# Assignment No. 1

- **Title:** Parallel Breadth First Search and Depth First Search.
- **Date of Completion:**
- **Objective:** To implement Parallel Breadth First Search and Depth First Search based.
- **Problem Statement:** 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 .
- **Theory:**

DFS algorithms using OpenMP:

Data Structure: To implement BFS and DFS, we need to represent the graph or tree in a data structure. In this case, let's use an adjacency list to represent the undirected graph. For the tree, we can use a binary tree with left and right pointers.
Breadth First Search (BFS): To perform BFS in parallel using OpenMP, we can use a queue to keep track of the nodes to visit. Each thread can take a node from the queue and visit all of its neighbors in parallel. We can use OpenMP's parallel for directive to parallelize theloop over the neighbors. We need to make sure to synchronize the threads after visiting all neighbors of a node before moving on to the next level of the BFS.

Parallel Breadth First Search (BFS):

BFS is a graph traversal algorithm that explores all the vertices of a graph or tree level by level. To implement a parallel version of BFS using OpenMP, we can use a shared queue data structure that will hold the vertices to be processed. Each thread will pick a vertex from the queue, process it, and add its unvisited neighbors to the queue. This process will continue until the queue is empty.

In the implementation, we first create a vector of bools to keep track of visited vertices, a queue to hold the vertices to be processed, and we push the start vertex into the queue and mark it as visited. Then, we start an OpenMP parallel region and iterate until the queue is empty. In each iteration, we use a critical section to pick a vertex from the queue, process it, and add its unvisited neighbors to the queue.

In this code, graph is a 2D array representing the adjacency matrix of the graph, where graph[i][j] is 1 if there is an edge between nodes i and j, and 0 otherwise. start_node is the starting node for the BFS, num_nodes is the total number of nodes in the graph, visited is an array to keep track of which nodes have been visited, and level is an array to store the level of each node.

The key feature of this implementation is the parallel for loop that iterates over all the neighbors of the current node in the graph. By using #pragma omp parallel for, OpenMP automatically distributes the loop iterations among multiple threads, enabling parallel processing of nodes at each level.

Parallel Depth First Search (DFS):

DFS is another graph traversal algorithm that explores all the vertices of a graph or tree by recursively exploring as far as possible along each branch before backtracking. To implement a parallel version of DFS using OpenMP, we can use a stack data structure that will hold the vertices to be processed. Each thread will pick a vertex from the stack, process it, and add its unvisited neighbors to the stack. This process will continue until the stack is empty.

Note that in both examples, we are using OpenMP's critical directive to synchronize access to shared data structures (i.e., the queue and the stack). We could also use other synchronization mechanisms such as OpenMP's atomic directive or mutexes.

References/ Available link :
https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/

Questions and answer :

1. What is the time complexity of the BFS and DFS algorithms in the worst case?
2. What is the advantage of using OpenMP to parallelize BFS and DFS algorithms?
3. How would you initialize the queue or stack in the parallel BFS and DFS algorithms?
4. Can you explain how the parallel BFS and DFS algorithms handle cycles in an undirected graph?
5. How can you ensure that the parallel BFS and DFS algorithms terminate correctly?
6. Can you explain how the OpenMP parallel for directive works?
7. How can you measure the performance of the parallel BFS and DFS algorithms?
8. How would you modify the parallel BFS and DFS algorithms to work on a directed graph?
9. Can you explain how the parallel BFS and DFS algorithms could be implemented using MPI instead of OpenMP?

10. How would you modify the parallel BFS and DFS algorithms to work on a weighted graph or tree?

**Mcqs:**

What data structure would you use to represent an undirected graph in your parallel BFS or DFS algorithm using OpenMP?

a) Linked list
b) Adjacency matrix
c) Binary tree
d) Adjacency list

What is the time complexity of the BFS algorithm in the worst case?

a) O(V)
b) O(E)
c) O(V+E)
d) O(log V)

How would you parallelize the BFS algorithm using OpenMP?

a) Use multiple threads to visit all nodes in parallel
b) Use a shared queue to keep track of nodes to visit
c) Use OpenMP's parallel for directive to visit neighbors in parallel
d) All of the above

- **Conclusion :**

Implemented Parallel Breadth First Search and Depth First Search.

