Sir Visvesvaraya Institute of Technology,

Nashik

Department Of Computer Engineering

BE COMPUTER (2019 COURSE)

LAB MANUAL

Semester – VIII

410250 : High Performance Computing

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: Students should be able to Write a program to implement Parallel BFS and DFS using a Tree or an undirected graph.

Prerequisite:

1. Basic of programming language
2. Concept of BFS and DFS
3. Concept of Parallelism

Theory:

Concept of OpenMP

● OpenMP (Open Multi-Processing) is an application programming interface (API) that supports shared-memory parallel programming in C, C++, and Fortran. It is used to write parallel programs that can run on multicore processors, multiprocessor systems, and parallel computing clusters.

● OpenMP provides a set of directives and functions that can be inserted into the source code of a program to parallelize its execution. These directives are simple and easy to use, and they can be applied to loops, sections, functions, and other program constructs. The compiler then generates parallel code that can run on multiple processors concurrently.

1] What is BFS?

Breadth-first search is a graph traversal algorithm that starts traversing the graph from the root node and explores all the neighboring nodes. Then, it selects the nearest node and explores all the unexplored nodes. While using BFS for traversal, any node in the graph can be considered as the root node. It is a recursive algorithm to search all the vertices of a tree or graph data structure. BFS puts every vertex of the graph into two categories - visited and non-visited.

Algorithm :

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Enqueue the starting node A and set its STATUS = 2 (waiting state)

Step 3: Repeat Steps 4 and 5 until QUEUE is empty

Step 4: Dequeue a node N. Process it and set its STATUS = 3 (processed state).

Step 5: Enqueue all the neighbours of N that are in the ready state (whose STATUS = 1) and set

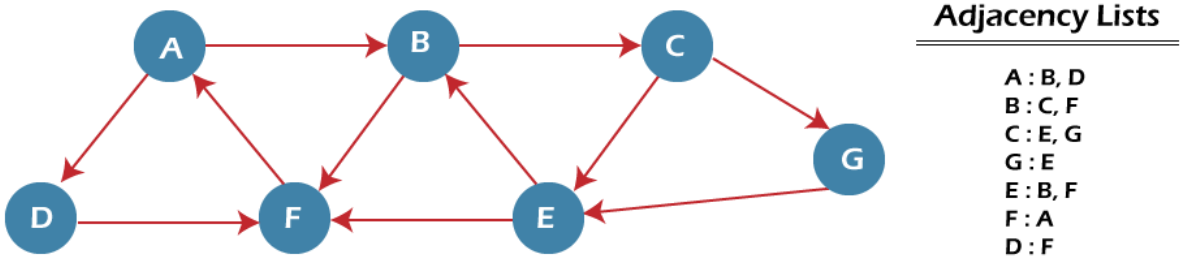
their STATUS = 2

(waiting state)

[END OF LOOP]

Step 6: EXIT

Example of BFS algorithm :



In the above graph, minimum path 'P' can be found by using the BFS that will start from Node A and end at Node E. The algorithm uses two queues, namely QUEUE1 and QUEUE2. QUEUE1 holds all the nodes that are to be processed, while QUEUE2 holds all the nodes that are processed and deleted from QUEUE1.

Now, let's start examining the graph starting from Node A.

Step 1 - First, add A to queue1 and NULL to queue2.

1. QUEUE1 = {A}
2. QUEUE2 = {NULL}

Step 2 - Now, delete node A from queue1 and add it into queue2. Insert all neighbors of node A to queue1.

1. QUEUE1 = {B, D}
2. QUEUE2 = {A}

Step 3 - Now, delete node B from queue1 and add it into queue2. Insert all neighbors of node B to queue1.

1. QUEUE1 = {D, C, F}
2. QUEUE2 = {A, B}

Step 4 - Now, delete node D from queue1 and add it into queue2. Insert all neighbors of node D to queue1. The only neighbor of Node D is F since it is already inserted, so it will not be inserted again.

1. QUEUE1 = {C, F}
2. QUEUE2 = {A, B, D}

Step 5 - Delete node C from queue1 and add it into queue2. Insert all neighbors of node C to queue1.

1. QUEUE1 = {F, E, G}
2. QUEUE2 = {A, B, D, C}

Step 5 - Delete node F from queue1 and add it into queue2. Insert all neighbors of node F to queue1. Since all the neighbors of node F are already present, we will not insert them again.

1. QUEUE1 = {E, G}
2. QUEUE2 = {A, B, D, C, F}

Step 6 - Delete node E from queue1. Since all of its neighbors have already been added, so we will not insert them again. Now, all the nodes are visited, and the target node E is encountered into queue2.

1. QUEUE1 = {G}
2. QUEUE2 = {A, B, D, C, F, E}

Program:

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

2] What is DFS?

It is a recursive algorithm to search all the vertices of a tree data structure or a graph. The depth-first search (DFS) algorithm starts with the initial node of graph G and goes deeper until we find the goal node or the node with no children. Because of the recursive nature, stack data structure can be used to implement the DFS algorithm.

Algorithm

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Push the starting node A on the stack and set its STATUS = 2 (waiting state)

Step 3: Repeat Steps 4 and 5 until STACK is empty

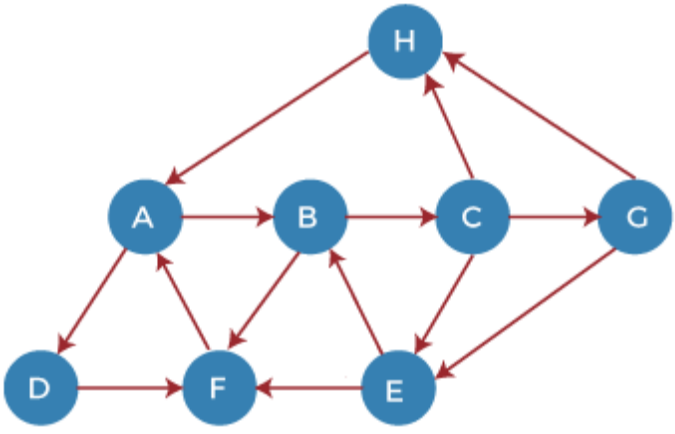
Step 4: Pop the top node N. Process it and set its STATUS = 3 (processed state)

Step 5: Push on the stack all the neighbors of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)

[END OF LOOP]

Step 6: EXIT

Example of DFS algorithm



Adjacency Lists

A : B, D
B : C, F
C : E, G, H
G : E, H
E : B, F
F : A
D : F
H : A

Step 1 - First, push H onto the stack.

1. STACK: H

Step 2 - POP the top element from the stack, i.e., H, and print it. Now, PUSH all the neighbors of H onto the stack that are in ready state.

1. Print: H]STACK: A

Step 3 - POP the top element from the stack, i.e., A, and print it. Now, PUSH all the neighbors of A onto the stack that are in ready state.

1. Print: A
2. STACK: B, D

Step 4 - POP the top element from the stack, i.e., D, and print it. Now, PUSH all the neighbors of D onto the stack that are in ready state.

