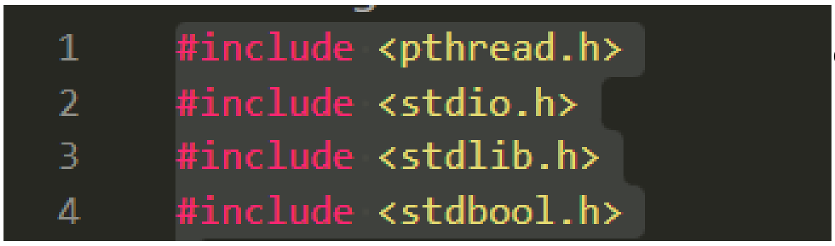
REPORT

Prepared By :
Soham Jiddewar
(ES22BTECH11017)

Low-Level Design of my program

Here our aims is to find is vampire number using multiple thread. Vampire numbers are numbers that can be expressed as the product of two numbers, each containing the same digits as the original number.

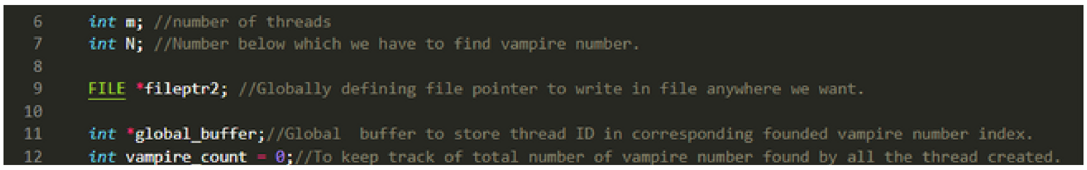
Library I used in my code



```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
```

Other than normal library i used here pthread.h library provides support for multithreading in C through the POSIX thread (pthread) API.

Global variable I used in my code



```c
int m; //number of threads
int N; //Number below which we have to find vampire number.

FILE *fileptr2; //Globally defining file pointer to write in file anywhere we want.

int *global_buffer;//Global  buffer to store thread ID in corresponding founded vampire number index.
int vampire_count = 0;//To keep track of total number of vampire number found by all the thread created.
```

Helper Function I used in my code

NumberofDigit(int num):
- Calculates the number of digits in a given number.

powerfunc(int base, int exp):
- Calculates the value of base^exp.

ifodd(int digiCount):
- Returns 1 if the digit count is odd, 0 otherwise

VampireNum(int num) Function:

- In these it Checks if a given number is a vampire number.
- Initializes arrays **array_original** and **array** to count occurrences of digits.
- Checks if the digit count is odd or 2 and returns false.
- Counts occurrences of digits in the original number using **array_original**.
- Iterates through possible divisors and checks if they form vampire numbers.
- If, divisor and quotient all digit matches with num then it is vampire number.That we get via counting over digit of occurrences of digits in both array.

Main Function:

- Here first of all my main function Reads **m** and **N** from the input file (**infile.txt**).
- Allocates memory for **global_buffer**.
- Creates an array of thread identifiers (**tid**).
- Creates threads, each with a unique index, using **pthread_create**.
- Waits for threads to finish using **pthread_join**.
- Writes the found vampire numbers and corresponding thread IDs to an output file (**outfile.txt**).
- Exits the program.

*runner(void *arg) Function:*

- Here it first of allTakes an index as an argument, representing the thread's identifier.
- Allocates a local buffer **(localbuffer)** to store vampire numbers found by the thread.
- Iterates through a range of numbers, checking for vampire numbers and updating global_buffer.
- Frees the memory allocated for localbuffer.
- Exits the thread.

Output Analysis:

- The program writes the found vampire numbers and the corresponding thread IDs to **outfile.txt**.
- The output includes a total count of the vampire numbers found.
- Each thread operates on a subset of the numbers, partitioned based on the thread index.
- Checks if input files are successfully opened.
- Prints an error message if thread creation or joining fails.

Complication arises :

Basically when i am creating thread at first I am doing that I first creating one thread and performing all its required task then creating next thread and performing its task then afterward I understand that I have to run all thread simultaneously which will give output fast .

Second one when I am storing local buffer data from each thread into global buffer ,At first I am trying to returning local buffer data to main thread at the time when **pthread_join** waits for each thread to finishes . But then I store thread ID corresponding vampire Number index in global_buffer which help me to directly access it main function.