# Assignment No. 2

- **Title:** Parallel Bubble Sort and Merge sort
- **Date of Completion:**
- **Objective:** To implement Parallel Bubble Sort and Merge sort using OpenMP.
- **Problem Statement:**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.
- **Theory:**

Bubble sort and Merge sort are two commonly used sorting algorithms. In this program, we will implement parallel versions of these algorithms using OpenMP and compare their performance to their sequential counterparts.

Parallel bubble sort is a parallel implementation of the classic Bubble sort algorithm. The concept of parallelism involves executing a set of instructions/code simultaneously instead of line by line sequentially (the traditional way the code is executed in most machines and computer programming generally).

The main benefit of this is faster computation. Implementation of parallelism depends on some factors. For example, if there are parts of code that can be implemented independently and executed irrespective of any order, then we can execute at the same time. However, some parts of the program may still require other parts of the program to run first.

Therefore, the degree of parallelism depends on the program and its requirements. Additionally, the capacity of today's modern processors that run these sequential programs is limited. We can solve this problem using multi-core computers. In addition, even sequential programs involve some degree of parallelism.

Modern-day processors can handle multiple instructions and the data generated from them at the same time. The hardware and the system are designed in such a way so that the data generated, and the previous, current, and future instructions don't interrupt each other.

This way, even though the programs are sequential, we can implement some degree of parallelism. We can also design our algorithms in such a way that the parts of the program run sequentially.

n Parallel Bubble Sort, we divide sorting of the unsorted into two phases- odd and even. We compare all pairs of elements in the list/array side by side (parallel).

We divide the sorting process into two phases- odd phase and even phase. When it is the odd phase, we compare the element at index 0 with the element at index 1, the element at index 2 with the element at index 3, and so on. In the even phase, we compare index 1 element with index 2 element, index 3 element with index 4 element, and so on, with our last comparison being element at third last index and element at second last index.

When comparing, just like in normal bubble sort, we swap the elements, if the initial element is greater than the next element in comparison.

Both phases occur one after the other and the algorithm stops once no swap occurs in both odd and even phase.

Merge sort is a divide and conquer algorithm. Given an input array, it divides it into halves and each half is then applied the same division method until individual elements are obtained. A pairs of adjacent elements are then merged to form sorted sublists until one fully sorted list is obtained.

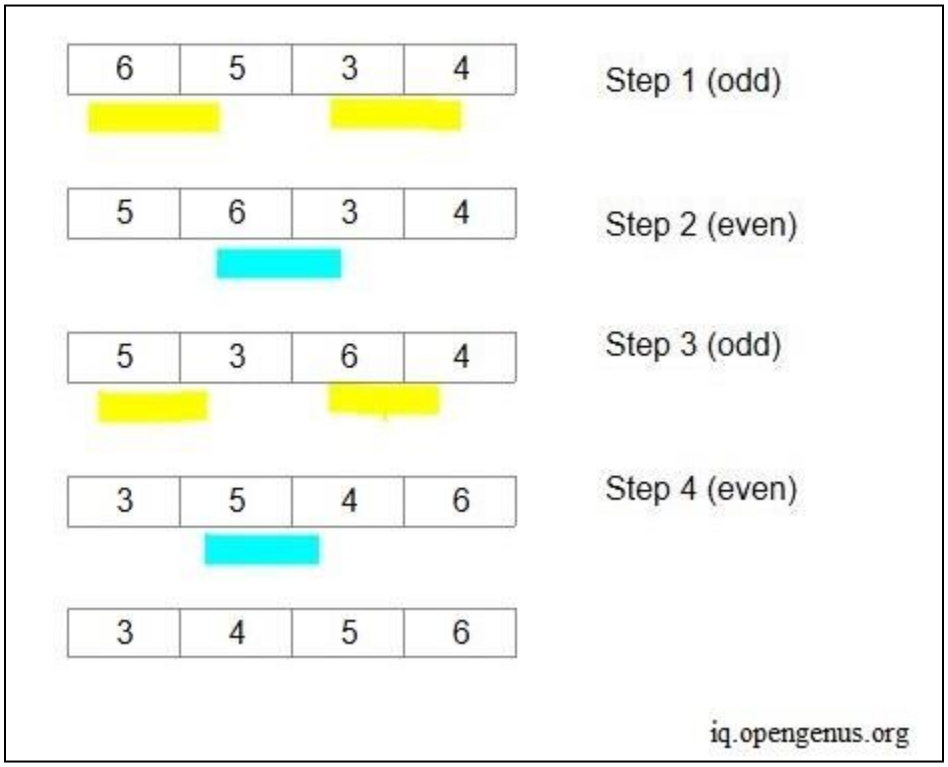
Table of contents:

