1 ASSIGNMENT NO: A2

Author: Ameeth Kanawaday
Roll No: 4430

2 Problem Definition

Using Divide and Conquer Strategies design a class for Concurrent Quick Sort using C++.

3 Learning Objectives:

1. To understand Divide and Conquer strategy.
2. To use Divide and Conquer to implement concurrent quick sort.

4 S/W and H/W requirements:

1. Open source 64 bit OS.
2. Gedit text editor.
3. C++ programming language.
4. g++ compiler.

5 Theory

Divide and Conquer:

In computer science, divide and conquer is an algorithm design paradigm based on multi-branched recursion. A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

This divide and conquer technique is the basis of efficient algorithms for all kinds of problems, such as sorting (e.g., quicksort, merge sort), multiplying large numbers (e.g. the Karatsuba algorithm), finding the closest pair of points, syntactic analysis (e.g., top-down parsers), and computing the discrete Fourier transform (FFTs).

Understanding and designing Divide and Conquer algorithms is a complex skill that requires a good understanding of the nature of the underlying problem to be solved. As when proving a theorem by induction, it is often necessary to replace the original problem with a more general or complicated problem in order to initialize the recursion, and there is no systematic method for finding

the proper generalization. These Divide and Conquer complications are seen when optimizing the calculation of a Fibonacci number with efficient double recursion.
The correctness of a divide and conquer algorithm is usually proved by mathematical induction, and its computational cost is often determined by solving recurrence relations.

Concurrent Quick Sort Method:

1. Find the pivot element and place it in its proper position in the given array.
2. Split the array into two parts at the correct position of pivot element.
3. Perform step 1 and 2 on the subarrays concurrently. If smallest array is reached then return sorted array.
4. Combine the sorted subarrays from lower levels to finally form the complete sorted array.

6 Related Mathematics

Let S be the solution perspective of the given problem.
The set S is defined as:
$S = \{\ s, e, X, Y, F, DD, NDD | \varnothing_s\}$
Where,
s= Start point
e= End point
F= Set of main functions
DD= set of deterministic data
NDD= set of non deterministic data

X= Input Set.
$X = \{\ < x_1, x_2, ..., x_n >\}$
where,

Y $= < x_1, x_2, ..., x_n >$, such that,
$x_1 <= x_2 <= ...x_{n-1} <= x_n$

s = unsorted array.
e = sorted array

F = $\{f_{split}, f_{swap}, f_{merge}, f_{concurr}\}$

f_{split} :function to split the list from position of pivot element.

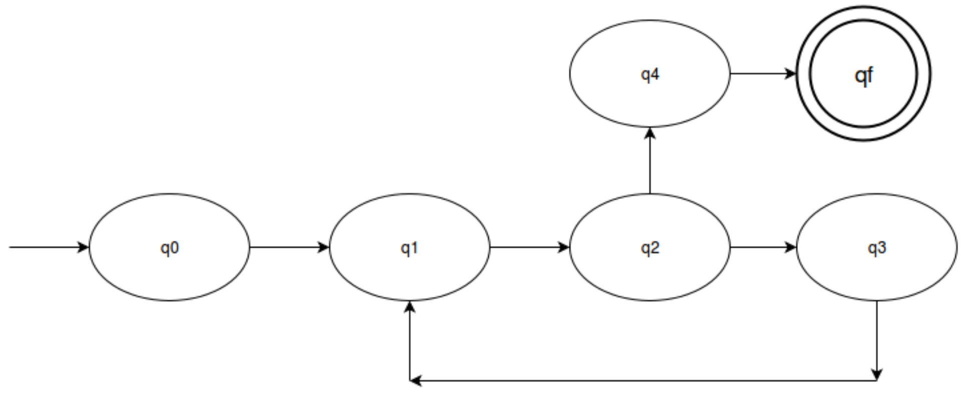
f_{swap} :function to exchange positions of two array elements.

f_{merge} :function to merge sorted subarrays.

$f_{concurr}$: function to assign thread to sorting of subarray.

DD = $\{< x_1 ... x_n >\}$
NDD = $\{threadID\}$

7 State Diagram



q0 = input state
q1 = find pivot element state
q2 = place pivot element state
q3 = split state
q4 = display sorted list state
qf = final state

8 Program

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

int k=0;

class array
{
public:
int partition(int arr[], int low_index, int high_index)
{
int i, j, temp, key;
key = arr[low_index];
i= low_index + 1;
j= high_index;
while(1)
{
while(i < high_index && key >= arr[i])
    i++;
while(key < arr[j])
```

```
    j--;
if(i < j)
{
temp = arr[i];
arr[i] = arr[j];
arr[j] = temp;
}
else
{
temp= arr[low_index];
arr[low_index] = arr[j];
arr[j]= temp;
return(j);
}
}
}

void quicksort(int arr[], int low_index, int high_index)
{
int j;
if(low_index < high_index)
{
j = partition(arr, low_index, high_index);
cout<<"Pivot element with index "<<j<<" has been found out by thread "<<k<<"\n";

#pragma omp parallel sections
{
    #pragma omp section
    {
k=k+1;
quicksort(arr, low_index, j - 1);
    }

    #pragma omp section
    {
k=k+1;
quicksort(arr, j + 1, high_index);
    }

}
}
}

};

int main()
{
array a;

int arr[100];
int n,i;

cout<<"Enter the value of n\n";
cin>>n;
cout<<"Enter the "<<n<<" number of elements \n";

for(i=0;i<n;i++)
{
cin>>arr[i];
}

a.quicksort(arr, 0, n - 1);

cout<<"Elements of array after sorting \n";
for(i=0;i<n;i++)
{
cout<<arr[i]<<"\t";
}

cout<<"\n";
}
```

```
OUTPUT:
ameeth@ubuntu-16.0.4:~/CL1$ g++ a2.cpp
ameeth@ubuntu-16.0.4:~/CL1$ ./a.out
Enter the value of n
10
Enter the 10 number of elements
1
6
32
5
7
54
8
9
3
2
Pivot element with index 0 has been found out by thread 0
Pivot element with index 4 has been found out by thread 2
Pivot element with index 2 has been found out by thread 3
Pivot element with index 9 has been found out by thread 6
Pivot element with index 8 has been found out by thread 7
Pivot element with index 5 has been found out by thread 8
Pivot element with index 6 has been found out by thread 10
Elements of array after sorting
1 2 3 5 6 7 8 9 32 54
ameeth@ubuntu-16.0.4:~/CL1$
```

9 Conclusion

Thus we have implemented concurrent quick sort using divide and conquer strategy.