****

**De La Salle University • College of Computer Studies**

**Concurrent Sorting Networks Application**

(Design and Evaluation of Its Performance)

Name (last name first) : Fernandez, Ryan Austin

Poblete, Clarisse Felicia M.

San Pedro, Marc Dominic

Tan, Johansson E.

Section : G01

Date of Submission : February 9, 2016

**I. Source Code**

**Driver.java**

import java.io.\*;

public class Driver2 {

public static int[] dummy;

public static int[] nums;

private static int numCount;

private static int threads;

private static int depth;

private static int compsPerThread;

private static int size;

private static int start;

public static void main(String[] args) throws Exception {

//number of numbers

numCount = (int)Math.pow(2,Integer.parseInt(args[0]));

//number of threads

threads = (int)Math.pow(2,Integer.parseInt(args[1]));

//max layers in a merger

depth = (int)(Math.log(numCount) / Math.log(2));

// comparators per thread

compsPerThread = numCount / 2 / threads;

//read nums

BufferedReader br2 = new BufferedReader(new FileReader("nums.sort"));

dummy = new int[numCount];

nums = new int[numCount];

for(int i=0; i<numCount; i++){

dummy[i] = Integer.parseInt(br2.readLine());

}

br2.close();

float ave = 0;

CompareThread[][] threadList;

long time;

int currDepth;

float maxTime = 0;

// depth = depth \* (depth + 1) / 2;

for(int i = 0; i < 6; i++){

float currTime = 0;

size = 2; //number of elements being sorted at a time

start = 0; //start of current block

for(int j = 0; j < numCount; j++) {

nums[j] = dummy[j];

}

int divisions[] = getDivisions(Integer.parseInt(args[0])

, Integer.parseInt(args[2]));

int actualDivs = divisions.length;

for(int g = 0; g < divisions.length; g++){

if(divisions[g] != 0){

if(g == 0){

threadList = generateThreadList(1, divisions[0]);

}

else{

threadList = generateThreadList(divisions[g - 1] + 1

, divisions[g]);

}

time = System.currentTimeMillis();

for(int layer = 0; layer < threadList.length; layer++) {

for(int j = 0; j < threads; j++) {

threadList[layer][j].start();

}

for(int j = 0; j < threads; j++) {

threadList[layer][j].join();

}

}

time = System.currentTimeMillis() - time;

currTime += time;

}

else{

actualDivs--;

}

}

//CPU caches on first iteration, making it slower.

//first iteration is not considered

ave += currTime;

if( currTime > maxTime) {

maxTime = currTime;

}

boolean isSorted = true;

for(int j = 0; j < numCount - 1; j++ ){

if(nums[j] > nums[j + 1]){

isSorted = false;

break;

}

}

System.out.println("List is " + (isSorted ? "" : "not ")

+ "sorted at " + (currTime / 1000.0)

+ " seconds with " + actualDivs + " divisions");

}

ave -= maxTime;

System.out.println("Average time: " + (ave / 5000.0) + "s");

}

private static CompareThread[][] generateThreadList(int gStart, int gEnd){

//list of threads

CompareThread[][] threadList

= new CompareThread[(gEnd \* (gEnd + 1) / 2)

- ((gStart - 1) \* gStart / 2)][threads];

long time = System.currentTimeMillis();

//what is being created at the moment

final int MERGE = 0;

final int BITONIC = 1;

int state = MERGE;

//level in network

int level = 0;

//for each block

for(int i = gStart; i <= gEnd; i++,size \*= 2) {

int currSize = size;

state = MERGE;

for(int j = i; j > 0; j--,currSize /= 2) {

start = 0;

int k = 0;

int chunk = currSize / 2; //number of comps before skip

int currComps = 0; //comparators already threads

int currIndex = 0; //current thread in layer

//for each wire

while(k < numCount) {

//no comps yet, create thread

if(currComps == 0) {

threadList[level][currIndex] = new CompareThread();

}

int partner;

switch(state) {

case MERGE:

partner = currSize - 1 - k + 2 \* start;

break;

case BITONIC:

partner = k + currSize / 2;

break;

default:

partner = 0;

}

threadList[level][currIndex].add(k,partner);

currComps++;

if(currComps == compsPerThread) {

currComps = 0;

currIndex++;

}

chunk--;

k++;

if( chunk == 0) {

chunk = currSize / 2;

k += chunk;

start = k;

}

}

state = BITONIC;

level++;

}

}

time = System.currentTimeMillis() - time;

return threadList;

}

public static int[] getDivisions(int blocks,int divisions) {

int[] answers = new int[divisions];

int total = blocks \* (blocks + 1) / 2;

int partition = total / divisions;

int runningPartition = partition;

int runningTotal = 0;

for(int i = 1,j = 0; i <= blocks; i++) {

runningTotal += i;

if(runningTotal >= runningPartition) {

runningPartition += partition;

answers[j] = i;

j++;

}

}

return answers;

}

}

**CompareThread.java**

import java.util.ArrayList;

public class CompareThread extends Thread {

private ArrayList<int[]> comparators;

public CompareThread() {

comparators = new ArrayList<int[]>();

}

public CompareThread(ArrayList<int[]> comps) {

comparators = comps;

}

public void add(int s, int e) {

comparators.add(new int[] { s, e });

}

public void run() {

for(int[] comp: comparators) {

if(Driver2.nums[comp[0]] > Driver2.nums[comp[1]]) {

int temp = Driver2.nums[comp[0]];

Driver2.nums[comp[0]] = Driver2.nums[comp[1]];

Driver2.nums[comp[1]] = temp;

}

}

}

public CompareThread copy() {

return new CompareThread(comparators);

}

public String toString(){

return comparators.get(0)[0] + " - " + comparators.get(0)[1];

}

}

**II. Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
| **lg x of x Threads** | **Merge Sort** | **Sorting Networks v1** | **Sorting Networks v2** |
| 0 | 0.8024 | 1.823 | 3.4732 |
| 1 | 0.7908 | 1.7564 | 7.9234 |
| 2 | 0.8344 | 2.164 | 8.7272 |
| 3 | 0.8068 | 1.999 | 12.507 |
| 4 | 0.7956 | 2.0074 | 26.7848 |
| 5 | 0.8372 | 2.8934 | 14.2652 |
| 6 | 0.7524 | 7.231 | 15.4566 |
| 7 | 0.7734 | 11.6108 | 18.025 |
| 8 | 0.7574 | 28.8356 | 20.5346 |
| 9 | 0.7862 | 63.6564 | 25.4828 |
| 10 | 0.776 | 128.3152 | 31.0526 |
| 11 | 0.9146 | 248.5446 | 54.9676 |
| **Fastest Time** | 0.7524 | 1.7564 | 3.4732 |
| **% improvement** |  | -133.4396598 | -361.6161616 |

**Table 1. – Comparative Results of All Three Algorithms**

**Figure 1. – Comparative Results of All Three Algorithms**

**III. Conclusion**

**Appendix A: Ticketing Solution**

GLOBALS

globalturn = 1;

newturn = 1;

lock[2] = [false,false];

ticket[2] = [-1,-1];

whoseTicket = 0;

p1

lock[0] = true;

whoseTicket = 2;

while(lock[1] && whoseTicket == 2);

ticket[0] = newturn++;

lock[0] = false;

while(ticket[0] != globalturn);

CS

ticket[0] = -1;

globalturn++;

p2

lock[1] = true;

whoseTicket = 1;

while(lock[0] && whoseTicket == 1);

ticket[1] = newturn++;

lock[1] = false;

while(ticket[1] != globalturn);

CS

ticket[1] = -1;

globalturn++;