as built-in, pre-initialized variables that can be accessed withi kernel functions. When a kernel function is executed, the coordinate variable blockIdx and

threadIdx are assigned to each thread by the CUDA runtime. Based on the coordinates, yo can assign portions of data to different threads. It is a structure containing three unsigne integers, and the 1st, 2nd, and 3rd components are accessible through the fields x, y, and respectively.

blockIdx.x
blockIdx.y
blockIdx.z

threadIdx.x
threadIdx.y
threadIdx.z

CUDA organizes grids and blocks in three dimensions. The dimensions of a grid and a bloc are specified by the following two built-in variables:

➢blockDim (block dimension, measured in threads)

➢gridDim (grid dimension, measured in blocks)

These variables are of type dim3, that is used to specify dimensions. When defining variable of type dim3, any component left unspecified is initialized to 1.Each component in variable of type dim3 is accessible through its x,y,, and z fields, respectively, as shown in th following example:

blockDim.x
blockDim.y
blockDim.

CUDA program for calculating Min for given N-element vector

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

#define SIZE 100

__global__ void min(int *a , int *c)// kernel function definition


{
int i = threadIdx.x;                    // initialize i to thread ID

*c = a[55];

        if(a[i] < *c)
                {
                *c = a[i];
                }
```

```
}

int main()
{
int i;
srand(time(NULL)); //makes use of the computer's internal clock to control the choice of the seed

int a[SIZE];
int c;

int *dev_a, *dev_c;                    //GPU / device parameters

cudaMalloc((void **) &dev_a, SIZE*sizeof(int)); //assign memory to parameters on GPU from CUDA runtir
//API

cudaMalloc((void **) &dev_c, SIZE*sizeof(int));


for( i = 0 ; i < SIZE ; i++)
{
  a[i] = rand();              // input the numbers
}
for( i = 0 ; i < SIZE ; i++)
{
  printf("%d", a[i]);                  // input the numbers
```

```
    }
cudaMemcpy(dev_a , a, SIZE*sizeof(int),cudaMemcpyHostToDevice); //copy the array from CPU
to GPU
    min<<<1,SIZE>>>(dev_a,dev_c);
                        // call kernel function <<<number of blocks, number of
threads
    cudaMemcpy(&c, dev_c, SIZE*sizeof(int),cudaMemcpyDeviceToHost);
      //    copy the result back from GPU to CPU

       printf("\nmin = %d ",c);

    cudaFree(dev_a);                        // Free the allocated memory
    cudaFree(dev_c);
    printf("");

    return 0;
      }
```

CUDA program for calculating Max for given N-element vector

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

      #define SIZE 1000

                  __global__ void max(int *a , int *c)          // kernel function definition
      {
      int i = threadIdx.x;                    // initialize i to thread ID

      *c = a[0];

            if(a[i] > *c)
                  {
                  *c = a[i];
                  }

      }

      int main()
      {
      int i;
      srand(time(NULL)); //makes use of the computer's internal clock to control the choice of the seed

      int a[SIZE];
      int c;

      int *dev_a, *dev_c;           //GPU / device parameters
                                                                        //assign memory
      cudaMalloc((void **) &dev_a, SIZE*sizeof(int));
//to parameters on GPU
      cudaMalloc((void **) &dev_c, SIZE*sizeof(int));
```

```
for( i = 0 ; i < SIZE ; i++)
{
a[i] = i; // rand()% 1000 + 1;                    // input the numbers
}

cudaMemcpy(dev_a , a, SIZE*sizeof(int),cudaMemcpyHostToDevice);

//copy the array from CPU to GPU

max<<<1,SIZE>>>(dev_a,dev_c); // call kernel function <<<number of blocks, number of threads

cudaMemcpy(&c, dev_c, SIZE*sizeof(int),cudaMemcpyDeviceToHost);

//   copy the result back from GPU to CPU

printf("\nmax = %d ",c);


cudaFree(dev_a);                    // Free the allocated memory
cudaFree(dev_c);
printf("");

return 0;
}
```

CUDA program for calculating standard deviation for given N-element vector

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

//   CUDA kernel. Each thread takes care of one element of c
__global__ void vecAdd(int *a, int *c, int n)
{
      // Get our global thread ID
      int id = blockIdx.x*blockDim.x+threadIdx.x;
    //   c[id]=0;
      //   Make sure we do not go out of bounds if (id < n)
           *c+= a[id];
    //   printf("\n%d", c[id]);
}

int main( int argc, char* argv[] )
{
     //   Size of vectors
     // int n = 100000;

        int n=5;
const int size = n * sizeof(int);
    //   Host input vectors
     int *h_a;
  //   double *h_b; //Host output vector
    int *h_c;

   //   Device input vectors
   int *d_a;
   //double *d_b; //Device output
   //vector
   int *d_c;
   int dev=0;
```

```
cudaMemcpy ( device = 1

    //    Size, in bytes, of each vector
    size_t bytes = n*sizeof(double);

    //    Allocate memory for each vector on host
    //    h_a = (int*)malloc(bytes);
    //h_b = (double*)malloc(bytes);
    h_c = (int*)malloc(bytes);

    //    Allocate memory for each vector on GPU
    cudaMalloc(&d_a, bytes);
      //cudaMalloc(&d_b, bytes);
       cudaMalloc(&d_c, bytes);

       int i;
       printf("Input array");
       //    Initialize vectors on host
       /*for( i = 0; i < n; i++ ) {
              //    h_a[i] = sin(i)*sin(i);
              //printf("\n",i);
          h_a[i]=i;
          //printf("\n%d", h_a[i]);
          //h_b[i]=i;
             //h_b[i] = cos(i)*cos(i);
       }*/

      int a[]= {0, 1, 2, 3, 4};

      cudaMalloc(&h_a, size);