1. Print: D
2. STACK: B, F

Step 5 - POP the top element from the stack, i.e., F, and print it. Now, PUSH all the neighbors of F onto the stack that are in ready state.

1. Print: F
2. STACK: B

Step 6 - POP the top element from the stack, i.e., B, and print it. Now, PUSH all the neighbors of B onto the stack that are in ready state.

1. Print: B
2. STACK: C

Step 7 - POP the top element from the stack, i.e., C, and print it. Now, PUSH all the neighbors of C onto the stack that are in ready state.

1. Print: C
2. STACK: E, G

Step 8 - POP the top element from the stack, i.e., G and PUSH all the neighbors of G onto the stack that are in ready state.

1. Print: G
2. STACK: E

Step 9 - POP the top element from the stack, i.e., E and PUSH all the neighbors of E onto the stack that are in ready state.

1. Print: E
2. STACK:

Now, all the graph nodes have been traversed, and the stack is empty.

Program:

```
#include <iostream>
#include <vector>
#include <stack>
#include <omp.h>
using namespace std;

const int MAX = 100000;
vector<int> graph[MAX];
bool visited[MAX];

void dfs(int node) {
        stack<int> s;
        s.push(node);
        while (!s.empty()) {
        int curr_node = s.top();
        s.pop();

        if (!visited[curr_node]) {
        visited[curr_node] = true;

        if (visited[curr_node]) {
        cout << curr_node << " ";
```

```
        }
        #pragma omp parallel for
        for (int i = 0; i < graph[curr_node].size(); i++) {
                int adj_node = graph[curr_node][i];
                if (!visited[adj_node]) {
                s.push(adj_node);
                }
        }
        }
        }
}

int main() {
        int n, m, start_node;
        cout << "Enter No of Node,Edges,and start node:" ;
        cin >> n >> m >> start_node;
       //n: node,m:edges

cout << "Enter Pair of edges:" ;
        for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
//u and v: Pair of edges
        graph[u].push_back(v);
        graph[v].push_back(u);
        }

        #pragma omp parallel for
```

```
        for (int i = 0; i < n; i++) {
        visited[i] = false;
        }
        dfs(start_node);

/*      for (int i = 0; i < n; i++) {
        if (visited[i]) {
        cout << i << " ";
        }
        }*/
        return 0;
}
```

Conclusion: In this way we can implement BFS and DFS using the concept of OpenMP with Tree or an undirected graph.

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: Students should be able to Write a program to implement Parallel Bubble Sort and Merge Sort and can measure the performance of sequential and parallel algorithms.

Prerequisite:

1. Basic of programming language

2. Concept of Bubble Sort and Merge Sort

3. Concept of Parallelism

Theory :

1] What is Bubble Sort?

Bubble Sort is a simple sorting algorithm that works by repeatedly swapping adjacent elements if they are in the wrong order. It is called "bubble" sort because the algorithm moves the larger elements towards the end of the array in a manner that resembles the rising of bubbles in a liquid.

The basic algorithm of Bubble Sort is as follows:

1. Start at the beginning of the array.

2. Compare the first two elements. If the first element is greater than the second element, swap them.

3. Move to the next pair of elements and repeat step 2.

4. Continue the process until the end of the array is reached.

5. If any swaps were made in step 2-4, repeat the process from step 1.

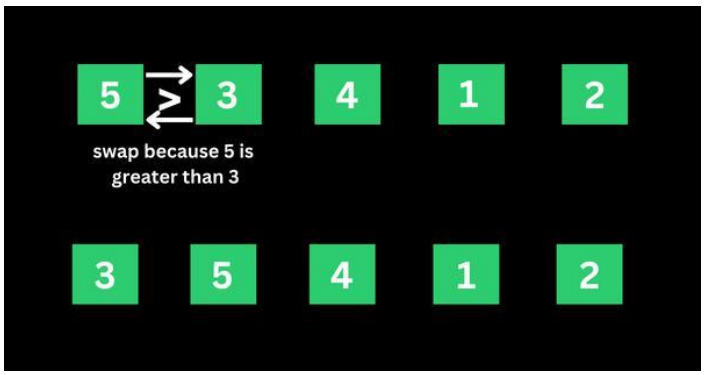
The time complexity of Bubble Sort is O(n^2), which makes it inefficient for large lists. However, it has the advantage of being easy to understand and implement, and it is useful for educational purposes and for sorting small datasets.

Example of Bubble Sort :

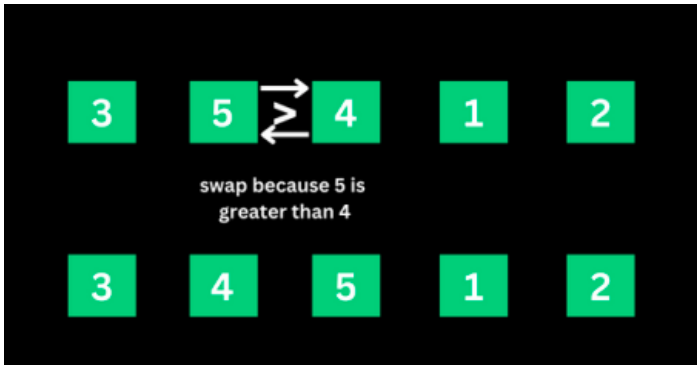
Let's say we want to sort a series of numbers 5, 3, 4, 1, and 2 so that they are arranged in ascending order... The sorting begins the first iteration by comparing the first two values. If the first value is greater than the second, the algorithm pushes the first value to the index of the second value.

First Iteration of the Sorting

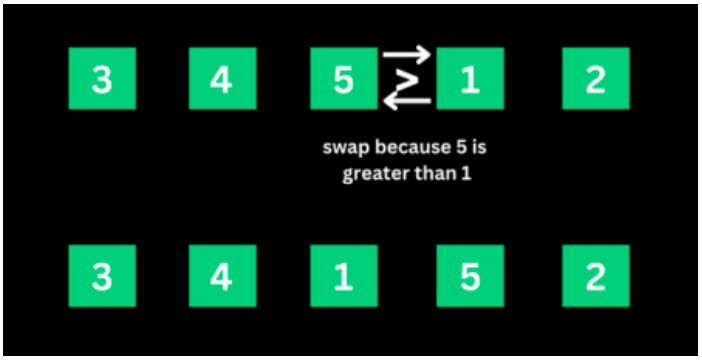
Step 1: In the case of 5, 3, 4, 1, and 2, 5 is greater than 3. So 5 takes the position of 3 and the numbers become 3, 5, 4, 1, and 2.



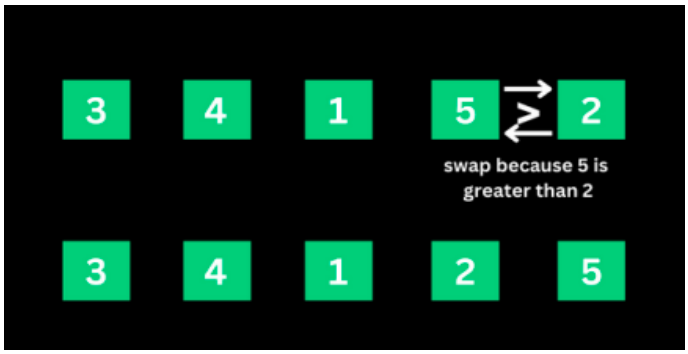
Step 2: The algorithm now has 3, 5, 4, 1, and 2 to compare, this time around, it compares the next two values, which are 5 and 4. 5 is greater than 4, so 5 takes the index of 4 and the values now become 3, 4, 5, 1, and 2.



