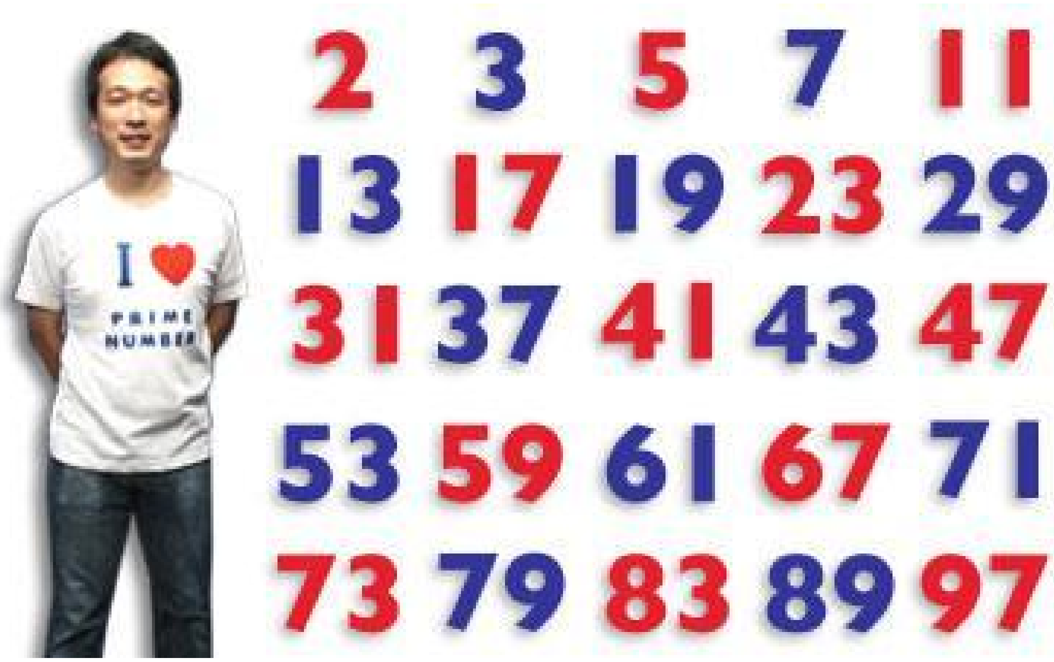
Concurrency Prime Numbers Report

Derry Brennan

C00231080



Source: https://puzzlefry.com/puzzles/prime-pairs-riddle/

Tutor: Joseph Kehoe

Date: 10/02/2021

Introduction

Tasked with producing a concurrent program using C++ and OpenMP to output the amount of prime numbers up to a given number (N) and the list of all the twin primes found up to N. Twin primes are prime numbers that are 2 different from each other e.g (3, 5) or (5, 7). The following was performed on a Linux virtual machine with the following specifications.

System info

Operating System	Linux Mint 20 Cinnamon
Cinnamon Version	4.6.6
Linux Kernel	5.4.0-26-generic
Processor	AMD Ryzen 9 3900X 12-Core Processor × 12
Memory	10.1 GiB
Hard Drives	33.3 GB
Graphics Card	VMware SVGA II Adapter (prog-if 00 [VGA controller])

Upload system information

The code along with a make file and a readme to replicat the tests carried out with are available at https://github.com/derrymb/CDDLabs/tree/main/Prime%20Project/PrimePairs.

Algorithm

Firstly I used a function to calculate the prime numbers up to the input number:

```
vector<int> primes;       // Global vector to store Primes
vector<int> primepairs;   // Global vector to store Primes Pairs
Void calculatePrime(int n)
{
      #OpenMP parallel for
      for(i =2; i <= n: i++)
      {
            bool not_prime = false
            for(j = 2; j < i; j++)
            {
                  if(i%j == 0)
                  {
```

```
                Not_prime = true;
                Break;
            }
        }
        if(!not_prime)
        {
            #OpenMP critical
            primes.push_back(i);
        }
        else if(i==2)
        {
            #OpenMP critical
            primes.push_back(i);
        }
    }
}
```