       // Copy host vectors to device
       cudaMemcpy( h_a, a, bytes, cudaMemcpyHostToDevice);
       cudaMemcpy( d_c, &dev, sizeof(int), cudaMemcpyHostToDevice);
//     cudaMemcpy( d_b, h_b, bytes, cudaMemcpyHostToDevice);

       int blockSize, gridSize;

       //    Number of threads in each thread block
       blockSize = 2;

       //    Number of thread blocks in grid
       gridSize = (int)ceil((float)n/blockSize);

       //    Execute the kernel
       vecAdd<<<gridSize, blockSize>>>(d_a,d_c,n);
       int result;
       // Copy array back to host
       cudaMemcpy( &result,d_c, sizeof(int), cudaMemcpyDeviceToHost );

       //    Sum up vector c and print result divided by n, this should equal 1 within error
       double sum = 0;
       //for(i=0; i<n; i++)
          //       sum += h_c[i];

       printf("final result: %f\n",result );

//     vecdev<<<gridSize, blockSize>>>(d_a,d_c, n);

       //    Release device memory
       cudaFree(d_a);
       //cudaFree(d_b);
       cudaFree(d_c);
```

```
        // Release host memory
        free(h_a);

    //free(h_b);
    free(h_c);

    return 0;
}
```

CUDA program for calculating arithmetic mean for given N-element vector

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

// CUDA kernel. Each thread takes care of one element of c
__global__ void vecAdd(double *a, double *b, double *c, int n)
{
    // Get our global thread ID
    int id = blockIdx.x*blockDim.x+threadIdx.x; // get global index


    //    Make sure we do not go out of bounds
     if (id < n)
          c[id] = a[id] + b[id];
}

int main( )
{

    //int n = 100000;
       int n=5;                          // Size of vectors


    double      *h_a;           // Host input vector
    double *h_b;                // Host input vector


    double *h_c;                //Host output vector


    double      *d_a;           // Device input vector
    double *d_b;                // Device input vector

    double *d_c;                 //Device output vector


    size_t bytes = n*sizeof(double);
            // Size, in bytes, of
```

```
//each vector

    //    Allocate memory for each vector on host
    h_a = (double*)malloc(bytes);
    h_b = (double*)malloc(bytes);
     h_c = (double*)malloc(bytes);

    //    Allocate memory for each vector on GPU
    cudaMalloc(&d_a, bytes);


        cudaMalloc(&d_b, bytes);
        cudaMalloc(&d_c, bytes);

        int i;

        //    Initialize vectors on host
         for( i = 0; i < n; i++ ) {

            h_a[i]=i;
            h_b[i]=i;

        }

        // Copy host vectors to device
        cudaMemcpy( d_a, h_a, bytes, cudaMemcpyHostToDevice);
         cudaMemcpy( d_b, h_b, bytes, cudaMemcpyHostToDevice);

        int blockSize, gridSize;

        //    Number of threads in each thread block
        blockSize = 1024;

        //    Number of thread blocks in grid
        gridSize = (int)ceil((float)n/blockSize);

        //    Execute the kernel
        vecAdd<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);

        // Copy array back to host
        cudaMemcpy( h_c, d_c, bytes, cudaMemcpyDeviceToHost );

        //    Sum up vector c and print result divided by n, this should equal 1 within error
        double sum = 0;
         for(i=0; i<n; i++)
               sum += h_c[i];
        printf("Average mean of 2 vectors: %f\n", sum/n);

        //    Release device memory
        cudaFree(d_a);
        cudaFree(d_b);
        cudaFree(d_c);

        //    Release host memory
        free(h_a); free(h_b); free(h_c);
```

```
    return 0;
}
```

Conclusion: We have implemented CUDA program for calculating Min, Max, Arithmetic mean and Stan deviation Operations on N-element vector.

Experiment No: 2

Title: **Vector and Matrix Operations-**
Design parallel algorithm to
1. Add two large vectors
2. Multiply Vector and Matrix
3. Multiply two N × N arrays using n^2 processors

Aim: Implement *nxn* matrix parallel addition, multiplication using CUDA, use shared memory.

Prerequisites:

- Concept of matrix addition, multiplication.
- Basics of CUDA programming

Objectives:

Student should be able to learn parallel programming, CUDA architecture and CUDA processing flow

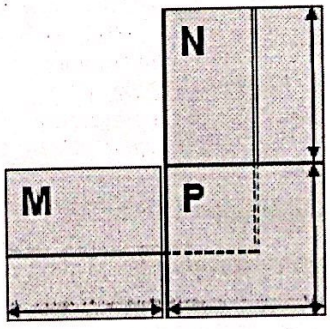
Theory:

A straightforward matrix multiplication example that illustrates the basic features of memory and thread management in CUDA programs

- Leave shared memory usage until later
- Local, register usage
- Thread ID usage

- *P = M * N of size WIDTH x WIDTH*

- One thread handles one element of P
- M and N are loaded WIDTH times from global memory



Matrix Multiplication steps

1. **Matrix Data Transfers**
2. **Simple Host Code in C**

3. **Host-side Main Program Code**
4. **Device-side Kernel Function**
5. **Some Loose Ends**

