HPC Practical 3

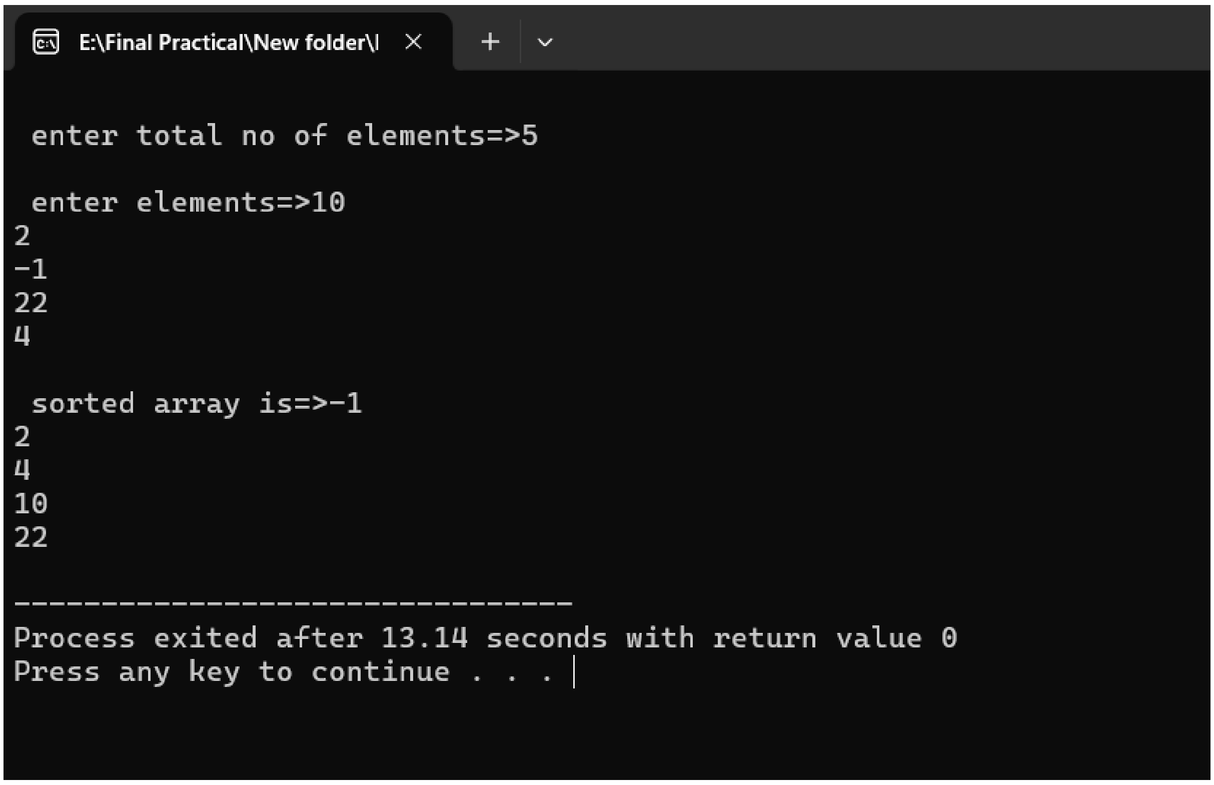
```
#include<iostream>
#include<stdlib.h>
#include<omp.h>
using namespace std;
void bubble(int *, int);
void swap(int &, int &);
void bubble(int *a, int n)
{
    for(  int i = 0;  i < n;  i++ )
     {
     int first = i % 2;
     #pragma omp parallel for shared(a,first)
     for(  int j = first;  j < 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;
 }
```



E:\Final Practical\New folder\I

```
 enter total no of elements=>5

 enter elements=>10
2
-1
22
4

 sorted array is=>-1
2
4
10
22

--------------------------------
Process exited after 13.14 seconds with return value 0
Press any key to continue . . .
```

Let's break down the Bubble Sort step by step for the given example:

