GROUP – A

Assignment No: 1

==========

Title:- Design and implement Parallel Breadth First Search and Depth First Search based on existing algorithms using OpenMP. Use a Tree or an undirected graph for BFS and DFS

==========

Objective:-

-To learn about OpenMP.

-To understand the concept of parallelism or parallel algorithms.

==========

Theory:-

- **What is parallel algorithm?**
  - An algorithm is a sequence of steps that take inputs from the user and after some computation, produces an output. A parallel algorithm is an algorithm that can execute several instructions simultaneously on different processing devices and then combine all the individual outputs to produce the final result.
  - In parallel algorithm, The problem is divided into sub-problems and are executed in parallel to get individual outputs. Later on, these individual outputs are combined together to get the final desired output.
  - While designing a parallel algorithm, proper CPU utilization should be considered to get an efficient algorithm.

- **What is OpenMP?**
  - OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared-memory multiprocessing programming in C, C++, and Fortran, on many platforms, instruction-set architectures and operating systems, including Solaris, AIX, FreeBSD, HP-UX, Linux, macOS, and Windows. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.
  - An application built with the hybrid model of parallel programming can run on a computer cluster using both OpenMP and Message Passing Interface (MPI), such that OpenMP is used for parallelism within a (multi-core) node while MPI is used for parallelism between nodes. There have also been efforts to run OpenMP on software distributed shared memory systems, to translate OpenMP into MPI and to extend OpenMP for non-shared memory systems.

- OpenMP is an implementation of multithreading, a method of parallelizing whereby a primary thread (a series of instructions executed consecutively) forks a specified number of sub-threads and the system divides a task among them. The threads then run concurrently, with the runtime environment allocating threads to different processors.

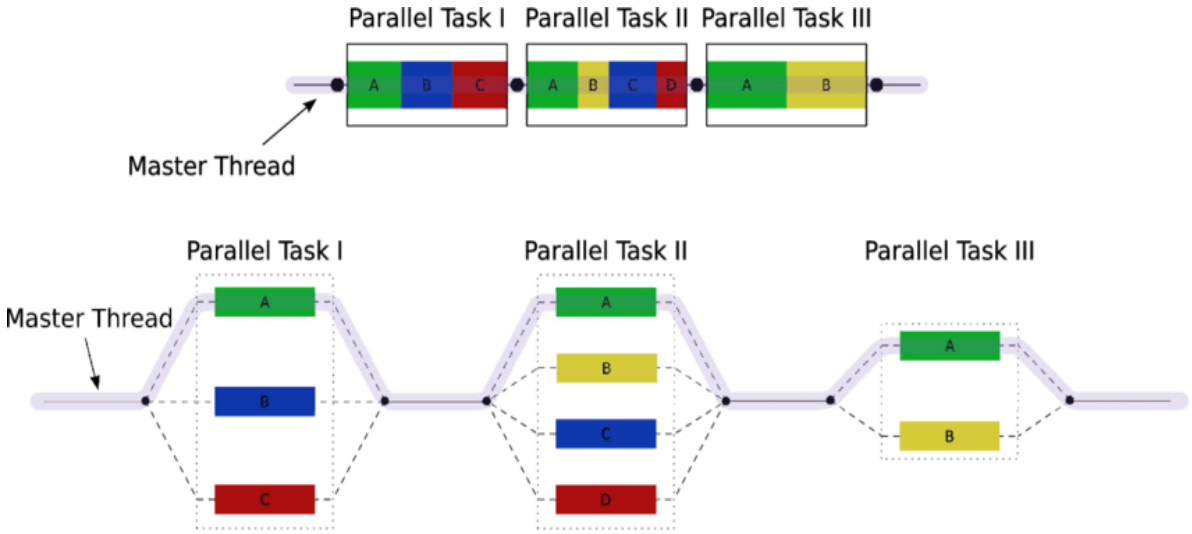


Figure 1: Multithreading

- The section of code that is meant to run in parallel is marked accordingly, with a compiler directive that will cause the threads to form before the section is executed.[3] Each thread has an ID attached to it which can be obtained using a function (called omp_get_thread_num()). The thread ID is an integer, and the primary thread has an ID of 0. After the execution of the parallelized code, the threads join back into the primary thread, which continues onward to the end of the program.
- By default, each thread executes the parallelized section of code independently. Work-sharing constructs can be used to divide a task among the threads so that each thread executes its allocated part of the code. Both task parallelism and data parallelism can be achieved using OpenMP
- The runtime environment allocates threads to processors depending on usage, machine load and other factors. The runtime environment can assign the number of threads based on environment variables, or the code can do so using functions.
- The OpenMP functions are included in a header file labelled **omp.h** in C/C++.
- The core elements of OpenMP are the constructs for thread creation, workload distribution (work sharing), data-environment management, thread synchronization, user-level runtime routines and environment variables.
- In C/C++, OpenMP uses **#pragmas** for thread creation.

- Sample code for thread creation:

```
#include <stdio.h>
#include <omp.h>
int main(void)
{
  #pragma omp parallel
  printf("Hello, world.\n");
  return 0;
}
```

- In above sample code, The pragma omp parallel is used to fork additional threads to carry out the work enclosed in the construct in parallel. The original thread will be denoted as master thread with thread ID 0.
- For compilation in GCC using fopenmp:

```
$ gcc -fopenmp hello.c -o hello -ldl
```

- **Breadth First Search (BFS):**
  - Traversal means visiting all the nodes of a graph. Breadth First Traversal or Breadth First Search is a recursive algorithm for searching all the vertices of a graph or tree data structure.
  - A standard BFS implementation puts each vertex of the graph into one of two categories:
    1. Visited
    2. Not Visited
  - The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

    - **Algorithm for BFS:**

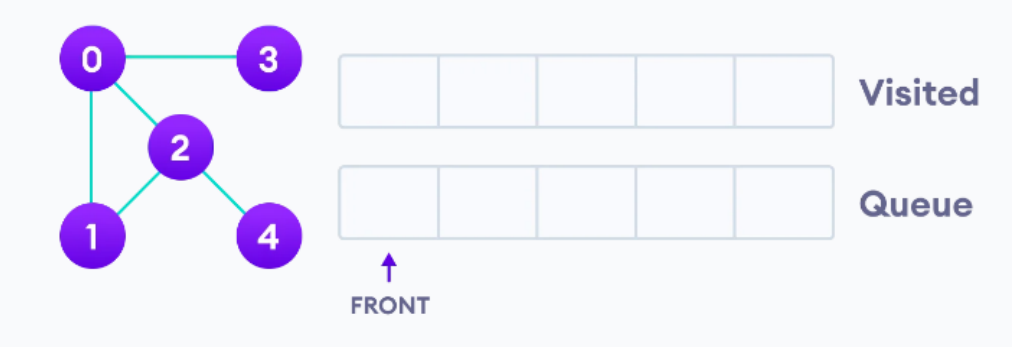
      Step 1: Start by putting any one of the graph's vertices at the back of a queue.

      Step 2: Take the front item of the queue and add it to the visited list.

