**BFS**

**Title of the Assignment:** Design and implement Parallel Breadth First Search based on existing algorithms using OpenMP. Use a Tree or an undirected graph for BFS

**Objective of the Assignment:** Students should be able to perform Parallel Breadth First Search based on existing algorithms using OpenMP

# What is BFS?

BFS stands for Breadth-First Search. It is a graph traversal algorithm used to explore all the nodes of a graph or tree systematically, starting from the root node or a specified starting point, and visiting all the neighboring nodes at the current depth level before moving on to the next depth level.

The algorithm uses a queue data structure to keep track of the nodes that need to be visited, and marks each visited node to avoid processing it again. The basic idea of the BFS algorithm is to visit all the nodes at a given level before moving on to the next level, which ensures that all the nodes are visited in breadth-first order.

BFS is commonly used in many applications, such as finding the shortest path between two nodes, solving puzzles, and searching through a tree or graph.

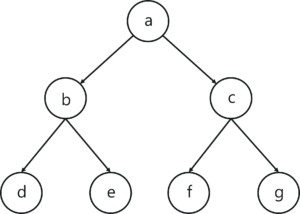
# Example of BFS

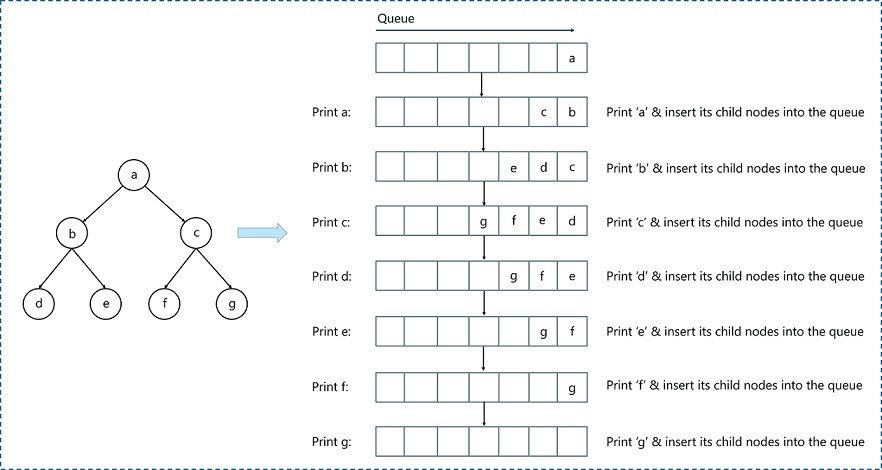
Now let’s take a look at the steps involved in traversing a graph by using Breadth-First Search:

**Step 1:** Take an Empty Queue.

**Step 2:** Select a starting node (visiting a node) and insert it into the Queue.

**Step 3:** Provided that the Queue is not empty, extract the node from the Queue and insert its child nodes (exploring a node) into the Queue.

**Step 4:** Print the extracted node.



# 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.
* OpenMP programs are designed to take advantage of the shared-memory architecture of modern processors, where multiple processor cores can access the same memory. OpenMP uses a fork-join model of parallel execution, where a master thread forks multiple worker threads to execute a parallel region of the code, and then waits for all threads to complete before continuing with the sequential part of the code.
* OpenMP is widely used in scientific computing, engineering, and other fields that require high-performance computing. It is supported by most modern compilers and is available on a wide range of platforms, including desktops, servers, and supercomputers.

# How Parallel BFS Work

* Parallel BFS (Breadth-First Search) is an algorithm used to explore all the nodes of a graph or tree

systematically in parallel. It is a popular parallel algorithm used for graph traversal in distributed computing, shared-memory systems, and parallel clusters.