1. **Initial Array**:
    - Original Array: 10, 2, -1, 22, 4
2. **First Pass**:
    - In the first pass, the algorithm compares adjacent pairs of elements and swaps them if they are in the wrong order.
    - After the first pass:
        - Step 1: Compare 10 and 2. Since 10 > 2, swap them.
        - Step 2: Compare 10 and -1. Since 10 > -1, swap them.
        - Step 3: Compare 10 and 22. Since 10 < 22, no swap.
        - Step 4: Compare 22 and 4. Since 22 > 4, swap them.
    - Array after first pass: 2, -1, 10, 4, 22
3. **Second Pass**:
    - In the second pass, the algorithm performs the same process as the first pass but stops one element before the end since the largest element has already been placed at the end in the previous pass.
    - After the second pass:
        - Step 1: Compare 2 and -1. Since 2 > -1, swap them.
        - Step 2: Compare 2 and 10. Since 2 < 10, no swap.
        - Step 3: Compare 10 and 4. Since 10 > 4, swap them.
    - Array after second pass: -1, 2, 4, 10, 22
4. **Third Pass**:
    - In the third pass, the algorithm again performs the same process but stops two elements before the end since the last two elements are already in their correct positions.
    - After the third pass:
        - Step 1: Compare -1 and 2. Since -1 < 2, no swap.
    - Array after third pass remains unchanged: -1, 2, 4, 10, 22
5. **Sorted Array**:

- After three passes, the array is sorted:
    - Sorted Array: -1, 2, 4, 10, 22

6. **Output**:
    - The program prints the sorted array: -1, 2, 4, 10, 22

This completes the Bubble Sort algorithm for the given example. Each pass of the algorithm iterates through the array, comparing adjacent elements and swapping them if necessary, gradually moving the largest elements to their correct positions.

Code Explanation

1. **Include Necessary Libraries**: We include the `iostream` library to enable input and output operations.
2. **Function Declarations**: We declare two functions, `bubble` and `swap`, which we'll define later.
3. **Function Definitions**:
    - `bubble`: This function implements the Bubble Sort algorithm. It iterates through the array and compares adjacent elements, swapping them if they're in the wrong order. It uses OpenMP to parallelize the sorting process, which can make the sorting faster.
    - `swap`: This function simply swaps the values of two variables.
4. **Main Function**:
    - We declare an array `a` and a variable `n` to store the number of elements.
    - We ask the user to input the total number of elements (`n`) and the elements themselves. We store these values in the array `a`.
    - We call the `bubble` function to sort the array.
    - Finally, we print the sorted array using a loop.
5. **Bubble Sort Process**:

- The Bubble Sort algorithm works by repeatedly swapping adjacent elements if they're in the wrong order.
- In each iteration, the largest unsorted element "bubbles up" to its correct position.
- The algorithm repeats this process for each element in the array until the entire array is sorted.

6. **Parallelization**:
    - OpenMP allows us to parallelize the sorting process, making it faster by utilizing multiple threads to perform the sorting concurrently.
    - The `#pragma omp parallel for` directive parallelizes the inner loop of the Bubble Sort algorithm, which is the most computationally intensive part.
7. **Output**:
    - The program outputs the sorted array, showing the elements in ascending order.

Overall, the code demonstrates how to use OpenMP to parallelize the Bubble Sort algorithm, making it faster and more efficient for sorting large arrays.

Theory for Practical

📌 What is Bubble sort

Bubble sort is a simple sorting algorithm that works by repeatedly stepping through the list of elements to be sorted, comparing each pair of adjacent items, and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted.

Here's how it works:

1. Start at the beginning of the list.
2. Compare the first two elements. If they are in the wrong order, swap them.
3. Move to the next pair of elements and repeat step 2.
4. Continue this process until the end of the list is reached.

5. If any swaps were made during a pass through the list, repeat the process from step 1. Otherwise, the list is sorted.

Bubble sort is not very efficient for large lists because it has a worst-case and average-case time complexity of O(n^2), where n is the number of elements in the list. However, it is easy to understand and implement, making it a good choice for small lists or educational purposes.