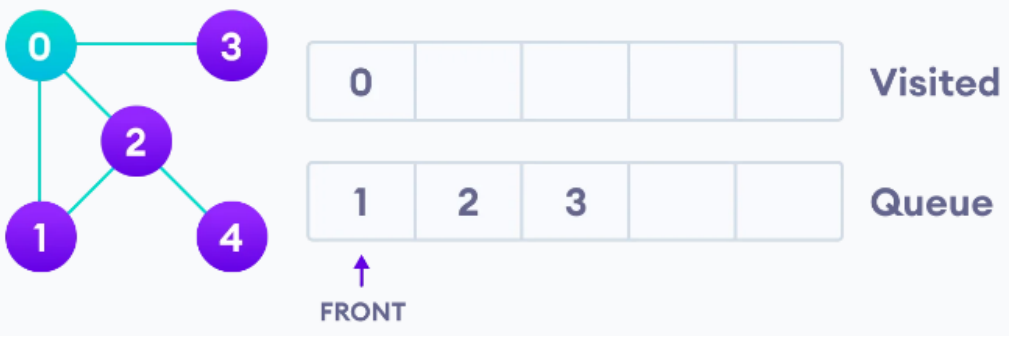
      Step 3: Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.

      Step 4: Keep repeating steps 2 and 3 until the queue is empty.

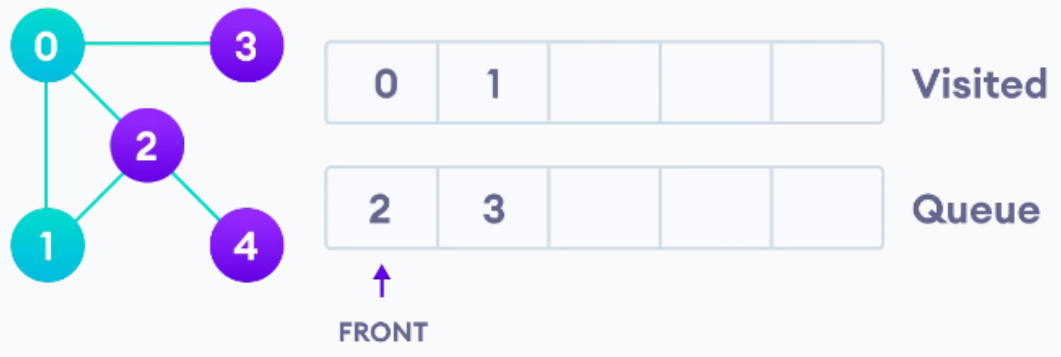
- **Example of BFS:**
  - An undirected graph with 5 vertices.



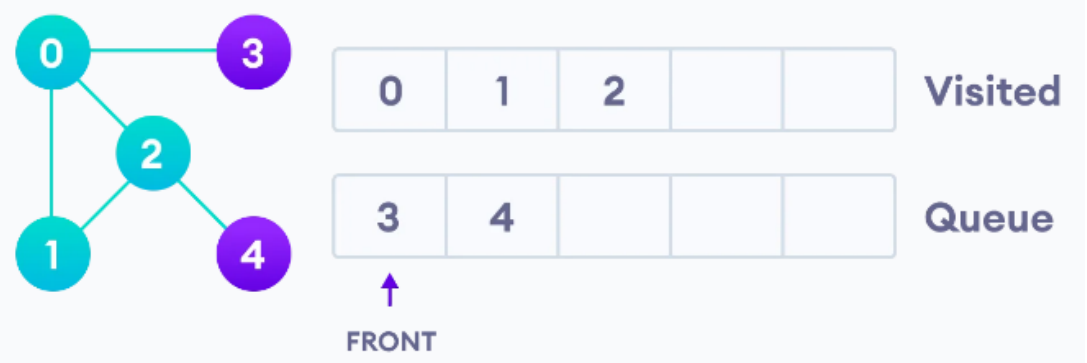
  - Start from vertex 0, the BFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.



  - Visit the element at the front of queue i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.



  - Vertex 2 has an unvisited adjacent vertex in 4, add that to the back of the queue and visit 3, which is at the front of the queue.



- Only 4 remains in the queue since the only adjacent node of 3 i.e. 0 is already visited.

- Visit last remaining item in the queue to check if it has unvisited neighbors. Since the queue is empty, we have completed the Breadth First Traversal of the graph.

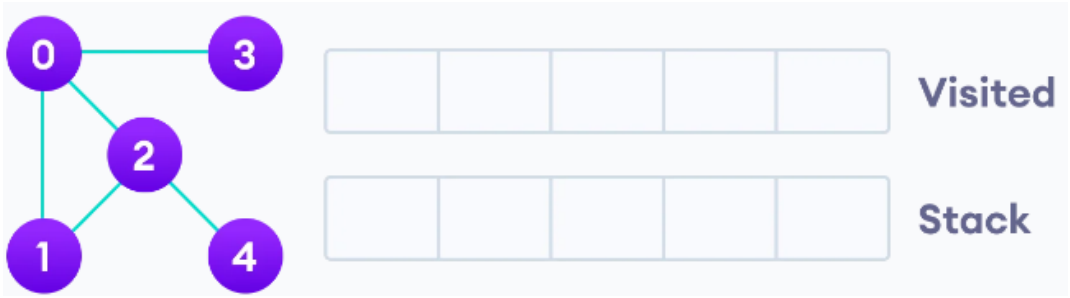
- **Depth First Search (DFS):**

- Depth first Search or Depth first traversal is a recursive algorithm for searching all the vertices of a graph or tree data structure. Traversal means visiting all the nodes of a graph.
- A standard DFS implementation puts each vertex of the graph into one of two categories:
  1. Visited
  2. Not Visited
- The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

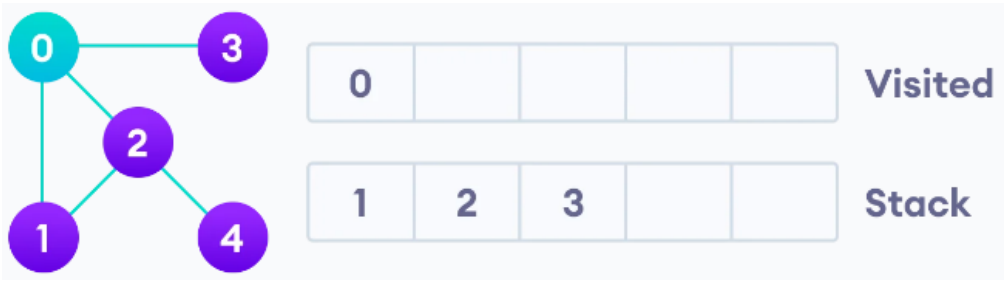
- **Algorithm for BFS:**

  Step 1: Start by putting any one of the graph's vertices on top of a stack.
  Step 2: Take the top item of the stack and add it to the visited list.
  Step 3: Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
  Step 4: Keep repeating steps 2 and 3 until the stack is empty.