Step 1: Matrix Data Transfers

```
// Allocate the device memory where we will copy M to Matrix
Md;
Md.width = WIDTH;
Md.height = WIDTH;
Md.pitch = WIDTH;
int size = WIDTH * WIDTH * sizeof(float);
cudaMalloc((void**)&Md.elements, size);
// Copy M from the host to the device
cudaMemcpy(Md.elements, M.elements, size, cudaMemcpyHostToDevice);
// Read M from the device to the host into P
cudaMemcpy(P.elements, Md.elements, size, cudaMemcpyDeviceToHost);
...
// Free device memory
cudaFree(Md.elements);
```

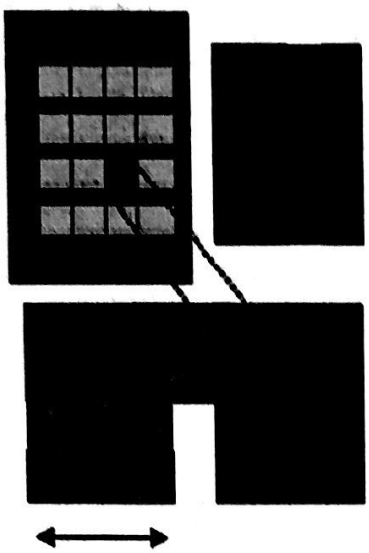
Step 2: Simple Host Code in C

```
// Matrix multiplication on the (CPU) host in double precision
// for simplicity, we will assume that all dimensions are equal
void MatrixMulOnHost(const Matrix M, const Matrix N, Matrix P)
{
    for (int i = 0; i < M.height; ++i)
        for (int j = 0; j < N.width; ++j) {
            double sum = 0;
            for (int k = 0; k < M.width; ++k) {
                double a = M.elements[i * M.width + k];
                double b = N.elements[k * N.width + j];
                sum += a * b;
            }
            P.elements[i * N.width + j] = sum;
    }
```

Multiply Using One Thread Block

- One Block of threads compute matrix P
  - Each thread computes one element of P
- Each thread
  - Loads a row of matrix M
  - Loads a column of matrix N
  - Perform one multiply and addition for each pair of M and N elements

– Compute to off-chip memory access ratio close to 1:1 (not very high)



1hread

- Size of matrix limited by the number of threads allowed in a thread block

Step 3: Host-side Main Program Code

```
int main(void) {
// Allocate and initialize the matrices
      Matrix      M   = AllocateMatrix(WIDTH, WIDTH, 1);
      Matrix      N   = AllocateMatrix(WIDTH, WIDTH, 1);
      Matrix      P   = AllocateMatrix(WIDTH, WIDTH, 0);
//    M * N on the device
      MatrixMulOnDevice(M, N, P);
// Free       Matrices
       FreeMatrix(M);
       FreeMatrix(N);
       FreeMatrix(P);
return 0;
}
```

Host-side code

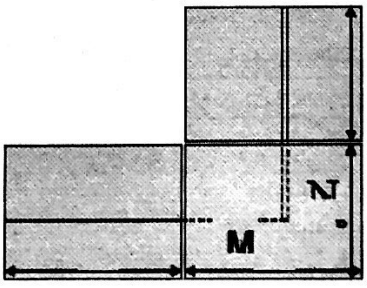
```
// Matrix multiplication on the device
void MatrixMulOnDevice(const Matrix M, const Matrix N, Matrix P)
{
      //    Load M and N to the device Matrix Md =
      AllocateDeviceMatrix(M);
      CopyToDeviceMatrix(Md, M);
      Matrix Nd = AllocateDeviceMatrix(N);
      CopyToDeviceMatrix(Nd, N);
```

```
    // Allocate P on the device


    // Setup the execution configuration dim3
    dimBlock(WIDTH, WIDTH);
    dim3 dimGrid(1, 1);
    // Launch the device computation threads!
    MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd);
    // Read P from the device
    CopyFromDeviceMatrix(P, Pd);
    // Free device matrices
  FreeDeviceMatrix(Md);
  FreeDeviceMatrix(Nd);
  FreeDeviceMatrix(Pd);
}
```

Step 4: Device-side Kernel Function

```
// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(Matrix M, Matrix N, Matrix P)
{
    // 2D Thread ID
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    // Pvalue is used to store the element of the matrix
    // that is computed by the thread
    float Pvalue = 0;
for (int k = 0; k < M.width; ++k)
    {
        float Melement = M.elements[ty * M.pitch + k];
        float Nelement = Nd.elements[k * N.pitch + tx];
        Pvalue += Melement * Nelement;
    }
    // Write the matrix to device memory;
    // each thread writes one element
    P.elements[ty * P.pitch + tx] = Pvalue;
}
```



Step 5: Some Loose Ends

- Free allocated CUDA memory

Facilities:

Latest version of 64 Bit Operating Systems, CUDA enabled NVIDIA Graphics card

Input:

Two matrices

Output:

Multiplication of two matrix

Software Engg.:

Mathematical Model:

Conclusion:

We learned parallel programming with the help of CUDA architecture.

Questions:

1. What is CUDA?
2. Explain Processing flow of CUDA programming.
3. Explain advantages and limitations of CUDA.
4. Make the comparison between GPU and CPU.
5. Explain various alternatives to CUDA.
6. Explain CUDA hardware architecture in detail.

Assignment No: 3

Title: For Bubble Sort and Merger Sort, based on existing sequential algorithms, design and implement parallel algorithm utilizing all resources available.

Aim: Understand Parallel Sorting Algorithms like Bubble sort and Merge Sort.

Prerequisites:

- Student should know basic concepts of Bubble sort and Merge Sort.

Objective: Study of Parallel Sorting Algorithms like Bubble sort and Merge Sort

Theory:

i) **What is Sorting?**

Sorting is a process of arranging elements in a group in a particular order, i.e., ascending order, descending order, alphabetic order, etc.