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.



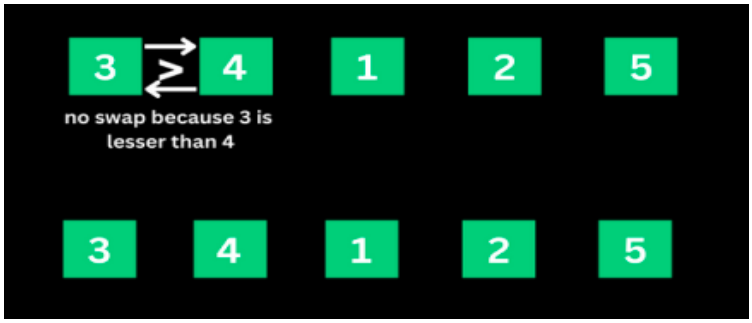
Step 4: The algorithm now has 3, 4, 1, 5, and 2 to compare. It compares the next two values, which are 5 and 2. 5 is greater than 2, so 5 takes the index of 2 and the numbers become 3, 4, 1, 2, and 5.



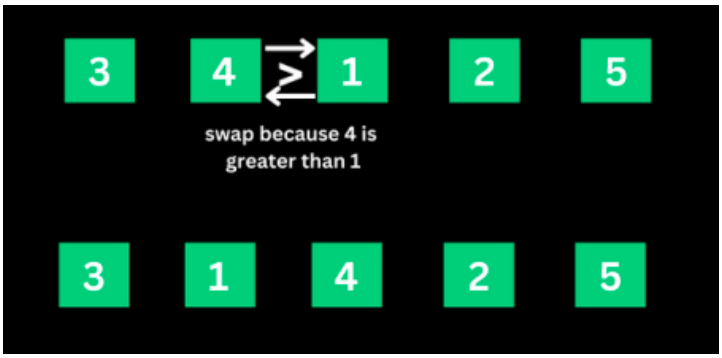
That's the first iteration. And the numbers are now arranged as 3, 4, 1, 2, and 5 – from the initial 5, 3, 4, 1, and 2. As you might realize, 5 should be the last number if the numbers are sorted in ascending order. This means the first iteration is really completed.

Second Iteration of the Sorting and the Rest

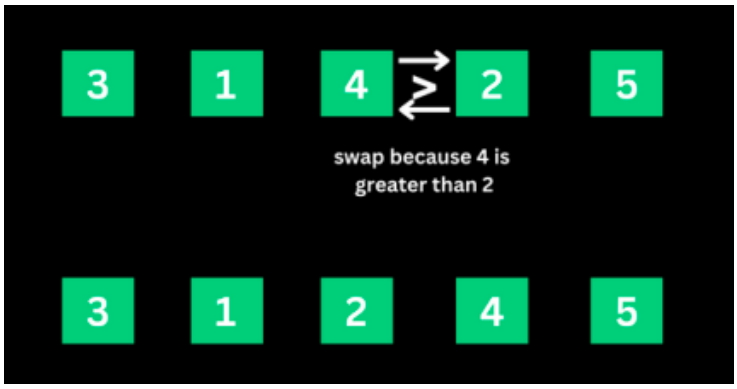
The algorithm starts the second iteration with the last result of 3, 4, 1, 2, and 5. This time around, 3 is smaller than 4, so no swapping happens. This means the numbers will remain the same.



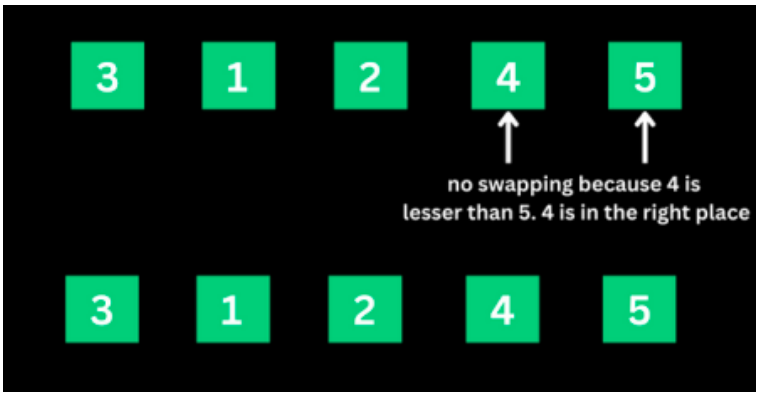
The algorithm proceeds to compare 4 and 1. 4 is greater than 1, so 4 is swapped for 1 and the numbers become 3, 1, 4, 2, and 5.



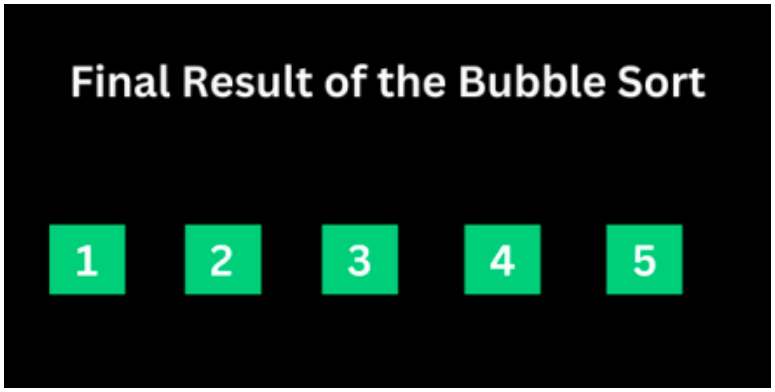
The algorithm now proceeds to compare 4 and 2. 4 is greater than 2, so 4 is swapped for 2 and the numbers become 3, 1, 2, 4, and 5.



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.



Concept of OpenMP

● OpenMP (Open Multi-Processing) is an application programming interface (API) that supports shared-memory parallel programming in C, C++, and Fortran. It is used to write parallel programs that can run on multicore processors, multiprocessor systems, and parallel computing clusters.

● OpenMP provides a set of directives and functions that can be inserted into the source code of a program to parallelize its execution. These directives are simple and easy to use, and they can be applied to loops, sections, functions, and other program constructs. The compiler then generates parallel code that can run on multiple processors concurrently.

How to measure the performance of sequential and parallel algorithms?

To measure the performance of sequential Bubble sort and parallel Bubble sort algorithms, you can follow these steps:

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.

When measuring the performance of the parallel Bubble sort algorithm, you will need to specify the number of threads to use. You can experiment with different numbers of threads to find the optimal value for your system.

