Assignment 1(b)

Title: Write a program to implement Parallel Bubble Sort using OpenMP

```
#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;
}
```

Here, we're using the #pragma omp parallel directive to parallelize the code using OpenMP. Within the parallel block, we have a while loop that runs until the array is sorted. Inside the while loop, we have a #pragma omp for directive that parallelizes the for loop that performs the bubble sort.

Note that the flag variable is used to keep track of whether any swaps were made during an iteration of the for loop. If no swaps were made, then the array is already sorted and we can exit the loop.

explanation of each line of the code:

These lines are the standard C++ header files used for input/output operations and OpenMP, respectively.

```c++
void parallel_bubble_sort(int arr[], int n) {
    #pragma omp parallel
    {
        int i, temp, flag = 1;
        while (flag) {
            flag = 0;
            #pragma omp for
            for (i = 0; i < n-1; i++) {
                if (arr[i] > arr[i+1]) {
                    temp = arr[i];
                    arr[i] = arr[i+1];
                    arr[i+1] = temp;
                    flag = 1;
                }
            }
        }
    }
}
```

This is the function parallel_bubble_sort() that implements the parallel bubble sort algorithm using OpenMP.

- #pragma omp parallel is a directive that creates a team of threads to execute the parallel code inside the block. In this case, the block contains the code for bubble sort algorithm.
- int i, temp, flag = 1; declares the variables i, temp, and flag that will be used inside the while loop.
- while (flag) is a loop that runs until the flag variable is 0.
- flag = 0; sets the flag variable to 0 before starting each iteration of the for loop.
- #pragma omp for is a directive that parallelizes the for loop, by dividing the loop iterations among the threads in the team. Each thread performs the sorting operation on a subset of the array, thereby making the sorting process faster.
- for (i = 0; i < n-1; i++) is a for loop that iterates over the array, from 0 to n-1.
- if (arr[i] > arr[i+1]) checks if the current element is greater than the next element.
- temp = arr[i]; arr[i] = arr[i+1]; arr[i+1] = temp; swaps the current element with the next element, using a temporary variable.
- flag = 1; sets the flag variable to 1, indicating that a swap has been made.
- Finally, the sorted array is printed using a for loop.

```c++
int main() {
    int arr[] = {5, 3, 1, 9, 8, 2, 4, 7, 6};
    int n = sizeof(arr)/sizeof(arr[0]);

    parallel_bubble_sort(arr, n);

    cout << "Sorted array: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```

This is the main() function, which initializes an array arr and its size n. The function parallel_bubble_sort() is called with these arguments to sort the array. The sorted array is then printed to the console using a for loop.

How to Run code in Ubuntu:

1. **Open a terminal window.**

Compile the code using the following command:

```
g++ -fopenmp filename.cpp -o executable
```

Run the program by executing the following command:

```bash
./executable
```

Output

```c
Sorted array: 1 2 3 4 5 6 7 8 9
```

This is because the input array `{5, 3, 1, 9, 8, 2, 4, 7, 6}` is sorted in ascending order using the parallel bubble sort algorithm implemented in the `parallel_bubble_sort()` function. The sorted array is then printed to the console in the `main()` function using a for loop.

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 utilisation 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.