****

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

**Concurrent Merge Sort 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 2, 2016

**I. Source Code**

**Driver.java**

import java.util.ArrayList;

import java.io.\*;

public class Driver {

private static int[] dummy;

private static int[] nums;

public static volatile int threads;

/\*\* operation constants \*/

public static final int SORT = 0;

public static final int BITONIC = 1;

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

//length of list to sort and no. of threads

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

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

int depth = lg(n);

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

//read nums

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

dummy = new int[n];

nums = new int[n];

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

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

}

br.close();

float ave = 0;

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

//copy original list

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

nums[j] = dummy[j];

}

long time = System.currentTimeMillis();

sort(0,n - 1);

time = System.currentTimeMillis() - time;

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

//first iteration is not considered

if( i > 0 ) {

ave += time;

}

//check if sorted

boolean isSorted = true;

for(int j = 0; isSorted && j < n - 1; j++ ){

isSorted = nums[j] <= nums[j + 1];

}

if( i > 0 ) {

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

+ "sorted at " + (time / 1000.0) + " seconds");

}

}

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

}

public static int lg(int n) {

return (int)(Math.log(n) / Math.log(2));

}

public static void compare(int s, int e) {

if( nums[s] > nums[e] ) {

int temp = nums[s];

nums[s] = nums[e];

nums[e] = temp;

}

}

public static synchronized Thread[] threadResources(int s, int m, int e

, int operation) {

if( threads >= 2) {

Thread[] threadList = new Thread[] {

new Thread(new Operation(operation,s,m)),

new Thread(new Operation(operation,m + 1,e))

};

threads -= 2;

return threadList;

} else {

return null;

}

}

public static synchronized void signal() {

threads++;

}

public static void sort(int s,int e) {

if( e - s == 1 ) {

compare(s,e);

} else {

int mid = s + (e - s) / 2;

int len = e - s + 1;

Thread[] threadList = threadResources(s,mid,e,SORT);

if( threadList == null ) {

sort(s,mid);

sort(mid + 1,e);

} else {

try {

threadList[0].start();

threadList[1].start();

threadList[0].join();

threadList[1].join();

} catch( InterruptedException ie) {

ie.printStackTrace();

}

}

merge(s,e);

}

}

public static void merge(int s, int e) {

int mid = s + (e - s) / 2;

int len = e - s + 1;

int s2 = s;

int e2 = e;

while(s2 < e2) {

compare(s2,e2);

s2++; e2--;

}

Thread[] threadList = threadResources(s,mid,e,BITONIC);

if( threadList == null ) {

bitonicSort(s,mid);

bitonicSort(mid + 1,e);

} else {

try {

threadList[0].start();

threadList[1].start();

threadList[0].join();

threadList[1].join();

} catch( InterruptedException ie) {

ie.printStackTrace();

}

}

}

public static void bitonicSort(int s, int e) {

if( e - s == 1 ) {

compare(s,e);

} else {

int mid = s + (e - s) / 2;

int len = e - s + 1;

for(int s2 = s,i = mid + 1; s2 <= mid; s2++,i++ ) {

compare(s2,i);

}

Thread[] threadList = threadResources(s,mid,e,BITONIC);

if( threadList == null ) {

bitonicSort(s,mid);

bitonicSort(mid + 1,e);

} else {

try {

threadList[0].start();

threadList[1].start();

threadList[0].join();

threadList[1].join();

} catch( InterruptedException ie) {

ie.printStackTrace();

}

}

}

}

private static class Operation implements Runnable {

public int operation;

public int s;

public int e;

public Operation(int operation,int s,int e) {

this.operation = operation;

this.s = s;

this.e = e;

}

public void run() {

switch(operation) {

case SORT:

sort(s,e);

break;

case BITONIC:

bitonicSort(s,e);

break;

default:

}

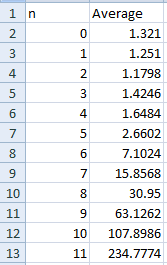
signal();

}

}

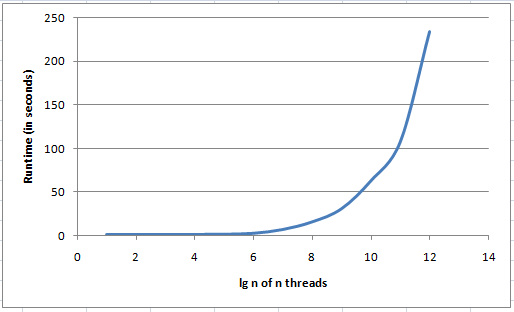
}

**II. Analysis**

****

**Table 1:** Average time of 5 runs per 2^n threads

The sorting network was run on 2^20 randomly generated integers, with the number of threads doubling with each subsequent run. Each run performed the sorting five times, and displayed the average time. Based on this experiment, we found that performance was optimal at 2^2, or 4, threads.

****

However, after that point, runtime seemed to increase exponentially in proportion to the number of threads being generated. At 4 threads, sorting on average took around 1 second. However, at 2048 threads, sorting took on average around 4 minutes.

**III. Conclusion**

As with the previous exercises, having more threads increases the speed of the program up to a certain point, after which having a greater number of threads starts to have the opposite effect.

However, what makes this different from the previous exercises is the rate at which the runtime increases. In the previous exercises, runtime increased linearly. However, during this exercise, runtime seemed to increase exponentially.

Still, In general, having a greater number of threads is useful in decreasing execution time, up to a certain point wherein time is already being wasted allocating and processing threads that have no distinct tasks to do.