Program:

```
#include<iostream>
#include<stdlib.h>
#include<omp.h>
using namespace std;

void bubble(int *, int);
void swap(int &, int &);

void bubble(int *a, int n)
{
    for(  int i = 0;  i < n;  i++ )
    {
        int first = i % 2;
        #pragma omp parallel for shared(a,first)
        for(  int j = first;  j < n-1;  j += 2  )
         {
                if(  a[ j ]  >  a[ j+1 ]  )
                {
                        swap(  a[ j ],  a[ j+1 ]  );
                }
                 }
    }
}
void swap(int &a, int &b)
{
    int test;
    test=a;
```

```
    a=b;
    b=test;
}

int main()
{
    int *a,n;
    cout<<"\n enter total no of elements=>";
    cin>>n;
    a=new int[n];
    cout<<"\n enter elements=>";
    for(int i=0;i<n;i++)
    {
        cin>>a[i];
    }
    bubble(a,n);

    cout<<"\n sorted array is=>";
    for(int i=0;i<n;i++)
    {
        cout<<a[i]<<endl;
    }
return 0;
}
```

2] What is Merge Sort?

Merge sort is a sorting algorithm that uses a divide-and-conquer approach to sort an array or a list of elements. The algorithm works by recursively dividing the input array into two halves, sorting each half, and then merging the sorted halves to produce a sorted output. T

he merge sort algorithm can be broken down into the following steps:

1. Divide the input array into two halves.

2. Recursively sort the left half of the array.

3. Recursively sort the right half of the array.

4. Merge the two sorted halves into a single sorted output array.

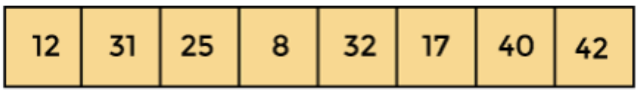
● The merging step is where the bulk of the work happens in merge sort. The algorithm compares the first elements of each sorted half, selects the smaller element, and appends it to the output array. This process continues until all elements from both halves have been appended to the output array.

● The time complexity of merge sort is O(n log n), which makes it an efficient sorting algorithm for large input arrays. However, merge sort also requires additional memory to store the output array, which can make it less suitable for use with limited memory resources.

● In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.

Example of Merge Sort :

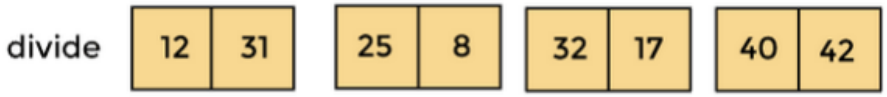
Now, let's see the working of merge sort Algorithm. To understand the working of the merge sort algorithm, let's take an unsorted array. It will be easier to understand the merge sort via an example. Let the elements of array are –



● 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.

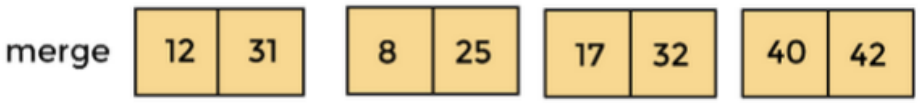


● Now, again divide these arrays to get the atomic value that cannot be further divided.

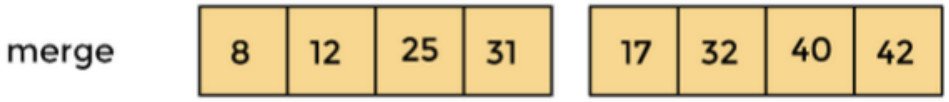
● Now, combine them in the same manner they were broken.

● In combining, first compare the element of each array and then combine them into another array in sorted order.

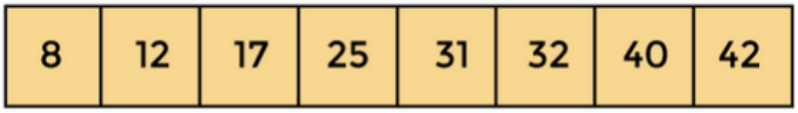
● So, first compare 12 and 31, both are in sorted positions. Then compare 25 and 8, and in the list of two values, put 8 first followed by 25. Then compare 32 and 17, sort them and put 17 first followed by 32. After that, compare 40 and 42, and place them sequentially.



● 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 –



How to measure the performance of sequential and parallel algorithms?

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.

Program :

```
#include<iostream>

#include<stdlib.h>

#include<omp.h>

using namespace std;


void mergesort(int a[],int i,int j);

void merge(int a[],int i1,int j1,int i2,int j2);


void mergesort(int a[],int i,int j)

{

        int mid;

        if(i<j)
```

```
        {
        mid=(i+j)/2;
        #pragma omp parallel sections
        {
        #pragma omp section
        {
                mergesort(a,i,mid);
        }
        #pragma omp section
        {
                mergesort(a,mid+1,j);
        }
        }

        merge(a,i,mid,mid+1,j);
        }
}

void merge(int a[],int i1,int j1,int i2,int j2)
{
        int temp[1000];
        int i,j,k;
        i=i1;
        j=i2;
        k=0;

        while(i<=j1 && j<=j2)
        {
```

```
        if(a[i]<a[j])
        {
        temp[k++]=a[i++];
        }
        else
        {
        temp[k++]=a[j++];
    }
        }

        while(i<=j1)
        {
        temp[k++]=a[i++];
        }

        while(j<=j2)
        {
        temp[k++]=a[j++];
        }

        for(i=i1,j=0;i<=j2;i++,j++)
        {
        a[i]=temp[j];
        }
}
int main()
{
        int *a,n,i;
```

```
        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];
        }
  //     start=.....
//#pragma omp.....
        mergesort(a, 0, n-1);
//          stop.....
        cout<<"\n sorted array is=>";
        for(i=0;i<n;i++)
        {
        cout<<"\n"<<a[i];
        }
        // Cout<<Stop-Start
        return 0;
}
```

Conclusion: In this way we can implement Bubble Sort and Merge Sort in parallel way using OpenMP also come to know how to how to measure performance of serial and parallel algorithm.

Assignment No: 3

Title: Implement Min, Max, Sum and Average operations using Parallel Reduction.

Objective: Students should be able to Write a program to implement listed operations using parallel reduction.

Prerequisite:

1. Basic of programming language

2. Concept of Parallel Reduction

Theory:

1] What is parallel reduction?

Parallel reduction refers to algorithms which combine an array of elements producing a single value as a result. Problems eligible for this algorithm include those which involve operators that are associative and commutative in nature. Some of them include:

1. Sum of an array
2. Minimum/Maximum of an array

let's take a problem statement of finding the minimum in an array of size 10^8 floating point elements. In a CPU this usually takes around 15–17s for an array of this size. The time for computation only increases on a larger scale as the computations and size increases.

To minimize the computation time on a CPU we might parallelize the computation using multiple threads on multiple cores. The parallelism that can be achieved with CPUs is very much limited. On the other hand, the compute power of GPUs is evolving continuously. Given the ubiquitous nature of GPUs recently and the ability to execute several hundreds of threads in parallel, it is vital to extract the compute power of GPUs whenever possible.

2] What is reduction computation?