📌 What is Parallel Bubble Sort

Parallel bubble sort is a variation of the traditional bubble sort algorithm that uses parallel processing to sort elements more quickly, especially for large lists. Instead of processing each element sequentially, parallel bubble sort divides the list into smaller sublists and sorts them concurrently using multiple threads or processors.

Here's how parallel bubble sort works:

1. **Divide the List**: The list of elements to be sorted is divided into smaller sublists, each of which can be sorted independently.
2. **Sort Sublists in Parallel**: Each sublist is sorted independently using the bubble sort algorithm. This sorting can be done in parallel, with each processor or thread handling a different sublist.
3. **Merge Sublists**: Once all sublists are sorted, they are merged back together to form the final sorted list. This merging process is usually done sequentially but can also be parallelized for further optimization.
4. **Repeat if Necessary**: If the list is not fully sorted after the first pass, the process is repeated until the list is sorted.

Parallel bubble sort can be more efficient than traditional bubble sort for large lists, especially on systems with multiple processors or cores. However, it requires careful management of parallelism to avoid issues such as race conditions and inefficient use of resources.

📌 What is OpenMP

OpenMP is a programming interface that helps you write programs that can run faster on computers with multiple processors or cores. It does this by allowing you to divide your program into parts that can be done at the same time by different processors. This can make your program run faster and more efficiently.

📌 Explain Parallel BFS to a 5-year-old kid

Alright, imagine you have a bunch of toys, and you want to arrange them in order from smallest to biggest.

For bubble sort, you start by comparing two toys next to each other. If the bigger toy is on the left, you swap them. Then, you move one position to the right and do the same thing with the next two toys. You keep doing this until all the toys are in order.

Now, imagine you have a few friends helping you. Each friend can compare and swap toys at the same time. This makes the sorting faster because you're all working together.

That's like parallel bubble sort! Instead of sorting one toy at a time, you and your friends can sort many toys together, making the job quicker and more fun!

Viva Questions ❓ ❓

1. **What is the Bubble Sort algorithm?**
    - Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.

2. **How does Bubble Sort work?**
    - Bubble Sort works by comparing adjacent elements and swapping them if they are in the wrong order. This process is repeated until the entire list is sorted.
3. **What is the time complexity of Bubble Sort?**
    - The time complexity of Bubble Sort is O(n^2) in the worst and average cases, where n is the number of elements in the list.
4. **Why is Bubble Sort not efficient for large lists?**
    - Bubble Sort's time complexity of O(n^2) means that it becomes very slow as the number of elements in the list increases. It has to compare and swap elements multiple times, making it inefficient for large lists.
5. **How can Bubble Sort be improved for large lists?**
    - One way to improve Bubble Sort for large lists is to use parallel processing, where different parts of the list are sorted simultaneously by multiple processors or threads. This can reduce the time it takes to sort the list.
6. **What is parallel processing?**
    - Parallel processing is a method of computation in which many calculations or processes are carried out simultaneously. It can be used to speed up tasks that can be divided into smaller parts that can be processed independently.
7. **How does parallel Bubble Sort work?**
    - Parallel Bubble Sort divides the list into smaller sublists and sorts them independently using multiple processors or threads. Once the sublists are sorted, they are merged back together to form the final sorted list.
8. **What are the advantages of parallel Bubble Sort?**
    - Parallel Bubble Sort can be faster than traditional Bubble Sort for large lists because it can take advantage of multiple processors or cores to sort the list more quickly.
9. **What are the challenges of parallel programming?**
    - Parallel programming can be challenging because it requires careful coordination and synchronization of multiple processes or threads to

ensure that they do not interfere with each other and produce incorrect results.

10. **Can you explain the concept of race conditions in parallel programming?**
    - Race conditions occur in parallel programming when the outcome of the program depends on the timing or interleaving of operations between different threads or processes. This can lead to unpredictable behavior and incorrect results.