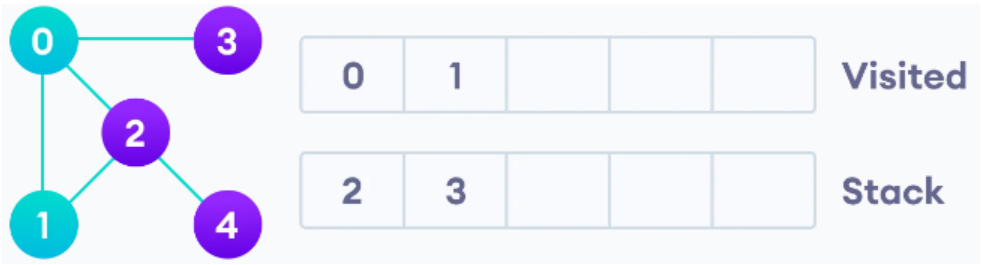
- **Examples of BFS:**
  - An undirected graph with 5 vertices.



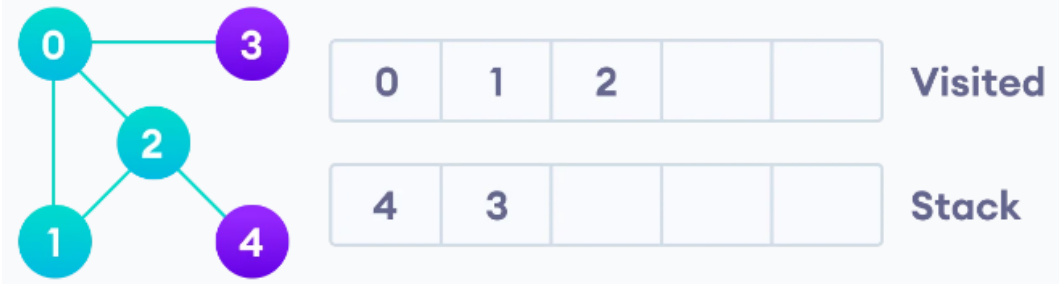
  - Start from vertex 0, the DFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.

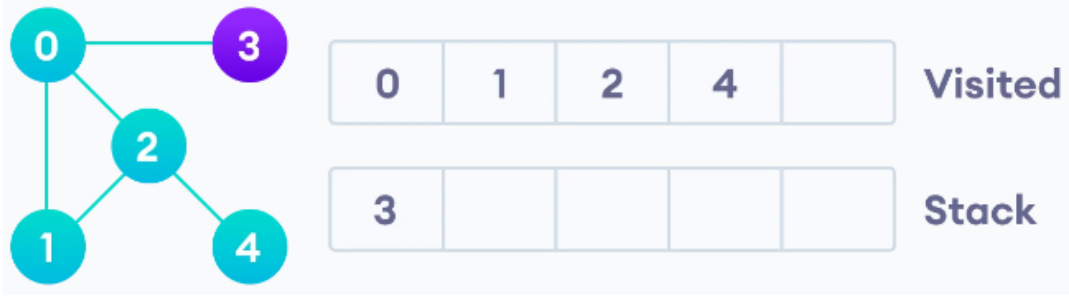


  - Visit the element at the top of stack i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.

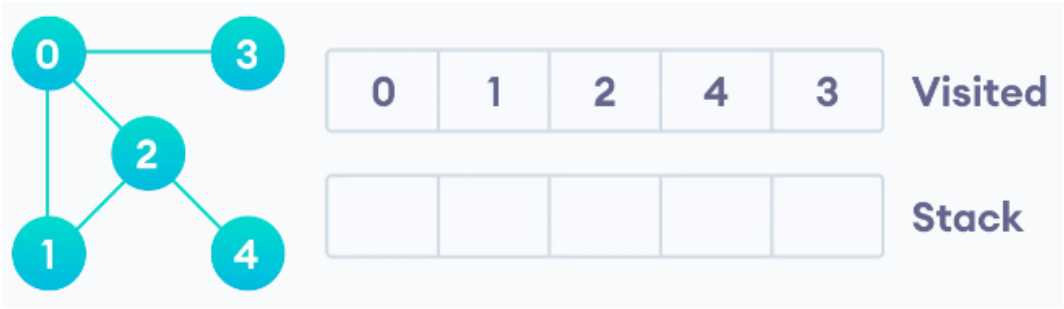


- Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.





- After visit the last element 3, it doesn't have any unvisited adjacent nodes, so completed the Depth First Traversal of the graph.



==========

Conclusion:-

Thus we have studied parallelism and OpenMP used in programming. Also, How DFS and BFS algorithm works for traversal of graph.

==========

GROUP – A

Assignment No: 2

==========

Title: - Write a program to implement Parallel Bubble Sort and Merge sort using OpenMP. Use existing algorithms and measure the performance of sequential and parallel algorithms.

==========

Objective:-

-To learn about OpenMP.

-To understand the concept of parallelism or parallel algorithms.

==========

Theory:-

- **What is parallel algorithm?**
  - An algorithm is a sequence of steps that take inputs from the user and after some computation, produces an output. A parallel algorithm is an algorithm that can execute several instructions simultaneously on different processing devices and then combine all the individual outputs to produce the final result.
  - In parallel algorithm, The problem is divided into sub-problems and are executed in parallel to get individual outputs. Later on, these individual outputs are combined together to get the final desired output.
  - While designing a parallel algorithm, proper CPU utilization should be considered to get an efficient algorithm.

- **What is OpenMP?**
  - OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared-memory multiprocessing programming in C, C++, and Fortran, on many platforms, instruction-set architectures and operating systems, including Solaris, AIX, FreeBSD, HP-UX, Linux, macOS, and Windows. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.
  - An application built with the hybrid model of parallel programming can run on a computer cluster using both OpenMP and Message Passing Interface (MPI), such that OpenMP is used for parallelism within a (multi-core) node while MPI is used for parallelism between nodes. There have also been efforts to run OpenMP on software distributed shared memory systems, to translate OpenMP into MPI and to extend OpenMP for non-shared memory systems.

- OpenMP is an implementation of multithreading, a method of parallelizing whereby a primary thread (a series of instructions executed consecutively) forks a specified number of sub-threads and the system divides a task among them. The threads then run concurrently, with the runtime environment allocating threads to different processors.

