Laboratory One - Lab Manual

Computer Engineering (Savitribai Phule Pune University)

Assignment No.1

Title: 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.

Objective: To study and implementation of directive based parallel programming model. 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 understand the implementation of sequential program augmented with compiler directives to specify parallelism. Students will 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 Tutorials.

Hardware Specification: x86_64 bit, 2 – 2/4 GB DDR RAM, 80 - 500 GB SATA HD, 1GB NIDIA TITAN X Graphics Card.

Software Specification: Ubuntu 14.04, GPU Driver 352.68, CUDA Toolkit 8.0, CUDNN Library v5.0

Theory:

Introduction:

OpenMP:

OpenMP is a set of C/C++ pragmas (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.

Parallel Programming:

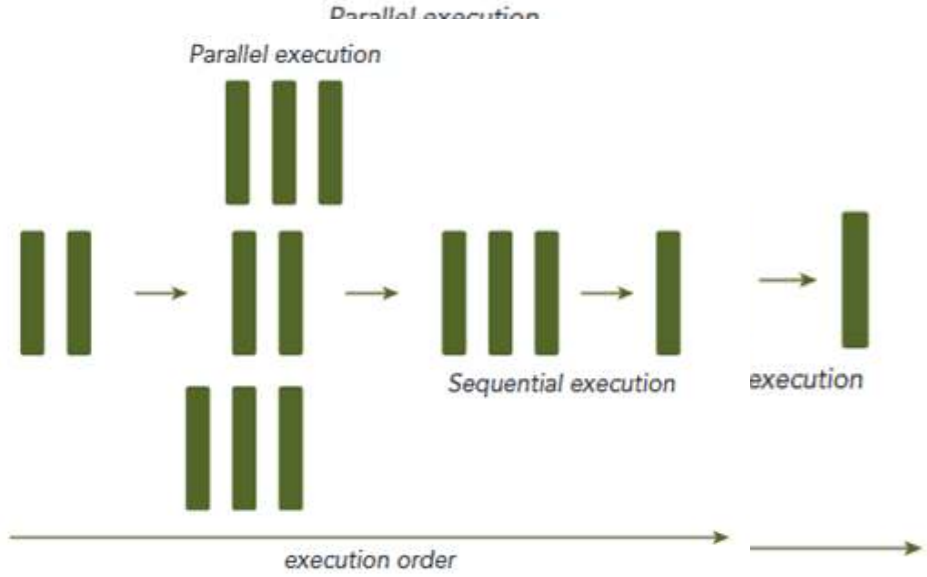
There are two fundamental types of parallelism in applications:

➤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 exe- cuted 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.

Conclusion: We have implemented parallel reduction using Min, Max, Sum and Average Operations. We have implemented CUDA program for calculating Min, Max, Arithmetic mean and Standard deviation Operations on N-element vector.

Assignment NO: 02

Title: Design & implementation of Parallel (CUDA) algorithm to Add two large Vector, Multiply Vector and Matrix and Multiply two N × N arrays using n^2.

Outcome: To offload parallel computations to the graphics card, when it is appropriate to do so, and to give some idea of how to think about code running in the massively parallel environment presented by today's graphics cards.

Outcome: Students should understand the basic of GPU computing in the CUDA environment.

Prerequisites: Strassen's Matrix Multiplication Algorithm and CUDA Tutorials.

Hardware Specification: x86_64 bit, 2 – 2/4 GB DDR RAM, 80 - 500 GB SATA HD, 1GB NIDIA TITAN X Graphics Card.

Software Specification: Ubuntu 14.04, GPU Driver 352.68, CUDA Toolkit 8.0, CUDNN Library v5.0

Introduction:

It has become increasingly common to see supercomputing applications harness the massive parallelism of graphics cards (Graphics Processing Units or GPUs) to speed up computations. One platform for doing so is NVIDIA's Compute Unified Device Architecture (CUDA). We use the example of Matrix Multiplication to introduce the basics of GPU computing in the CUDA environment.