This function takes in the input number N and starts off a loop which openMP splits into threads workloads, it goes from 2, the first prime number up to N. We make a boolean variable not_prime false initially and then enter the next loop which is not run concurrently.
This inner loop also starts from 2 and goes up to the index of the first loop, does a modulo of (i%j == 0) to determine if that i number is not a prime, if this condition is true, we make our bool variable true and break this loop, If this condition was never met that means we found a prime and it is entered into the primes vector. An additional condition of if(i == 2) push_back(i) was entered to catch this tricky prime, since it is the only even prime it would fail our other checks.
The next function was constructed to find the prime pairs within our vector of primes:

```
void findPrimePairs()
{
    #OpenMP parallel for
    for(i = 0; i < primes.size() -1; i++)
    {
        if(primes[i] + 2 == primes[i+1])
        {
            #OpenMP critical
            {
                primepairs.push_back(primes[i]);
                primepairs.push_back(primes[i+1]);
            }
        }
    }
```

}
This function simply runs through the prime vector and checks if the value at pimes[i] +2 is equal to the value at primes[i+1], add int two to the first value and check it against its right neighbour. If this is true, both values are added to the primepairs vector.

Then the main function will sort both vectors after each respective function is called as the parallel nature of the for loops does not have things in order for doing the required calculations on them and will then print out the results on to the screen.

The Result

On my system the default number of threads openMP spins up is 12 and running the program concurrently finding the primes and prime pairs up to 1 million takes roughly 13.5 seconds as shown below.

```
Parallel execution for 1000000 numbers was : 13296 milliseconds.
Total Prime numbers from 0 to 1000000 = 78498
Total Prime Pairs from 0 to 1000000 = 8169

real    0m13.298s
user    2m36.342s
sys     0m1.007s
```

Figure 1: Speed for parallel program on 1 million numbers 12 threads

Average time for 1,000,000 numbers parallel 12 threads

First run	Second run	Third run	Forth run	Fifth run	Average
13.627 seconds	13.936 seconds	14.008 seconds	13.298 seconds	13.859 seconds	13.7456 seconds

```
Parallel execution for 1000000 numbers was : 134303 milliseconds.
Total Prime numbers from 0 to 1000000 = 78498
Total Prime Pairs from 0 to 1000000 = 8169

real    2m14.305s
user    2m14.268s
sys     0m0.018s
```

Figure 2: Speed for sequential program on 1 million numbers

Average time for 1,000,000 numbers sequential

First run	Second run	Third run	Forth run	Fifth run	Average
134.051 seconds	136.125 seconds	134.447 seconds	136.573 seconds	134.720 seconds	135.183 seconds

While running the same code sequentially finding the primes and pairs up to 1 million takes roughly 135 seconds.

We can force OpenMP to use a set number of threads using the num_threads(#) in the openMP pragma, doing this on 2, 4, 8, 16, 32 and 64 threads to see the difference in speed ups.

```
999959 999961
Parallel execution on 2 threads for 1000000 numbers was : 66 seconds.
Total Prime numbers from 0 to 1000000 = 78498
Total Prime Pairs from 0 to 1000000 = 8169

real    1m6.700s
user    2m13.330s
sys     0m0.016s
derry@derryVM:~/Desktop/CDDLabs/Prime Project$
```

Figure 3: Two threaded execution time

Average time for 1,000,000 numbers 2 threads

First run	Second run	Third run	Forth run	Fifth run	Average
68.268 seconds	66.700 seconds	67.908 seconds	67.518 seconds	68.053 seconds	67.7294 seconds

```
Parallel execution for 1000000 numbers was : 35145 milliseconds.
Total Prime numbers from 0 to 1000000 = 78498
Total Prime Pairs from 0 to 1000000 = 8169

real    0m35.148s
user    2m20.364s
sys     0m0.119s
```

Figure 4: Four threaded execution time

Average time for 1,000,000 numbers 4 threads

First run	Second run	Third run	Forth run	Fifth run	Average
35.148 seconds	34.326 seconds	34.649 seconds	34.191 seconds	34.361 seconds	34.535 seconds

```
Parallel execution for 1000000 numbers was : 18443 milliseconds.
Total Prime numbers from 0 to 1000000 = 78498
Total Prime Pairs from 0 to 1000000 = 8169

real    0m18.446s
user    2m27.061s
sys     0m0.260s
```