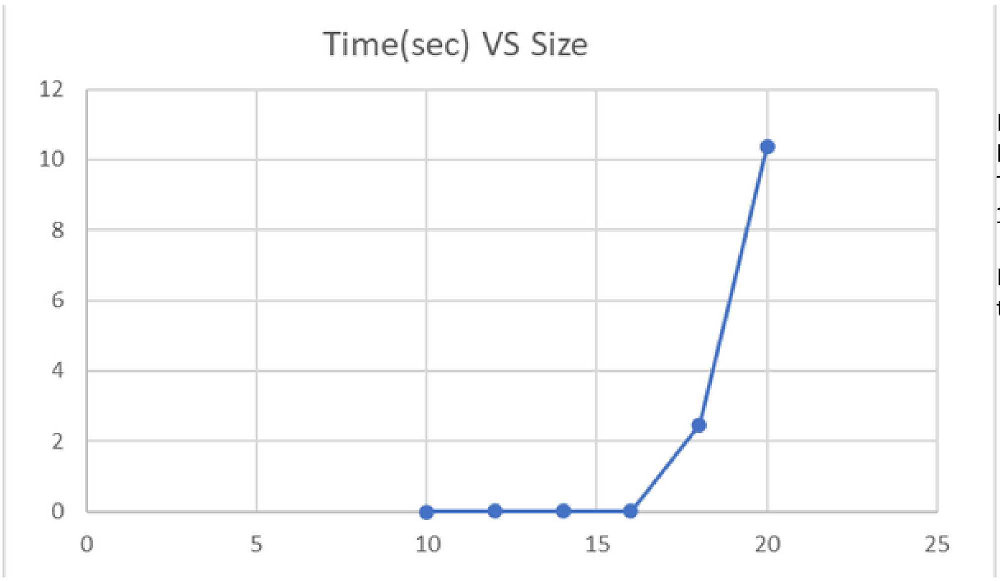
Logic of partitioning:

Here I partition N numbers in m threads such that each thread get N/m sets of number.And it works as first m threads will work on first m number then m+1 th number will be tested by 1st thread so on to m thread.here it help us as it divide works between every thread.So it is good partitioning.
for example take N=12 and m=4 then 1st thread will work on 1 ,2nd will work on 2,3rd will work on 3,4th will work on 4 and here our partitioning structure start now 5 will be worked by 1 , 6 will be worked by 2nd , 7 will be worked by 3rd and...

These is how I partition my N numbers in m threads..

Graphs which shows the performance of my program:

<u>*These is time(sec) Vs size graph*</u>



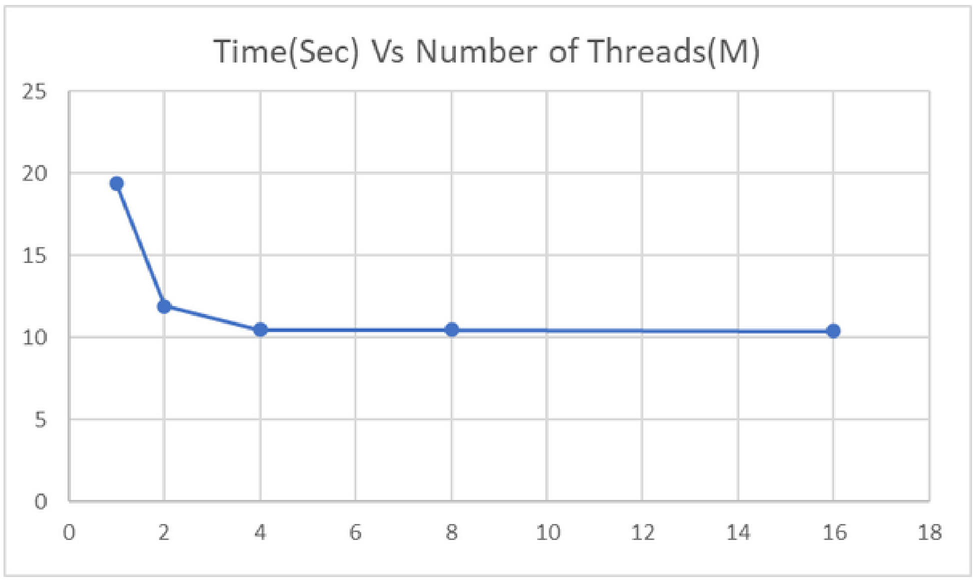
In this graph, the **y-axis** will show the time taken by my algorithms.
The **x-axis** will show the values of k varying from 10 to 20 in increments of 2 where N=2k.

Have M, the number of threads fixed at 8 for all these experiments.

Here if i calculate for all intermediate value we get to know that curves is like exponential.

Here as we get to know that as size increases means 2^{10}=1024 to 2^{20}=1048576 and 8 threads fix for all the time increases as threads have to work more as number of sets (i.e N/m) each threads have increases therefore over overall time increases.

These is time(sec) Vs Number of thread(m) graph



The **y-axis** will show
the time taken by your algorithm.
The **x-axis** will be the values of M, the
number of threads varying from 1 to 16 (in
powers of 2 i.e, 1,2,4,8,16). Have N fixed at
1000000 (10 lakhs) for all
these experiments.

Here we have fixed N=10 lakh and we are changing number of thread from 1-2-4-8-16 we get to know that for m=1 we get so much time as it will be like our simple code but when we start to share works among multiple thread it will lead to decrease time and hence improved program efficiency.

Also there is big time drop from 1 thread to 2 threads and nearly same time for 2,4,8,16 threads these is beacuse Increasing the number of threads introduces additional overhead, such as thread creation, synchronization, and context switching. If the overhead becomes a bottleneck, further increasing the number of threads may not lead to significant performance improvement.