HPC 1

Bfs

This code is an implementation of the Breadth-First Search (BFS) algorithm using OpenMP for parallel execution. It allows you to perform a BFS traversal on a graph represented by an adjacency matrix.

Let's break down the code step by step:

1. The code includes the necessary header files `iostream`, `queue`, and `omp.h` for input/output operations, queue data structure, and OpenMP functionality, respectively.
2. The constant `MAX` is defined with a value of 1000, which represents the maximum size of the graph.
3. The global variables `graph` and `visited` are declared. `graph` is a two-dimensional array representing the adjacency matrix of the graph, and `visited` is an array to keep track of visited vertices during the BFS traversal.
4. The `bfs` function is defined to perform the BFS traversal. It takes two parameters: `start`, which is the starting vertex, and `n`, which is the number of vertices in the graph.
5. Inside the `bfs` function, a `queue` object `q` is created to store the vertices to be visited. The `visited` array is initialized, and the starting vertex is marked as visited and enqueued.
6. The BFS algorithm is implemented using a `while` loop that continues until the queue is empty. Inside the loop, the front element of the queue is dequeued and stored in the variable `curr`.
7. The `for` loop following the `#pragma omp parallel for` directive is the parallelized section of the code. It iterates over all vertices `i` from 0 to `n-1`. The `#pragma omp parallel for` directive enables parallel execution of the loop by distributing iterations among multiple threads. The `shared(graph, visited, q)` clause specifies that the variables `graph`, `visited`, and `q` are shared among all threads. The `schedule(dynamic)` clause dynamically distributes loop iterations among threads to improve load balancing.
8. Inside the `for` loop, the code checks if there is an edge between the current vertex `curr` and the vertex `i`, and if `i` has not been visited before. If both

conditions are true, `i` is marked as visited, and it is enqueued in the queue `q` for further exploration.

9. The `main` function is where the execution of the program starts. It prompts the user to enter the number of vertices `n` and the adjacency matrix of the graph.
10. After reading the input graph, the user is asked to enter the starting vertex.
11. The `#pragma omp parallel num_threads(4)` directive parallelizes the execution of the following block of code. It specifies that the parallel region should use 4 threads.
12. Inside the parallel region, the `bfs` function is called with the starting vertex `start` and the number of vertices `n`.
13. Once the parallel region ends, the main thread continues execution and outputs the BFS traversal of the graph.
14. The BFS traversal is printed by iterating over the vertices and checking if they have been visited. If a vertex is visited, its index is printed.
15. Finally, the program ends with the `return 0` statement.

That's the explanation of the given code. It uses OpenMP to parallelize the BFS traversal of a graph.

Here's a shorter explanation of the given code:

1. The code implements the Breadth-First Search (BFS) algorithm using OpenMP for parallel execution.
2. It defines a graph represented by an adjacency matrix and an array to track visited vertices.
3. The `bfs` function performs the BFS traversal starting from a given vertex.
4. The main function prompts the user for the number of vertices and adjacency matrix.
5. It also prompts for the starting vertex and calls the `bfs` function in parallel using OpenMP.
6. After the parallel execution, the BFS traversal is printed.
7. The code utilizes OpenMP directives like `parallel for` and `num_threads` to parallelize the BFS algorithm.
8. The program terminates after printing the BFS traversal.

Dfs

This code is an implementation of the Depth-First Search (DFS) algorithm using OpenMP for parallel execution. It allows you to perform a DFS traversal on a graph represented by an adjacency matrix.

Here's a breakdown of the code:

1. The code includes the necessary header files `iostream`, `stack`, and `omp.h` for input/output operations, stack data structure, and OpenMP functionality, respectively.
2. The constant `MAX` is defined with a value of 1000, which represents the maximum size of the graph.
3. The global variables `graph` and `visited` are declared. `graph` is a two-dimensional array representing the adjacency matrix of the graph, and `visited` is an array to keep track of visited vertices during the DFS traversal.
4. The `dfs` function is defined to perform the DFS traversal. It takes two parameters: `start`, which is the starting vertex, and `n`, which is the number of vertices in the graph.
5. Inside the `dfs` function, a `stack` object `s` is created to store the vertices to be visited. The starting vertex is pushed onto the stack.
6. The DFS algorithm is implemented using a `while` loop that continues until the stack is empty. Inside the loop, the top element of the stack is popped and stored in the variable `curr`.
7. The code checks if the current vertex `curr` has not been visited. If it hasn't been visited, the vertex is marked as visited, and the `for` loop following the `#pragma omp parallel for` directive is executed.
8. The `for` loop iterates over all vertices `i` from 0 to `n-1`. The `#pragma omp parallel for` directive enables parallel execution of the loop by distributing iterations among multiple threads. The `shared(graph, visited, s)` clause specifies that the variables `graph`, `visited`, and `s` are shared among all threads. The `schedule(dynamic)` clause dynamically distributes loop iterations among threads to improve load balancing.
9. Inside the `for` loop, the code checks if there is an edge between the current vertex `curr` and the vertex `i`, and if `i` has not been visited before. If both conditions are true, `i` is pushed onto the stack for further exploration.
10. The `main` function is where the execution of the program starts. It prompts the user to enter the number of vertices `n` and the adjacency matrix of the graph.
11. After reading the input graph, the user is asked to enter the starting vertex.

