###### Experiment Number: A2

**TITLE:**

**PROBLEM STATEMENT: Using Divide and Conquer Strategies to design an efficient class for Concurrent Quick Sort and the input data is stored using XML. Use object oriented software design method and Modelio/ StarUML2.x Tool.**

**Perform the efficiency comparison with any two software design methods. Use necessary USE-CASE diagrams and justify its use with the help of mathematical modeling. Implement the design using Scala/Python/Java/C++.**

**OBJECTIVES:**

1. To develop problem solving abilities using Mathematical Modeling
2. To apply algorithmic strategies while solving problems
3. To develop time and space efficient algorithms
4. To implement software design and testing in distributed, concurrent and parallel environments

**THEORY:**

***Divide and Conquer***

Divide-and-conquer is a top-down technique for designing algorithms that consists of dividing the problem into smaller **subproblems** hoping that the solutions of the **subproblems** are easier to find and then composing the partial solutions into the solution of the original problem.

***Quick Sort***

Quicksort is a divide-and-conquer sorting algorithm in which division is dynamically carried out (as opposed to static division in Mergesort). The Quicksort algorithm is based on choosing some element from the list to be sorted, and partitioning the remaining elements into those less than and greater than the chosen element, then recursively applying the algorithm to these partitions until we get down to single elements.

This can be implemented in a concurrent form, by representing the partitions as subtasks of the sorting task.

**MATHEMATICAL MODEL:**

Let P be the solution perspective.

P ={ S , E , I , O, F }

S = { start }

I = Input of the system → { I1 , I2 }

where,

I1 = { Path of xml file entered by user}

I2 = { Input XML file }

O = Output of the system → { O1,O2 }

where,

O1 = { Sorted Array}

O2 = {File created at user specified site containing sorted array}

F = Functions used → { f1 , f2 , f3 }

where

f1 = {Reading Input XML File }

f2 = {Concurrent Quick Sort}

f3 = {Storing output in file}

E = End state of the system shows successful implementation of concurrent quicksort on an input xml file.

**IMPLEMENTATION DETAILS / DESIGN LOGIC:**

**Algorithm:-**

1. Start

2. Take path of input xml file from user.

3. Read the xml file from the specified path

4. Perform partitioning of elements

5. Call concurrent quick sort function

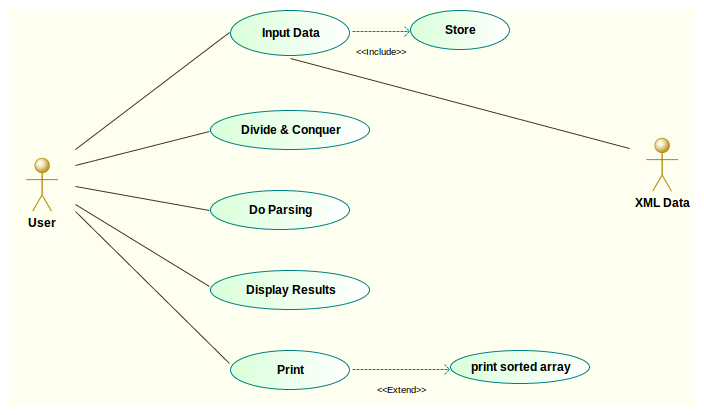
6. Print the sorted array on the terminal.

7. Take path of output file from user

8. Store the sorted array in a text file created at the specified path.

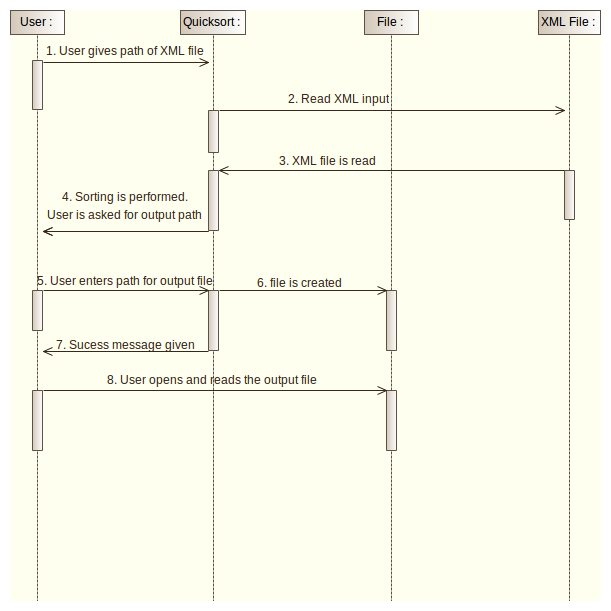
**UML Diagrams:-**

**1.USE-CASE Diagram**

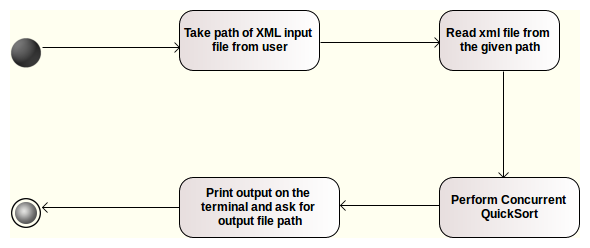


Reverse the direction of extend

**2. Sequence diagram**



**3. State Machine Diagram use “ing” or “ed” to verb that shows state instead of activity**



**INPUT**

An XML file.