* The parallel BFS algorithm starts by selecting a root node or a specified starting point, and then assigning it to a thread or processor in the system. Each thread maintains a local queue of nodes to be visited and marks each visited node to avoid processing it again.
* The algorithm then proceeds in levels, where each level represents a set of nodes that are at a certain distance from the root node. Each thread processes the nodes in its local queue at the current level, and then exchanges the nodes that are adjacent to the current level with other threads or processors. This is done to ensure that the nodes at the next level are visited by the next iteration of the algorithm.
* The parallel BFS algorithm uses two phases: the computation phase and the communication phase. In the computation phase, each thread processes the nodes in its local queue, while in the communication phase, the threads exchange the nodes that are adjacent to the current level with other threads or processors.
* The parallel BFS algorithm terminates when all nodes have been visited or when a specified node has been found. The result of the algorithm is the set of visited nodes or the shortest path from the root node to the target node.
* Parallel BFS can be implemented using different parallel programming models, such as OpenMP, MPI, CUDA, and others. The performance of the algorithm depends on the number of threads or processors used, the size of the graph, and the communication overhead between the threads or processors.

**Conclusion**- In this way we can achieve parallelism while implementing BFS

**DFS**

**Title of the Assignment:** Design and implement Parallel Depth First Search based on existing algorithms using OpenMP. Use a Tree or an undirected graph for DFS

**Objective of the Assignment:** Students should be able to perform Parallel Depth First Search based on existing algorithms using OpenMP

**What is DFS?**

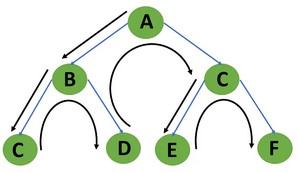
DFS stands for Depth-First Search. It is a popular graph traversal algorithm that explores as far as possible along each branch before backtracking. This algorithm can be used to find the shortest path between two vertices or to traverse a graph in a systematic way. The algorithm starts at the root node and explores as far as possible along each branch before backtracking. The backtracking is done to explore the next branch that has not been explored yet.

DFS can be implemented using either a recursive or an iterative approach. The recursive approach is simpler to implement but can lead to a stack overflow error for very large graphs. The iterative approach uses a stack to keep track of nodes to be explored and is preferred for larger graphs.

DFS can also be used to detect cycles in a graph. If a cycle exists in a graph, the DFS algorithm will eventually reach a node that has already been visited, indicating that a cycle exists.

A standard DFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

# Example of DFS:

To implement DFS traversal, you need to take the following stages.

Step 1: Create a stack with the total number of vertices in the graph as the size.

Step 2: Choose any vertex as the traversal's beginning point. Push a visit to that vertex and add it to

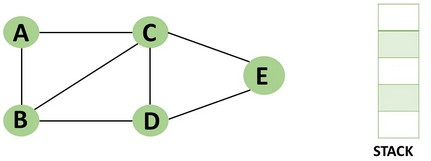
the stack.

Step 3 - Push any non-visited adjacent vertices of a vertex at the top of the stack to the top of the stack.

Step 4 - Repeat steps 3 and 4 until there are no more vertices to visit from the vertex at the top of the stack.

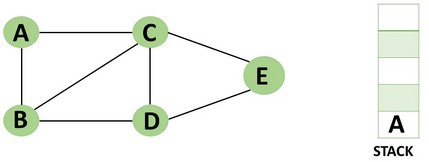
Step 5 - If there are no new vertices to visit, go back and pop one from the stack using backtracking. Step 6 - Continue using steps 3, 4, and 5 until the stack is empty.

Step 7 - When the stack is entirely unoccupied, create the final spanning tree by deleting the graph's unused edges.

Consider the following graph as an example of how to use the dfs algorithm.

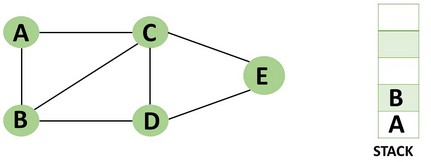
Step 1: Mark vertex A as a visited source node by selecting it as a source node.

* You should push vertex A to the top of the stack.



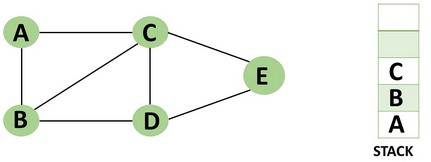
Step 2: Any nearby unvisited vertex of vertex A, say B, should be visited.