Figure 5: Eight threaded execution time

Average time for 1,000,000 numbers 8 threads

First run	Second run	Third run	Forth run	Fifth run	Average
18.446 seconds	18.191 seconds	17.953 seconds	17.828 seconds	18.261 seconds	18.1358 seconds

```
Parallel execution for 1000000 numbers was : 13123 milliseconds.
Total Prime numbers from 0 to 1000000 = 78498
Total Prime Pairs from 0 to 1000000 = 8169

real    0m13.125s
user    2m35.266s
sys     0m0.530s
```

Figure 6: Sixteen threaded execution time

Average time for 1,000,000 numbers 16 threads

First run	Second run	Third run	Forth run	Fifth run	Average
13.125 seconds	13.567 seconds	13.622 seconds	13.571 seconds	13.561 seconds	13.4892 seconds

```
Parallel execution for 1000000 numbers was : 13572 milliseconds.
Total Prime numbers from 0 to 1000000 = 78498
Total Prime Pairs from 0 to 1000000 = 8169

real    0m13.575s
user    2m40.004s
sys     0m0.842s
```

Figure 7: Thirty two threaded execution time

Average time for 1,000,000 numbers 32 threads

First run	Second run	Third run	Forth run	Fifth run	Average
13.753 seconds	13.668 seconds	13.575 seconds	13.685 seconds	13.761 seconds	13.6844 seconds

```
Parallel execution for 1000000 numbers was : 13631 milliseconds.
Total Prime numbers from 0 to 1000000 = 78498
Total Prime Pairs from 0 to 1000000 = 8169

real    0m13.636s
user    2m40.482s
sys     0m0.777s
```

Figure 8: Sixty four threaded execution time

Average time for 1,000,000 64 threads

First run	Second run	Third run	Forth run	Fifth run	Average
13.636 seconds	13.693 seconds	13.630 seconds	13.651 seconds	13.680 seconds	13.658 seconds

As the treads go up above the number of cores available to the system the time no longer increases, but neither does the extra context switching prove to be a great slow down on the system, taking things to an extreme we will do one last test on 2048 threads.

```
Parallel execution for 1000000 numbers was : 13711 milliseconds.
Total Prime numbers from 0 to 1000000 = 78498
Total Prime Pairs from 0 to 1000000 = 8169

real    0m13.724s
user    2m41.670s
sys     0m0.948s
```

Figure 9: 2048 threaded execution time

Average time for 1,000,000 numbers 2048 threads

First run	Second run	Third run	Forth run	Fifth run	Average
13.724 seconds	13.496 seconds	13.692 seconds	13.645 seconds	13.574 seconds	13.6262 seconds

To show that openMP is actually creating all the threads asked for a simple program was written to have a for loop and have every thread print out their tread number once they enter the for loop. This was run on a static scheduler and all 2048 threads did their thing.