Characteristics of Sorting are:

- Arrange elements of a list into certain order
- Make data become easier to access
- Speed up other operations such as searching and merging. Many sorting algorithms with different time and space complexities

ii) **What is Parallel Sorting?**

A sequential sorting algorithm may not be efficient enough when we have to sort a huge volume of data. Therefore, parallel algorithms are used in sorting.

Design methodology:

- Based on an existing sequential sort algorithm
  - Try to utilize all resources available
  - Possible to turn a poor sequential algorithm into a reasonable parallel algorithm (bubble sort and parallel bubble sort)
- Completely new approach
  - New algorithm from scratch
  - Harder to develop
  - Sometimes yield better solution

Bubble Sort

The idea of bubble sort is to compare two adjacent elements. If they are not in the right order,switch them. Do this comparing and switching (if necessary) until the end of the array is reached. Repeat this process from the beginning of the array n times.

- One of the straight-forward sorting methods
  - Cycles through the list
  - Compares consecutive elements and swaps them if necessary
  - Stops when no more out of order pair
- Slow & inefficient
- Average performance is $O(n^2)$

Bubble Sort Example

Here we want to sort an array containing [8, 5, 1]. The following figure shows how we can sort this array using bubble sort. The elements in consideration are shown in **bold.**

8, 5, 1	Switch 8 and 5
5, **8, 1**	Switch 8 and 1
5, 1, 8	Reached end start again.
5, 1, 8	Switch 5 and 1
1, **5, 8**	No Switch for 5 and 8
1, 5, 8	Reached end start again.
1, 5, 8	No switch for 1, 5
1, **5, 8**	No switch for 5, 8
1, 5, 8	Reached end.

But do not start again since this is the nth iteration of same process.

Parallel Bubble sort

- Implemented as a pipeline.
- Let local_size = n / no_proc. We divide the array in no_proc parts, and each process executes the bubble sort on its part, including comparing the last element with the first one belonging to the next thread.
- Implement with the loop (instead of j<i)
  for (j=0; j<n-1; j++)
- For every iteration of i, each thread needs to wait until the previous thread has finished that iteration before starting.
- We'll coordinate using a barrier.

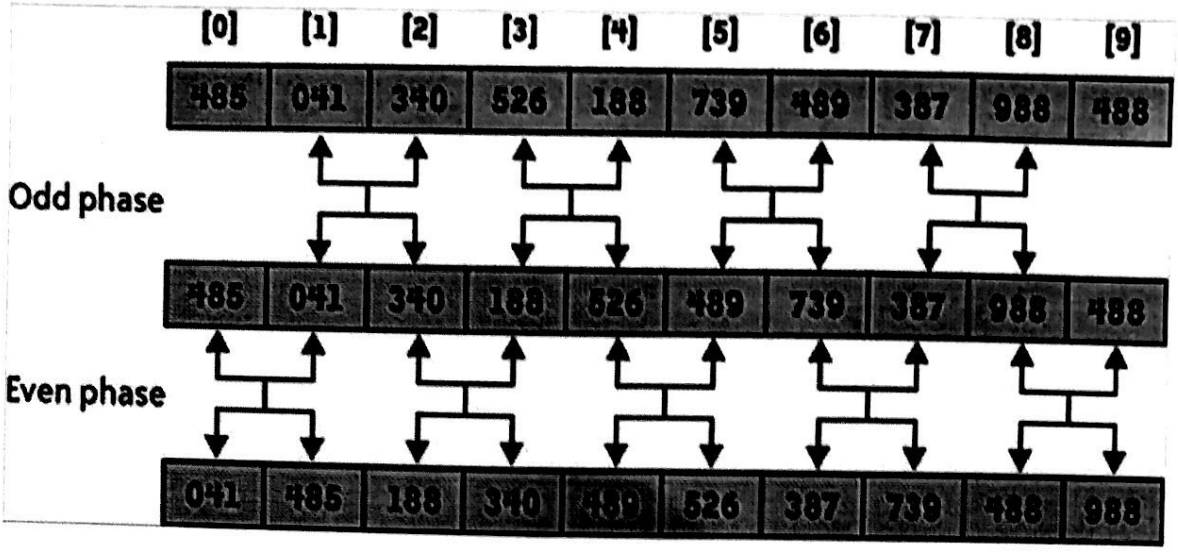
Algorithm for Parallel Bubble Sort

1. For $k = 0$ to n-2
2. If k is even then
3. for $i = 0$ to $(n/2)$-1 do in parallel
4. If $A[2i] > A[2i+1]$ then
5. Exchange $A[2i] \leftrightarrow A[2i+1]$
6. Else
7. for $i = 0$ to $(n/2)$-2 do in parallel
8. If $A[2i+1] > A[2i+2]$ then
9. Exchange $A[2i+1] \leftrightarrow A[2i+2]$
10. Next k

Parallel Bubble Sort Example 1

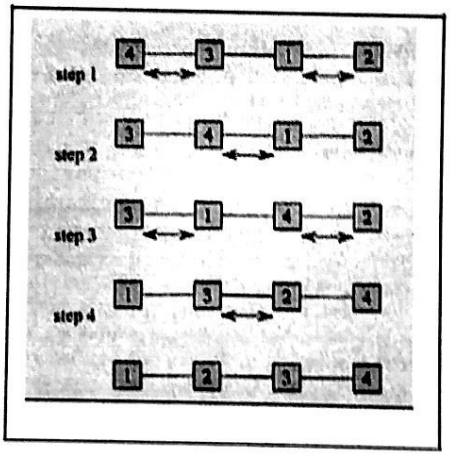
- Compare all pairs in the list in parallel