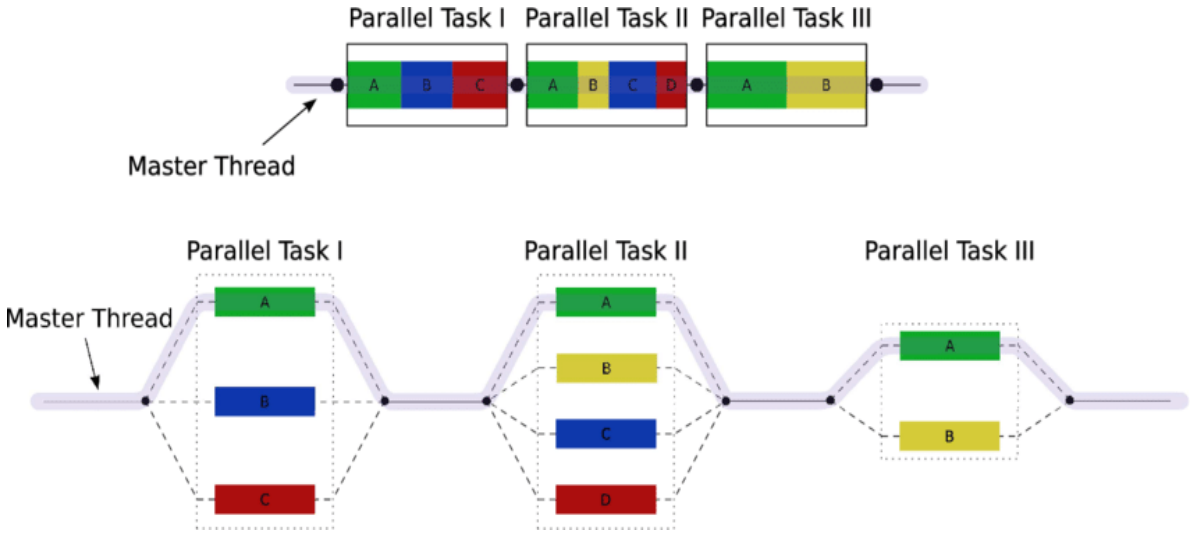


Figure 1: Multithreading

- The section of code that is meant to run in parallel is marked accordingly, with a compiler directive that will cause the threads to form before the section is executed.[3] Each thread has an ID attached to it which can be obtained using a function (called omp_get_thread_num()). The thread ID is an integer, and the primary thread has an ID of 0. After the execution of the parallelized code, the threads join back into the primary thread, which continues onward to the end of the program.
- By default, each thread executes the parallelized section of code independently. Work-sharing constructs can be used to divide a task among the threads so that each thread executes its allocated part of the code. Both task parallelism and data parallelism can be achieved using OpenMP
- The runtime environment allocates threads to processors depending on usage, machine load and other factors. The runtime environment can assign the number of threads based on environment variables, or the code can do so using functions.
- The OpenMP functions are included in a header file labelled **omp.h** in C/C++.
- The core elements of OpenMP are the constructs for thread creation, workload distribution (work sharing), data-environment management, thread synchronization, user-level runtime routines and environment variables.
- In C/C++, OpenMP uses **#pragmas** for thread creation.

- For compilation in GCC using fopenmp:

```
$ gcc -fopenmp hello.c -o hello -ldl
```

- **Bubble Sort**
  - Parallel Bubble Sort is a modification of the classic Bubble Sort algorithm that takes advantage of parallel processing to speed up the sorting process.
  - In parallel Bubble Sort, the list of elements is divided into multiple sublists that are sorted concurrently by multiple threads. Each thread sorts its sublist using the regular Bubble Sort algorithm. When all sublists have been sorted, they are merged together to form the final sorted list.
  - The parallelization of the algorithm is achieved using OpenMP, a programming API that supports parallel processing in C++, Fortran, and other programming languages. OpenMP provides a set of compiler directives that allow developers to specify which parts of the code can be executed in parallel.
  - In the parallel Bubble Sort algorithm, the main loop that iterates over the list of elements is divided into multiple iterations that are executed concurrently by multiple threads. Each thread sorts a subset of the list, and the threads synchronize their work at the end of each iteration to ensure that the elements are properly ordered.
  - Parallel Bubble Sort can provide a significant speedup over the regular Bubble Sort algorithm, especially when sorting large datasets on multi-core processors. However, the speedup is limited by the overhead of thread creation and synchronization, and it may not be worth the effort for small datasets or when using a single-core processor.
  - To measure the performance of sequential Bubble sort and parallel Bubble sort algorithms follow steps are use:
    1. Implement both the sequential and parallel Bubble sort algorithms.
    2. Choose a range of test cases, such as arrays of different sizes and different degrees of sortedness, to test the performance of both algorithms.
    3. Use a reliable timer to measure the execution time of each algorithm on each test case.
    4. Record the execution times and analyze the results.
    5. Run each algorithm multiple times on each test case and take the average execution time to reduce the impact of variations in system load and other factors.
    6. Monitor system resource usage during execution, such as CPU utilization and memory consumption, to detect any performance bottlenecks.
    7. Visualize the results using charts or graphs to make it easier to compare the performance of the two algorithms.

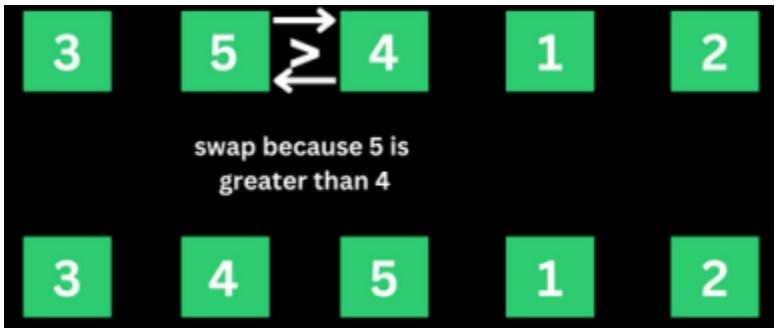
  - **Example of bubble sort:**

Sort a series of numbers 5, 3, 4, 1, and 2 in ascending order. The sorting begins the first iteration by comparing the first two values. If the first value is greater than the second, the algorithm pushes the first value to the index of the second value.

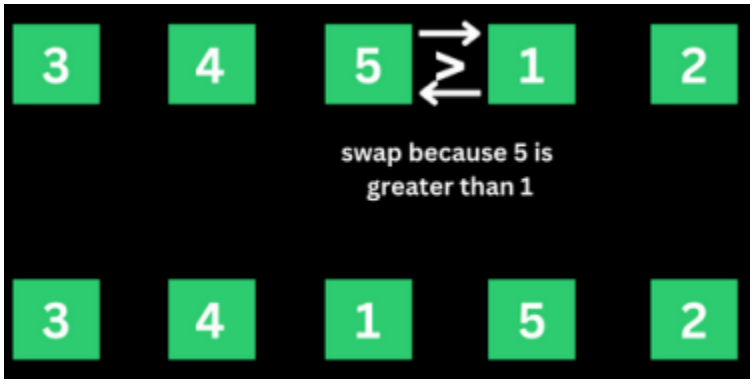
- First Iteration of the Sorting

Step 1: In the case of 5, 3, 4, 1, and 2, 5 is greater than 3. So 5 takes the position of 3 and the numbers become 3, 5, 4, 1, and 2.

Step 2: The algorithm now has 3, 5, 4, 1, and 2 to compare, this time around, it compares the next two values, which are 5 and 4. 5 is greater than 4, so 5 takes the index of 4 and the values now become 3, 4, 5, 1, and 2.



Step 3: The algorithm now has 3, 4, 5, 1, and 2 to compare. It compares the next two values, which are 5 and 1. 5 is greater than 1, so 5 takes the index of 1 and the numbers become 3, 4, 1, 5, and 2.



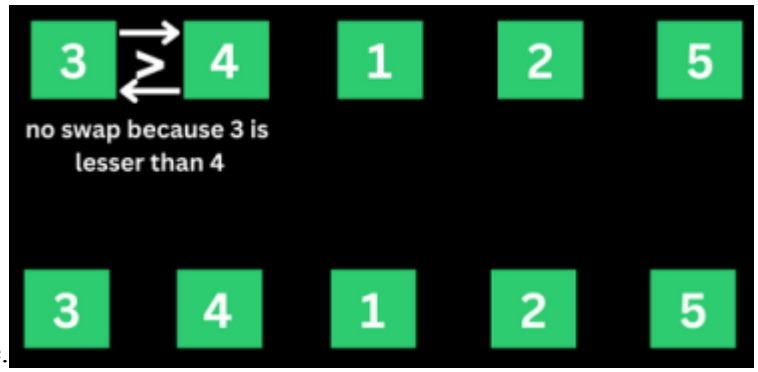
Step 4: The algorithm now has 3, 4, 1, 5, and 2 to compare. It compares the next two values, which are 5

and 2. 5 is greater than 2, so 5 takes the index of 2 and the numbers become 3, 4, 1, 2, and 5.