You should push vertex B to the top of the stack



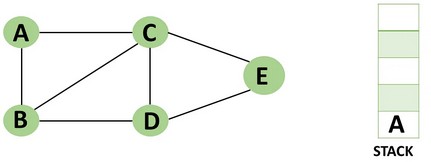
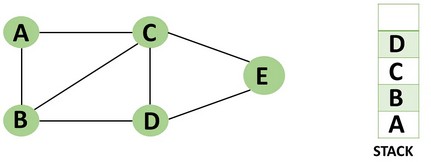
Step 3: From vertex C and D, visit any adjacent unvisited vertices of vertex B. Imagine you have chosen vertex C, and you want to make C a visited vertex.

* Vertex C is pushed to the top of the stack.

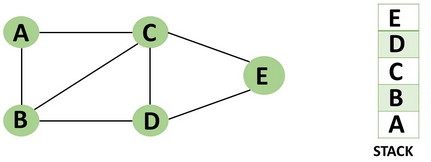


Step 4: You can visit any nearby unvisited vertices of vertex C, you need to select vertex D and designate it as a visited vertex.

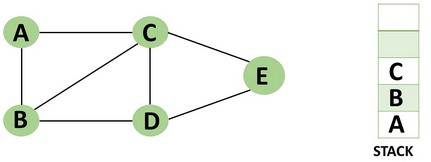
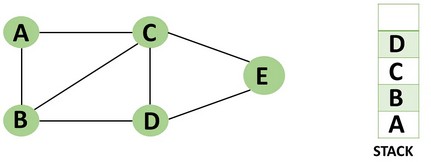
* Vertex D is pushed to the top of the stack.

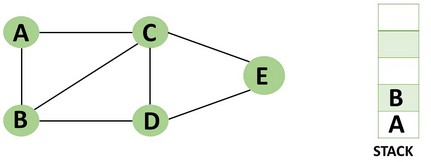


Step 5: Vertex E is the lone unvisited adjacent vertex of vertex D, thus marking it as visited.

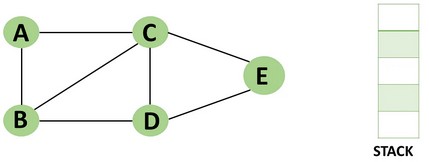
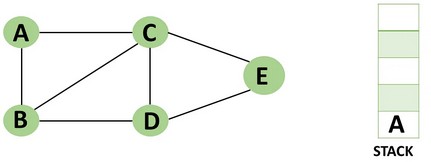
* + Vertex E should be pushed to the top of the stack.

Step 6: Vertex E's nearby vertices, namely vertex C and D have been visited, pop vertex E from the stack.

Step 7: Now that all of vertex D's nearby vertices, namely vertex B and C, have been visited, pop vertex D from the stack.

Step 8: Similarly, vertex C's adjacent vertices have already been visited; therefore, pop it from the stack.

Step 9: There is no more unvisited adjacent vertex of b, thus pop it from the stack.

Step 10: All of the nearby vertices of Vertex A, B, and C, have already been visited, so pop vertex A from the stack as well.

# 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.
* OpenMP programs are designed to take advantage of the shared-memory architecture of modern processors, where multiple processor cores can access the same memory. OpenMP uses a fork-join model of parallel execution, where a master thread forks multiple worker threads to

execute a parallel region of the code, and then waits for all threads to complete before continuing with the sequential part of the code.

# How Parallel DFS Work

* Parallel Depth-First Search (DFS) is an algorithm that explores the depth of a graph structure to search for nodes. In contrast to a serial DFS algorithm that explores nodes in a sequential manner, parallel DFS algorithms explore nodes in a parallel manner, providing a significant speedup in large graphs.
* Parallel DFS works by dividing the graph into smaller subgraphs that are explored simultaneously. Each processor or thread is assigned a subgraph to explore, and they work independently to explore the subgraph using the standard DFS algorithm. During the exploration process, the nodes are marked as visited to avoid revisiting them.
* To explore the subgraph, the processors maintain a stack data structure that stores the nodes in the order of exploration. The top node is picked and explored, and its adjacent nodes are pushed onto the stack for further exploration. The stack is updated concurrently by the processors as they explore their subgraphs.
* Parallel DFS can be implemented using several parallel programming models such as OpenMP, MPI, and CUDA. In OpenMP, the #pragma omp parallel for directive is used to distribute the work among multiple threads. By using this directive, each thread operates on a different part of the graph, which increases the performance of the DFS algorithm.