- Alternate between odd and even phases
- Shared flag, **sorted**, initialized to true at beginning of each iteration (2 phases), if any processor perform swap, **sorted** = false



Parallel Bubble Sort Example 2

- How many steps does it take to sort the following sequence from least to greatest using the Parallel Bubble Sort? How does the sequence look like after 2 cycles?
- Ex: 4,3,1,2



Merge Sort

- Collects sorted list onto one processor
- Merges elements as they come together
- Simple tree structure
- Parallelism is limited when near the root

Theory:

To sort $A[p \,..\, r]$:

1. **Divide Step**

If a given array A has zero or one element, simply return; it is already sorted. Otherwise, split $A[p .. r]$ into two subarrays $A[p .. q]$ and $A[q + 1 .. r]$, each containing about half of the elements of $A[p .. r]$. That is, q is the halfway point of $A[p .. r]$.

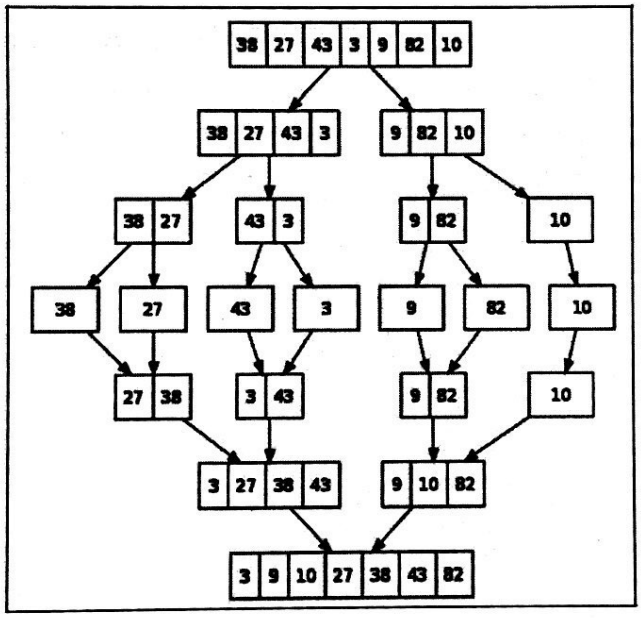
2. **Conquer Step**

Conquer by recursively sorting the two subarrays $A[p .. q]$ and $A[q + 1 .. r]$.

3. **Combine Step**

Combine the elements back in $A[p .. r]$ by merging the two sorted subarrays $A[p .. q]$ and $A[q + 1 .. r]$ into a sorted sequence. To accomplish this step, we will define a procedure MERGE (A, p, q, r).

Example:

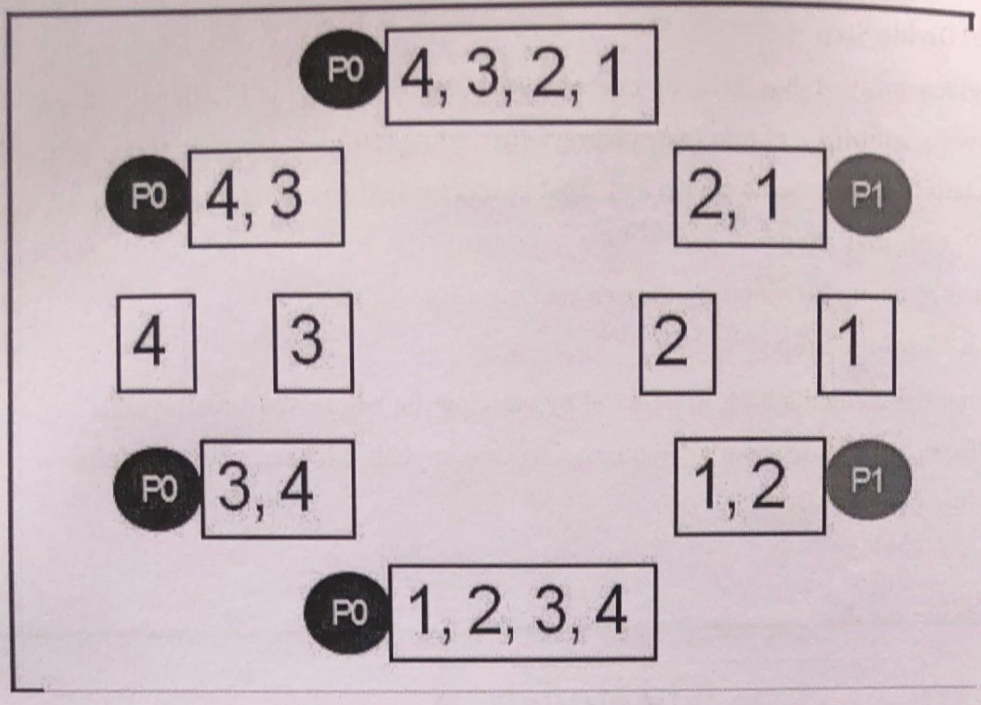


Parallel Merge Sort

- Parallelize processing of sub-problems
- Max parallelization achieved with one processor per node (at each layer/height)

Parallel Merge Sort Example

- Perform Merge Sort on the following list of elements. Given 2 processors, P0 & P1, which processor is responsible for which comparison?
- 4,3,2,1