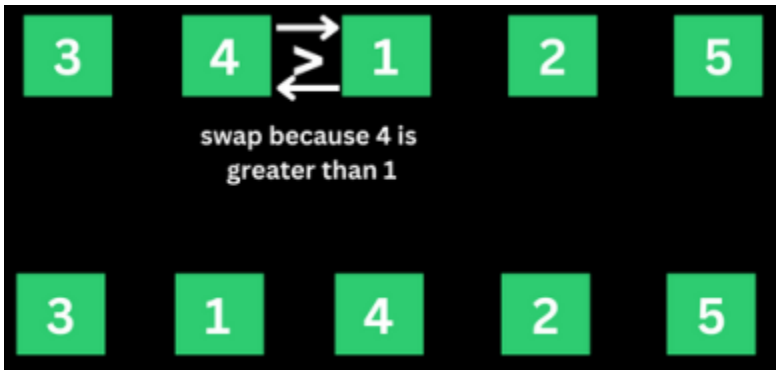
That's the first iteration. And the numbers are now arranged as 3, 4, 1, 2, and 5 – from the initial 5, 3, 4, 1, and 2. As you might realize, 5 should be the last number if the numbers are sorted in ascending order. This means the first iteration is really completed.

- Second Iteration of the Sorting and the Rest

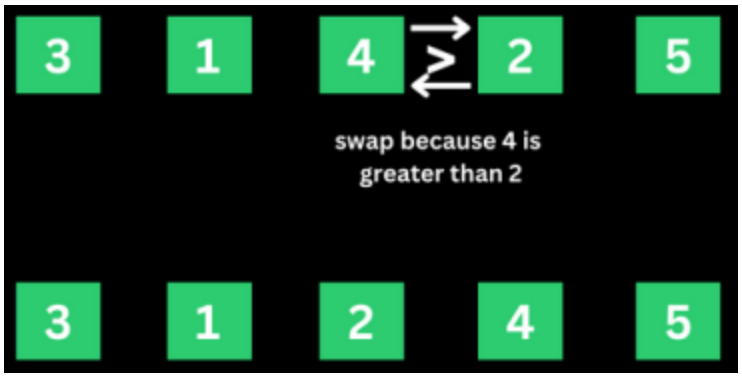
The algorithm starts the second iteration with the last result of 3, 4, 1, 2, and 5. This time around, 3 is smaller than 4, so no swapping happens. This means the numbers will remain the same.



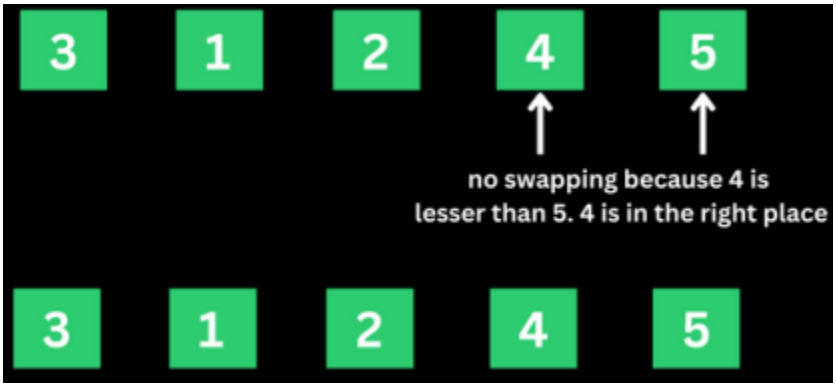
The algorithm proceeds to compare 4 and 1. 4 is greater than 1, so 4 is swapped for 1 and the numbers become 3, 1, 4, 2, and 5.



The algorithm now proceeds to compare 4 and 2. 4 is greater than 2, so 4 is swapped for 2 and the numbers become 3, 1, 2, 4, and 5.



4 is now in the right place, so no swapping occurs between 4 and 5 because 4 is smaller than 5.



That's how the algorithm continues to compare the numbers until they are arranged in ascending order of 1, 2, 3, 4, and 5.

- **Merge Sort:**

- Parallel merge sort is a parallelized version of the merge sort algorithm that takes advantage of multiple processors or cores to improve its performance.
- In parallel merge sort, the input array is divided into smaller subarrays, which are sorted in parallel using multiple processors or cores. The sorted subarrays are then merged together in parallel to produce the final sorted output.
- The parallel merge sort algorithm can be broken down into the following steps:
  1. Divide the input array into smaller subarrays.
  2. Assign each subarray to a separate processor or core for sorting.
  3. Sort each subarray in parallel using the merge sort algorithm.
  4. Merge the sorted subarrays together in parallel to produce the final sorted output.
  5. The merging step in parallel merge sort is performed in a similar way to the merging step in the sequential merge sort algorithm. However, because the subarrays are sorted in parallel, the merging step can also be performed in parallel using multiple

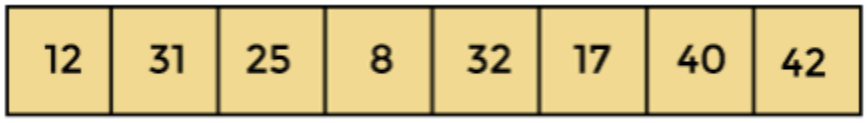
processors or cores. This can significantly reduce the time required to merge the sorted subarrays and produce the final output.

6. Parallel merge sort can provide significant performance benefits for large input arrays with many elements, especially when running on hardware with multiple processors or cores. However, it also requires additional overhead to manage the parallelization, and may not always provide performance improvements for smaller input sizes or when run on hardware with limited parallel processing capabilities.

- There are several metrics that can be used to measure the performance of sequential and parallel merge sort algorithms:
  1. Execution time: Execution time is the amount of time it takes for the algorithm to complete its sorting operation. This metric can be used to compare the speed of sequential and parallel merge sort algorithms.
  2. Speedup: Speedup is the ratio of the execution time of the sequential merge sort algorithm to the execution time of the parallel merge sort algorithm. A speedup of greater than 1 indicates that the parallel algorithm is faster than the sequential algorithm.
  3. Efficiency: Efficiency is the ratio of the speedup to the number of processors or cores used in the parallel algorithm. This metric can be used to determine how well the parallel algorithm is utilizing the available resources.
  4. Scalability: Scalability is the ability of the algorithm to maintain its performance as the input size and number of processors or cores increase. A scalable algorithm will maintain a consistent speedup and efficiency as more resources are added.