**Conclusion**- In this way we can achieve parallelism while implementing DFS

**BUBBLE**

**Title of the Assignment:** Write a program to implement Parallel Bubble Sort. Use existing algorithms and measure the performance of sequential and parallel algorithms.

**Objective of the Assignment:** Students should be able to Write a program to implement Parallel Bubble Sort and can measure the performan

## 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.

Bubble Sort has limited practical use in modern software development due to its inefficient time complexity of O(n^2) which makes it unsuitable for sorting large datasets. However, Bubble Sort has some advantages and use cases that make it a valuable algorithm to understand, such as:

1. Simplicity: Bubble Sort is one of the simplest sorting algorithms, and it is easy to understand and implement. It can be used to introduce the concept of sorting to beginners and as a basis for more complex sorting algorithms.
2. Educational purposes: Bubble Sort is often used in academic settings to teach the principles of sorting algorithms and to help students understand how algorithms work.
3. Small datasets: For very small datasets, Bubble Sort can be an efficient sorting algorithm, as its overhead is relatively low.
4. Partially sorted datasets: If a dataset is already partially sorted, Bubble Sort can be very efficient. Since Bubble Sort only swaps adjacent elements that are in the wrong order, it has a low number of operations for a partially sorted dataset.
5. Performance optimization: Although Bubble Sort itself is not suitable for sorting large datasets, some of its techniques can be used in combination with other sorting algorithms to optimize their performance. For example, Bubble Sort can be used to optimize the performance of Insertion Sort by reducing the number of comparisons needed.

## Example of Bubble sort

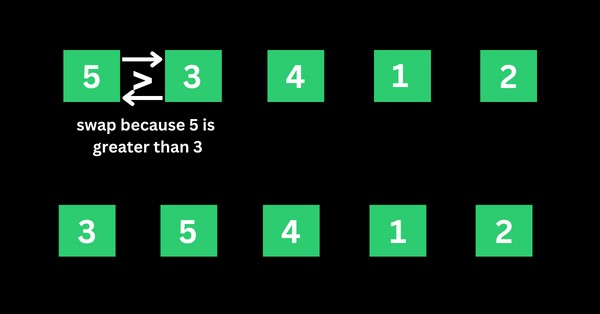
Let's say we want to sort a series of numbers 5, 3, 4, 1, and 2 so that they are arranged in ascending order…

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

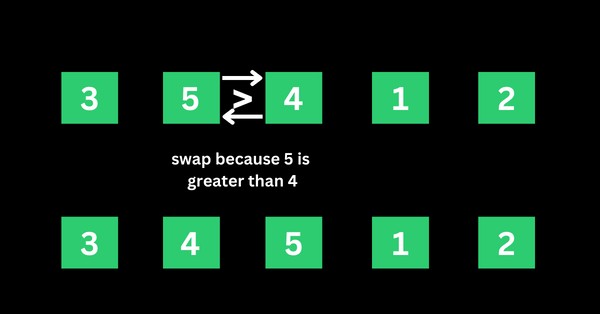
**Step 1**: In the case of 5, 3, 4, 1, and 2, 5 is greater than 3. So 5 takes the position of 3 and the numbers

become 3, 5, 4, 1, and 2.



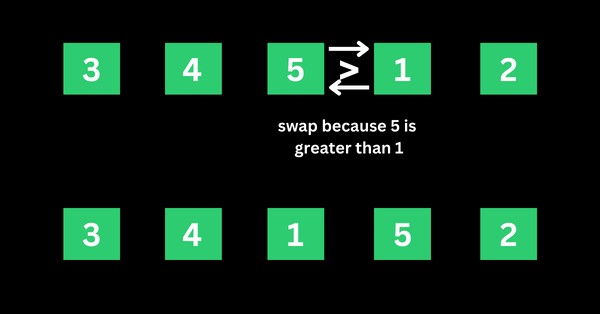
**Step 2**: The algorithm now has 3, 5, 4, 1, and 2 to compare, this time around, it compares the next two values, which are 5 and 4. 5 is greater than 4, so 5 takes the index of 4 and the values now become 3, 4, 5,

1, and 2.