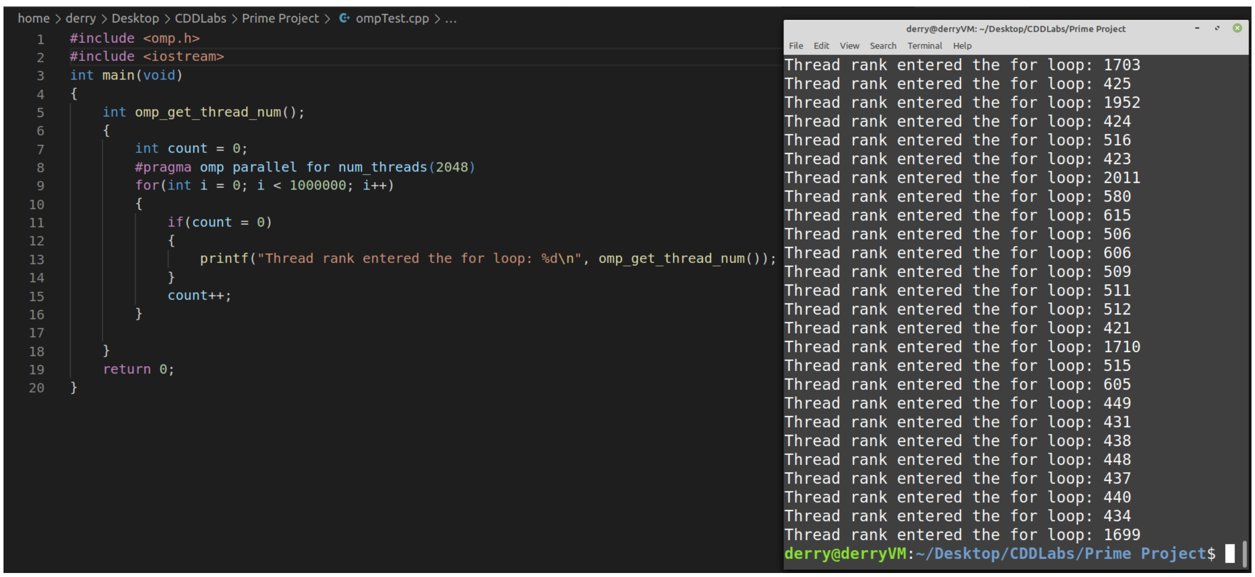


Figure 10: 2048 threaded for loop test

With the addition of the schedule(dynamic) tag to the #pragma this did reduce the amount of threads actually used, but some were just used multiple times instead.

Speedup

Absolute Speedup

Since the sequential time for the program was 135 seconds and a calculation says that 85% of the code is parallelizable. The absolute speed up of the program can be calculated using

$$S_n = \frac{T_s}{T_p(n)}.$$

- Ts = time of sequential program = 135.183
- Tp(n) time of parallel program with n processors = 13.7456(12)
- 0 < Sn <= n (always?)

$135.183/13.7456(12)$ = 0.819553166

Absolute Efficiency

$$E_n = \frac{S_n}{n}.$$

- 0 < En <= 1 (always?)

$0.819553166/12 \ = \ 0.068296097$

Amdahl's Law

$$S_n \leq \frac{1}{f + \frac{1-f}{n}}$$

- Sn = speedup
- F = Sequinital section of code
- 1 -F = parallel section of code
- N = number of cores