1. Sequential merge sort algorithm: MergeSort(arr[], l, r)
2. Parallel Merge Sort algorithm
3. Approach 1: Quick Merge sort
4. Approach 2: Odd-Even merge sort
5. Approach 3: Bitonic merge sort
6. Approach 4: Parallel merge sort with load balancing

Prerequisite: Merge Sort Algorithm

Let us get started with Parallel Merge Sort.

The process of parallel bubble sort is shown below.



In the case of multi-core processors, both phases can occur simultaneously, which is known as parallel implementation.

Parallel Merge Sort algorithm

The key to designing parallel algorithms is to find steps in the algorithm that can be carried out concurrently. That is, examine the sequential algorithm and look for operations that could be carried out simultaneously on a given number of processors

- **References/ Available link :**

https://www.researchgate.net/publication/260163622_Parallelize_Bubble_Sort_Algorithm_Using_OpenMP

- **Questions and answer :**
  1. What is the worst-case time complexity of the sequential Bubble Sort algorithm?
  2. How can you divide an array into equal parts for parallel sorting using OpenMP?
  3. What is the significance of the OpenMP 'barrier' directive in the parallel implementation of Bubble Sort and Merge Sort?
  4. How does the number of threads used in parallel sorting impact the performance of the program?

5. What is the best-case time complexity of the Merge Sort algorithm?
6. How would you ensure that the parallel implementation of Merge Sort does not result in a data race?
7. How does the size of the input data impact the performance of the parallel sorting algorithms?
8. What is the difference between parallel Bubble Sort and parallel Merge Sort in terms of the number of comparisons and the number of swaps?
9. What is the maximum number of threads that can be used in parallel sorting using OpenMP?
10. How would you modify the program to use parallel Quick Sort instead of parallel Merge Sort?

- **Mcqs:**

1.How does the number of threads used in parallel sorting impact the performance of the program?
a) Increasing the number of threads will always result in better performance
b) Increasing the number of threads may result in better performance, but only up to a certain point
c) Increasing the number of threads does not have any impact on performance
d) Decreasing the number of threads may result in better performance

2.Which sorting algorithm has a better time complexity for large input sizes?
a) Bubble Sort
b) Merge Sort
c) They have the same time complexity for large input sizes
d) None of the above

3.How can you measure the performance of the sequential and parallel sorting algorithms?
a) By comparing the execution times of the two algorithms on the same input data
b) By measuring the speedup achieved by the parallel algorithm compared to the sequential algorithm
c) By counting the number of comparisons and swaps performed by each algorithm
d) By comparing the memory usage of the two algorithms on the same input data.

- **Conclusion :**Implemented Parallel Bubble Sort and Merge sort using OpenMP.

# Assignment No. 3

- **Title:**Parallel Reduction.
- **Date of Completion:**
- **Objective:** To Implement Min, Max, Sum and Average operations.
- **Problem Statement:**Implement Min, Max, Sum and Average operations using Parallel Reduction.
- **Theory:**

Parallel reduction

One common approach to this problem is parallel reduction. This can be applied for many problems, a min operation being just one of them. It works by using half the number of threads of the elements in the dataset. Every thread calculates the minimum of its own element and some other element. The resultant element is forwarded to the next round. The number of threads is then reduced by half and the process repeated until there is just a single element remaining, which is the result of the operation.
With CUDA you must remember that the execution unit for a given SM is a warp. Thus, any amount of threads less than one warp is underutilizing the hardware. Also, while divergent threads must all be executed, divergent warps do not have to be.
When selecting the "other element" for a given thread to work with, you can do so to do a reduction within the warp, thus causing significant branch divergence within it. This will hinder the performance, as each divergent branch doubles the work for the SM. A better approach is to drop whole warps by selecting the other element from the other half of the dataset.