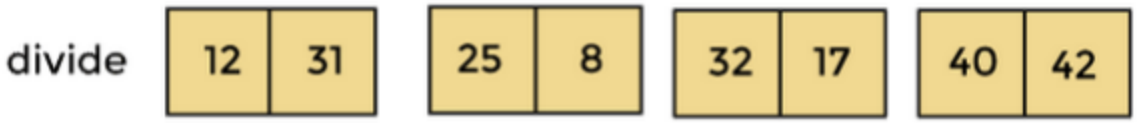
- **Examples of Merge Sort:**

- To understand the working of the merge sort algorithm, let's take an unsorted array. It will be easier to understand the merge sort via an example. Let the elements of array are:



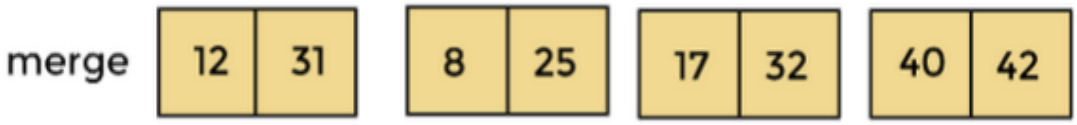
- According to the merge sort, first divide the given array into two equal halves. Merge sort keeps dividing the list into equal parts until it cannot be further divided. As there are eight elements in the given array, so it is divided into two arrays of size 4.

- Now, again divide these two arrays into halves. As they are of size 4, divide them into new arrays of size 2.

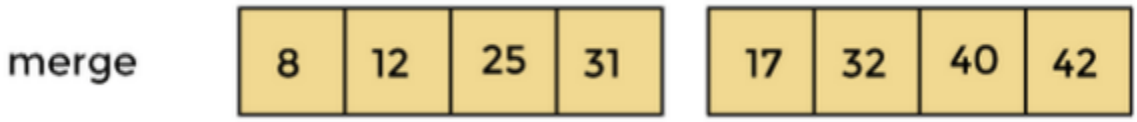


- Now, again divide these arrays to get the atomic value that cannot be further divided.

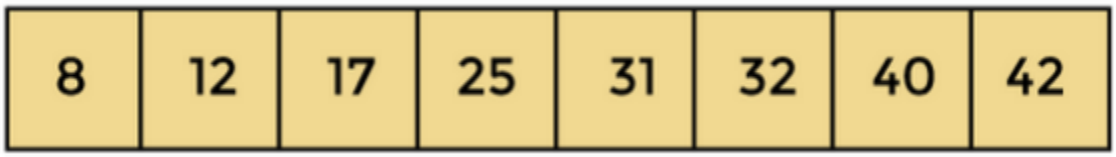
- Now, combine them in the same manner they were broken. In combining, first compare the element of each array and then combine them into another array in sorted order.
- So, first compare 12 and 31, both are in sorted positions. Then compare 25 and 8, and in the list of two values, put 8 first followed by 25. Then compare 32 and 17, sort them and put 17 first followed by 32. After that, compare 40 and 42, and place them sequentially.



- In the next iteration of combining, now compare the arrays with two data values and merge them into an array of found values in sorted order.



- Now, there is a final merging of the arrays. After the final merging of above arrays, the array will look like –



==========

Conclusion:-

Thus we have studied parallelism and OpenMP used in programming. Also, How Bubble sort and Merge sort algorithm works in parallel programming.

==========

GROUP – A

Assignment No: 3

==========

Title: - Write a program to implement Min, Max, Sum and Average operations using Parallel Reduction.

==========

Objective:-

-To learn about OpenMP

-To understand the concept of Parallel Reduction

==========

Theory:-

- **What is OpenMP?**
  - OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared-memory multiprocessing programming in C, C++, and Fortran, on many platforms, instruction-set architectures and operating systems, including Solaris, AIX, FreeBSD, HP-UX, Linux, macOS, and Windows. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.
  - An application built with the hybrid model of parallel programming can run on a computer cluster using both OpenMP and Message Passing Interface (MPI), such that OpenMP is used for parallelism within a (multi-core) node while MPI is used for parallelism between nodes. There have also been efforts to run OpenMP on software distributed shared memory systems, to translate OpenMP into MPI and to extend OpenMP for non-shared memory systems.
  - For compilation in GCC using fopenmp:

```
$ gcc -fopenmp hello.c -o hello -ldl
```

- **What is Parallel Reduction?**
  - Reduction : specifies that 1 or more variables that are private to each thread are subject of reduction operation at end of parallel region.

    reduction(operation:var)

    where, operation is operator to perform on the variables (var) at the end of the parallel region and Var is one or more variables on which to perform scalar reduction.

- **Min Operation**
  - The function takes in a vector of integers as input and finds the minimum value in the vector using parallel reduction.
  - The OpenMP reduction clause is used with the "min" operator to find the minimum value across all threads.
  - The minimum value found by each thread is reduced to the overall minimum value of the entire array.
  - The final minimum value is printed to the console.

- **Max Operation**
  - The function takes in a vector of integers as input and finds the maximum value in the vector using parallel reduction.
  - The OpenMP reduction clause is used with the "max" operator to find the maximum value across all threads.
  - The maximum value found by each thread is reduced to the overall maximum value of the entire array.
  - The final maximum value is printed to the console.

- **Sum Operation**
  - The function takes in a vector of integers as input and finds the sum of all the values in the vector using parallel reduction.
  - The OpenMP reduction clause is used with the "+" operator to find the sum across all threads.
  - The sum found by each thread is reduced to the overall sum of the entire array.
  - The final sum is printed to the console.

- **Average Operation**
  - The function takes in a vector of integers as input and finds the average of all the values in the vector using parallel reduction.
  - The OpenMP reduction clause is used with the "+" operator to find the sum across all threads.
  - The function takes in a vector of integers as input and finds the average of all the values in the vector using parallel reduction.
  - The OpenMP reduction clause is used with the "+" operator to find the sum across all threads.

- **Main Function**
  - The function takes in a vector of integers as input and finds the average of all the values in the vector using parallel reduction.

- The OpenMP reduction clause is used with the "+" operator to find the sum across all threads.

==========

Conclusion:-

Thus we have studied Min, Max, Sum and Average operation using parallel reduction in C++ with OpenMP. Also, Parallel reduction is a powerful technique that allows us to perform these operation on large arrays more efficiently by dividing the work among multiple threads running in parallel.

