410255:LP-V. HPC	
Experiment No: 2	Parallel Bubble Sort and Merge sort using OpenMP

Aim : 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 :

Parallel Bubble Sort:

A way to implement the *Bubble Sort* in parallel is to divide the domain of the list (more or less) equally between the N-1 nodes 1 to (N-1) of an N nodes parallel machine, keeping node 0 to administer the calculation. Each node 1 to (N-1) can then sort its partial list and send it back to node 0 for a final global merge.

A possible idea is to run multiple iterations in a pipeline fashion, i.e., start the bubbling action of the next iteration before the preceding iteration has finished in such a way that it does not overtakes it. R. Rocha and F. Silva (DCC-FCUP) Parallel Sorting Algorithms Parallel Computing

Odd-even transposition sort is a variant of bubble sort which operates in two alternating phases: Even Phase: even processes exchange values with right neighbors (P0 ↔ P1, P2 ↔P3, ...) Odd Phase: odd processes exchange values with right neighbors (P1 ↔ P2, P3 ↔P4, ...) For sequential programming, odd-even transposition sort has no particular advantage over normal bubble sort. However, its parallel implementation corresponds to a time complexity of O(n).

Parallel Merge Sort:

Mergesort is a classical sorting algorithm using a divide-and-conquer approach. The initial unsorted list is first divided in half, each half sublist is then applied the same division method until individual elements are obtained. Pairs of adjacent elements/sublists are then merged into sorted sublists until the one fully merged and sorted list is obtained.

Program For Parallel Bubble Sort

```
#include<iostream>
#include<omp.h>
using namespace std;

void swap(int &a, int &b)
{
  int temp;
  temp=a;
  a=b;
  b=temp;
}

void bubble(int *a, int n)
{
  double start=omp_get_wtime();
for(int i=0;i<n;i++)
  {
    #pragmaomp parallel
for(int j=i+1;j<n;j++)
    {
      if(a[j]<a[i])
      {
              swap(a[j],a[i]);
      }
    }
  }
  double end=omp_get_wtime();
  double time=end-start;
cout<<"\nTime taken => "<<time<<endl;
```

```
}

int main()
{
omp_set_num_threads(4);
  double start,end;
  int *a,n;
cout<<"\nEnter total number of elements =>";
cin>>n;
  a=new int[n];
cout<<"\nEnter elements =>";
for(int i=0;i<n;i++)
  {
cin>>a[i];
  }
  bubble(a,n);
cout<<"\nSorted Array =>";
for(int i=0;i<n;i++)
  {
cout<<a[i]<<" ";
  }
  return 0;
}
```

Output –

```
Enter total number of elements => 5
Enter elements => 5 4 3 2 1
Time taken => 0.00200009
Sorted Array => 1 2 3 4 5
```

Program For Parallel Merge Sort

```
#include<iostream>
#include<omp.h>
using namespace std;

void merge(int *,int,int,int);

void merge_sort(int *arr, int low, int high)
{
   int mid;
   if(low<high)
   {
      mid=(low+high)/2;
      #pragmaomp parallel sections
      {
         #pragmaomp section
         {
merge_sort(arr,low,mid);
         }
         #pragmaomp section
         {
merge_sort(arr,mid+1,high);
         }
      }
      merge(arr,low,high,mid);
   }
}

void merge(int *arr,intlow,inthigh,int mid)
{
```

```
    int i,j,k,c[50];
i=low;
    k=low;
    j=mid+1;
while(i<=mid && j<=high)
    {
        if(arr[i]<arr[j])
        {
            c[k]=arr[i];
            k++;
i++;
        }
        else
        {
            c[k]=arr[j];
            k++;
j++;
        }
    }
    while(i<=mid)
    {
        c[k]=arr[i];
        k++;
i++;
    }
    while(j<=high)
    {
        c[k]=arr[j];
        k++;
j++;
    }
```

```
    for(i=low;i<k;i++)
    {
arr[i]=c[i];
    }
}

int main()
{
omp_set_num_threads(4);
    int myarray[30],num;
cout<<"\nEnter number of elements to be sorted : ";
cin>>num;
cout<<"\nEnterelements : ";
for(int i=0;i<num;i++)
    {
cin>>myarray[i];
    }
merge_sort(myarray,0,num-1);
cout<<"\nSortedarray :"<<" ";
for(int i=0;i<num;i++)
    {
cout<<myarray[i]<<" ";
    }
}
```

Output –

Enter number of elements to be sorted : 5

Enter elements : 5 4 3 2 1

Sorted array : 1 2 3 4 5