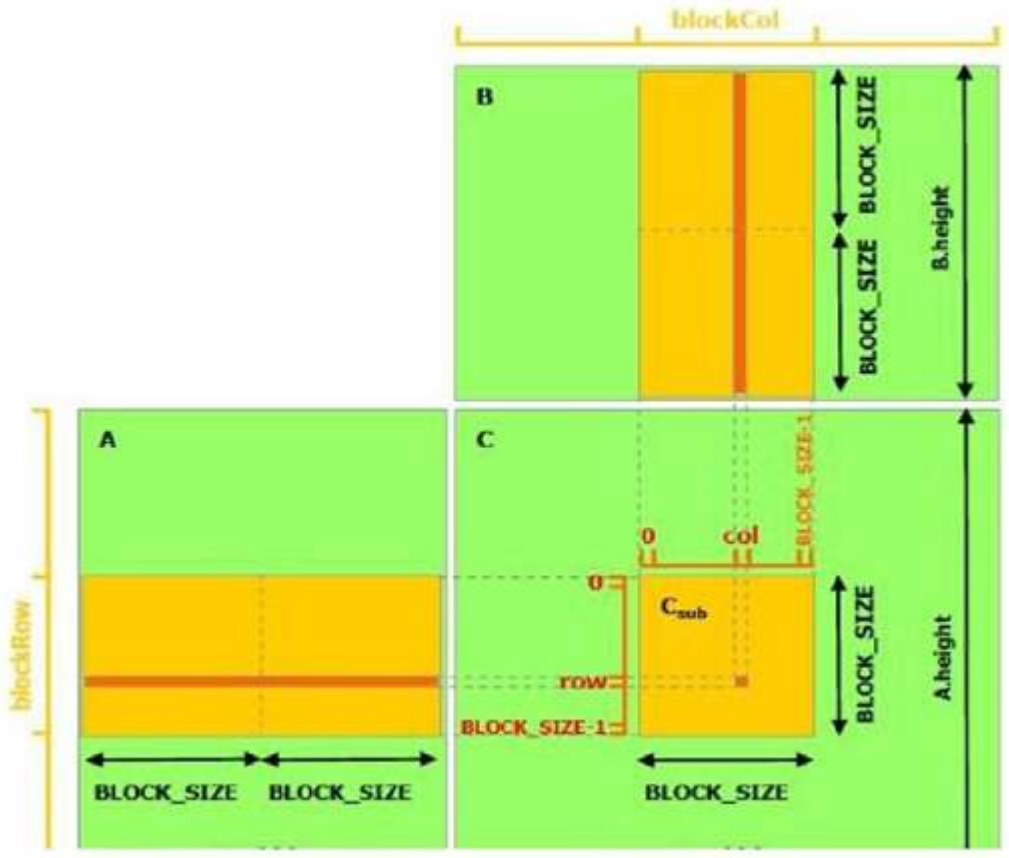
Matrix Multiplication is a fundamental building block for scientific computing. Moreover, the algorithmic patterns of matrix multiplication are representative. Many other algorithms share similar optimization techniques as matrix multiplication. Therefore, matrix multiplication is one of the most important examples in learning parallel programming.

A kernel that allows host code to offload matrix multiplication to the GPU. The kernel function is shown below,

```
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C) {
  // Each thread computes one element of C
  // by accumulating results into Cvalue
  float Cvalue = 0;
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  if(row > A.height || col > B.width) return;
  for (int e = 0; e < A.width; ++e)
    Cvalue += A.elements[row * A.width + e] *
                          B.elements[e * B.width + col];
  C.elements[row * C.width + col] = Cvalue;
}
```

The first line contains the **__global__** keyword declaring that this is an entry-point function for running code on the device. The declaration float **Cvalue = 0** sets aside a register to hold this float value where we will accumulate the product of the row and column entries. The next two lines help the thread to discover its row and column within the matrix. It is a good idea to make sure you understand those two lines before moving on. The ***if*** statement in the next line terminates the thread if its row or column place it outside the bounds of the product matrix. This will happen only in those blocks that overhang either the right or bottom side of the matrix.
The next three lines loop over the entries of the row of **A** and the column of **B** (these have the same size) needed to compute the (row, col)-entry of the product, and the sum of these products are accumulated in the **Cvalue** variable. Matrices **A** and **B** are stored in the device's global memory in row major order, meaning that the matrix is stored as a one-dimensional array, with the first row followed by the second row, and so on. Thus to find the index in this linear array of the **(i, j)** entry of matrix **A.**
Finally, the last line of the kernel copies this product into the appropriate element of the product matrix C, in the device's global memory.

