**Assignment No.**

**Title: Concurrent Quick Sort**

**Aim :**Implementation and Modelling of Concurrent Quick Sort

**Objective:**

Modelling and Designing of Concurrent Quick Sort.

**Theory :**

**Quick sort**

Quick sort is an efficient sorting algorithm, serving as a systematic method for placing the elements of array in order. It is a comparison sort, meaning that it can sort any types of items for which 'less than' relation is defined. It is not a stable sort, meaning that the relative order of equal sort items is not preserved. It can operate on an array, required small additional amount of memory to sort.

Quicksort is a divide and conquer algorithm. Quicksort first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quicksort can then recursively sort the sub-arrays.

The steps are:

* Pick an element, called a *pivot*, from the array.
* *Partitioning*: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the *partition* operation.
* Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

The base case of the recursion is arrays of size zero or one, which never need to be sorted.

The pivot selection and partitioning steps can be done in several different ways; the choice of specific implementation schemes greatly affects the algorithm's performance.

Worst case O(n2)

Best case O(nlogn)

Average case O(nlogn)

**Open MP**

Open multi processing (MP) is an API that supports multi platform shared memory multiprocessing programming in c/c++.It consists of a set of compiler directives, library routines and environment variables that influence runtime behaviour.It uses a portable, scalable model that gives programs a flexible and simple interface for developing parallel applications for platforms ranging from standard desktop computers to supercomputers.

OpenMP is an implementation of multithreading, a method of parallelizing whereby a master *thread* (a series of instructions executed consecutively) *forks* a specified number of slave *threads* and the system divides a task among them. The threads then run concurrently, with the runtime environment allocating threads to different processors.

The section of code that is meant to run in parallel is marked accordingly, with a pre-processor directive that will cause the threads to form before the section is executed. Each thread has an *id* attached to it which can be obtained using a function (called omp\_get\_thread\_num()). The thread id is an integer, and the master thread has an id of *0*. After the execution of the parallelized code, the threads *join* back into the master thread, which continues onward to the end of the program.

By default, each thread executes the parallelized section of code independently. *Work-sharing constructs* can be used to divide a task among the threads so that each thread executes its allocated part of the code. Both task parallelism and data parallelism can be achieved using OpenMP in this way.

The runtime environment allocates threads to processors depending on usage, machine load and other factors. The runtime environment can assign the number of threads based on environment variables, or the code can do so using functions. The OpenMP functions are included in a headerfile labelled omp.h in C/C++.

**Mathematical Model:**

Let S be the system such that :

S={s,e,X,Y,F,T,Sc,Fc,CPUcorecnt}

Where,

s= initial state

e= end state

X= set of inputs

Y= set of outputs

T=threads

F= set of function

Sc= Success cases

Fc= Failure cases

CPUcorecnt= No. of CPU cores

Let S’ be system in observation

Where S’C S

S’ = {s,e,X,Y,F,T,Sc,Fc,DD,NDD}

* S= start state

{init\_arr }

* e= end state

exit(0) ….success

* X= {(Xi,n ) | XiЄ i, 0 ≤ i < n }
* Y= {Y1, Y2, Y3} Є Y

Where ,

Y1 Є success

{Y2, Y3} Є failure

* T=Threading.thread()

thread=start()

thread=join()

* F= {Fme, Ffriend}

Fme = {Fk1}

Fk1 = quicksort (x,st,end)

= “X”

Where “X” = {Xi , n/2}

Such that xi< xi+1 ,0 ≤ x < n & 0 ≤st,end

Ffriend = {FAs1,FAs2}

FAs1 = partition (x,st,end)

= pivot\_index

Such that (Xj) <x ,pivot\_index< {xk…} , 0<pivot\_index< end ’n’

* CPUcorecnt = lscpu = {1,2,…,8}
* Sc= {Y1}

Y1Є Xi where,

XiЄ {xi , n | xiЄ x}

And xi<x , for all 0 <xi<n

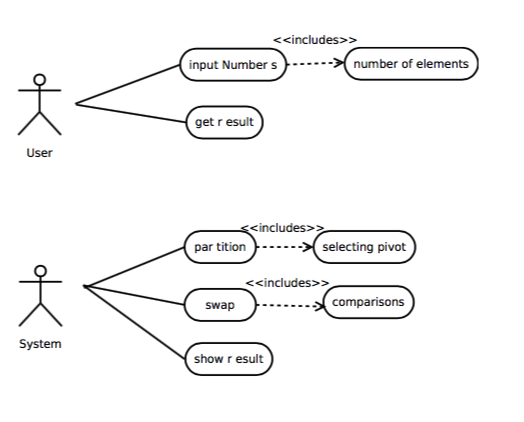
* Fc = {Y2,Y3}

Y2 = {Thread\_allocation} = error

Y3 = {x ≠sorted(x)} = error

**Design:**

**Use Case Diagram:**

****

**Fig: Use Case Diagram**

**Algorithm :**

Algorithm QuickSort(A,Low,High)

//A is the array to be sorted

//low = first index of array

//Hogh = last index of array

If low < high then

q = partition(A,low,high)

QuickSort(A,Low,q-1)

QuickSort(A,q+1,High)

end

Algorithm Partition(A,Low,High)

x = A[high]

i = low - 1

for j=low to high-1

if A[j]<=x then

i=i+1

swap(A[i],A[j])

end

end

swap(A[i+1],A[high])

return(i+1)

**Input :** No. of elements, Integers to be sorted

**Output :** Sorted array of numbers,Time taken to sort the array

**Platform :** Ubuntu 14.04

**Conclusion:**

The concurrent quick sort is implemented and modelled.