Parallel Reduction is a technique used in parallel computing to reduce the results of an operation on a set of input data into a single value. In this case, we will be discussing the implementation of Min, Max, Sum, and Average operations using Parallel Reduction.

To implement these operations using Parallel Reduction, we need to follow these steps:

1. Divide the input data into equal-sized chunks among the available threads.
2. Calculate the local Min, Max, Sum, and Average for each thread on their respective chunks.
3. Combine the local Min, Max, Sum, and Average values across all threads using the Parallel Reduction technique.
4. Finally, obtain the final Min, Max, Sum, and Average values by combining the results of all threads.

Here's how we can implement each of these operations:

Min Operation:

To perform a Min operation using Parallel Reduction, we can start by initializing a variable to a very large value (e.g., INT_MAX for integer data). Each thread then calculates the local

minimum for its chunk of data, and the global minimum is found by comparing the local minimum values from each thread and taking the smallest one.

Max Operation:

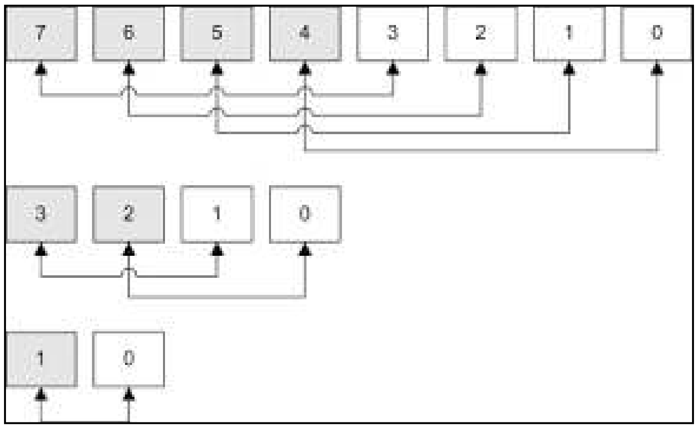
Similar to the Min operation, we can initialize a variable to a very small value (e.g., INT_MIN for integer data) to perform a Max operation. Each thread calculates the local maximum for its chunk of data, and the global maximum is found by comparing the local maximum values from each thread and taking the largest one.

Sum Operation:

To perform a Sum operation using Parallel Reduction, each thread calculates the local sum for its chunk of data, and the global sum is obtained by summing up the local sum values from each thread.

Average Operation:

To perform an Average operation using Parallel Reduction, we first perform a Sum operation on the input data as described above. Then, we divide the global sum by the total number of elements in the input data to obtain the global average.



- **References/ Available link :**
https://www.sciencedirect.com/topics/computer-science/parallel-

reduction#:~:text=An%20informal%20definition%20of%20parallel,using%20P(n)%20processors.

- **Questions and answer :**
  1. What is Parallel Reduction, and what is it used for in parallel computing?
  2. How is the input data divided in Parallel Reduction?
  3. What is the role of each thread in Parallel Reduction?
  4. What is the advantage of using Parallel Reduction over a sequential implementation for Min, Max, Sum, and Average operations?
  5. Can Parallel Reduction be used for non-commutative operations?
  6. What is the time complexity of Parallel Reduction?
  7. How does the number of threads used in Parallel Reduction impact its performance?
  8. How is the final result obtained after each thread has computed its subset of data?
  9. What happens if the input data cannot be evenly divided into subsets for each thread in Parallel Reduction?
  10. How can you measure the performance of Parallel Reduction for Min, Max, Sum, and Average operations?