In light of the memory hierarchy described above, each threads loads **(2 × A. width)** elements in Kernel .



From global memory — two for each iteration through the loop, one from matrix **A** and one from matrix **B**. Since accesses to global memory are relatively slow, this can bog down the kernel, leaving the threads idle for hundreds of clock cycles, for each access.
Matrix **A** is shown on the left and matrix **B** is shown at the top, with matrix **C**, their product, on the bottom-right. This is a nice way to lay out the matrices visually, since each element of **C** is the product of the row to its left in **A** and the column above it in **B**. In the above figure, square thread blocks of dimension **BLOCK_SIZE** × **BLOCK_SIZE** and will assume that the dimensions of **A** and **B** are all multiples of **BLOCK_SIZE**. Again, each thread will be responsible for computing one element of the product matrix **C**.

It decomposes matrices **A** and **B** into non-overlapping submatrices of size **BLOCK_SIZE** × **BLOCK_SIZE**. It shows in above figure in red row and red column. It passes through the same number of these submatrices, since they are of equal length. If it load the left- most of those submatrices of matrix **A** into shared memory, and the top-most of those submatrices of matrix **B** into shared memory, then it compute the first **BLOCK_SIZE** products and add them together just by reading the shared memory. But here is the benefit, as long as it have those submatrices in shared memory, every thread's thread block (computing the BLOCK_SIZE × BLOCK_SIZE submatrix of C) can compute that portion of their sum as well from the same data in shared memory. When each thread has computed this sum, it loads the next **BLOCK_SIZE** × **BLOCK_SIZE** submatrices from **A** and **B**, and continue adding the term-by-term products to our value in **C**.

Applications:

1. Fast Video Trans-coding & Enhancement.
2. Medical Imaging.
3. Neural Networks.
4. Gate-level VLSI Simulation.

Conclusion: Strassen's Matrix Multiplication Algorithm have been implemented parallel using GPU computing in the CUDA environment

Assignment NO: 03

Title: Design & implementation of Parallel (OpenMP) Sorting algorithms (Bubble Sort & Merge Sort) based on existing sequential algorithms.

Objectives: To offload parallel computations to the graphics card, when it is appropriate to do so, and to give some idea of how to think about code running in the massively parallel environment presented by today's graphics cards.

Outcome: Students should understand the basic of OpenMP Programming Module using existing Sorting techniques.

Prerequisites: Data Structure and Files, Design and Analysis of Algorithm, OpenMP Tutorials. **Hardware Specification:** x86_64 bit, 2 – 2/4 GB DDR RAM, 80 - 500 GB SATA HD. **Software Specification:** Ubuntu 14.04, G Edit, GCC Complier.

Introduction:

An Application Program Interface **(API)** that may be used to explicitly direct multithreaded, shared memory parallelism. An Open Multi-Processing (**OpenMP)** consists of a set of compiler ***#pragmas*** that control how the program works. The **pragmas** are designed so that even if the compiler does not support them, the program will still yield correct behavior, but without any parallelism. Sorting is the process of putting data in order; either numerically or alphabetically. It is necessary to arrange the elements in an array in numerical or lexicographical order, sorting numerical values in descending order or ascending order and alphabetical value like addressee key. Many existing sorting algorithms were observed in terms of the efficiency of the algorithmic complexity. All sorting algorithms are appropriate for specific kinds of problems. Some sorting algorithms work on less number of elements, some are used for huge number of data, some are used if the list has repeated values, and some are suitable for floating point numbers. Sorting is used in many important applications and there have been a plenty of performance analysis.