– Summarize a set of input values into one value using a "reduction operation"
– Max
– Min

– Sum
– Product
– Often used with a user defined reduction operation function as long as the operation

– Is associative and commutative
– Has a well-defined identity value (e.g., 0 for sum)
– For example, the user may supply a custom "max" function for 3D coordinate data sets where the magnitude for the each coordinate data tuple is the distance from the origin.

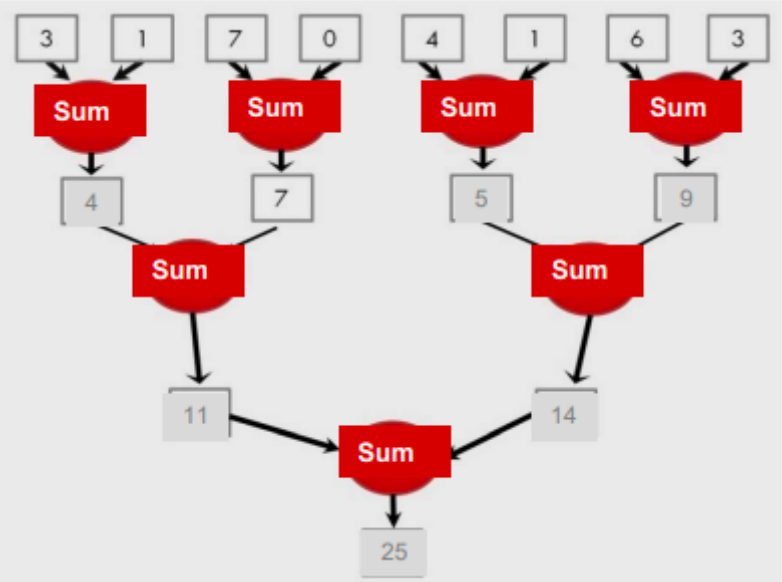
Parallel Sum Reduction :

– Parallel implementation

– Each thread adds two values in each step

– Recursively halve # of threads

– Takes log(n) steps for n elements, requires n/2 threads.



– Assume an in-place reduction using shared memory

– The original vector is in device global memory

– The shared memory is used to hold a partial sum vector

– Initially, the partial sum vector is simply the original vector

– Each step brings the partial sum vector closer to the sum

– The final sum will be in element 0 of the partial sum vector

– Reduces global memory traffic due to reading and writing partial sum values

– Thread block size limits n to be less than or equal to 2,048.

Program:

```
#include <iostream>
//#include <vector>
#include <omp.h>
#include <climits>
using namespace std;
void min_reduction(int arr[], int n) {
  int min_value = INT_MAX;
  #pragma omp parallel for reduction(min: min_value)
  for (int i = 0; i < n; i++) {
        if (arr[i] < min_value) {
        min_value = arr[i];
        }
  }
  cout << "Minimum value: " << min_value << endl;
}

void max_reduction(int arr[], int n) {
  int max_value = INT_MIN;
  #pragma omp parallel for reduction(max: max_value)
  for (int i = 0; i < n; i++) {
        if (arr[i] > max_value) {
        max_value = arr[i];
```

```
        }
  }
  cout << "Maximum value: " << max_value << endl;
}

void sum_reduction(int arr[], int n) {
  int sum = 0;
   #pragma omp parallel for reduction(+: sum)
   for (int i = 0; i < n; i++) {
        sum += arr[i];
  }
  cout << "Sum: " << sum << endl;
}

void average_reduction(int arr[], int n) {
  int sum = 0;
  #pragma omp parallel for reduction(+: sum)
  for (int i = 0; i < n; i++) {
        sum += arr[i];
  }
  cout << "Average: " << (double)sum / (n-1) << endl;
}

int main() {
    int *arr,n;
    cout<<"\n enter total no of elements=>";
    cin>>n;
    arr=new int[n];
```

```
    cout<<"\n enter elements=>";
    for(int i=0;i<n;i++)
    {
            cin>>arr[i];
    }

//   int arr[] = {5, 2, 9, 1, 7, 6, 8, 3, 4};
//   int n = size(arr);

  min_reduction(arr, n);
  max_reduction(arr, n);
  sum_reduction(arr, n);
  average_reduction(arr, n);
}
```

Output:

```
6
3
4
2
Minimum value: 2
Maximum value: 8
Sum: 23
Average: 5.75
```

Conclusion: In this way we can implement min, max, sum and average operations using parallel reduction.

Assignment No: 4

Title: Implement HPC applications for AI/ML domain.

Objective: Students should be able to implement the applications of HPC over AI and ML domain.

Theory:

1] What is High Performance Computing (HPC)?

HPC refers to high-speed parallel processing of complex computations on multiple servers. A group of these servers is called a cluster, which consists of hundreds or thousands of computing servers connected through a network. In an HPC cluster, each computer performing computing operations is called a node.

HPC clusters commonly run batch calculations. At the heart of an HPC cluster is a scheduler used to keep track of available resources. This allows for efficient allocation of job requests across different compute resources (CPUs and GPUs) over high-speed networks.

Modern HPC solutions can run in an on-premises data center, at the edge, or in the cloud. They can solve large-scale computing problems at a reasonable time and cost, making them suitable for a wide range of problems.

2] High-Performance Computing & AI

AI can be used in High-Performance Computing to augment the analysis of datasets and produce faster results at the same accuracy level. The implementation of HPC and AI requires similar architectures – both achieve results by processing large data sets that typically grow in size using high computing and storage capacity, large bandwidth, and high-bandwidth fabrics. The following HPC use cases can benefit from AI capabilities.

- Financial analysis like risk and fraud detection, manufacturing, and logistics.
- Astrophysics and astronomy.
- Climate Science and meteorology.
- Earth Sciences.
- Computer-aided design (CAD), computational fluid dynamics (CFD), and computer-aided engineering (CAE).

- Scientific visualization and simulation.

How HPC can help build better AI applications

- Massive Parallel Computing speeds up calculations significantly, allowing the processing of large datasets in less time.
- More storage and memory make it easy to process large volumes of data, thereby increasing the accuracy of AI models.
- Graphical Processing Units (GPUs) can be used to process AI algorithms more effectively.
- HPC as a service can be accessed on the cloud; hence, upfront costs can be reduced.

Integration of AI and HPC

The integration of HPC and AI, however, requires a few changes to workload tools and management. Following are a few ways High-Performance Computing is evolving to address the challenges associated with integrating AI and HPC.

1. **Programming Languages**

HPC programs are generally written in languages like C, C++, and Fortran, and the extensions and libraries of these languages support the processes of HPC. AI, on the other hand, relies on python and Julia.

For HPC and AI to use the same infrastructure, the software and interface should be compatible. In most cases, AI frameworks and languages are overlaid on existing software to allow both sets of programmers to continue using their current tool without migrating to a different language.

2. **Virtualization and Containers**

Containers allow the developers to adapt their infrastructure to the changing work needs and enable them to deploy the same consistently. Containers allow the Julia and Python applications to be more scalable.

With containers, teams can create HPC configurations that are quick and easy to deploy without time-consuming configuration.

