Name: Arches, Keith Nicole M.
CMSC 180 PROJECT Laboratory Report

INTRODUCTION

To help understand the concept of parallelism further, a project has been conducted, which focuses on creating parallel algorithms for the notable Binary Search, and Depth First Search. Just like during laboratory 3, the project focuses more on using Core Affine Parallel Programming, which is about creating threads that will be computed equally, and then will be distributed according to the number of the computer's processors. Through this, we can be able to acknowledge the value of parallelism compared to its serial counterpart.

With this concept in mind, I had my code run in an **Intel(R) Core(™) i5-8300H CPU @ 2.30 GHz, with an 8GB installed memory of RAM.** It has a **64-bit Operating System,** which is a **x64-based processor.** According to the task manager, (which is also then verified through code), the machine's current CPU count is **8.**

To help compare serial and parallel computations, the programming language chosen for both Binary Search and Depth-First Search is the **C Programming Language.** They are then divided into functions, thoroughly separating computations between serial and parallel.

ANALYSIS [BINARY SEARCH]

The following are the core functions that emphasize serial and parallel programming. For visual representations and comments of the other functions, please refer to the source code.

Task 1: serial_search

This function aims to create the serial version of the binary search. It accepts 2 array parameters (the list itself, and the array that will contain target indexes). It will revolve around 3 variables (the left, right and mid), which will be the ones traversing around the array. It basically is a serial algorithm that will search until the target values are found. Considering the theoretical analysis, the time complexity for the serial binary search is considered as O(log n). In comparison to that, considering the code below:

```
1 int* serial_search(int arr[],int arryA[]){
2     //defining the first left and right index value
3     int left = 0;
4     int right = N-1;
5     int mid;
6     while(left<=right){
7           mid = (left+right)/2;
8           if (arr[mid]==target){
9
```

```
10              counter=0;
11              int answers[N];
12              for(int i=mid-1;i>=0;i--){
13                   if (target == arr[i]){
14                        answers[counter] = i;
15                        counter++;
16                   }
17              }
18              answers[counter] = mid;
19              counter++;
20              for (int i=mid+1;i<N;i++){
21                   if(target==arr[i]){
22                        answers[counter] = i;
23                        counter++;
24                   }
25              }
26
27              for (int i=0;i<counter;i++){
28                   arryA[i] = answers[i];
29              }
30              arryA[counter+1] = counter;
31              return arryA;
32         }else if(target<arr[mid]){
33              right = mid-1;
34         }else{
35              left = mid+1;
36         }
37    }
38    arryA[0] = -1;
39    arryA[1] = 0;
40    return arryA;
41 }
```

Line 6 can mostly base around the size of the array itself. Considering the additional for-loops found inside, it could safely be defined to have the time complexity of O(N^2).

Task 2: parallel_binary_search

This function aims to create the parallel version of the binary search. It accepts two parameters (the left variable being the first on the subarray index and the right variable being the last on the subarray index). The subarray size is computed by dividing N (size of the whole array) by the number of the machine's processor.

Considering the theoretical analysis (from the serial binary search concept) the time complexity for binary search is O(log n). In comparison to that, considering the code below:

```
1 void parallel_binary_search_function(){
2     // printf("P: %d\n",p);
3     // printf("Size*P: %d\n",N);
4
5     int subarray_size = N/p;
6
7     // printf("subarray_size: %d\n",subarray_size);
8
9     pthread_t binary_array[p];
10    BINARIES binary_structure[p];
11
12    //set affinity variables
13    pthread_attr_t attr;
14    cpu_set_t cpus;
15    pthread_attr_init(&attr);
16
17    int marker = (-1);
18    for(int i = 0; i<p; i++){
19          CPU_ZERO(&cpus);
20          CPU_SET(i, &cpus);
21
22pthread_attr_setaffinity_np(&attr,sizeof(cpu_set_t),&cpu
s)23;
24
25          binary_structure[i].s = (marker+1);
26          binary_structure[i].f = (marker+subarray_size);
27
28
29pthread_create(&binary_array[i],&attr,parallel_binary_se
ar30ch,&binary_structure[i]);
31          pthread_join(binary_array[i],NULL);
32
33          marker = marker+subarray_size;
34    }
35
36    //parameters to pass: starting index and ending index
37
38
39 }
```

```
1 void *parallel_binary_search(void *arg){
```

```
2
3     //setting up index to be left and right
4     BINARIES * temp;
5     temp = (BINARIES *) arg;
6
7     int left = temp->s;
8     int right = temp->f;
9     int mid;
10    while(left<=right){
11          mid = (left+right)/2;
12          if (arr_parallel[mid]==target){
13
14                counter=0;
15                int answers[N];
16                for(int i=mid-1;i>=0;i--){
17                      if (target == arr_parallel[i]){
18                            answers[counter] = i;
19                            counter++;
20                      }
21                }
22                answers[counter] = mid;
23                counter++;
24                for (int i=mid+1;i<N;i++){
25                      if(target==arr_parallel[i]){
26                            answers[counter] = i;
27                            counter++;
28                      }
29                }
30
31                for (int i=0;i<counter;i++){
32                      insert(answers[i]);
33                      // printf("%d\t",answers[i]);
34                      // arryA[i] = answers[i];
35                }
36
37                break;
38                // arryA[counter+1] = counter;
39                // return arryA;
40          }else if(target<arr_parallel[mid]){
41                right = mid-1;
42          }else{
43                left = mid+1;
44          }
45    }
46    // arryA[0] = -1;
```

```
47    // arryA[1] = 0;
48    // return arryA;
49 }
```