There are several sorting algorithms available to sort the elements of an array. Some of the sorting algorithms are,

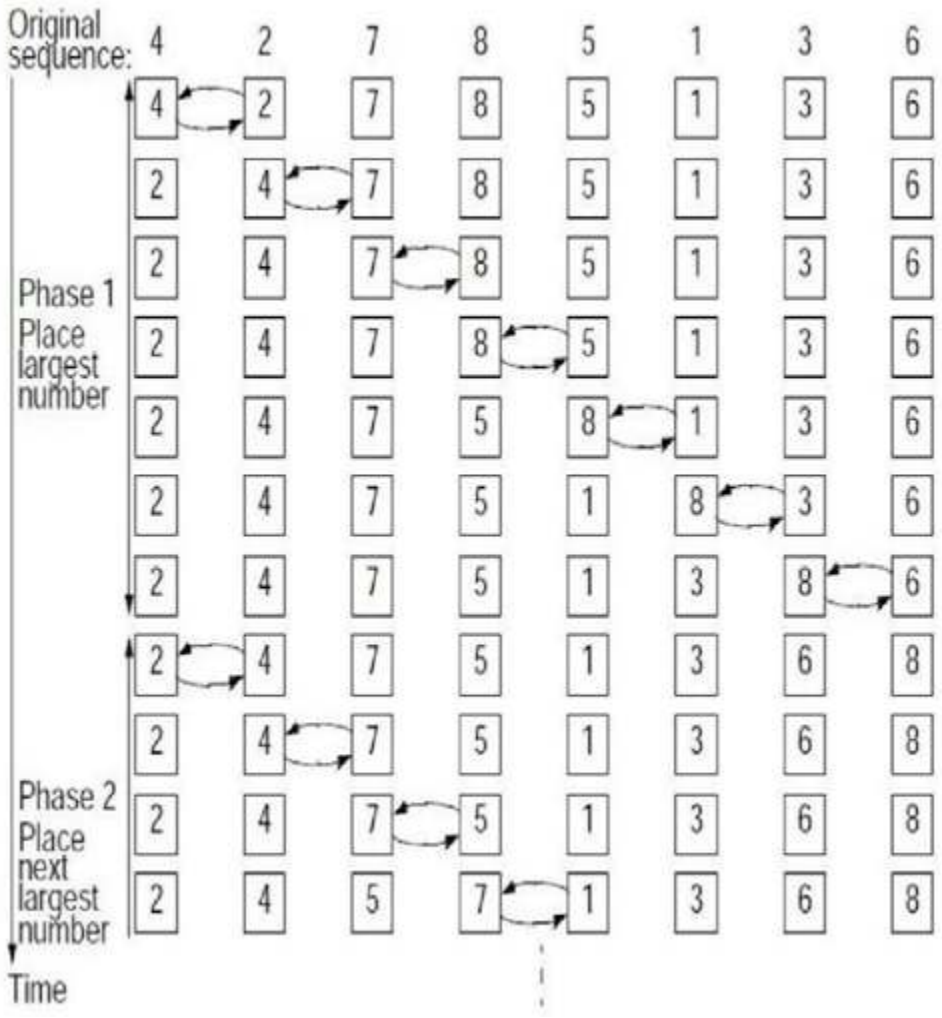
A. **Bubble Sort**

B. **Merge Sort.**

Introduction:

A. Bubble Sort:

In bubble sort, the largest number is first moved to the very end of the list by a series of compare-and-exchange operations, starting at the opposite end. The procedure repeats, stopping just before the previously positioned largest number, to get the next-largest number. In this way, the larger numbers move (like a bubble) toward the end of the list.

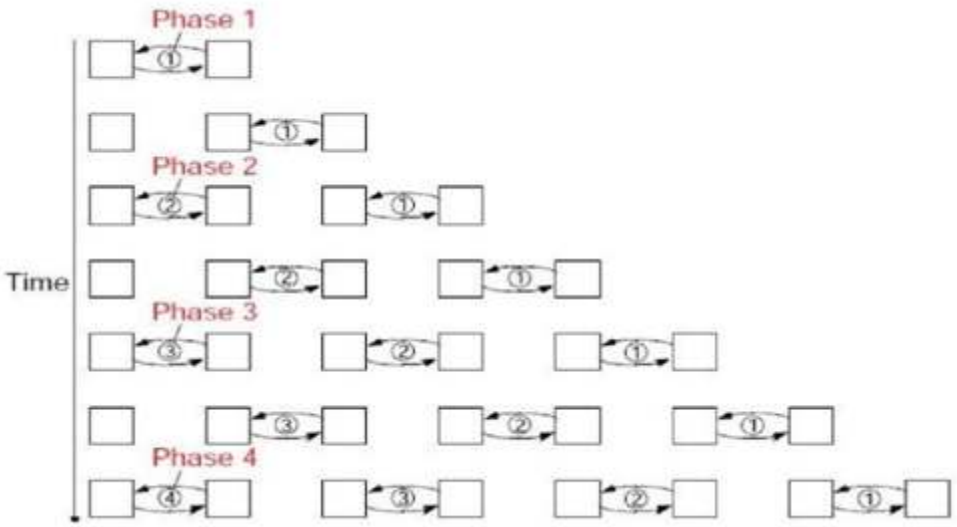



```
for (i = N - 1; i > 0; i--)
for (j = 0; j < i; j++) { k = j + 1;
if (a[j] > a[k]) { temp = a[j];
a[j] = a[k];
a[k] = temp;
}}
```

