OS Report Assign_2 ES22BTECH11026

02 December 2023 12:59

How I have designed my function which checks whether an input no. is Vampire?? (bool isVampire() in the code)

I run a loop from 10->999, checking for whether any no. divides the input num.
IF, it does, I store its value in a variable ' i ' and the value of quotient in 'num2'.
I then store all the digits of 'num2' and ' i ' in an array '**result[]**' (I have made a function for that', and then **SORT** it.
Similarly, I store the digits of 'num' in an array '**final[]**', and **SORT** it.
I then compare the two arrays, if they are equal, input num is Vampire and return true.
ELSE
I **continue** the loop.

For each of the operations described above, I have made a **function**(as you can see in the code), trying to make the code as **modular** as possible.

One of the good decisions I took during the course of the Assignment, was to write the code in **Linux**. Since pthreads are a feature of **POSIX**, I was able to avoid any possible complications like 'extended time durations to process the output', 'header file missing complications' etc.

In the above function, I faced may **complications**. One of them was to figure out how to determine whether an input num is **Vampire or not**. I also initially missed to include the condition that if both 'num2' and ' i ' have **trailing zeroes**, **return false**.

Logic of partitioning the N numbers among the M threads

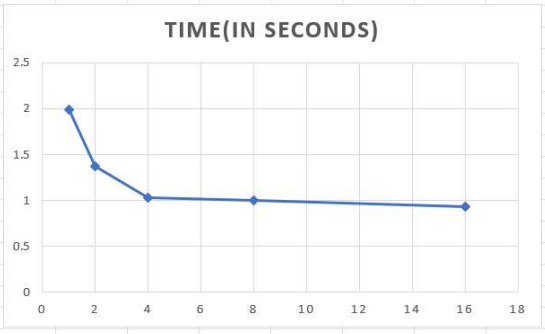
So basically, I have partitioned using the following logic:

- Suppose N = 13, M = 5
  Then I give the first thread(i.e. M = 1) the nos. 1, 6, 11
  Similarly, the second thread 2, 7, 12
  The third, 3, 8, 13 and son for the remaining threads. Using such a logic distributes almost equal nos. among the **'M'** threads, and each thread (say the **'kth'** thread) gets the numbers **k, k + M, k + 2M..**
  In such a way, **I am able to use every no. if the 'N' nos.**, making it an **efficient** algorithm

  IN THE CODE, I have passed the thread no. as an argument to the function (void store_Vampire()), which can be seen as the variable 'a'(also dynamically allocating memory in heap to the variable).

Below Attached are the graphs, Time v/s Size, N & Time v/s Number Of Threads, M show the performance of my program

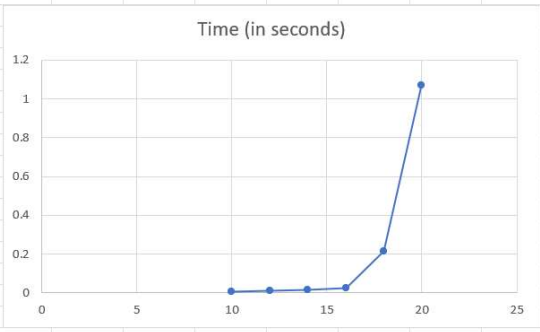
M	Time(in seconds)
1	1.989
2	1.369
4	1.027
8	0.999
16	0.934



Y- Axis: Time(in sec.) X- Axis: Values of M

******M Values correspond to the no. of threads, in powers of 2, as M = 1, 2, 4, 8, 16******

K	Time (in seconds)
10	0.008
12	0.01
14	0.014
16	0.023
18	0.215
20	1.069



Y- Axis: Time(in sec.) X- Axis: Values of K

******K values correspond to the input Size as N = 2^k, with no. of threads fixed at M = 8.*******

DESIGN OF THE PROJECT

```c
FILE* fp;
fp = fopen("InputFile.txt", "r");
if(fp == NULL){
    perror("Error");
}
else{
    fscanf(fp, "%d %d", &N, &M);
}
fclose(fp);
```

The program starts by reading the input values N and M from a file named "InputFile.txt." This file contain the values for N (the upper limit for finding Vampire numbers) and M (the number of threads to be used).

```c
void* store_Vampire(void* arg){

    int index = *(int*)arg;
    int local_buffer[N/M];
    int j = 0;
    for(int i = index; i<=N; i += M){
        if(isVampire(i)){
            fprintf(fptr, "%d%s %s %s %s %d\n", i, ":", "Found", "by", "Thread", index);
            local_buffer[j] = i;
            j++;
        }
    }
    for(int i = 0; i< j; i++){
        memory->global_buffer[global_counter] = local_buffer[i];
        global_counter++;
    }
    if(index == M){
        fprintf(fptr, "%s %s %s%s %d", "Total", "Vampire", "numbers", ":", global_counter);
    }
}
```

The store_Vampire function is the main logic for each thread. It iterates through the numbers from the index (thread number) to N with a step of M. For each number in this range, it checks if it's a Vampire number using the isVampire function. If so, it writes the information about the Vampire number & the thread ID into the "OutFile.txt" and stores it in the local buffer.

```c
struct buffer{
    int* global_buffer;
};

struct buffer* memory;
memory = (struct buffer*)malloc(sizeof(struct buffer));
memory->global_buffer = (int*)malloc(N*sizeof(int));
```

A structure named buffer is defined to hold the global buffer, which is shared among all threads. The global buffer is an array that stores Vampire numbers found by each thread.

Memory is allocated for the buffer structure and the global buffer array. This is where the Vampire numbers found by each thread will be stored.

```c
bool isVampire(int num) {
    int digitCount = no_Of_digits(num);
    if ((digitCount % 2) != 0){
        return false;
    }
    if (digitCount == 2){
        return false;
    }
    int upper_bound;
    int lower_bound;
    lower_bound = pow(10, (digitCount/2) -1);
    upper_bound = pow(10, digitCount/2);
    for(int i = lower_bound; i< upper_bound; i++){
        if(num%i == 0){
            int num2 = num/i;
            if(no_Of_digits(num2) > (no_Of_digits(num)/2) || no_Of_digits(i) > (no_Of_digits(num)/2)){
                continue;
            }
            if(num2%10 == 0 && i%10 == 0){
                continue;
            }
            int result[100];
            int final[100];
            build_Array(i, num2, result);                          //Result array stores the digits of 'num2' and 'i'
            bubble_Sort(result, no_Of_digits(i) + no_Of_digits(num2));
            makeArr(num, final);                                   //Final Array stores the digits of 'num'
            bubble_Sort(final, digitCount);
            if(compareArrays(result, final, no_Of_digits(i) + no_Of_digits(num2), digitCount)){
                return true;
            }
        }
    }
    return false;
}
```

The isVampire function. The logic has already been explained on the first page

```
for (i = 0; i < M; i++) {
    if (pthread_join(th[i], NULL) != 0) {
        return 2;
    }
}
fclose(fptr);
```

The program waits for all threads to finish using pthread_join. This ensures that the main program does not exit before all threads have completed their tasks.

At last, the file "fptr" is closed.