OS Assignment #2

By Ahmad Almomani (134081) and Mamdouh

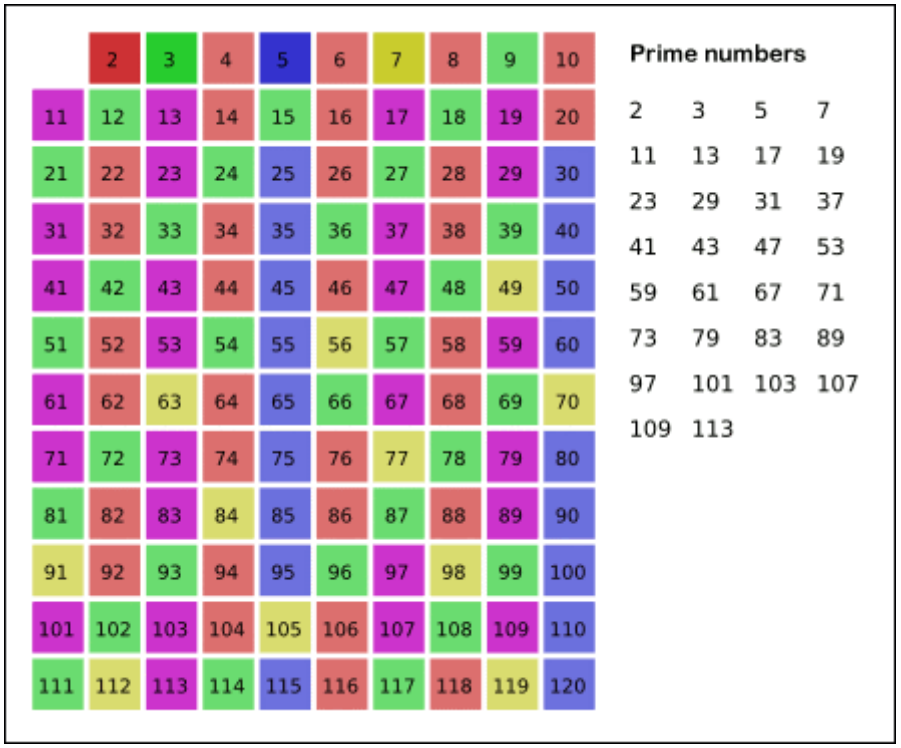
Supervision: Dr Mohammad Alshboul

Table of contents

Introduction

Prime numbers

Prime numbers are a very crucial topic in modern computing, this is due to multiple reasons, one of them being fundamental for nowadays encryption. Prime numbers have been researched even in ancient since, such as the ancient algorithm ***sieve of Eratosthenes*** and the earlier ***Euler's totient function***.



What is PrimeFinder

PrimeFinder is a set of programs that calculate and find the prime numbers within a given range. The study of this project includes various computing and math topics, including threading, Operating Systems, Profiling, algorithms and mathematics.

PrimeFinder Components

The project contains a set of tools and files described as flow:

- **PrimeFinder:** a tool that finds prime numbers efficiently but inaccurately.
    - **syncedPrimeFinder:** a version that finds prime numbers accurately. This version does some threads synchronization to ensure accuracy.
    - **OptimizedPrimeFinder:** an optimized and efficient version. This tool does synchronization by avoiding sharing resources to get rid of any racing issues. This version also optimizes CPU CacheLine.
    - **Sieve:** this version uses the much faster ***sieve of Eratosthenes*** algorithm. Due to incompatibility with the requested tasks and defeating the purpose of threading, this version is **deprecated and not used**.
- **Text files**
    - **In.txt:** inputs the range to find primes within.
    - **Out.txt:** output of the found prime numbers. This filename can be changed in the command line.
- **Evaluate:** a program (close to a script) that evaluates and benchmarks PrimeFinder on multiple amounts of threads to find the best options.

Glossary

Term	Meaning
Prime number	a natural number greater than 1 that is not a product of two smaller natural numbers
#threads	Number of threads
#logica_cores	Number of physical + logical cores (total number of hardware threads)
pthread	Thread created under the POSIX standards

The Code

Main code:

```
// convert 2d char arr to string arr
std::string args[argc];
for (int i = 0; i < argc; i++)
{
    const std::string x(argv[i]);
    args[i] = x;
}

const std::string usage = "primesFinder number_of_threads [primes_file]";
if (args[1] == "--help" || argc == 1)
{
    std::cout << "This program finds prime numbers (range from in.txt), usage:" <<
usage << "\n";
    return 0;
}

if (argc < 2)
{
    std::cerr << "Invalid inputs! Please use" << usage << "\n";
    return -1;
}

// convert second arg to int
int threadsCount = std::stoi(args[1]);

// read in.txt
readRange("in.txt", startTarget, endTarget);

// resize primes to range
primes.resize(endTarget - startTarget + 1);

find_primes(startTarget, endTarget, threadsCount);

// write output file
if (argc > 2)
{
    writePrimes(args[2]);
}

// print statistics
std::cout << "numOfPrimes=" << numOfPrimes << ", totalNums="
          << totalNums << "\n";

pthread_exit(NULL);
```

Thread Functions

count_primes

```
void *count_primes(void *args)
{
    // covert args to int vars
    struct prime_args *temp = (struct prime_args *)args;

    int l = temp->l;

    // edge case, skip if l=1
    if (l == 1)
    {
        l = 2;
        pthread_mutex_lock(&mutex);
        ++totalNums;
        pthread_mutex_unlock(&mutex);
    }

    int r = temp->r;

    delete temp;

    for (int i = l; i <= r; ++i)
    {
        bool prime = true;

        // check if i is divisible by 2:sqrt(i)
        for (int j = 2; j * j <= i; ++j)
        {
            if (i % j == 0)
            {
                prime = false;
                break;
            }
        }

        pthread_mutex_lock(&mutex);
        if (prime)
        {
            primes[i - startTarget] = true;

            ++numOfPrimes;
        }

        ++totalNums;
        pthread_mutex_unlock(&mutex);
    }

    pthread_exit(NULL);

    return NULL;
}
```