Algorithm for Parallel Merge Sort

1. Procedure parallelMergeSort
2. Begin
3. Create processors Pi where i = 1 to n
4. if i > 0 then receive size and parent from the root
5. receive the list, size and parent from the root
6. endif
7. midvalue= listsize/2
8. if both children is present in the tree then
9. send midvalue, first child
10. send listsize-mid,second child
11. send list, midvalue, first child
12. send list from midvalue, listsize-midvalue, second child
13. call mergelist(list,0,midvalue,list, midvalue+1,listsize,temp,0,listsize)
14. store temp in another array list2
15. else
16. call parallelMergeSort(list,0,listsize)
17. endif
18. if i >0 then
19. send list, listsize,parent
20. endif
21. end

INPUT:

1. Array of integer numbers.

OUTPUT:

1. Sorted array of numbers

FAQ

1. What is sorting?
2. What is parallel sort?

3. How to sort the element using Bubble Sort?
4. How to sort the element using Parallel Bubble Sort?
5. How to sort the element using Parallel Merge Sort?
6. How to sort the element using Merge Sort?
7. What is searching?
8. Different types of searching methods.
9. Time complexities of sorting and searching methods.
10. How to calculate time complexity?
11. What are space complexity of all sorting and searching methods?
12. Explain what is best, worst and average case for each method of searching and sorting.

ALGORITHM ANALYSIS

1. Time Complexity Of parallel Merge Sort and parallel Bubble sort in best case is(when all data is already in sorted form):O(n)
2. Time Complexity Of parallel Merge Sort and parallel Bubble sort in worst case is: O(n logn)
3. Time Complexity Of parallel Merge Sort and parallel Bubble sort in average case is: O(n logn)

APPLICATIONS

1. Representing Linear data structure & Sequential data organization : structure & files
2. For Sorting sequential data structure

CONCLUSION

Thus, we have studied Parallel Bubble and Parallel Merge sort implementation.

Laboratory Practice – I BE (Comp Engg)

Experiment No: 4

Aim: Design and implement parallel algorithm utilizing all resources available. For

- Binary Search for Sorted Array
- Depth-First Search (tree or an undirected graph) OR
- Breadth-First Search (tree or an undirected graph) OR
- Best-First Search that (traversal of graph to reach a target in the shortest possible path)

Objective: To study and implementation of searching techniques.

Outcome: Students will be understand the implementation of Binary search and BFS, DFS

Pre-requisites:

64-bit Open source Linux or its derivative

Programming Languages: C++/JAVA/PYTHON/R

Theory:

Binary Search:

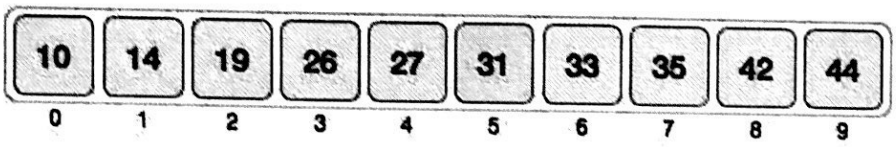
In computer science, binary search, also known as half-interval search, logarithmic search, or binary chop, is a search algorithm that finds the position of a target value within a sorted array. Binary search compares the target value to the middle element of the array. If they are not equal, the half in which the target cannot lie is eliminated and the search continues on the remaining half, again taking the middle element to compare to the target value, and repeating this until the target value is found. If the search ends with the remaining half being empty, the target is not in the array. Even though the idea is simple, implementing binary search correctly requires attention to some subtleties about its exit conditions and midpoint calculation.

Binary search runs in logarithmic time in the worst case, making $O(\log n)$ comparisons,
where n is the number of elements in the array, the O is Big O notation, and log is the logarithm. Binary search takes constant ($O(1)$) space, meaning that the space taken by the algorithm is the same for any number of elements in the array.Binary search is faster than linear search except for small arrays, but the array must be sorted first. Although specialized data structures designed for fast

searching, such as hash tables, can be searched more efficiently, binary search applies to a wider range of problems.

How Binary Search Works?

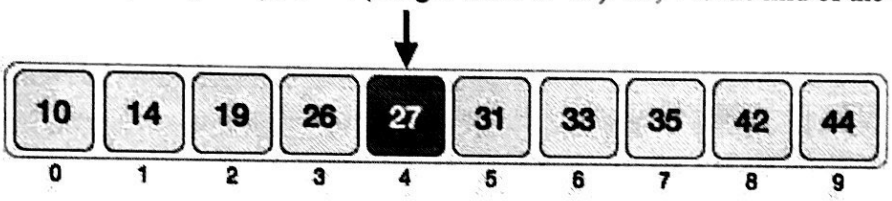
For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.



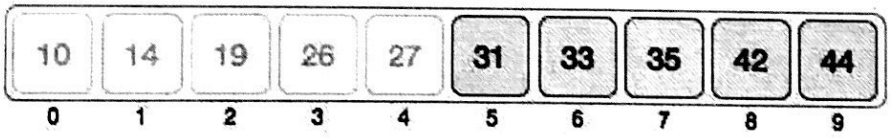
First, we shall determine half of the array by using this formula –

mid = low + (high - low) / 2

Here it is, 0 + (9 - 0) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



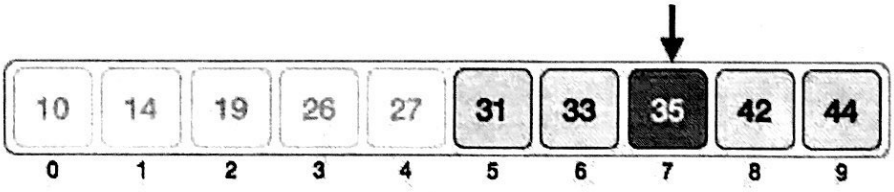
Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.