1/0.15+(0.85/12) = 4.385

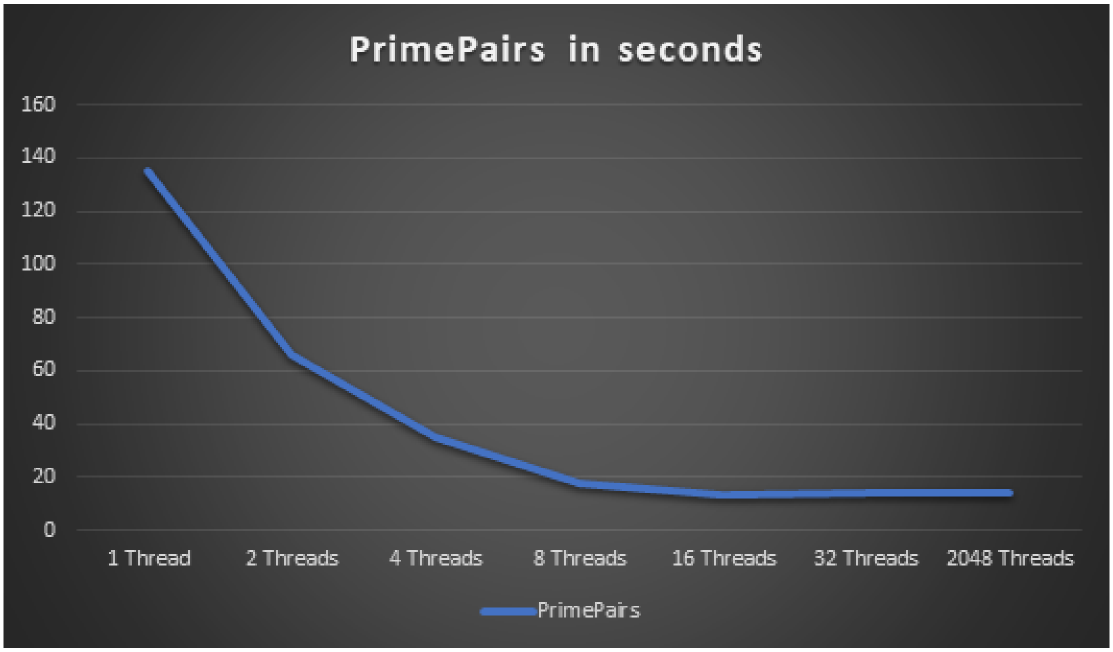


Figure 11: Time in seconds for 1 - 32 threads with a bonus 2048

The time to complete the program on n = 1000000 roughly halves each time the thread count doubles until it goes beyond the available cores on the machine and then levels out with no noticeable increase in context switching time even as the thread count goes to great heights.

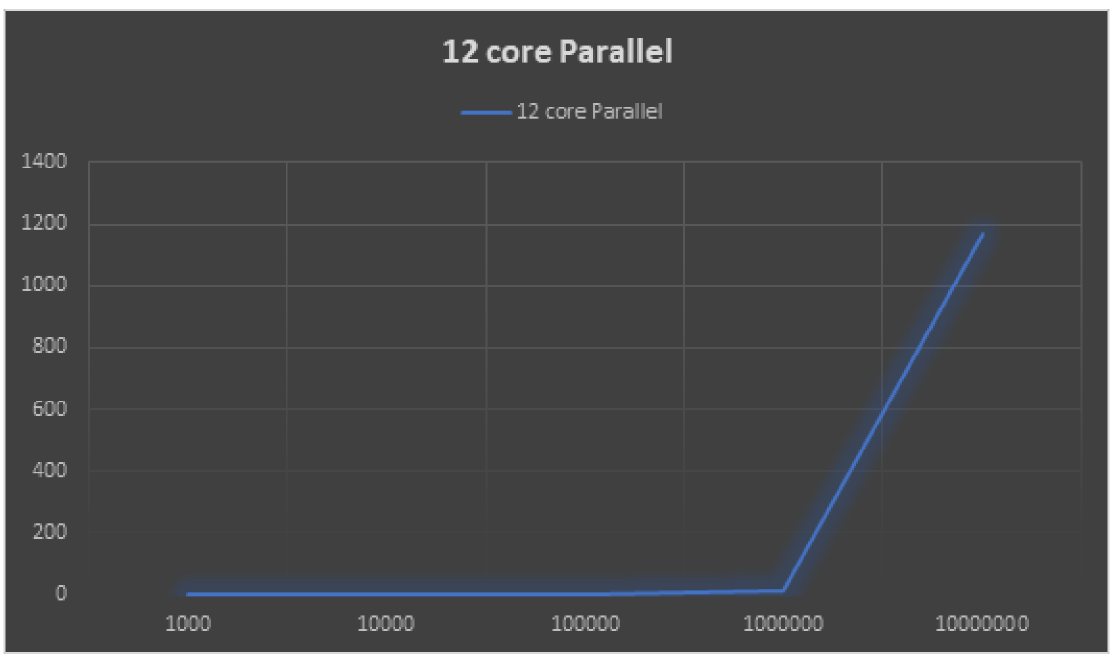


Figure 12: time in seconds for N = 1000 - 10000000

As can be seen in figure 12 this program does not scale very well, it increases exponentially as the number n increases. After some research it appears that the algorithm that I came up with was not the most efficient going and selecting a better algorithm such as the sieve of Atkins, Eratosthenes or Sundaram would have led to a far scalable program and will be kept in mind for future prime number projects. There are some sample programs running the different sieves on GitHub at https://github.com/derrymb/CDDLabs/tree/main/Prime%20Project/PrimePairs.

Conclusion

The use of OpenMp greatly sped up the process of running parallelizable code up until the number of cores on the system was reached. It is very easy and intuitive to implement, greatly increasing the likelihood of being used in every project henceforth where sections are parallelizable. The only thing limiting the program at the moment is the algorithm, which can be optimized in a number of ways to improve the scalability.