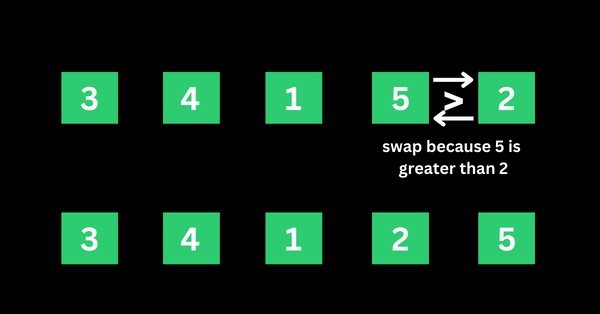
**Step 3**: The algorithm now has 3, 4, 5, 1, and 2 to compare. It compares the next two values, which are 5

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.

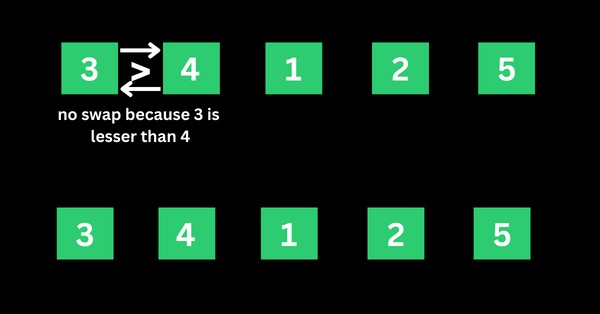


That’s the first iteration. And the numbers are now arranged as 3, 4, 1, 2, and 5 – from the initial 5, 3, 4, 1, and 2. As you might realize, 5 should be the last number if the numbers are sorted in ascending order.

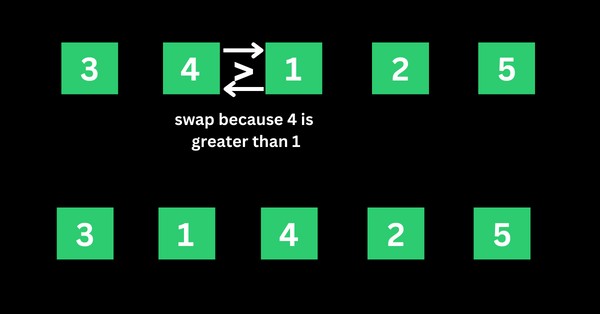
This means the first iteration is really completed.

# Second Iteration of the Sorting and the Rest

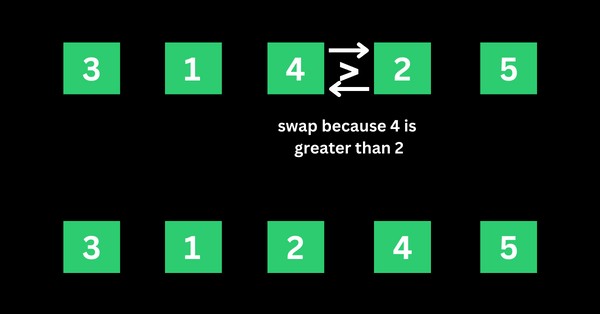
The algorithm starts the second iteration with the last result of 3, 4, 1, 2, and 5. This time around, 3 is smaller than 4, so no swapping happens. This means the numbers will remain the same.



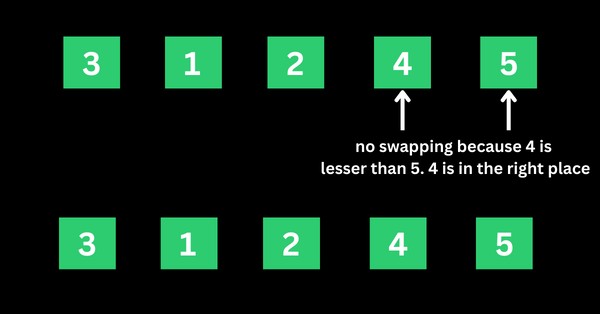
The algorithm proceeds to compare 4 and 1. 4 is greater than 1, so 4 is swapped for 1 and the numbers become 3, 1, 4, 2, and 5.