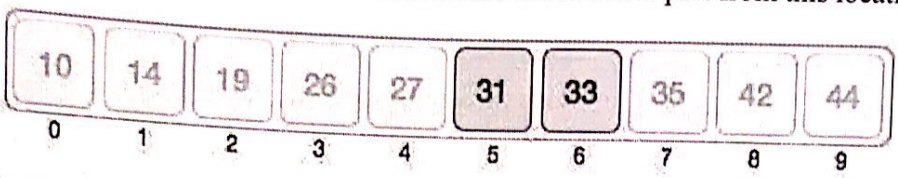
We change our low to mid + 1 and find the new mid value again.

low = mid + 1
mid = low + (high - low) / 2

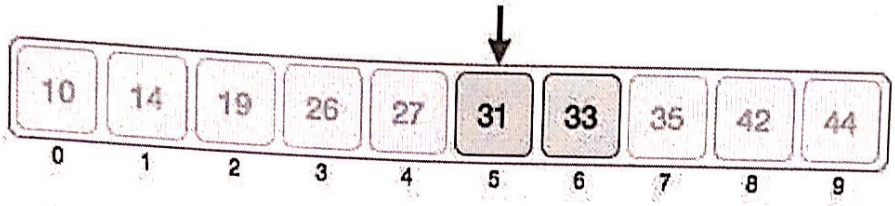
Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.



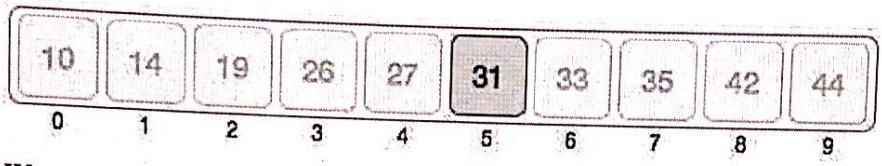
The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.



Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.



We conclude that the target value 31 is stored at location 5.

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

Binary Search Code:

```
#include<iostream>

#include<stdlib.h>

#include<omp.h> using
namespace std;

int binary(int *, int, int, int);

int binary(int *a, int low, int high, int key){
 int mid; mid=(low+high)/2;

int low1,low2,high1,high2,mid1,mid2,found=0,loc=-1;
#pragma omp parallel sections { #pragma omp section {
 low1=low; high1=mid;

while(low1<=high1) {

if(!(key>=a[low1] && key<=a[high1])) { low1=low1+high1; continue; } cout<<"here1";

mid1=(low1+high1)/2;

if(key==a[mid1]) { found=1; loc=mid1; low1=high1+1; } else
if(key>a[mid1]) { low1=mid1+1; } else if(key<a[mid1]) high1=mid1-1;}
  }


  #pragma omp section {
```

```
        low2=mid+1; high2=high;
        while(low2<=high2) {

        if(!(key>=a[low2] && key<=a[high2])) { low2=low2+high2; continue; }
        cout<<"here2";

mid2=(low2+high2)/2;

if(key==a[mid2]) { found=1; loc=mid2;
low2=high2+1; } else if(key>a[mid2]) {
low2=mid2+1; } else if(key<a[mid2])
high2=mid2-1;

} } }

return loc;

}

int main(){

int *a,i,n,key,loc=-1;

cout<<"\n enter total no of elements=>";
cin>>n; a=new int[n]; cout<<"\n enter elements=>";
for(i=0;i<n;i++) { cin>>a[i]; }
cout<<"\n enter key to find=>"; cin>>key;
loc=binary(a,0,n-1,key);

if(loc==-1) cout<<"\n Key not found.";
      else cout<<"\n Key found at
      position=>"<<loc+1; return 0;
      }
```

Breadth-First Search :

Graph traversals

Graph traversal means visiting every vertex and edge exactly once in a well-defined order. While using certain graph algorithms, you must ensure that each vertex of the graph is visited exactly once. The order in which the vertices are visited are important and may depend upon the algorithm or question that you are solving.

During a traversal, it is important that you track which vertices have been visited. The most common way of tracking vertices is to mark them.

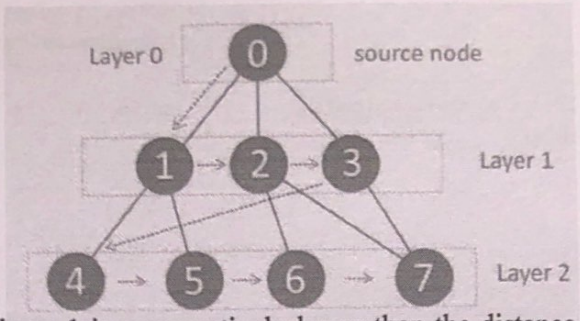
Breadth First Search (BFS)

There are many ways to traverse graphs. BFS is the most commonly used approach. BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbour nodes.

As the name BFS suggests, you are required to traverse the graph breadthwise as follows:

1. First move horizontally and visit all the nodes of the current layer
2. Move to the next layer

Consider the following diagram.



The distance between the nodes in layer 1 is comparatively lesser than the distance between the nodes in layer 2. Therefore, in BFS, you must traverse all the nodes in layer 1 before you move to the nodes in layer 2.

Traversing child nodes

A graph can contain cycles, which may bring you to the same node again while traversing the graph. To avoid processing of same node again, use a boolean array which marks the node after it is processed. While visiting the nodes in the layer of a graph, store them in a manner such that you can traverse the corresponding child nodes in a similar order.