Parallel Bubble Sort

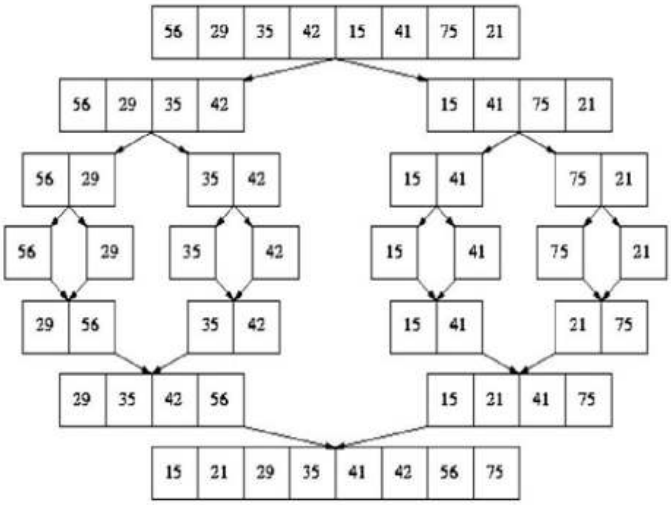
A possible idea is to run multiple iterations in a pipeline fashion, i.e., start the bubbling action of the next iteration before the preceding iteration has finished in such a way that it does not overtake it.

Introduction:



B. Merge Sort

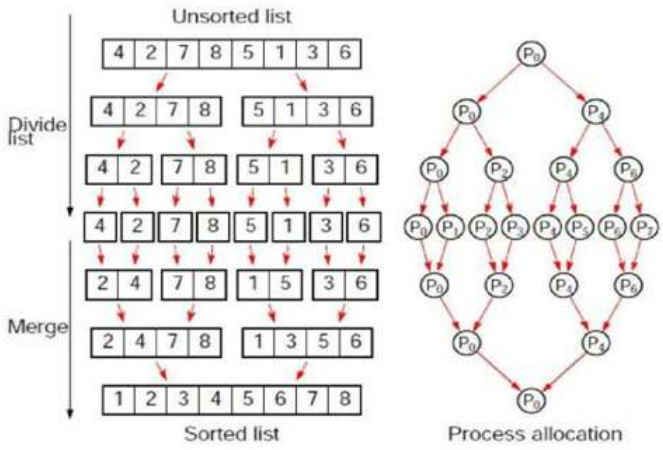
Merge Sort is a classical sorting algorithm using a divide-and-conquer approach. The initial unsorted list is first divided in half, each half sub list is then applied the same division method until individual elements are obtained. Pairs of adjacent elements / sub lists are then merged into sorted sub lists until the one fully merged and sorted list is obtained.



Introduction:

Parallel Merge Sort

The idea is to take advantage of the tree structure of the algorithm to assign work to processes. If communication time ignore, computations still only occur when merging the sub lists. But now, in the worst case, it takes ***2s − 1*** steps to merge all sub lists (two by two) of size s in a merging step



Applications: 1. Bubble sort is used in programming TV remote to sort channels on the basis of longer viewing time.
2. Databases use an external merge sort to sort set of data that are too large to be loaded entirely into memory.

Conclusion: We have tested both the Sorting algorithms for different number elements with respective to time. As the number of elements increased time required for OpenMP is reduced.

Assignment NO: 04

Title: Design & implementation of Parallel (OpenMP) Searching algorithms for Binary search for Sorted Array and 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).

Objectives: To offload parallel computations to the graphics card, when it is appropriate to do so, and to give some idea of how to think about code running in the massively parallel environment presented by today's graphics cards.