12. The `#pragma omp parallel num_threads(4)` directive parallelizes the execution of the following block of code. It specifies that the parallel region should use 4 threads.
13. Inside the parallel region, the `dfs` function is called with the starting vertex `start` and the number of vertices `n`.
14. Once the parallel region ends, the main thread continues execution and outputs the DFS traversal of the graph.
15. The DFS traversal is printed by iterating over the vertices and checking if they have been visited. If a vertex is visited, its index is printed.
16. Finally, the program ends with the `return 0` statement.

That's the explanation of the given code. It uses OpenMP to parallelize the DFS traversal of a graph.

Here's a shorter explanation of the given code:

1. The code implements the Depth-First Search (DFS) algorithm using OpenMP for parallel execution.
2. It defines a graph represented by an adjacency matrix and an array to track visited vertices.
3. The `dfs` function performs the DFS traversal starting from a given vertex.
4. The main function prompts the user for the number of vertices and adjacency matrix.
5. It also prompts for the starting vertex and calls the `dfs` function in parallel using OpenMP.
6. After the parallel execution, the DFS traversal is printed.
7. The code utilizes OpenMP directives like `parallel for` and `num_threads` to parallelize the DFS algorithm.
8. The program terminates after printing the DFS traversal.

HPC2

Bubble

This code is an implementation of the bubble sort algorithm with parallelization using OpenMP. It sorts an array of integers in ascending order.

Here's a breakdown of the code:

1. The code includes the necessary header files `omp.h`, `stdio.h`, and `stdlib.h` for OpenMP functionality, standard input/output operations, and standard library functions.
2. The `swap` function is defined, which swaps two integer values passed as pointers.
3. The `main` function is the entry point of the program.
4. It defines the `SIZE` of the array and initializes an array `A` of size `SIZE` with random integers.
5. It also defines the variable `N` to store the size of the array.
6. Inside the main function, there are variables `i`, `j`, and `first`.
7. The program records the start time using `omp_get_wtime()`.
8. The main sorting algorithm is implemented using a nested loop structure. The outer loop `i` iterates from 0 to `N-1`.
9. Inside the outer loop, the variable `first` is set to `i % 2`, which determines the starting point for the inner loop.
10. The `#pragma omp parallel for` directive parallelizes the inner loop. It distributes the iterations of the inner loop among multiple threads.
11. The inner loop `j` starts from `first` and iterates up to `N-1`, incrementing by 1.
12. Inside the inner loop, the code checks if the current element `A[j]` is greater than the next element `A[j+1]`. If true, it swaps the elements using the `swap` function.
13. After completing the outer loop, the program records the end time using `omp_get_wtime()`.
14. It then prints the sorted array `A` using a loop.
15. Finally, it prints the time taken for the parallel execution of the sorting algorithm.
16. The `swap` function is defined outside the `main` function and performs the swapping of two integers using a temporary variable.

That's the explanation of the given code. It utilizes OpenMP to parallelize the bubble sort algorithm for sorting an array.

Here's a shorter explanation of the given code:

1. The code implements the bubble sort algorithm with parallelization using OpenMP.
2. It defines an array `A` of integers and initializes it with random values.
3. The code uses a nested loop structure to perform the bubble sort.
4. The outer loop iterates from 0 to `N-1`, where `N` is the size of the array.
5. Inside the outer loop, the variable `first` is set based on the current iteration.
6. The inner loop is parallelized using the `#pragma omp parallel for` directive.
7. The inner loop compares adjacent elements of the array and swaps them if necessary.
8. After the sorting is complete, the sorted array is printed.
9. The execution time of the parallel sorting is also printed.
10. The `swap` function is defined to swap two integer values.
11. That's the explanation of the given code. It utilizes OpenMP to parallelize the bubble sort algorithm for sorting an array.

Merge -

This code implements the merge sort algorithm using OpenMP for parallel execution. Merge sort is a divide-and-conquer algorithm that recursively divides the input array into smaller halves, sorts them, and then merges them back together.

Here's a breakdown of the code:

1. The code includes the necessary header files `iostream`, `stdlib.h`, and `omp.h` for input/output operations, standard library functions, and OpenMP functionality.
2. The `mergesort` function is defined to perform the merge sort algorithm. It takes an array `a`, and the indices `i` and `j` representing the range of elements to be sorted.
3. Inside the `mergesort` function, there is a base case that checks if `i` is less than `j`. If true, the array is divided into two halves, and each half is sorted using parallel sections.
4. The `#pragma omp parallel sections` directive parallelizes the execution of the following sections.
5. The first section sorts the first half of the array by recursively calling `mergesort` with appropriate indices.
6. The second section sorts the second half of the array by recursively calling `mergesort` with appropriate indices.
7. After the two halves are sorted, the `merge` function is called to merge them back together.
8. The `merge` function takes the array `a` and four indices representing the subarrays to be merged.
9. Inside the `merge` function, a temporary array `temp` is created to store the merged result.
10. Using a while loop, the elements from the two subarrays are compared, and the smaller element is placed in the `temp` array.
11. The remaining elements from any of the subarrays are then copied to the `temp` array.
12. Finally, the elements from the `temp` array are copied back to the original array `a`.
13. The `main` function is where the execution of the program starts.
14. It prompts the user to enter the total number of elements `n` and the elements of the array.
15. The `mergesort` function is called with the array `a`, starting index 0, and ending index `n-1`.
16. After the sorting is complete, the sorted array is printed.