3] Convergence of AI and HPC

The HPC industry is eager to integrate AI and HPC, improving support for AI use cases. HPC has been successfully used to run large-scale AI models in fields like cosmic theory, astrophysics, high-energy physics, and data management for unstructured data sets.

However, it is important to realize that the methods used to accelerate training of AI models on HPC are still experimental. It is not well understood how to optimize hyperparameters as the number of GPUs increases in an HPC setup.

Another challenge is that when vendors test performance of AI on HPC platforms, they use classical neural network models, such as ResNet trained on the standard ImageNet dataset. This provides a general idea of HPC performance for AI. However, in real life, there are noisy, incomplete, and heterogeneous AI architectures, which may perform very differently from these benchmark results.

The following developments will promote the convergence of AI and HPC in the future:

- Creating better mathematical frameworks for selecting AI architectures and optimization plans that will be most compatible with HPC systems.
- Building a community that can share experiences across interdisciplinary tasks, such as informatics, AI models, data and software management.
- Understanding how artificial intelligence data and models interact and creating commercial solutions that can be used across multiple domains and use cases.
- Increasing the use of open source tools and platforms. This can increase adoption of AI on HPC and improve the support of standardized tooling.

Program: Training prot_t5_xl_uniref50 model for predicting protein secondary structure using SageMaker Model Parallel.

```
import sys
!{sys.executable} -m pip install "sagemaker>=2.48.0" "transformers>=4.12.3" "datasets" --upgrade
!{sys.executable} -m pip install ipywidgets
!{sys.executable} -m pip install s3fs
import sagemaker.huggingface
import datasets


import sagemaker

sess = sagemaker.Session()
# sagemaker session bucket -> used for uploading data, models and logs
# sagemaker will automatically create this bucket if it not exists
sagemaker_session_bucket=None
if sagemaker_session_bucket is None and sess is not None:
    # set to default bucket if a bucket name is not given
    sagemaker_session_bucket = sess.default_bucket()

role = sagemaker.get_execution_role()
sess = sagemaker.Session(default_bucket=sagemaker_session_bucket)

print(f"sagemaker role arn: {role}")
print(f"sagemaker bucket: {sess.default_bucket()}")
print(f"sagemaker session region: {sess.boto_region_name}")


rom datasets import load_dataset
from transformers import AutoTokenizer

# dataset used
dataset_name = 'agemagician/NetSurfP-SS3'

# s3 key prefix for the data
s3_prefix = 'samples/datasets/netsurfp-ss3'
# load dataset
dataset = load_dataset(dataset_name)


storage_options = {"anon": True}  # for anonymous connection
```

```
# for private buckets, uncomment the following code and add your aws access key id and secret key.
# storage_options = {"key": aws_access_key_id, "secret": aws_secret_access_key}
import botocore
s3_session = botocore.session.Session()
storage_options = {"session": s3_session}


import s3fs
fs = s3fs.S3FileSystem(**storage_options)


# save dataset to s3
data_input_path = f's3://{sess.default_bucket()}/{s3_prefix}/data1'
dataset.save_to_disk(data_input_path,fs=fs)

data_input_path
fs.ls(data_input_path)



# training script
!pygmentize ./code/train.py
model_name = "Rostlab/prot_t5_xl_uniref50"

from sagemaker.huggingface import HuggingFace

# hyperparameters, which are passed into the training job
hyperparameters={'epochs': 1,
                 'train_batch_size': 1,
                 'model_name': model_name
                 }

# configuration for running training on smdistributed Model Parallel
mpi_options = {
    "enabled" : True,
    "processes_per_host" : 8,
}
smp_options = {
    "enabled":True,
    "parameters": {
        "microbatches": 1,
        "placement_strategy": "cluster",
        "pipeline": "interleaved",
```

```
        "optimize": "memory",
        "partitions": 4,
        "ddp": True,
        # "tensor_parallel_degree": 2,
        "shard_optimizer_state": True,
        "activation_checkpointing": True,
        "activation_strategy": "each",
        "activation_offloading": True,
    }
}

distribution={
    "smdistributed": {"modelparallel": smp_options},
    "mpi": mpi_options
}

huggingface_estimator = HuggingFace(entry_point='train.py',
                        source_dir='./code',
                        instance_type='ml.p3dn.24xlarge',
                        instance_count=1,
                        role=role,
                        transformers_version='4.12',
                        pytorch_version='1.9',
                        py_version='py38',
                        distribution= distribution,
                        hyperparameters = hyperparameters,
                        keep_alive_period_in_seconds=60*60) # managed warm pool for
60mins

# starting the train job with our uploaded datasets as input
huggingface_estimator.fit({'data': data_input_path})

predictor = huggingface_estimator.deploy(1,"ml.g4dn.xlarge")
predictor.delete_endpoint()
```

Conclusion: The potential of combining AI with HPC is phenomenal, although of course that doesn't mean it is suited to every task in scientific computing, and part of the challenge is to identify where it can be best put to work.

Mini Project

Title: Implement Huffman encoding on GPU.

Theory:

Huffman coding :

• Lossless data compression scheme

• Used in many data compression formats:

  • gzip, zip, png, jpg, etc.

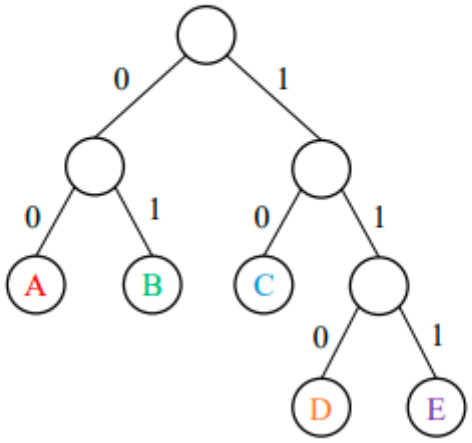
• Uses a codebook: mapping of fixed-length (usually 8-bit) symbols into codewords bits.

• Entropy coding: Symbols appear more frequently are assigned codewords with fewer bits.

• Prefix code: Every codeword is not a prefix of the other codewords.

codebook

symbols	A	B	C	D	E
codeword bits	00	01	10	110	111



• Huffman Encoding can be done by converting each symbol to the corresponding codeword: parallel encoding is easy.

• Huffman Decoding can be done by reading the codeword sequence from the beginning

  1. identifying each codeword

2. converting it into the corresponding codeword

GPU Huffman encoding with a gap array :

• **Naive Parallel GPU encoding**

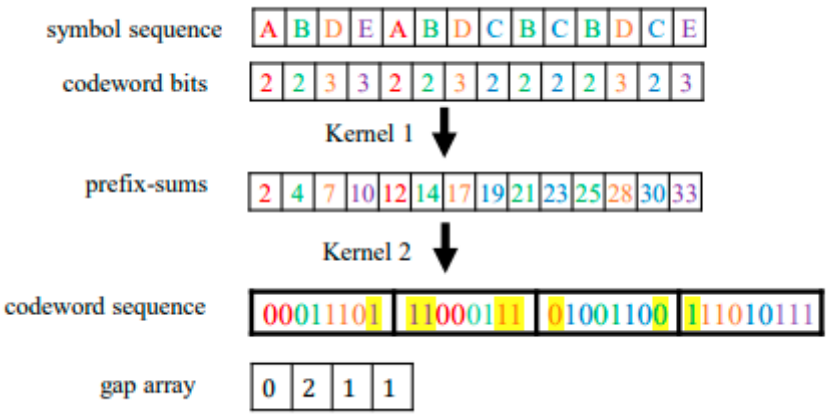
• Kernel 1: The prefix-sums of codeword bits are computed.