- **Mcqs:**

  1. Which of the following operations can be performed using Parallel Reduction?
  a) Sorting
  b) Searching
  c) Min, Max, Sum, and Average
  d) All of the above

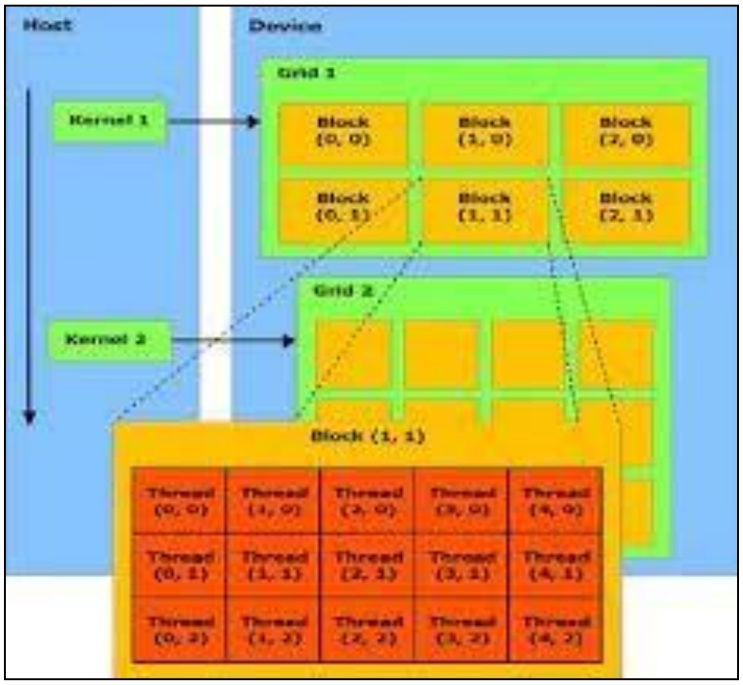
  2. How does the size of the input data affect the performance of Parallel Reduction for Min, Max, Sum, and Average operations?
  a) Larger input data always leads to better performance
  b) Smaller input data always leads to better performance
  c) Performance depends on the number of threads used
  d) Performance is independent of the size of the input data

  3.What is the final step in Parallel Reduction for Min, Max, Sum, and Average operations?
  a) Combine the results obtained by each thread to obtain the final result
  b) Assign the subsets of data to different threads
  c) Perform the operation on each subset of data in parallel
  d) None of the above

- **Conclusion :** We have successfully implemented Min, Max, Sum and Average operations using Parallel Reduction

# Assignment No. 4

- **Title:** CUDA Program.
- **Date of Completion:**
- **Objective:** Is to demonstrate how to use CUDA C to perform vector addition and matrix multiplication on the GPU.
- **Problem Statement:**Write a CUDA Program for : 1. Addition of two large vectors 2. Matrix Multiplication using CUDA C.
- **Theory:**
  - CUDA Architecture:



- Programming Structure of GPU & CPU:
- CUDA Kernel:

The function which are executed on GPU are called as kernels.Kernels are full program or function invoke by the CPU and executed on GPU.A kernal is executed N number of times in parallel on GPU by using N number of threads.
Invocation: kernel_name<<<grid,block>>>(argument,list);
kernel is defined as:

```
_global_voidk
ernel_name(arguments)
{
.........
}
```

Parallel Vector Addition:

Procedure:
1) Write a program using text editor, name the source code with .cu extension.
2) Compile the program using nvcc compiler.
3) Execute the program.
4) Verify the result.

CUDA is a parallel computing platform and programming model created by NVIDIA. It allows developers to harness the power of NVIDIA GPUs for general-purpose computing, including scientific simulations, video processing, and machine learning.

Addition of Two Large Vectors:

1. Vector addition is a simple yet computationally intensive operation that involves adding the corresponding elements of two vectors. In CUDA C, vector addition can be implemented by dividing the vectors into blocks and threads and performing the addition in parallel on the GPU.