==========

GROUP – A

Assignment No: 4

==========

Title: - Write a CUDA Program for -

1. Addition of two large vectors
2. Matrix Multiplication using CUDA C.

==========

Objective:-

-To learn about CUDA

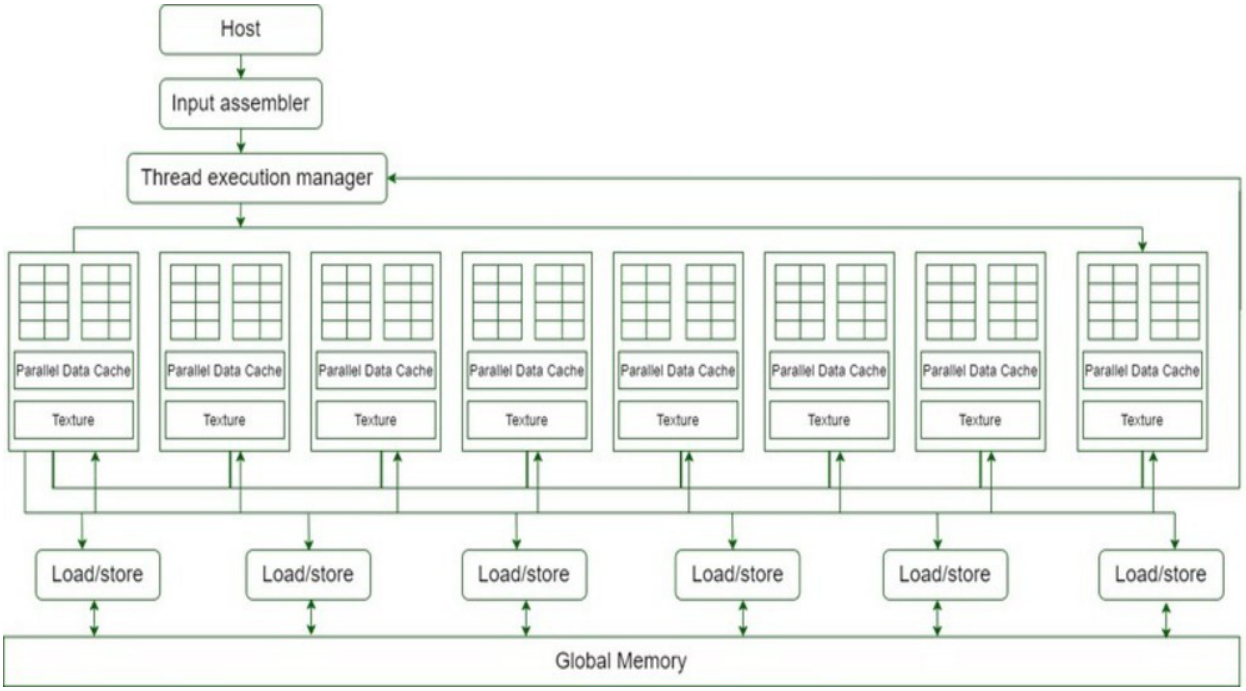
-To understand the concept of vector addition and matrix multiplication using CUDA

==========

Theory:-

- **What is CUDA?**
  - NVIDIA's CUDA is a general purpose parallel computing platform and programming model that accelerates deep learning and other compute-intensive apps by taking advantage of the parallel processing power of GPUs.
  - CUDA enables developers to speed up compute-intensive applications by harnessing the power of GPUs for the parallelizable part of the computation.
  - In 2003, a team of researchers led by Ian Buck unveiled Brook, the first widely adopted programming model to extend C with data-parallel constructs. Buck later joined NVIDIA and led the launch of CUDA in 2006, the first commercial solution for general purpose computing on GPUs.
  - Within the supported CUDA compiler, any piece of code can be run on GPU by using the **__global__** keyword. Programmers must change their malloc/new and free/delete calls as per the CUDA guidelines so that appropriate space could be allocated on GPU. Once the computation on GPU is finished, the results are synchronized and ported to the CPU.

- Architecture of CUDA



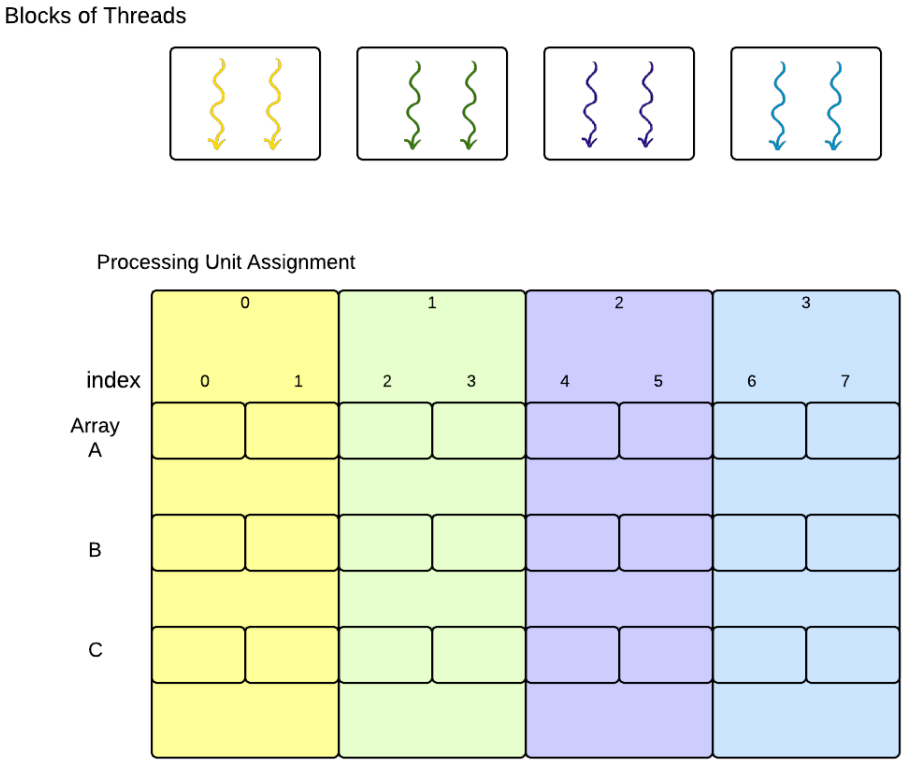
  - GPUs run one kernel (a group of tasks) at a time.
  - Each kernel consists of blocks, which are independent groups of ALUs.
  - Each block contains threads, which are levels of computation.
  - The threads in each block typically work together to calculate a value.
  - Threads in the same block can share memory.
  - In CUDA, sending information from the CPU to the GPU is often the most typical part of the computation.
  - For each thread, local memory is the fastest, followed by shared memory, global, static, and texture memory the slowest.

- For compilation in CUDA using nvcc. The CUDA compiler is called nvcc. This will exist on your machine if you have installed the CUDA development toolkit. Program name must be save with ".cu" extension.

