OS ASSIGNMENT-2 REPORT

Rajiv Shailesh Chitale
CS21BTECH11051

Input:
A file name is taken as a command line argument or input. The file contains parameters N and K. The program finds perfect numbers from 1 to N using K posix threads.

Creation of threads:
A for loop runs with k iterations of setid. A new thread is created in each iteration using `pthread_create()`. It is made to run the function `perfectFinder()` with a set of numbers corresponding to that setid.

Details for each thread are stored under `struct SolverThread`.

- ➔ pthread_t tid; // id of pthread
- ➔ pthread_attr_t attr; // attributes of a pthread
- ➔ void *param; // pointer to array on heap which stores parameters
- ➔ void *retval; // pointer to array on heap which stores return values

An array of such structs is made and indexed using setid.
Note: the pointers are void* due to the requirement of pthread functions. They are typecasted to long long* before usage.

Passing arguments to threads:
N, K and setid need to be fed to the threads. These are first stored in an array in the heap, in that order. The pointer to the array is then passed as an argument to `perfectFinder()`, via the third argument of `pthread_create()`.

Data Distribution:
Each thread generates its own set of data using its unique setid. The first element is decided as setid+1 and the remaining are obtained by repeatedly adding K (until the value exceeds N) . For any number val, we can find its setid = val%k -1. This distribution gives threads an almost equal number of tasks which are of similar difficulty.

Checking Perfect Numbers:
A number is perfect if the sum of its factors excluding the number itself is equal to the number. Let us consider the number n. Suppose a is a factor of n, then there is a corresponding factor b such b = n/a. Further, for every b>sqrt(n) we obtain a< sqrt(n).

The code uses this knowledge to narrow the search for factors down to floor(sqrt(n)). The other factors are obtained as b = n/a. Each such pair of factors is added to a partial sum (which is initialized with 1). The case b=a=sqrt(n) is handled separately to avoid repetition of factors. If the partial sum exceeds the number, the number is immediately deemed to not be perfect (return value 0). Finally, if the sum equals the number, then isPerfect() returns the value 1.

Numbers which are found to be perfect are stored in a local array in each solver thread.

Returning values from threads:
Upon completely searching their sets, the threads prepare to return the results. The count of perfect numbers and their values are stored in an array in the heap. The threads use `pthread_exit()` to terminate and return the pointer to the above array.

Joining threads:
The main thread loops over each setid: It calls `join`() using the tid stored since thread creation. The second argument of `join`() receives the pointer to the results, returned by the thread. Output is generated using these results. The arrays are freed from the heap.

Output:
The main thread combines results of each solver thread into one text file, OutMain. The perfect numbers are grouped by the thread that found them (only those with count > 0). Additionally, every solver thread has a log file, OutFileX. It writes the result of every number it checks into this file using `fprintf`(). Upon completion of the program, “Completed in X seconds!” is printed to the terminal. The time taken is calculated by subtracting the start time from the finish time.

Analysis:

Outfile for N=2000, K=6:
Thread3: 28 496
Thread5: 6

Outfile for N=50000, K=8:
Thread3: 28
Thread5: 6
Thread7: 496 8128

Outfile for N=2000000, K=4:
Thread1: 6
Thread3: 28 496 8128

Thus the occurrence of perfect numbers is very rare. There are some order of magnitudes without a single perfect number.

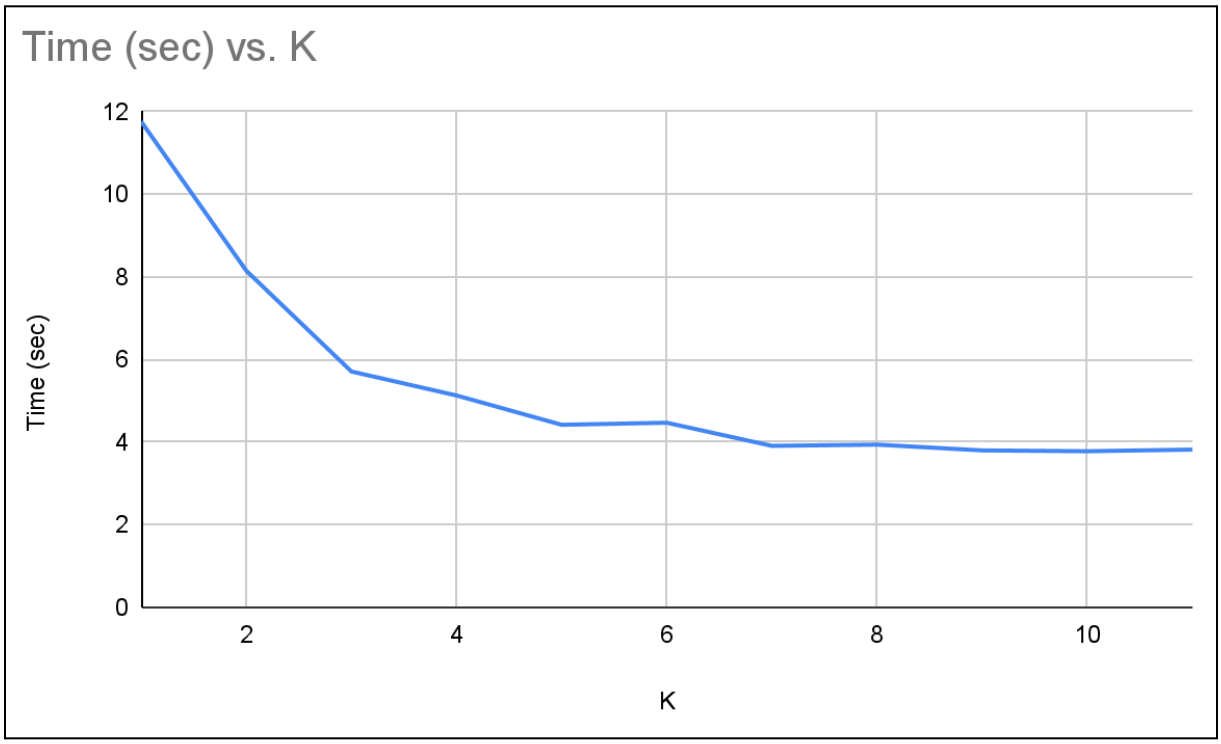
Multiple threads were successful in finding results. But half of the threads never found any perfect number, because the first four perfect values are all even.

The distribution of data worked successfully as the amount of numbers in each log file only differed by at most one.

Time for program execution:

Provided below in seconds, for N=2000000 (ie. 2×10^6)

K	Time (sec)	K	Time (sec)
1	11.76	7	3.91
2	8.15	8	3.94
3	5.71	9	3.8
4	5.13	10	3.78
5	4.42	11	3.82
6	4.47	100	4.33



Analysis:

- ➔ Time decreases as the number of threads (K) increases. This speedup implies that pthreads on linux are mapped to multiple kernel threads which can run concurrently.
- ➔ My computer has 4 cores with 2 threads each, which equates to 8 CPUs. Executing more than 8 threads concurrently is not possible with my hardware, thus there is no benefit beyond K = 8 on my hardware.
- ➔ Even at 100 threads, the time did not increase significantly. Thus the context switch time for threads is very low.

Note: for small values of N, even having 8 threads was not useful because the context switch time was not negligible.
For example with N=2000,
K = 1 took 0.006026 seconds
K = 5 took 0.002672 seconds
K = 8 took 0.006634 seconds