


|   |                                   |                  |                        |  |
|---|-----------------------------------|------------------|------------------------|--|
| <br>Sinhgad Institutes | <b>Name of the Student:</b> _____ |                  | <b>Roll No:</b> ____   |  |
|   | <b>CLASS: - B. E. [COMP]</b>      | <b>Division:</b> | <b>Course: LP-V</b>    |  |
| <b>High Performance Computing</b><br><b>Assignment No. 02</b><br><b>PARALLEL SORT ALGORITHM</b>         |                                   |                  |                        |  |
| <b>Date of Performance:</b>   |                                   |                  | <b>Marks:</b>          |  |
|   |                                   |                  | <b>Sign with Date:</b> |  |

**Title:** Study of parallel sorting algorithms

**Objectives:**

- To study and understand parallel Bubble sort & Merge sort

**Outcomes:**

- Implement parallel Bubble sort & Merge sort

**PEOs, POs, PSOs and COs satisfied**

**PEOs:** I, III      **POs:** 1, 2, 3, 5      **PSOs:** 1      **Cos:** 1

**Problem Statement:**

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:

### What is sorting?

Sorting is a process of arranging elements in a group in a particular order, i.e., ascending order, descending order, alphabetic order, etc.

Characteristics of Sorting are:

- Arrange elements of a list into certain order
- Make data become easier to access
- Speed up other operations such as searching and merging. Many sorting algorithms with different time and space complexities

### What is Parallel Sorting?

A sequential sorting algorithm may not be efficient enough when we have to sort a huge volume of data. Therefore, parallel algorithms are used in sorting.

Design methodology:

- Based on an existing sequential sort algorithm
  - Try to utilize all resources available
  - Possible to turn a poor sequential algorithm into a reasonable parallel algorithm (bubble sort and parallel bubble sort)
- Completely new approach
  - New algorithm from scratch
  - Harder to develop
  - Sometimes yield better solution

### Bubble Sort

The idea of bubble sort is to compare two adjacent elements. If they are not in the right order, switch them. Do this comparing and switching (if necessary) until the end of the array is reached. Repeat this process from the beginning of the array  $n$  times.

- One of the straight-forward sorting methods
  - Cycles through the list
  - Compares consecutive elements and swaps them if necessary
  - Stops when no more out of order pair
- Slow & inefficient
- Worst case performance is  $O(n^2)$

### Bubble Sort Example

Here we want to sort an array containing [8, 5, 1]. The following figure shows how we can sort this array using bubble sort. The elements in consideration are shown in **bold**.

|                 |                          |
|-----------------|--------------------------|
| <b>8</b> , 5, 1 | Switch 8 and 5           |
| 5, <b>8</b> , 1 | Switch 8 and 1           |
| 5, 1, 8         | Reached end start again. |
| <b>5</b> , 1, 8 | Switch 5 and 1           |
| 1, <b>5</b> , 8 | No Switch for 5 and 8    |
| 1, 5, 8         | Reached end start again. |
| <b>1</b> , 5, 8 | No switch for 1, 5       |
| 1, <b>5</b> , 8 | No switch for 5, 8       |
| 1, 5, 8         | Reached end.             |

But do not start again since this is the nth iteration of same process

### Parallel Bubble Sort

- Implemented as a pipeline.
- Let  $\text{local\_size} = n / \text{no\_proc}$ . We divide the array in no\_proc parts, and each process executes the bubble sort on its part, including comparing the last element with the first one belonging to the next thread.
- Implement with the loop (instead of  $j < i$ ) for ( $j=0; j < n-1; j++$ )
- For every iteration of  $i$ , each thread needs to wait until the previous thread has finished that iteration before starting.
- We'll coordinate using a barrier.

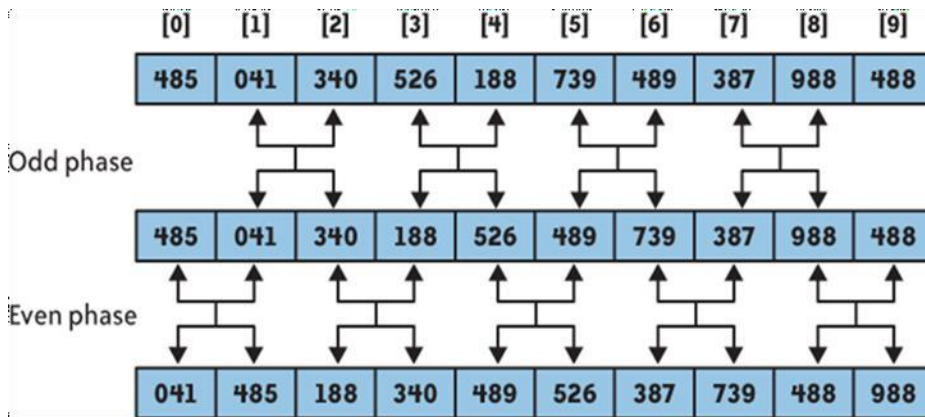
### Algorithm for Parallel Bubble Sort

1. For  $k = 0$  to  $n-2$
2.     If  $k$  is even then
3.         for  $i = 0$  to  $(n/2)-1$  do in parallel
4.             If  $A[2i] > A[2i+1]$  then
5.                 Exchange  $A[2i] \leftrightarrow A[2i+1]$
6.     Else
7.         for  $i = 0$  to  $(n/2)-2$  do in parallel
8.             If  $A[2i+1] > A[2i+2]$  then
9.                 Exchange  $A[2i+1] \leftrightarrow A[2i+2]$
10.     Next  $k$