Find_primes

```
void find_primes(int l, int r, int t)
{
    pthread_t threads[t];

    // calculate the number range of each thread
    int ranges[t][2];
    for (int i = 0; i < t; ++i)
    {
        ranges[i][0] = l + ((r - l + 1) / t) * i + (i < (r - l + 1) % t ? i : (r -
l + 1) % t);
        ranges[i][1] = ranges[i][0] + ((r - l + 1) / t) - !(i < (r - l + 1) % t);
    }

    for (int i = 0; i < t; ++i)
    {
        prime_args *args = new prime_args;

        args->l = ranges[i][0];
        args->r = ranges[i][1];
        if (args->r >= args->l)
            std::cout << "ThreadIndex=" << i << ", startNum=" << args->l << ",
endNum=" << args->r << "\n";
        else
            std::cout << "ThreadIndex=" << i << " Not Needed\n";
        pthread_create(&threads[i], NULL, count_primes, (void *)args);
    }

    // wait for all the threads to finish
    for (int i = 0; i < t; ++i)
    {
        pthread_join(threads[i], NULL);
    }
}
```

Thread-safe

You can see throw the count_primes that we used mutex lock/unlock twice:

- In the edge case if statement to update ***totoalNums*** :

```
pthread_mutex_lock(&mutex);
++totalNums;
pthread_mutex_unlock(&mutex);
```

- Inside the check-if prime for loop to update ***totalNums*** and ***numOfPrimes***:

```
pthread_mutex_lock(&mutex);
if (prime)
{
primes[i - startTarget] = true;

++numOfPrimes;
}

++totalNums;
pthread_mutex_unlock(&mutex);
```

Commenting out these lock/unlock statements will result in race conditions as described next.

Testing race condition

For testing race condition in multiple number of threads we will use the bash script:

```
for i in 1 4 16 64 256 1024;do echo "testing $i threads"; ./prog $i; done
```

Where ***prog*** is either primeFinder or syncedPrimeFinder. (in.txt= 1 1024)

- primeFinder (not synced) result:

testing 1 threads
ThreadIndex=0, startNum=1, endNum=1024
numOfPrimes=172, **totalNums=1024**
testing 4 threads
ThreadIndex=0, startNum=1, endNum=256
....
numOfPrimes=172, totalNums=1024
testing 16 threads
ThreadIndex=0, startNum=1, endNum=64
....
numOfPrimes=172, totalNums=1024
testing 64 threads
ThreadIndex=0, startNum=1, endNum=16
.....
numOfPrimes=171, totalNums=1024
testing 256 threads
ThreadIndex=0, startNum=1, endNum=4
....
numOfPrimes=171, **totalNums=1022**
testing 1024 threads
ThreadIndex=0, startNum=1, endNum=1
numOfPrimes=172, totalNums=1024

We can see that the result for numOfPrimes on T=64 and T=256 is 171, which is incorrect, definitely because of racing.

- syncedPrimeFinder

testing 1 threads
ThreadIndex=0, startNum=1, endNum=1024
numOfPrimes=172, totalNums=1024
testing 4 threads
ThreadIndex=0, startNum=1, endNum=256
....
numOfPrimes=172, totalNums=1024
testing 16 threads
ThreadIndex=0, startNum=1, endNum=64
....
numOfPrimes=172, totalNums=1024
testing 64 threads
ThreadIndex=0, startNum=1, endNum=16
.....
numOfPrimes=172, totalNums=1024
testing 256 threads
ThreadIndex=0, startNum=1, endNum=4
....
numOfPrimes=172, totalNums=1022
testing 1024 threads
ThreadIndex=0, startNum=1, endNum=1
numOfPrimes=172, totalNums=1024

There is no racing here, and output is always correct -regardless of T-.

Evaluation

Best #threads

The project contains a C++ file with the name **"evaluation.cpp"**, this program is used to evaluate and benchmark our project on multiple numbers of T, more specifically, single-thread to twice the number of the machine logical cores.

The file "benchmarks.txt" has some tests we did on a machine -with 8 logical cores-. We tested all three versions of the code to get the average run-time for each number of T cases.

This snippet shows benchmarks for the synced version from T=1 to T=16:

```
[2]: syncedPrimesFinder:
Avg time on 1-threads =5114.12 <==== worst
Avg time on 2-threads =3330.97
Avg time on 3-threads =2535.12
Avg time on 4-threads =2053.43
Avg time on 5-threads =1882.02
Avg time on 6-threads =1901.45
Avg time on 7-threads =1874.23
Avg time on 8-threads =1830.21
Avg time on 9-threads =1796.33
Avg time on 10-threads =1783.12
Avg time on 11-threads =1758.12
Avg time on 12-threads =1778.8
Avg time on 13-threads =1777.87
Avg time on 14-threads =1747.85 <==== best
Avg time on 15-threads =1781.7
Avg time on 16-threads =1764.34
```

We can see the best result we got was on 14 threads. Although the machine has 8 logical cores, it would be assumed that the best #threads should be also 8. This estimation - #threads is best when equal to #logical_cores- is quite reasonable and even possible in some cases.

A possible reason that T is best on 14 and not 8 in our case, is that when creating more threads, the OS Scheduler will have higher chances of giving the cores to any thread from our program, hence decreasing the overall run-time. We can be more sure that decreasing the #threads any less than #logical_cores gives slower run-time (assuming context switching overhead is minimum).