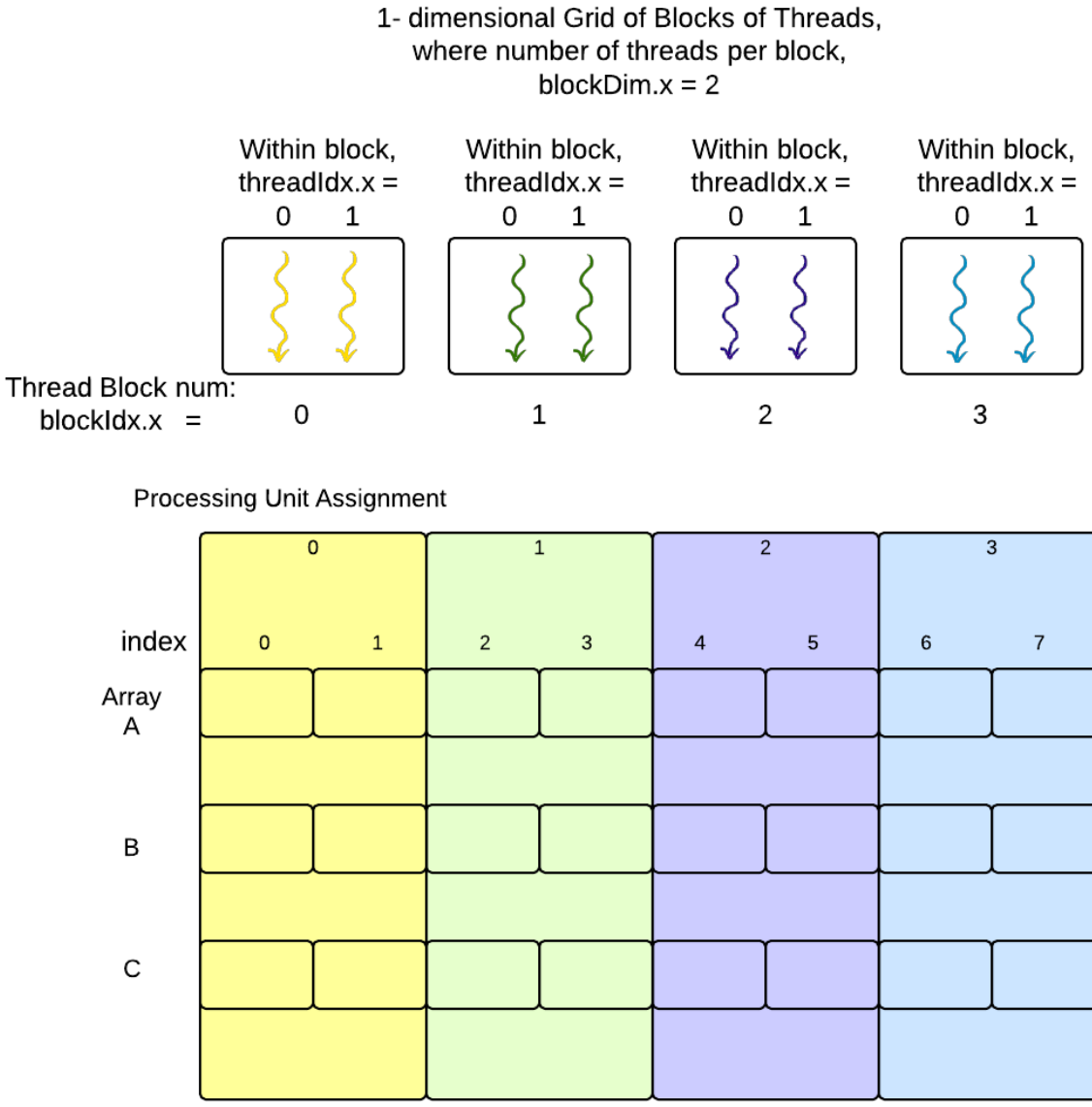
```
$ nvcc filename.cu
$ ./filename
```

- **What is vector?**
  - A vector is a quantity which has both magnitude and direction. A vector quantity, unlike scalar, has a direction component along with the magnitude which helps to determine the position of one point relative to the other.
  - the length of the segment of the directed line is called the magnitude of a vector and the angle at which the vector is inclined shows the direction of the vector.

- **How to do vector addition?**
  - CUDA convention has been to depict treads as scribbly lines with arrow heads, as shown below. In this case, the 4 blocks of threads that form the 1-dimensional grid become analogous to the processing units that we would assign to the portions of the array that would be run in parallel. In contrast to OpenMP and MPI, however, the individual threads within the blocks would each work on one data element of the array.



  - In CUDA programs, we always determine which thread is running and use that to determine what portion of data to work on. The mapping of work is up to the programmer.
  - to variables supplied by the CUDA library that help us define just what thread is executing. The following diagram shows how three of the available variables, corresponding to the grid, blocks, and threads within the blocks would be assigned for our above example decomposition:



- The three variables that we can access in CUDA code for this example shown above are:
  1. **threadIdx.x -** represents a thread's index along the x dimension within the block.
  2. **blockIdx.x** - represents a thread's block's index along the x dimension within the grid.
  3. **blockDim.x** - represents the number of threads per block in the x direction.
- In our simple example, there are a total of eight threads executing, each one numbered from 0 through 7. Each thread executing code on the GPU can determine which thread it is by using the above variables like this:

```
int tid = blockDim.x * blockIdx.x + threadIdx.x;
```

- In CUDA programming, the primary motherboard with the CPU is referred to as the host and the GPU co-processor is usually called the device. The GPU device has separate memory and different circuitry for executing instructions. Code to be executed on the GPU must be compiled for its instrution set.
- The overall structure of a CUDA program that uses the GPU for computation is as follows:
  1. Define the the code that will run on the device in a separate function, called the kernel function.
  2. In the main program running on the host's CPU:
     a) allocate memory on the host for the data arrays.
     b) initialze the data arrays in the host's memory.
     c) allocate separate memory on the GPU device for the data arrays.
     d) copy data arrays from the host memory to the GPU device memory.
  3. On the GPU device, execute the kernel function that computes new data values given the original arrays. Specify how many blocks and threads per block to use for this computation.
  4. After the kernel function completes, copy the computed values from the GPU device memory back to the host's memory.

- **How to do matrix multiplication using CUDA?**
  - 2D matrices can be stored in the computer memory using two layouts − row-major and column-major.
    1. Column Major

M0,0	M0,1	M0,2	M0,3
M1,0	M1,1	M1,2	M1,3
M2,0	M2,1	M2,2	M2,3
M3,0	M3,1	M3,2	M3,3

    2. Row Major

M0,0	M1,0	M2,0	M3,0	M0,1	M1,1	M2,1	M3,1	M0,2	M1,2	M2,2	M3,2	M0,3	M1,3	M2,3	M3,3

In row-major layout, element(x,y) can be addressed as: x*width + y. In the above example, the width of the matrix is 4. For example, element (1,1) will be found at position − 1*4 + 1 = 5 in the 1D array.

- Mapping each data element to a thread. The following mapping scheme is used to map data to thread. This gives each thread its unique identity.

```
row=blockIdx.x*blockDim.x+threadIdx.x;
col=blockIdx.y*blockDim.y+threadIdx.y;
```

- a grid is made-up of blocks, and that the blocks are made up of threads. All threads in the same block have the same block index.
- Matrix multiplication between a (IxJ) matrix d_M and (JxK) matrix d_N produces a matrix d_P with dimensions (IxK). The formula used to calculate elements of d_P is –

$$d\_P_{x,y} = \sum d\_M_{x,,k} * d\_N_{k,y}, \text{ for } k=0,1,2,....width$$

- A d_P element calculated by a thread is in 'blockIdx.y*blockDim.y+threadIdx.y' row and 'blockIdx.x*blockDim.x+threadIdx.x' column. Here is the actual kernel that implements the above logic.

```
__global__ void simpleMatMulKernell(float* d_M, float* d_N, float* d_P, int width)
{

}
```

- This helps to calculate row and col to address what element of d_P will be calculated by this thread.

```
int row = blockIdx.y*width+threadIdx.y;
int col = blockIdx.x*width+threadIdx.x;
```

- Matrix Multiplication:

```
if(row<width && col <width) {
 float product_val = 0
for(int k=0;k<width;k++) {
   product_val += d_M[row*width+k]*d_N[k*width+col];
   }
d_p[row*width+col] = product_val;
}
```

==========

Conclusion:-

Thus we have studied vector addition and matrix multiplication using CUDA in C.

==========