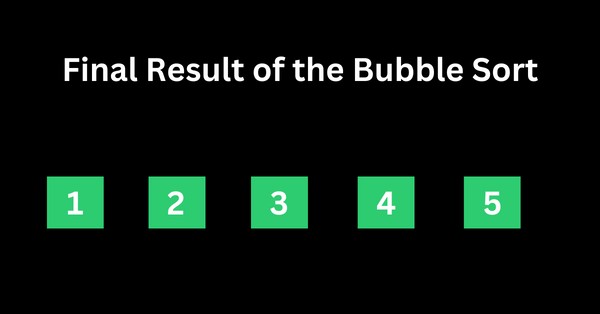
The algorithm now proceeds to compare 4 and 2. 4 is greater than 2, so 4 is swapped for 2 and the numbers become 3, 1, 2, 4, and 5.



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.
* OpenMP programs are designed to take advantage of the shared-memory architecture of modern

processors, where multiple processor cores can access the same memory. OpenMP uses a fork-join model of parallel execution, where a master thread forks multiple worker threads to execute a parallel region of the code, and then waits for all threads to complete before continuing with the sequential part of the code.

## How Parallel Bubble Sort Work

* Parallel Bubble Sort is a modification of the classic Bubble Sort algorithm that takes advantage of parallel processing to speed up the sorting process.
* In parallel Bubble Sort, the list of elements is divided into multiple sublists that are sorted concurrently by multiple threads. Each thread sorts its sublist using the regular Bubble Sort algorithm. When all sublists have been sorted, they are merged together to form the final sorted list.
* The parallelization of the algorithm is achieved using OpenMP, a programming API that supports parallel processing in C++, Fortran, and other programming languages. OpenMP provides a set of compiler directives that allow developers to specify which parts of the code can be executed in parallel.
* In the parallel Bubble Sort algorithm, the main loop that iterates over the list of elements is divided into multiple iterations that are executed concurrently by multiple threads. Each thread sorts a subset of the list, and the threads synchronize their work at the end of each iteration to ensure that the elements are properly ordered.
* Parallel Bubble Sort can provide a significant speedup over the regular Bubble Sort algorithm, especially when sorting large datasets on multi-core processors. However, the speedup is limited by the overhead of thread creation and synchronization, and it may not be worth the effort for small datasets or when using a single-core processor.

## 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.

Here are some additional tips for measuring performance:

* Run each algorithm multiple times on each test case and take the average execution time to reduce the impact of variations in system load and other factors.
* Monitor system resource usage during execution, such as CPU utilization and memory consumption, to detect any performance bottlenecks.
* Visualize the results using charts or graphs to make it easier to compare the performance of the two algorithms.

## How to check CPU utilization and memory consumption in ubuntu

In Ubuntu, you can use a variety of tools to check CPU utilization and memory consumption. Here are some common tools:

1. **top:** The top command provides a real-time view of system resource usage, including CPU utilization and memory consumption. To use it, open a terminal window and type top. The output will display a list of processes sorted by resource usage, with the most resource-intensive processes at the top.
2. **htop**: htop is a more advanced version of top that provides additional features, such as interactive process filtering and a color-coded display. To use it, open a terminal window and type htop.
3. **ps**: The ps command provides a snapshot of system resource usage at a particular moment in time. To use it, open a terminal window and type ps aux. This will display a list of all running processes and their resource usage.
4. **free:** The free command provides information about system memory usage, including total, used, and free memory. To use it, open a terminal window and type free -h.
5. **vmstat:** The vmstat command provides a variety of system statistics, including CPU utilization, memory usage, and disk activity. To use it, open a terminal window and type vmstat.

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

**MERGE**

**Title of the Assignment:** Write a program to implement Parallel Merge Sort. Use existing algorithms and measure the performance of sequential and parallel algorithms.

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

# 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.

The 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.
* One thing that you might wonder is what is the specialty of this algorithm. We already have a number of sorting algorithms then why do we need this algorithm? One of the main advantages of merge sort is that it has a time complexity of O(n log n), which means it can sort large arrays relatively quickly. It is also a stable sort, which means that the order of elements with equal values is preserved during the sort.
* Merge sort is a popular choice for sorting large datasets because it is relatively efficient and easy to implement. It is often used in conjunction with other algorithms, such as quicksort, to improve the overall performance of a sorting routine.