Considering that there is an outer loop that creates a thread for each processor, it can be said that the code will have a running time of O(N^2*p).

Upon entering the size of the array, the code will automatically multiply it by p, which is the number of the computer's processor.

If the two running times are compared, it can be seen that the serial algorithm might prove to be much faster than using the parallel algorithm for it. Furthermore, upon testing out the following methods with the time command found on C, the following results are as follows:

Size	Serial (In Seconds)	Parallel (In Seconds)
16000x16000	5.357s	21.060s
	11.348s	4.689s
	4.773s	8.248s
AVERAGE	**7.1593333s**	11.3323333s
24000x24000	5.650s	9.798s
	4.327s	4.337s
	5.414s	5.007s
AVERAGE	**5.130333333s**	6.38066667s
32000x32000	7.627s	4.824s
	5.331s	7.531s
	7.767s	6.234s
AVERAGE	6.90833333s	**6.1963333s**
40000x40000	8.628s	8.800s
	9.147s	9.698s
	6.453s	8.638s
AVERAGE	**8.076s**	9.04533333s
56000x56000	13.373s	9.680s

	8.820s	8.430s
	10.156s	11.137s
AVERAGE	10.783s	**9.749s**

Some of the seconds indicate that the serial algorithm is much faster than using the parallel one. It could be said that their runtimes are almost the same, but there is little indication that the serial has the upper hand when it comes to the run time.

ANALYSIS [DEPTH-FIRST SEARCH]

The following are the core functions that emphasized serial and parallel programming for depth-first search.

Task 1: dfs_serial

This function is the serial version of the depth-first search algorithm. It acquires the parameter index, which is the algorithm's identifier to keep track of the visited nodes.

The theoretical time complexity for the dfs_serial is O(V) with V as the number of indexes. Comparing that to the serial code below:

```
1 void dfs_serial(int index){
2     visited[index] = 1;
3     for(int i = 1;i<=N;i++){
4           if(g[index][i] && !visited[i]){
5                 printf("[%d,%d]\n",index,i);
6                 dfs_serial(i);
7           }
8     }
9}
```

Since the code follows the serial version and follows the needed algorithm for it, it could be seen that the theoretical time complexity for the dfs_serial function is O(V) as well, with V as the number of the indexes.

Task 2: dfs_parallel

This function is the parallel version of the depth-first search algorithm. It acquires the indexes that are needed, based on the computed subarray.

Following the theoretical time complexity, it is expected that there will be at least one for-loop outside the thread function, in order to distribute the computed subarray and the changing of core affinities to each thread made. With this in mind, time complexity would be at least O(V^2).