• The bit position of the codeword corresponding to each symbol can be determined from the prefix-sums.

• Kernel 2: The codeword of corresponding to each symbol is written.

• Gap arrays can be written if necessary.

• Both Kernels 1 and 2 perform global memory access.



• GPU encoding by the Single Kernel Soft Synchronization (SKSS)

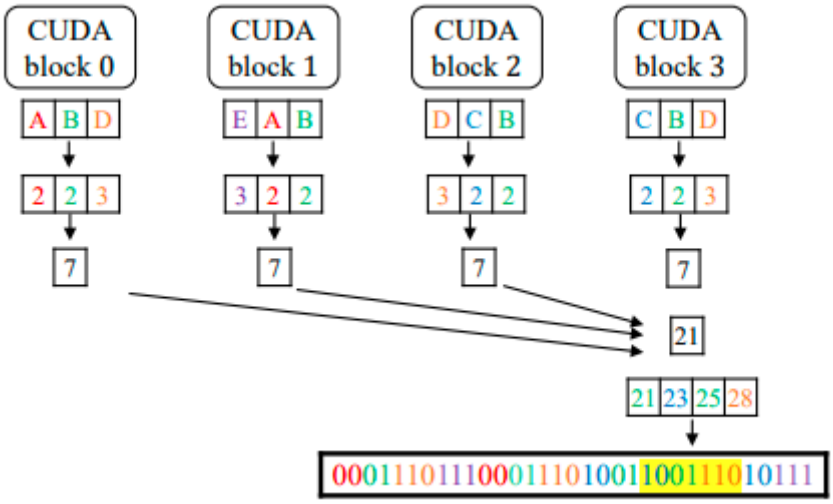
• Only one kernel call is performed.

• Reduce global memory access

• The codeword sequence are partitioned into equal-sized segments.

• Each CUDA block i (this number is assigned by a global counter) works for encoding segment i

• The Prefix-sums for each segment i are computed by looking back previous CUDA blocks.



GPU Huffman decoding with a gap array :

SKSS technique:

• The codeword sequence is partitioned into equal-sized segments and the gap value of each segment is available.

• Each CUDA block i (this number is assigned by a global counter) works for decoding a segment i

• Since the gap value is available, each CUDA block can start decoding from the first complete codeword.

• Similarly to GPU Huffman decoding, the prefix-sums of the number of symbols corresponding to segments are computed by the SKSS.

• From the prefix-sums, each CUDA block can determine the position in the symbol sequence where it writes the decoded symbols.

Compact codebook:

• A 64Kbyte codebook is separated into several small codebooks.

• Primary codebook: stores codewords with no more than 11 bits

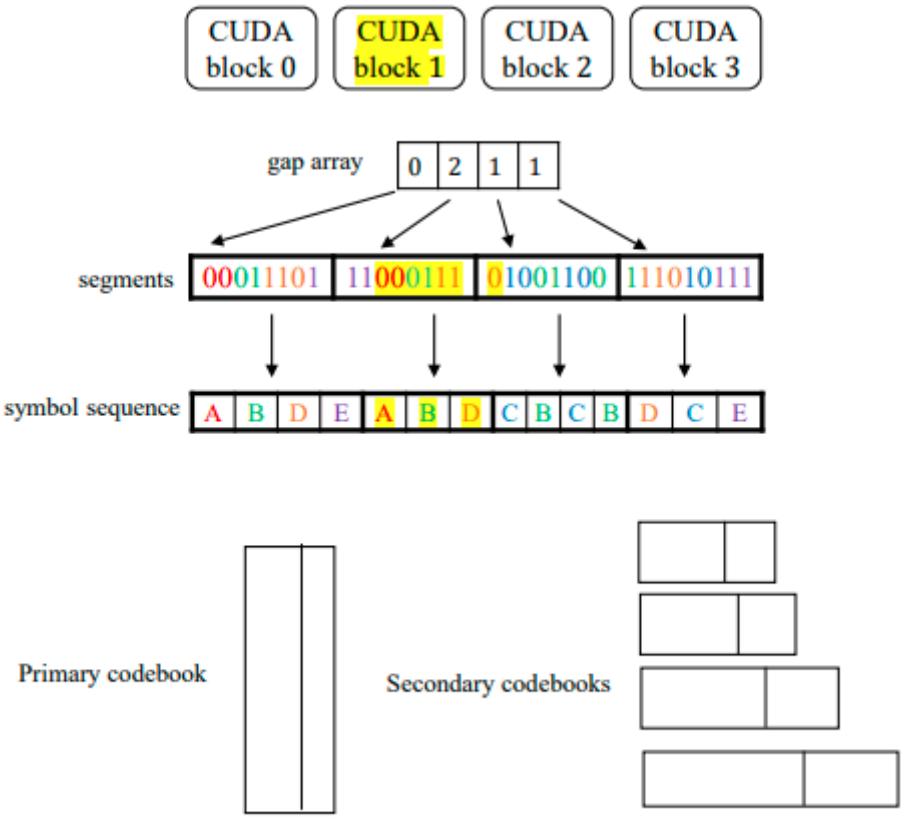
• Secondary codebooks: store codewords with 11 bits or more

• The total size is less than 3 Kbytes.

Wordwise memory access:

• 4 symbols are written as a 32-bit word.

• Global memory access throughput can be improved.