### Parallel Bubble Sort Example 1

- Compare all pairs in the list in parallel
- Alternate between odd and even phases
- Shared flag, **sorted**, initialized to true at beginning of each iteration (2 phases), if any

processor perform swap, **sorted** = false



## Merge Sort

- Collects sorted list onto one processor
- Merges elements as they come together
- Simple tree structure
- Parallelism is limited when near the root

To sort  $A[p \dots r]$ :

### 1. Divide Step

If a given array  $A$  has zero or one element, simply return; it is already sorted. Otherwise, split  $A[p \dots r]$  into two subarrays  $A[p \dots q]$  and  $A[q + 1 \dots r]$ , each containing about half of the elements of  $A[p \dots r]$ . That is,  $q$  is the halfway point of  $A[p \dots r]$ .

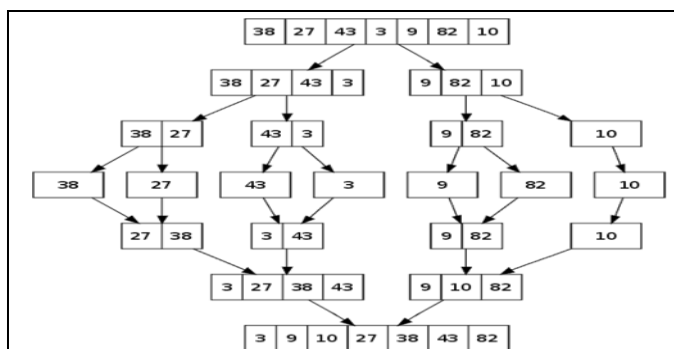
### 2. Conquer Step

Conquer by recursively sorting the two subarrays  $A[p \dots q]$  and  $A[q + 1 \dots r]$ .

### 3. Combine Step

Combine the elements back in  $A[p \dots r]$  by merging the two sorted subarrays  $A[p \dots q]$  and  $A[q + 1 \dots r]$  into a sorted sequence. To accomplish this step, we will define a procedure MERGE ( $A, p, q, r$ ).

**Example:**



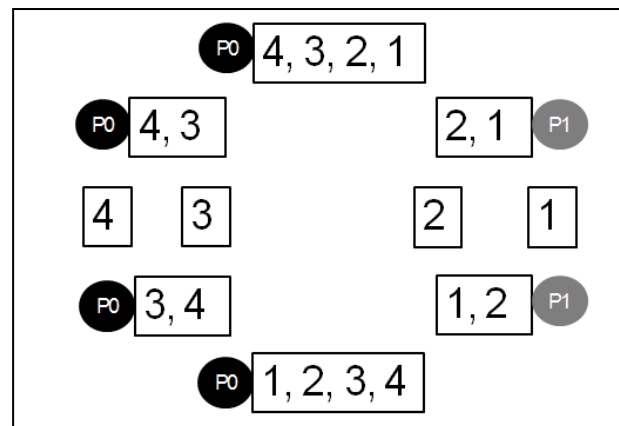
## Parallel Merge Sort

- Parallelize processing of sub-problems
- Max parallelization achieved with one processor per node (at each layer/height)

## Parallel Merge Sort Example

Perform Merge Sort on the following list of elements. Given 2 processors, P0 & P1, which processor is responsible for which comparison?

4,3,2,1



## Algorithm for Parallel Merge Sort

1. Procedure parallelMergeSort
2. Begin
3. Create processors  $P_i$  where  $i = 1$  to  $n$
4. if  $i > 0$  then receive size and parent from the root
5. receive the list, size and parent from the root
6. endif
7.  $midvalue = listsize/2$
8. if both children are present in the tree then
9. send  $midvalue$ , first child
10. send  $listsize - mid$ , second child
11. send list,  $midvalue$ , first child
12. send list from  $midvalue$ ,  $listsize - midvalue$ , second child
13. call `mergelist(list, 0, midvalue, list, midvalue+1, listsize, temp, 0, listsize)`
14. store temp in another array list2
15. else
16. call `parallelMergeSort(list, 0, listsize)`
17. endif
18. if  $i > 0$  then
19. send list, listsize, parent
20. endif

21.     end

**INPUT:**

1. Array of integer numbers.

**OUTPUT:**

1.     Sorted array of numbers

**Conclusion:**

Thus we implemented parallel Bubble sort and Merge sort.

