5.1

\* Function: Gen\_bins

\* Purpose: Compute max value for each bin, and store 0 as the

\* number of values in each bin

\* In args: min\_meas: the minimum possible measurement

\* max\_meas: the maximum possible measurement

\* bin\_count: the number of bins

\* Out args: bin\_maxes: the maximum possible value for each bin

\* bin\_counts: the number of data values in each bin

\*/

void Gen\_bins(

float min\_meas /\* in \*/,

float max\_meas /\* in \*/,

float bin\_maxes[] /\* out \*/,

int bin\_counts[] /\* out \*/,

int bin\_count /\* in \*/) {

float bin\_width;

int i;

bin\_width = (max\_meas - min\_meas)/bin\_count;

#pragma omp parallel for num\_threads(thread\_count) \

default(none) \

shared(min\_meas, max\_meas, bin\_maxes, bin\_counts, bin\_count, bin\_width) \

private(i)

for (i = 0; i < bin\_count; i++) {

bin\_maxes[i] = min\_meas + (i+1)\*bin\_width;

bin\_counts[i] = 0;

}

# ifdef DEBUG

printf("bin\_maxes = ");

for (i = 0; i < bin\_count; i++)

printf("%4.3f ", bin\_maxes[i]);

printf("\n");

# endif

} /\* Gen\_bins \*/

/\*-

\* Function: Gen\_data

\* Purpose: Generate random floats in the range min\_meas <= x < max\_meas

\* In args: min\_meas: the minimum possible value for the data

\* max\_meas: the maximum possible value for the data

\* data\_count: the number of measurements

\* Out arg: data: the actual measurements

\*/

void Gen\_data(

float min\_meas /\* in \*/,

float max\_meas /\* in \*/,

float data[] /\* out \*/,

int data\_count /\* in \*/) {

int i;

srandom(0);

#pragma omp parallel for num\_threads(thread\_count) \

default(none) shared(data, min\_meas, max\_meas, data\_count)

for (i = 0; i < data\_count; i++) {

data[i] = min\_meas + (max\_meas - min\_meas) \* random() / ((double) RAND\_MAX);

}

# ifdef DEBUG

printf("data = ");

for (i = 0; i < data\_count; i++)

printf("%4.3f ", data[i]);

printf("\n");

# endif

} /\* Gen\_data \*/

/\* Count number of values in each bin \*/

#pragma omp parallel for num\_threads(thread\_count) default(none) \

shared(data\_count, data, bin\_maxes, bin\_count, min\_meas, bin\_counts) \

private(bin, i)

for (i = 0; i < data\_count; i++) {

bin = Which\_bin(data[i], bin\_maxes, bin\_count, min\_meas);

#pragma omp critical

bin\_counts[bin]++;

}

5.2

number in circle = 0;

for ( toss = 0; toss < number of tosses ; toss ++) {

x = random double between − 1 and 1;

y = random double between − 1 and 1;

distance squared = x ∗ x + y ∗ y ;

if ( distance squared <= 1) number in circle ++;

}

pi estimate = 4∗ number in circle /((double) number of tosses );

5.3 d.

int main(int argc, char\* argv[])

{

long int number\_tosses, number\_in\_circle;

int thread\_count, i, j, n, count;

srandom(0);

thread\_count = strtol(argv[1], NULL, 10);

n = strtol(argv[2], NULL, 10);

int \* a = malloc(n\* sizeof(int));

geraMatriz(a, n);

//imprimeMatriz(a, n);

int \* temp = malloc(n\* sizeof(int));

double start = omp\_get\_wtime();

#pragma omp parallel for num\_threads(thread\_count) \

default(none) private(i, j, count) shared(a, n, temp, thread\_count)

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

{

count = 0;

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

if (a[j] < a[i]) count++;

else if (a[j] == a[i] && j < i)

count++;

temp[count] = a[i];

}

memcpy ( a , temp, n \* sizeof(int));

double finish = omp\_get\_wtime();

free(temp );

printf("Tempo estimado %e segundos\n", finish - start);

//imprimeMatriz(a, n);

return 0;

}

/\* main \*/

5.3 e

|  |  |  |  |
| --- | --- | --- | --- |
| p/n | 5000 | 10000 | 20000 |
| 1 | 190,62 | 725,81 | 2577 |
| 2 | 166,63 | 361,2 | 1431 |
| 4 | 116,55 | 293,13 | 1123 |
| Qsort(p=1) | 2,54 | 3,79 | 8,02 |

The quicksort has complexity of nlog (n) in the best case and in the average case, and n² in the worst case, compared to the nost algorithm of the count\_sort method presented.

6.6

Node {

// declaration of Node class

// elements

// declaration of variables

// left and right of Node type

// these are used as reference

// left and right children

Node left, right;

// declaration of variable

// element which holds the

// data at the node

int data;

// parameterized constructor

public Node(int n) {

left = null;

right = null;

data = n;

}

}

public class BTree {

private Node root;

// inserts root element into the tree

public void insert(int data) {

root = insert(root, data);

}

The insert(Node, int) method performs insertion of elements iteratively, If the value of the node is

null then create new node with the data. Else determine the position of the node either to left side

or right side with respect to parent, iteratively run through the tree and insert the element into the

tree .

// inserts elements into the tree iteratively

public Node insert(Node node, int data) {

if (node == null)

node = new Node(data);

else {

//don’t disturb the variable “node” value, instead use

//temporary variables of Node type

Node temp1 = node;

Node temp2 = temp1;

while (temp1 != null) {

temp2 = temp1;

if (data < temp1.data)

temp1 = temp1.left;

else if (data > temp1.data)

temp1 = temp1.right;

//when data equals to value at the node break

else if (data == temp1.data)

break;

}

insert the element into the left or right child depending on the value at the parent node, if data

value is less than parent then assign new node as left child else right child

if (data < temp2.data)

temp2.left = new Node(data);

else if (data > temp2.data)

temp2.right = new Node(data);

}

//return node to main

return node;

}

// inorder traversal

public void inorder() {

inorder(root);

}

private void inorder(Node r) {

if (r != null) {

}

// function for preorder traversal

public void preorder() {

preorder(root);

}

private void preorder(Node r) {

if (r != null) {

System.*out*.print(r.data + " ");

preorder(r.left);

preorder(r.right);

}

}

// Function for postorder traversal

public void postorder() {

postorder(root);

}

private void postorder(Node r) {

if (r != null) {

postorder(r.left);

postorder(r.right);

System.*out*.print(r.data + " ");

}

}

}

import java.util.Scanner;

public class LinkedListBTree

{

public static void main(String[] args)

{

Scanner scan = new Scanner(System.*in*);

//object instantiation for BTree class

BTree bt = new BTree();

System.*out*.println("Linked List Binary Search Tree\n"); char ch;

Provide a continuous choice for user to select insert. This can be done using a do-while loop with

an user choices of continuation. If ‘Y’/’ y’ is given as choice then insertion is prompted, the input

other than the above choice breaks the loop and terminates the program.

//accepts input

do

{

System.*out*.println("\nEnter integer element to insert");

bt.insert( scan.nextInt() ); //displays tree

System.*out*.print("\nPost order : ");

bt.postorder();

System.*out*.print("\nPre order : ");

System.*out*.println("\nDo you want to insert more (Type y or n)");

ch = scan.next().charAt(0);

} while (ch == 'Y'|| ch == 'y');

}

}

References :

Coding

[1]. <https://www.codeproject.com/KB/cpp/>