Comparing that to the parallel implementation below:

```
1 void parallel_dfs_function(){
2
3
4     int subarray_size = N/p;
5
6
7     pthread_t dfs_array[p];
8     DFS dfs_structure[p];
9
10    //set affinity variables
11    pthread_attr_t attr;
12    cpu_set_t cpus;
13    pthread_attr_init(&attr);
14
15    int marker = 0;
16
17    for(int i=0;i<p;i++){
18          CPU_ZERO(&cpus);
19          CPU_SET(i,&cpus);
20
21pthread_attr_setaffinity_np(&attr,sizeof(cpu_set_t),&cpu
s)22;
23
24          dfs_structure[i].s = (marker+1);
25          dfs_structure[i].f = (marker+subarray_size);
26
27
28pthread_create(&dfs_array[i],&attr,parallel_dfs,&dfs_str
uc29ture[i]);
30          pthread_join(dfs_array[i],NULL);
31          marker = marker+subarray_size;
32    }
33
34}
35 void *parallel_dfs(void *arg){
36
37    DFS * temp;
38    temp = (DFS *)arg;
39
40    int start = temp->s;
41    int end = temp->f;
42
```

```
43    int counterr = start;
44
45    while(counterr<=end){
46
47          dfs_parallel(counterr);
48          counterr = counterr + 1;
49    }
50
51}
52void dfs_parallel(int index){
53    parallel_visited[index] = 1;
54    for(int i = 1; i<=N; i++){
55
56          if(g[index][i] && !parallel_visited[i]){
57                printf("[%d,%d]\n",index,i);
58                dfs_parallel(i);
59          }
60    }
61 }
```

Considering that the following functions have nested loops within each, it could be said that the parallel contains a triple nested loop. With this in mind, the parallel implementation in the code is expected to be much slower than what was intended.

If the two task's running times are compared, it can be seen that the serial algorithm will be much better to use for DFS. To test it out, the following run times are executed.

Size	Serial (In Seconds)	Parallel (In Seconds)
8000x8000	3.959s	3.541s
	3.291s	4.284s
	3.222s	5.076s
AVERAGE	**3.4906666667s**	4.30033333s
12000x12000	5.321s	7.201s
	6.093s	6.223s
	6.738s	6.245s
AVERAGE	**6.050666667s**	6.55633333s

Same with the results in the previous algorithm, the serial and parallel methods run at mostly the same time. Although, it could be seen that the serial has a shorter run time than that of the parallel method. Factors contributing to this can be the nested loop found in the algorithm interpretation, compared to the serial part, which is just a recursive method of finding the edges.

CONCLUSION

While serial programming does tasks one at a time, the parallel programming method makes it so that tasks can be done concurrently. They both have their own advantages and disadvantages, one of which is the usage of processors. In parallel programming, all processors can be used at a time, indicating a much higher usage or performance. While that may be the case however, parallel programming tends to be much more costly than serial, mostly because of the usage of all of the processors. What needs to be checked before using either one is to know what the algorithm's or code's objective is, in order to know the most appropriate one to use at the situation.

REFERENCES

C program to implement Depth First Search(DFS). C program to implement Depth First Search(DFS) | Basic , medium ,expert programs example in c,java,c/++. (n.d.). https://scanftree.com/programs/c/c-program-to-implement-depth-first-search-dfs/.

Complexity Analysis of Binary Search. GeeksforGeeks. (2019, September 30). https://www.geeksforgeeks.org/complexity-analysis-of-binary-search/.

Depth First Search (DFS). Programiz. (n.d.). https://www.programiz.com/dsa/graph-dfs.

Find the number of CPU Cores, for Windows, Mac or Linux source code. Find the number of CPU Cores, for Windows, Mac or Linux source code - Cprogramming.com. (n.d.). http://www.cprogramming.com/snippets/source-code/find-the-number-of-cpu-cores-for-windows-mac-or-linux.

Introduction to Parallel Computing. GeeksforGeeks. (2021, June 4). https://www.geeksforgeeks.org/introduction-to-parallel-computing/.

JudgeDeath, JudgeDeathJudgeDeath 14111 gold badge11 silver badge99 bronze badges, greltgrelt 37622 silver badges55 bronze badges, Pauli NieminenPauli Nieminen 96066 silver badges77 bronze badges, & JarkkoLJarkkoL 1. (1963, April 1). *Pthread affinity before create threads*. Stack Overflow. https://stackoverflow.com/questions/25472441/pthread-affinity-before-create-threads.

Mido KammiMido Kammi 4366 bronze badges, & jh314jh314 24.6k1515 gold badges5858 silver badges7979 bronze badges. (1964, May 1). *Generate adjacency matrix of*

undirected graph. Stack Overflow. https://stackoverflow.com/questions/33150887/generate-adjacency-matrix-of-undirected-graph.