Program: Length-limited Huffman encoding

```
#include "llhuffman_encoder.h"

#include <map>
#include <unordered_map>
#include <algorithm>
#include <cmath>

std::shared_ptr<std::vector<llhuff::LLHuffmanEncoder::Symbol>>
    llhuff::LLHuffmanEncoder::get_symbol_lengths(
    SYMBOL_TYPE* data, size_t size) {
    // collect source symbol frequencies
    std::unordered_map<SYMBOL_TYPE, size_t> freq;

    for(size_t i = 0; i < size; ++i)
        ++freq[data[i]];
    // if number of symbols exceeds maximum number of leaf nodes,
    // cancel and return nullptr
    if(log2(freq.size()) > MAX_CODEWORD_LENGTH) {
        return nullptr;
    }

    // sort symbols by frequencies in increasing order
    std::vector<std::pair<SYMBOL_TYPE, size_t>>
        symbols(freq.begin(), freq.end());
```

```
if(symbols.size() == 1) {
    // input consists of only one symbol
    Symbol s; s.length = 1; s.symbol = symbols.at(0).first;
    std::vector<Symbol> vec(1);
    vec.at(0) = s;
    return std::make_shared<std::vector<Symbol>>(vec);
}

std::sort(symbols.begin(), symbols.end(),
    [](std::pair<SYMBOL_TYPE, size_t> &a,
        std::pair<SYMBOL_TYPE, size_t> &b) -> bool
        {return a.second < b.second;});

// create lists of coins (denomination, numismatic value)
std::vector<std::vector<CoinPackage>> denom_lists;
denom_lists.resize(MAX_CODEWORD_LENGTH);

// create coins
for(size_t i = 0; i < MAX_CODEWORD_LENGTH; ++i) {
    size_t second_counter = 0;
    std::vector<CoinPackage> denom;
    denom.resize(symbols.size());

    for(size_t j = 0; j < symbols.size(); ++j) {
```

```
        CoinPackage coin;

        const float denomination = 1.0f /
            (float) (1 << (MAX_CODEWORD_LENGTH - i));
        const float numismatic = (float) (symbols[j].second)
            / (float) (size);
        const SYMBOL_TYPE sym = symbols[j].first;
        coin.denomination = denomination;
        coin.numismatic = numismatic;
        coin.coins.push_back(Coin{denomination, numismatic, sym});
        denom.at(second_counter) = coin;
        ++second_counter;
    }
    denom_lists.at(i) = denom;
}

// apply package / merge algorithm
for(size_t i = 0; i < MAX_CODEWORD_LENGTH - 1; ++i) {
    std::vector<CoinPackage> new_row;

    // package
    if(denom_lists.at(i).size() % 2 != 0) {
        denom_lists.at(i).pop_back();
    }
```

```
    for(size_t j = 0; j <= (denom_lists.at(i).size() - 2); ++++j) {
        CoinPackage new_coin = denom_lists.at(i).at(j).package(
            &(denom_lists.at(i).at(j + 1)));
        new_row.push_back(new_coin);
    }
    std::vector<CoinPackage> dst;

    // merge
    std::merge(
        new_row.begin(), new_row.end(),
        denom_lists.at(i + 1).begin(), denom_lists.at(i + 1).end(),
        std::back_inserter(dst),
        [](CoinPackage &a, CoinPackage &b) -> bool {
            return a.numismatic <= b.numismatic;
        });
    denom_lists.at(i + 1) = dst;
}

// keep coin packages whose denominations add up to symbols.size() - 1,
// delete the remaining coins
denom_lists.back().resize(2 * (symbols.size() - 1));

// merge all coins into a vector
std::vector<Coin> coins;
```

```
for(size_t i = 0; i < denom_lists.back().size(); ++i) {
    coins.reserve(coins.size() + denom_lists.back().at(i).coins.size());
    coins.insert(coins.end(), denom_lists.back().at(i).coins.begin(),
        denom_lists.back().at(i).coins.end());
}

// count coins with equal symbol
std::unordered_map<SYMBOL_TYPE, size_t> num_coins_of_symbol;
for(size_t i = 0; i < coins.size(); ++i)
    ++num_coins_of_symbol[coins[i].symbol];

// assign codeword length to symbols
std::shared_ptr<std::vector<Symbol>> symbol_length
    = std::make_shared<std::vector<Symbol>>();
symbol_length->resize(num_coins_of_symbol.size());

size_t second_counter = 0;
for(auto &i: num_coins_of_symbol) {
    symbol_length->at(second_counter).symbol = i.first;
    symbol_length->at(second_counter).length = i.second;
    symbol_length->at(second_counter).count = freq[i.first];
    ++second_counter;
}

// sort symbols by increasing codeword length
```

```
    std::sort(symbol_length->begin(), symbol_length->end(),
        [](Symbol &a, Symbol &b) -> bool
            {return a.length < b.length;});
    return symbol_length;
}

std::shared_ptr<llhuff::LLHuffmanEncoderTable>
    llhuff::LLHuffmanEncoder::get_encoder_table(
    std::shared_ptr<std::vector<Symbol>> symbol_lengths) {

    llhuff::LLHuffmanEncoderTable table;



    // calculate total size of compressed output in bytes
    size_t size = 0;
    for(size_t i = 0; i < symbol_lengths->size(); ++i)
        size += symbol_lengths->at(i).length * symbol_lengths->at(i).count;

    // ceil to integer byte count
    if(size % 8 != 0)
        size += 8 - (size % 8);
    size /= 8;

    // convert to unit size
    table.compressed_size = size % sizeof(UNIT_TYPE) == 0 ?
```

```
        size / sizeof(UNIT_TYPE) : (size / sizeof(UNIT_TYPE)) + 1;

    // generate canonical codewords
    UNIT_TYPE code = 0;
    size_t current_length = symbol_lengths->at(0).length;

    for(size_t i = 0; i < symbol_lengths->size(); ++i) {
        table.dict[symbol_lengths->at(i).symbol] = {code, current_length};
        size_t next_length = (i + 1 < symbol_lengths->size())
            ? symbol_lengths->at(i + 1).length : current_length;
        code = (code + 1) << (next_length - current_length);
        current_length = next_length;
    }
    return std::make_shared<llhuff::LLHuffmanEncoderTable>(table);
}

void llhuff::LLHuffmanEncoder::encode_memory(UNIT_TYPE* out, size_t size_out,
    SYMBOL_TYPE* in, size_t size_in,
    std::shared_ptr<llhuff::LLHuffmanEncoderTable> encoder_table) {
    UNIT_TYPE* out_ptr = out;
    const size_t max_bits = sizeof(UNIT_TYPE) * 8;
    UNIT_TYPE window = 0;
    size_t at = 0;
    size_t in_unit = 0;
```

```
    for(size_t i = 0; i < size_out && in_unit < size_in; ++i) {
        auto code = encoder_table->dict[in[in_unit]]
        while(at + code.length < max_bits && in_unit < size_in) {
            window <<= code.length;
            window += code.codeword;
            at += code.length;
            ++in_unit;
            if(in_unit < size_in)
                code = encoder_table->dict[in[in_unit]];
        }

        const size_t diff = at + code.length - max_bits;
        window <<= code.length - diff;
        window += (code.codeword >> diff);
        out_ptr[i] = window;
        window = code.codeword & ~(~0 << diff);
        at = diff;
        ++in_unit;
    }
}
std::shared_ptr<cuhd::CUHDCodetable>
    llhuff::LLHuffmanEncoder::get_decoder_table(
    std::shared_ptr<llhuff::LLHuffmanEncoderTable> enc_table) {
    std::shared_ptr<cuhd::CUHDCodetable> table
        = std::make_shared<cuhd::CUHDCodetable>(enc_table->dict.size());
```

```
    cuhd::CUHDCodetableItemSingle* dec_ptr = table->get();


    for(auto &i: enc_table->dict) {
        const SYMBOL_TYPE symbol = i.first;
        const UNIT_TYPE codeword = i.second.codeword;
        const BIT_COUNT_TYPE length = i.second.length;
        const size_t shift = MAX_CODEWORD_LENGTH - length;
        const size_t num_entries = 1 << shift;


        for(size_t j = 0; j < num_entries; ++j)
            dec_ptr[(codeword << shift) + j] = {length, symbol};
    }
    return table;
}
```

Conclusion: In this way we have learned to implement Huffman encoding on GPU.