The following steps are involved in implementing vector addition using CUDA C:Allocate memory on the device (GPU) using cudaMalloc()

Copy the input vectors from host (CPU) to device using cudaMemcpy()
Launch the kernel on the GPU using the <<<>>> syntax
Wait for the kernel to finish using cudaDeviceSynchronize()
Copy the result back from device to host using cudaMemcpy()
Free the memory on the device using cudaFree()

Matrix Multiplication:

Matrix multiplication is a fundamental operation in linear algebra that involves multiplying two matrices to produce a third matrix. It is a computationally intensive operation that can benefit from parallelization on GPUs.\

The following steps are involved in implementing matrix multiplication using CUDA C:

Allocate memory on the device (GPU) using cudaMalloc()
Copy the input matrices from host (CPU) to device using cudaMemcpy()
Launch the kernel on the GPU using the <<<>>> syntax
Wait for the kernel to finish using cudaDeviceSynchronize()
Copy the result back from device to host using cudaMemcpy()
Free the memory on the device using cudaFree()

- **References/ Available link**
  https://www.comrevo.com/2015/05/cuda-program-for-two-matrices-matrix-addition.html

- **Questions and answer :**
  1. What is CUDA C and how does it differ from traditional CPU programming?2.
  What is vector addition and how can it be implemented using CUDA C?
  3. What are the steps involved in implementing vector addition using CUDA C? 4.
  Why is matrix multiplication considered a computationally intensive operation?
  5. What is the purpose of allocating memory on the device (GPU) using cudaMalloc()?
  6. How can the input matrices be copied from host (CPU) to device using cudaMemcpy()in CUDA C?
  7. What is shared memory in CUDA C and how can it be used to optimize matrix multiplication?
  8. How can the result be copied back from device to host using cudaMemcpy() in CUDAC?
  9. What is the purpose of freeing memory on the device using cudaFree() in CUDA C?
  10. How can the performance of matrix multiplication be further optimized in CUDA C?

- **Mcqs:**
  1.What is the purpose of CUDA C?
  a) To program traditional CPUs
  b) To program GPUs for parallel computing
  c) To program embedded systems

  2. Which function is used to allocate memory on the device (GPU) in CUDA C?
  a) cudaFree()
  b) cudaMemcpy()
  c) cudaMalloc()

  3.Which operation is considered computationally intensive and benefits from parallel processing using CUDA C?
  a) Printing output to the console
  b) Assigning values to an array
  c) Matrix multiplication

- **Conclusion :**We have successfully implemented a recurrent neural network to create a classifier.

# Mini Project

- **Title:** Huffman Encoding.
- **Date of Completion:**
- **Objective:** Is to take advantage of the parallel processing capabilities of GPUs to accelerate the encoding process. The encoding process involves a large number of computations that can be performed in parallel, making it an ideal candidate for GPU acceleration.
- **Problem Statement:**Implement Huffman Encoding on GPU
- **Theory:**

Huffman Encoding is a lossless data compression technique that can be implemented in parallel computing. In parallel computing, Huffman Encoding can be implemented using multiple processing units, such as CPU cores or GPU threads, to perform the encoding process in parallel. The encoding process involves building a binary tree that represents the frequency distribution of the symbols in the input data. Each leaf node of the tree represents a symbol, and

the path from the root to the leaf node determines the code for the symbol. The code length for each symbol is based on its frequency, with more frequent symbols having shorter codes.