Outcome: Students should understand the basic of OpenMP Programming Module using exiting Searching techniques.

Prerequisites: Data Structure and Files, Design and Analysis of Algorithm, OpenMP Tutorials.

Hardware Specification: x86_64 bit, 2 – 2/4 GB DDR RAM, 80 - 500 GB SATA HD.

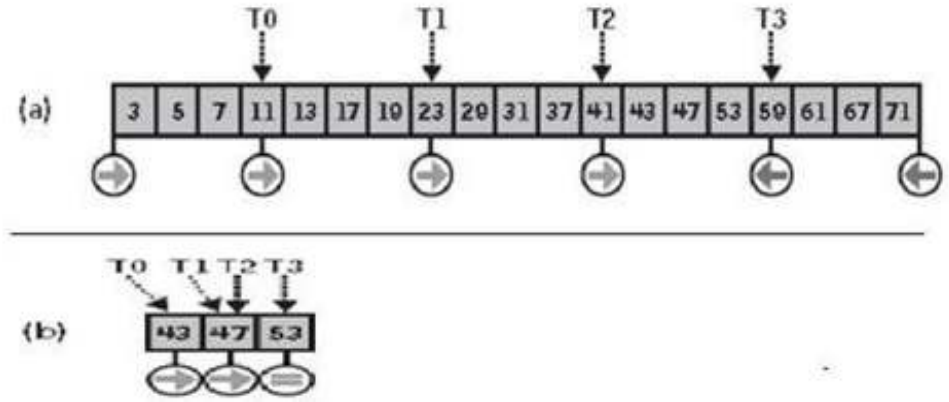
Software Specification: Ubuntu 14.04, G Edit, GCC Complier.

Introduction:

An Application Program Interface **(API)** that may be used to explicitly direct multithreaded, shared memory parallelism. An Open Multi-Processing (**OpenMP)** consists of a set of compiler ***#pragmas*** that control how the program works. The **pragmas** are designed so that even if the compiler does not support them, the program will still yield correct behavior, but without any parallelism. Searching identifies N well-spaced points within the search array bounds and compares the key of the corresponding records to the search key. Each thread does one of the N comparisons. There are three possible outcomes from these comparisons. The first is that the item of interest is found and the search is complete; the second is that the item key examined is less than the search key; the third is that the item key examined is greater than the search key. If no search key match is found, a new, smaller search array is defined by the two consecutive index points whose record keys were found to be less than the search key and greater than the search key.

The N-ary search is then performed on this refined search array. As with the serial version of binary search, the process is repeated until a match is found or the number of items in the search array is zero.

Given the sorted a rray of prime numbers in Figure (a), let's say we want to determine whether the value 53 is in the array and where it can be found. If there are four threads (T0 to T3), each computes an index into the array and compares the key value found there to the search key. Threads T0, T1, and T2 all find that the key value at the examined position is less than the search key value. Thus, an item with the matching key value must lie somewhere to the right of each of these thread's current search positions (indicated by the circled arrows).



Thread T3 determines that the examined key value is greater than the search key, and the matching key can be found to the left of this thread's search position (left-pointing circled arrow). Notice the circled arrows at each end of the array. These are attached to "phantom" elements just outside the array bounds. The results of the individual key tests define the subarray that is to become the new search array. Where we find two consecutive test results with opposite outcomes, the corresponding indexes will be just outside of the lower and upper bounds of the new search array.

Figure (a) shows that the test results from threads T2 (less than search key) and T3 (greater than search key) are opposite. The new search array is the array elements between the elements tested by these two threads.

Figure (b) shows this sub array and the index positions that are tested by each thread. The figure shows that during this second test of element key values, thread T3 has found the element that matches the search key.

Consider the case, where find a composite value, like 52, in a list of prime numbers. The individual key results by the four threads shown in Figure (a) would be the same. The sub array shown in Figure (b) would have the same results, except that the test by T3 would find that the key value in the assigned position was greater than the search key (and the equals sign would be a left-pointing arrow). The next round of key comparisons by threads would be from a sub array with no elements, bounded by the array slots holding the key values of 47 and 53. When threads are confronted with the search of an empty search space, then the key is not to be found.