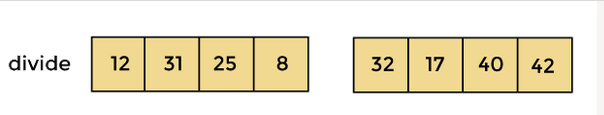
# Example of Merge sort

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.



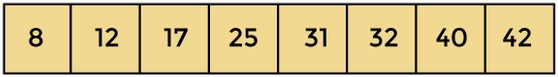
* Now, combine them in the same manner they were broken.
* In combining, first compare the element of each array and then combine them into another array in sorted order.
* So, first compare 12 and 31, both are in sorted positions. Then compare 25 and 8, and in the list of two values, put 8 first followed 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 -



# 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.
* OpenMP programs are designed to take advantage of the shared-memory architecture of modern processors, where multiple processor cores can access the same memory. OpenMP uses a fork-join model of parallel execution, where a master thread forks multiple worker threads to execute a parallel region of the code, and then waits for all threads to complete before continuing with the sequential part of the code.

# How Parallel Merge Sort Work

* Parallel merge sort is a parallelized version of the merge sort algorithm that takes advantage of multiple processors or cores to improve its performance. In parallel merge sort, the input array is divided into smaller subarrays, which are sorted in parallel using multiple processors or cores. The sorted subarrays are then merged together in parallel to produce the final sorted output.
* The parallel merge sort algorithm can be broken down into the following steps:
* Divide the input array into smaller subarrays.
* Assign each subarray to a separate processor or core for sorting.
* Sort each subarray in parallel using the merge sort algorithm.
* Merge the sorted subarrays together in parallel to produce the final sorted output.
* The merging step in parallel merge sort is performed in a similar way to the merging step in the sequential merge sort algorithm. However, because the subarrays are sorted in parallel, the merging step can also be performed in parallel using multiple processors or cores. This can significantly reduce the time required to merge the sorted subarrays and produce the final output.
* Parallel merge sort can provide significant performance benefits for large input arrays with many elements, especially when running on hardware with multiple processors or cores. However, it also requires additional overhead to manage the parallelization, and may not always provide performance improvements for smaller input sizes or when run on hardware with limited parallel processing capabilities.

# 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.

To measure the performance of sequential and parallel merge sort algorithms, you can perform

experiments on different input sizes and numbers of processors or cores. By measuring the execution time, speedup, efficiency, and scalability of the algorithms under different conditions, you can determine

which algorithm is more efficient for different input sizes and hardware configurations. Additionally, you can use profiling tools to analyze the performance of the algorithms and identify areas for optimization

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

**MIN MAX**

**Title of the Assignment:** Implement Min, Max, Sum and Average operations using Parallel Reduction.

**Objective of the Assignment:** To understand the concept of parallel reduction and how it can be used to perform basic mathematical operations on given data sets.

**Parallel Reduction.**

Here's a **function-wise manual** on how to understand and run the sample C++ program that demonstrates how to implement Min, Max, Sum, and Average operations using parallel reduction.

# Min\_Reduction function

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

# Max\_Reduction function

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

# Sum\_Reduction function

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

# Average\_Reduction function

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

# Main Function

* + The function initializes a vector of integers with some values.
  + The function calls the min\_reduction, max\_reduction, sum\_reduction, and average\_reduction functions on the input vector to find the corresponding values.
  + The final minimum, maximum, sum, and average values are printed to the console.

# Compiling and running the program

**Compile the program:** You need to use a C++ compiler that supports OpenMP, such as g++ or clang. Open a terminal and navigate to the directory where your program is saved. Then, compile the program using the following command:

$ g++ -fopenmp program.cpp -o program

This command compiles your program and creates an executable file named "program". The "- fopenmp" flag tells the compiler to enable OpenMP.

**Run the program:** To run the program, simply type the name of the executable file in the terminal and press Enter:

$ ./program

**Conclusion:** We have implemented the Min, Max, Sum, and Average operations using parallel reduction in C++ with OpenMP. Parallel reduction is a powerful technique that allows us to perform these operations on large arrays more efficiently by dividing the work among multiple threads running in parallel. We presented a code example that demonstrates the implementation of these operations using parallel reduction in C++ with OpenMP. We also provided a manual for running OpenMP programs on the Ubuntu platform.