17. The memory allocated for the array `a` is freed.
18. That's the explanation of the given code. It utilizes OpenMP to parallelize the merge sort algorithm for sorting an array.

Certainly! Here's a shorter explanation of the given code:

1. The code implements the merge sort algorithm with parallelization using OpenMP.
2. It defines two functions: `mergesort` and `merge`.
3. The `mergesort` function recursively divides the array into smaller halves and sorts them using parallel sections.
4. The `merge` function merges two sorted subarrays into a single sorted array.
5. The `main` function prompts the user for the number of elements and the array elements.
6. It calls the `mergesort` function to sort the array in parallel.
7. After sorting, the sorted array is printed.
8. That's the explanation of the given code. It utilizes OpenMP to parallelize the merge sort algorithm for sorting an array.

HPC 3

This code demonstrates the use of OpenMP directives for parallel reduction operations on an array. It calculates the minimum value, maximum value, sum, and average of the elements in the array.

Here's a breakdown of the code:

1. The code includes the necessary header files `iostream`, `omp.h`, `cstdlib`, and `ctime` for input/output operations, OpenMP functionality, and random number generation.
2. The main function is defined.
3. The code initializes the size of the array `n` to 1000 and declares an integer array `arr` of size `n`.
4. The loop generates random values and assigns them to the array elements using the `rand()` function.
5. The minimum operation is performed by initializing `min_val` to the first element of the array and using the `#pragma omp parallel for reduction(min: min_val)` directive. Each thread independently compares its assigned portion of the array with `min_val` and updates `min_val` if a smaller value is found.
6. The maximum operation is performed similarly to the minimum operation but with the `reduction(max: max_val)` directive.
7. The sum operation is performed by initializing `sum` to 0 and using the `#pragma omp parallel for reduction(+ : sum)` directive. Each thread independently calculates the sum of its assigned portion of the array, and the partial sums are combined to obtain the final sum.
8. The average operation is performed similarly to the sum operation, and the average is calculated by dividing the sum by the number of elements in the array.
9. The calculated minimum value, maximum value, sum, and average are printed.
10. The main function returns 0 to indicate successful execution.

In summary, the code showcases how to use OpenMP directives to parallelize reduction operations (min, max, sum, average) on an array, improving the performance by utilizing multiple threads.

Certainly! Here's an even shorter explanation of the given code:

1. The code uses OpenMP directives for parallel reduction operations on an array.
2. It initializes an array `arr` of size 1000 with random values.

3. The minimum value, maximum value, sum, and average of the array elements are calculated in parallel.
4. The calculated values are printed.
5. That's the explanation of the given code. It utilizes OpenMP to perform parallel reduction operations (min, max, sum) on an array.

HPC 4

Add

Certainly! Here's a shorter explanation of the given CUDA code:

1. The code defines the size of the arrays as N=500.
2. The `add` function is declared as a CUDA kernel, which performs element-wise addition of two arrays.
3. In the `main` function, host and device arrays are declared along with device pointers for device memory.
4. Memory allocation is performed on the device using `cudaMalloc`.
5. Host arrays are initialized with values.
6. CUDA events are created to measure the execution time.
7. Data is transferred from host to device memory using `cudaMemcpy`.
8. The `add` kernel is launched with N threads.
9. The result array is copied back from device to host memory.
10. The execution time is recorded and printed.
11. The elements of the arrays are printed.
12. Device memory is freed and the program terminates.

MM

This code demonstrates matrix multiplication using CUDA. Here's a breakdown of the code:

1. The code defines the size of the matrices as N=4.
2. The `matrixMul` function is a CUDA kernel that performs matrix multiplication of matrices `a` and `b` and stores the result in matrix `c`. It uses thread indices and block indices to determine the elements to be computed by each thread.
3. The `displayMatrix` function is a helper function to print the elements of a matrix.
4. In the `main` function, host and device matrices are declared along with device pointers for device memory.
5. Memory allocation is performed on the host matrices.
6. Host matrices `a` and `b` are initialized with values.
7. Memory allocation is performed on the device matrices using `cudaMalloc`.
8. Data is transferred from host to device memory using `cudaMemcpy`.

9. The kernel function `matrixMul` is launched on the GPU. The number of threads per block and the number of blocks per grid are calculated based on the matrix size.
10. The result matrix `c` is copied back from device to host memory using `cudaMemcpy`.
11. The matrices `a`, `b`, and the product matrix `c` are printed using the `displayMatrix` function.
12. Device memory is freed, and host memory is deallocated.

Overall, the code demonstrates how to perform matrix multiplication using CUDA, leveraging parallelism to accelerate the computation.