This is obviously more complex than a simple binary search. The algorithm must coordinate the choices of index positions that each thread needs to test, keep and store the results of each test such that multiple threads can examine those results, and compute the new search array bounds for the next round of key tests. From this quick description, it's clear that it needs some globally accessible data and, more importantly, it needs a barrier between the completion of the key tests and the examination of the results of those tests.

Applications: 1. Information retrieval in the required format is the central activity in all computer applications.

2. Searching methods are designed to take advantage of the file organization and

optimize the search for a particular record or to establish its absence.

Conclusion: We have tested both the Searching algorithms for different number elements with respective to time. As the number of elements increased time required for OpenMP is reduced.

Assignment NO: 05

Title: Design & implementation of Parallel (OpenMP) algorithm for K - Nearest Neighbors Classifier.

Objectives: To offload parallel computations to the graphics card, when it is appropriate to do so, and to give some idea of how to think about code running in the massively parallel environment presented by today's graphics cards.

Outcome: Students should understand the basic of OpenMP Programming Module using exiting K - Nearest Neighbors Classifier.

Prerequisites: Data Structure and Files, Design and Analysis of Algorithm, OpenMP Tutorials.

Hardware Specification: x86_64 bit, 2 – 2/4 GB DDR RAM, 80 - 500 GB SATA HD.

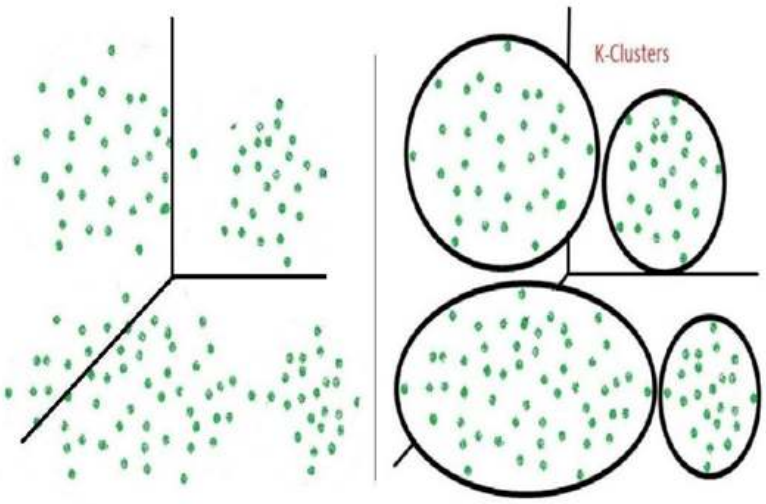
Software Specification: Ubuntu 14.04, G Edit, GCC Complier.

Introduction:

An Application Program Interface **(API)** that may be used to explicitly direct multithreaded, shared memory parallelism. An Open Multi-Processing (**OpenMP)** consists of a set of compiler ***#pragmas*** that control how the program works. The **pragmas** are designed so that even if the compiler does not support them, the program will still yield correct behavior, but without any parallelism.

K-means is a popular clustering method because it is simple to program and is easy to compute on large samples . The k-means clustering algorithm classifies N vectors with d dimension into k clusters by assigning each vector to the cluster whose average value is nearest to it by some distance measure on that set. The algorithm computes iteratively, until reassigning points and recompiling averages reaches optimal value. Initially given k, total number of clusters, the training samples are assigned to k clusters randomly. Alternately the first k training sample can be considered as single-element cluster and then each of the remaining (N-k) training samples is assigned to the nearest cluster.

After each assignment, the cluster centers are recomputed as referred as centroid of the cluster. Again each sample is taken in a sequence and computed its distance from the centroid If a sample is not currently in the cluster with the closest centroid, switch this sample to that cluster and update the centroid of the cluster gaining the new sample and the cluster losing the sample.



Repeat step 3 until convergence is achieved, that is until sample causes no new assignments. We calculate the distance to all centroid and get the minimum distance. At the termination of the K means algorithms all the ponts will belong to a cluster with shortest distance from its centroid. Above Figure, shows the clusters formed with number of points and 4 clusters.

Conclusion: We have tested both the Searching algorithms for K - clusters with respective to time. As the number of Clusters increased time required for OpenMP is reduced.

Applications:

1. In Wireless Sensor Network's based Application.
2. In Search Engines and Drug Activity Prediction.