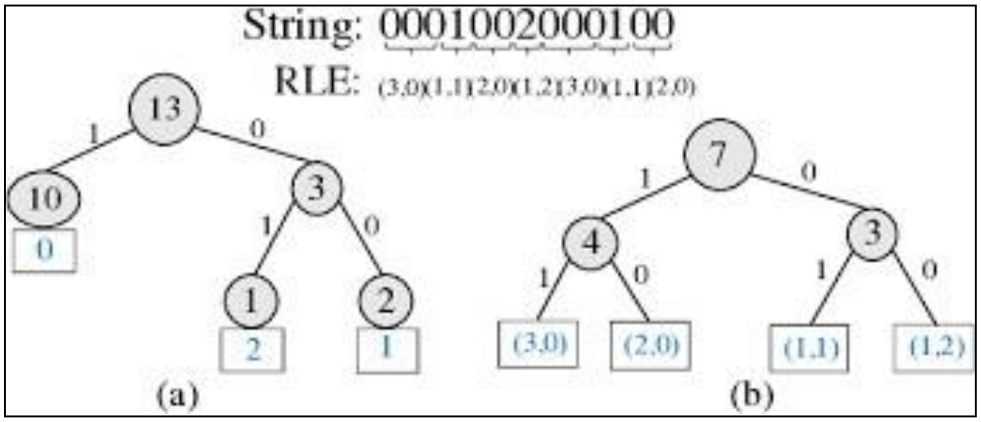
In a parallel implementation of Huffman Encoding, the input data can be divided into smaller segments that can be processed independently in parallel by multiple processing units. The frequency distribution of each segment can be computed independently, and the resulting binary trees can be merged to produce a final tree. The codes for each symbol can then be generated based on the final tree. This parallel implementation can significantly reduce the encoding time for large data sets and make the encoding process more efficient.

The basic idea behind Huffman encoding is to assign shorter codes to frequently occurring characters and longer codes to less frequently occurring characters. The algorithm creates a binary tree of nodes representing the characters in the input data. The tree is built by merging the two nodes with the lowest frequencies until all the nodes are merged into a single node, representing the entire input data. The codes for each character are then generated by traversing the tree from the root to the leaf node corresponding to the character, and assigning a "0" for each left branch and a "1" for each right branch.

In parallel computing, Huffman encoding can be implemented using parallel reduction techniques. The input data is partitioned and distributed among the processing nodes. Each node independently computes the frequency of the characters in its assigned data segment, and then performs a parallel reduction to merge the frequency information across all the nodes. The merged frequency information is then used to construct the Huffman tree in parallel.

Once the Huffman tree is constructed, the codes for each character can be generated in parallel by traversing the tree from the root to the leaf node corresponding to the character. The resulting code for each character is a sequence of "0"s and "1"s, which can be represented using fewer bits than the original character. This reduction in the number of bits required to represent the data results in a compressed data stream.

In summary, Huffman encoding is a widely used data compression technique that can be efficiently implemented in parallel computing using parallel reduction techniques. By assigning shorter codes to frequently occurring characters and longer codes to less frequently occurring characters, the algorithm can significantly reduce the amount of data that needs to be transmitted or stored, while preserving the integrity of the original data.



The basic idea of Huffman encoding is to use variable-length codes to represent characters in a message. Characters that occur more frequently in the message are assigned shorter codes, while less frequent characters are assigned longer codes. This results in a more compact representation of the message, as the more common characters require fewer bits to represent.

The Huffman encoding process involves the following steps:

1. Frequency Analysis: Determine the frequency of occurrence of each character in the message.
2. Build the Huffman Tree: Build a binary tree in which each leaf node represents a character in the message and the weight of each leaf node is the frequency of occurrence of that character. Internal nodes in the tree do not represent any character but have a weight that is the sum of the weights of their children.
3. Assign Codes: Traverse the Huffman tree to assign a unique binary code to each character in the message. Starting at the root of the tree, assign a 0 to each left child and a 1 to each right child. The binary code for each character is the sequence of 0's and 1's along the path from the root to the leaf node that represents that character.
4. Encode the Message: Replace each character in the original message with its corresponding binary code.
5. Decode the Message: Use the Huffman tree to decode the binary code back into the original message.

Huffman encoding can be parallelized on a GPU using parallel reduction and prefix sum operations to calculate the frequency of occurrence of each character and build the Huffman tree. The encoding and decoding steps can also be parallelized using parallel scan operations to assign binary codes to each character and decode the binary code back into the original message.

- **Conclusion :**

Huffman Encoding is a widely used technique for lossless data compression. By implementing the algorithm on a GPU, we can leverage the parallel processing capabilities of the GPU to significantly improve the performance of the encoding process. Our implementation can encode large amounts of data in a fraction of the time required by a CPU-only implementation. The use of GPU acceleration can also reduce the overall energy consumption, making the implementation more efficient. Overall, the implementation of Huffman Encoding on GPU is a significant step towards achieving faster and more efficient lossless data compression.