**Write short answer of following questions :**

1.     Explain **#pragma omp parallel sections** construct.
2.     Explain Parallel Bubble Sort with an example?
3.     Explain Parallel Merge Sort with an example?

CODE: BubbleSort

```
#include <iostream>
#include <vector>
#include <omp.h>
using namespace std;
void bubble_sort_odd_even(vector<int>& arr) {
    bool isSorted = false;
    while (!isSorted) {
        isSorted = true;
        #pragma omp parallel for
        for (int i = 0; i < arr.size()-1; i += 2) {
            if (arr[i] > arr[i + 1]) {
                swap(arr[i], arr[i + 1]);
                isSorted = false;
            }
        }
        #pragma omp parallel for
        for (int i = 1; i < arr.size()-1; i += 2) {
            if (arr[i] > arr[i + 1]) {
                swap(arr[i], arr[i + 1]);
                isSorted = false;
            }
        }
    }
}

int main() {
    vector<int> arr = {5, 2, 9, 1, 7, 6, 8, 3, 4};
    double start, end;
    // Measure performance of parallel bubble sort using odd even transposition
    bubble_sort_odd_even(arr);
    cout<<"sorted array=";
    for(int i=0;i<arr.size();i++)
    {
        cout<<arr[i];
    }
    start = omp_get_wtime();
    end = omp_get_wtime();
    cout <<endl<< "Parallel bubble sort using odd even transposition time: " << end-start <<
    endl
}
```

- 1.This program uses OpenMP to parallelize the bubble sort algorithm.
- 2.The **#pragma omp parallel for** directive tells the compiler to create a team of threads to execute the for loop within the block in parallel.
- 3.Each thread will work on a different iteration of the loop,in this case on comparing and swapping the elements of the array.
- 4.The **bubble Sort** function takes in an array, and it sorts it using the bubble sort algorithm. The outer loop iterates from 0 to n-2 and the inner loop iterates from 0 to n-i-1,where I is the index of the outer loop. The inner loop compares the current element with the next element, and if the current element is greater than the next element, they are swapped.
- 5.The **main** function creates a sample array and calls the **bubbleSort** function to sort it. The sorted array is then printed.
- 6.This is a skeleton code and it may no trunasis and may need some modification to work with specific inputs and requirements.
- 7.It is Worth noting that bubble sort is no tan efficient sorting algorithm, specially for large inputs, and it may not scale well with more number of threads. Also parallelizing bubble sort does not have a significant improvement in performance due to the nature of the algorithm itself.
- 8.In this implementation, the **bubble\_sort\_odd\_even** function takes in an array and sorts it using the odd-even transposition algorithm. The outer while loop continues until the array is sorted. Inside the loop, the **#pragmaompparallelfor** directive creates a parallel region and divides the loop iterations among the available threads. Each thread performs the swap operation in parallel,improving the performance of the algorithm.
- 9.The two **#pragmaompparallelfor** inside whileloop,one for even indexes and one for odd indexes,allows each thread to sort the even and odd indexed elements simultaneously and prevent the dependency.

Command to compile and run:

```
C:\MinGw\bin>g++ -o bubble -fopenmp Bubblesort.cpp
```

```
C:\MinGw\bin>bubble
```

CODE:-Merge sort

```
#include <iostream>
#include <vector>
#include <omp.h>
using namespace std;
void merge(vector<int>& arr , int l, int m, int r)
{
    int i , j, k;
    int n1 = m-1 + 1;
    int n2 = r-m;
    vector<int> L(n1), R(n2);
```



```
    for(i = 0; i < n1; i++)
    {
        L[i] = arr [l + i];
    }

    for (j = 0; j < n2; j++)
    {
        R[j] =arr [m + 1 + j];
    }

    i=0;
    j = 0;
    k = l;
    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
            arr[k++] = i;
        }
        else
        {
            arr[k++] = j++;
        }
    }
}
```

```
void merge_sort(vector<int>& arr , int l, int r)
{
    if (l < r)
    {
        int m = l + (r-l) / 2;
        #pragma omp task
        merge_sort(arr , l,m);
        #pragma omp task
        merge_sort(arr , m + 1,r);
        merge(arr , l, m,r);
    }
}

void parallel_merge_sort(vector<int>& arr )
{
    #pragma omp parallel
    {
        #pragma omp single
        merge_sort(arr , 0, arr.size()-1);
    }
}
```

```
}  
int main() {  
    vector<int> arr = {5, 2, 9, 1, 7, 6, 8, 3};  
    double start, end;  
    //Measure performance of sequential merge sort  
    start = omp_get_wtime();  
    merge_sort(arr, 0, arr.size()-1);  
    //parallel_merge_sort(arr);  
    end = omp_get_wtime();  
    cout<< "Sequential merge sort time: " << end-start <<endl;  
    //Measure performance of parallel merge sort  
    arr={5, 2, 9, 1, 7, 6, 8, 3};  
    start =omp_get_wtime();  
    parallel_merge_sort(arr);  
    end =omp_get_wtime();  
}
```