In the earlier diagram, start traversing from 0 and visit its child nodes 1, 2, and 3. Store them in the order in which they are visited. This will allow you to visit the child nodes of 1 first (i.e. 4 and 5), then of 2 (i.e. 6 and 7), and then of 3 (i.e. 7) etc.

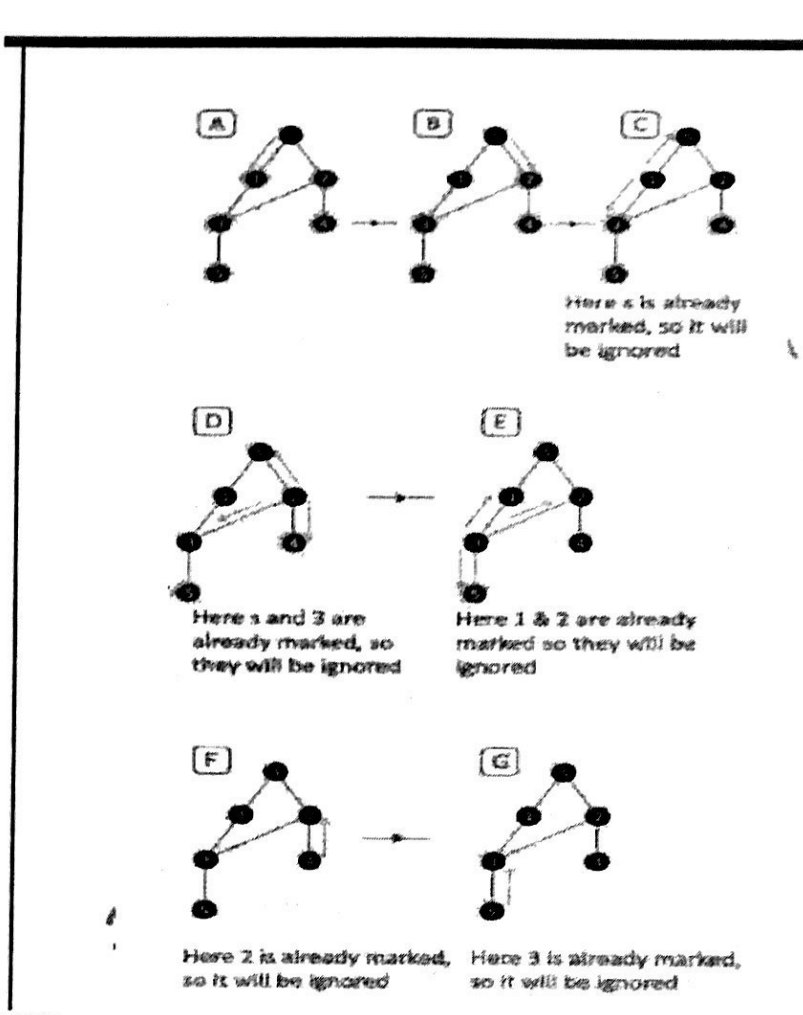
To make this process easy, use a queue to store the node and mark it as 'visited' until all its neighbors (vertices that are directly connected to it) are marked. The queue follows the First In First Out (FIFO) queuing method, and therefore, the neighbors of the node will be visited in the order in which they were inserted in the node i.e. the node that was inserted first will be visited first, and so on.

Pseudo code for BFS:

```
FS (G, s) //Where G is the graph and s is the source node let Q
be queue.
Q.enqueue( s ) //Inserting s in queue until all its neighbor vertices are marked.
marks as visited.
while ( Q is not
empty)
  //Removing that vertex from queue,whose neighbour will be visited now
   v = Q.dequeue( )
  //processing all the
  neighbours of v for all
  neighbours w of v in Graph G
```

if w is not visited

Q.enqueue(w) //Stores w in Q to further visit its neighbour mark w as visited.



Traversing process

Code for BFS:

```
#include<iostream>
#include<stdlib.h>
#include<queue> using
namespace std;
class node{ public: node *left, *right; int data;};

class Breadthfs{ public: node *insert(node *, int); void bfs(node *); }; node *insert(node *root, int data){
 / /inserts a node in tree if(!root) {


        root=new node; root-
        >left=NULL; root-
        >right=NULL; root-
        >data=data; return root;

    }

    queue<node *> q; q.push(root);
    while(!q.empty()) {

    node *temp=q.front(); q.pop();
    if(temp->left==NULL) {
temp->left=new node; temp->left-
```

```
>left=NULL; temp->left->right=NULL ;
temp->left->data=data; return root;

}

else { q.push(temp->left); }
if(temp->right==NULL) { temp-
>right=new node; temp->right-
>left=NULL; temp->right-
>right=NULL; temp->right-
>data=data;
return root; }

else { q.push(temp->right); }


} }


void bfs(node *head){

queue<node*> q; q.push(head); int
qSize; while (!q.empty()) {

qSize = q.size();

#pragma omp parallel for / /creates
parallel threads for (int i = 0; i <
qSize; i++) { node* currNode; #pragma
omp critical {

currNode = q.front();

 q.pop(); cout<<"\t"<<currNode->data; } / /prints parent node #pragma omp
critical {

if(currNode->left) q.push(currNode->left); / /push parent's left node in queue if(currNode->right)
q.push(currNode->right); } / /push parent's right node in queue } }

}

int main(){

node *root=NULL; int data; char
ans; do { cout<<"\n enter
data=>"; cin>>data;
root=insert(root,data);

cout<<"do you want insert one more node?"; cin>>ans;
}while(ans=='y'||ans=='Y');

bfs(root);
return 0;
}
```

Conclusion: We have implemented Binary searching and BFS .