The path of this file needs to be specified by the user.

**OUTPUT**

A sorted array using concurrent quick sort.

**EXECUTION STEPS**

Execute the program with the following commands.

rajani@rajani-Inspiron-5521:~/Desktop$ c++ input1.cpp

rajani@rajani-Inspiron-5521:~/Desktop$ ./a.out

When prompted give path of input xml file

Ex: /home/oct/num.xml

The program will then sort the array stored in xml file

Give path of output file( give .txt extension)

Go to that path and open the file. It will contain the sorted array.

**TEST CASES:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test ID** | **Description** | **Input** | **Expected output** | **Actual output** |
| 1. | Getting XML file as input | Path of xml file from user | Should be able to parse and read the xml file | Successful parsing of file |
| 2. | Perform concurrent quick sort | Array stored in xml file | Should sort the array | Array gets sorted |
| 3. | Print output on terminal | Sorted array to display function | Print sorted array on terminal | Sorted Array printed on terminal |
| 4. | Create output text file containing the sorted array | Path of the file to be created from the user | Should create the file at user specified location and store the sorted array into this file | Successful creation of file at user specified location and sorted array is stored into the file which can be read when the file is opened. |

**CONCLUSION**

Thus we have successfully implemented concurrent quick sort using an xml file as input.

**COURSE OUTCOMES ACHIEVED**

To solve problems using mathematical modeling.

To use software design methods and testing.

To solve problems for multi-core or distributed, concurrent/Parallel environments

**FAQ’s**

1. What algorithmic strategy does concurrent Quicksort follow? How should we select a pivot element?

2. Differentiate between concurrency and parallelism.

3. What is OpenMP? How is fork and join implemented in OpenMP?

4. Enlist the ways in which XML parsing can be done?

5. Give the time and space complexity of concurrent Quicksort.

**CODE**

#include<iostream>

#include<omp.h>

#include<stdlib.h>

#include<string.h>

#include<fstream>

#include"rapidxml\_utils.hpp"

#include <vector>

using namespace std ;

using namespace rapidxml ;

class Quicksort

{

public :

string input\_xml;

string line;

char inputstring [50], outputstring[50];

int input[20],output[20],i1;

void display()

{

for(int i1=0;i1<20;i1++)

{

cout<<"\n"<<input[i1];

}

}

void take\_input()

{

cout<<"Enter the input file pathname: ";

cin>> inputstring;

ifstream in(inputstring);

// read file into input\_xml

while(getline(in,line))

input\_xml += line;

int i=0;

// make a safe-to-modify copy of input\_xml

vector<char> xml\_copy(input\_xml.begin(), input\_xml.end());

xml\_copy.push\_back('\0'); // '\0' appened at the end of vector

// only use xml\_copy from here on!

xml\_document<> doc; // character type defaults to char

/\* we are choosing to parse the XML declaration

parse\_no\_data\_nodes prevents RapidXML from using the somewhat surprising

behavior of having both values and data nodes, and having data nodes take

precedence over values when printing\*/

doc.parse<parse\_declaration\_node | parse\_no\_data\_nodes>(&xml\_copy[0]);

xml\_node<>\* cur\_node = doc.first\_node("array"); // taking the first node as a root node (array).

cur\_node = cur\_node->first\_node("number"); //

string content = "";

for(i=0; i<20; i++)

{

content = cur\_node->value(); // storing the number in content

input[i]=atoi(content.c\_str()); // converting char to int ; content to input[i]

cur\_node = cur\_node->next\_sibling("number"); // going to next node

}

}

void store()

{

cout<<"\nEnter output file pathname: ";

cin>>outputstring;

ofstream ofs (outputstring , ofstream::out);

for (int i=0;i<20;i++)

{

ofs<<input[i]<<endl;

}

ofs.close();

cout<<"\n Sortred array stored in: "<<outputstring<<endl;

}

void quicksort(int left,int right)

{

int index;

if(left<right)

{

index = partition(left,right);

#pragma omp parallel sections

{

#pragma omp section

quicksort(left,index-1);

#pragma omp section

quicksort(index+1,right);

}

}

}

int partition(int low,int high)

{

int pivot ,temp,i,j;

pivot = input[low];

i=low;

j=high+1;

while(1)

{

do ++i;

while(pivot>=input[i] && i<=high);

do --j;

while(pivot<input[j]);

if(i>=j)

break;

temp = input[i];

input[i] = input[j];

input[j] = temp;

}

temp = input[low];

input[low] = input[j];

input[j] = temp;

return j;

}

};

int main()

{

Quicksort q;

q.take\_input();

cout<<"Input array from xml file :\n";

q.display();

cout<<endl<<endl;

q.quicksort(0,20-1);

cout<<endl<<endl;

cout<<"Sorted array :\n";

q.display();

q.store();

cout<<endl<<endl;

return 0;

}

**OUTPUT**

Enter the input file pathname: num.xml

Input array from xml file :

16

17

29

37

61

12

14

26

38

65

13

17

24

38

64

32

66

11

6666

27

Sorted array :

11

12

13

14

16

17

17

24

26

27

29

32

37

38

38

61

64

65

66

6666

Enter output file pathname: num2.txt

Sorted array stored in: num2.txt

rajani@rajani-Inspiron-5521:~/Desktop$