Counting Prime Numbers in Paralell - Faster by Reducing the Synchronization Overhead

Duraković Adna*, Pajić Enil*, Branković Ilvana*, Kušundžija Elma*, Karkelja Sead*
*Faculty of Electrical Engineering/Department for Computer Science, University of Sarajevo
Sarajevo, Bosnia and Herzegovina
(adurakovic4, epajic1, ibrankovic1, ekusundzij1, skarkelja1)@etf.unsa.ba

***Abstract*— This paper illustrates comparison of sequential and parallel execution of algorithm that counts prime numbers between 1 and N. Many systems rely on problems in number theory, where primes have an important role. Primes are used for encryption and hash functions too. In order to reduce the complexity of sequential algorithm execution, which is O(n^2), and also to achieve speedup, the algorithm was parallelized and some parts of it were additionally optimized by saving time on barriers. OpenMP tool is used as an application programming interface for parallelization. Sequential and parallel algorithm performances were measured on different architectures, based on Intel i3 and i7 processors, and the results, as well as the achieved speedup, are presented in the paper.**

***IndexTerms*—Primes, speedup, Paralellization.**

I. INTRODUCTION

It's known, since Euclid, that there is an infinite number of prime numbers[1]. There are also a lot of proofs that support this thesis. Mathematicians were always keen to investigate existence of an infinite number of primes. As time passed by, they came up with idea to define the function π(x) which calculates number of primes that are smaller than a certain number (value of x).

$$\boldsymbol{\pi(x) = \#\{p \leq x, where\ p\ is\ prime\ number\}}$$

For example, if we look at all primes smaller than 25, then we have following numbers: 2, 3, 5, 7, 11, 13, 17, 19 and 23. So, $\boldsymbol{\pi(25) = 9}$.

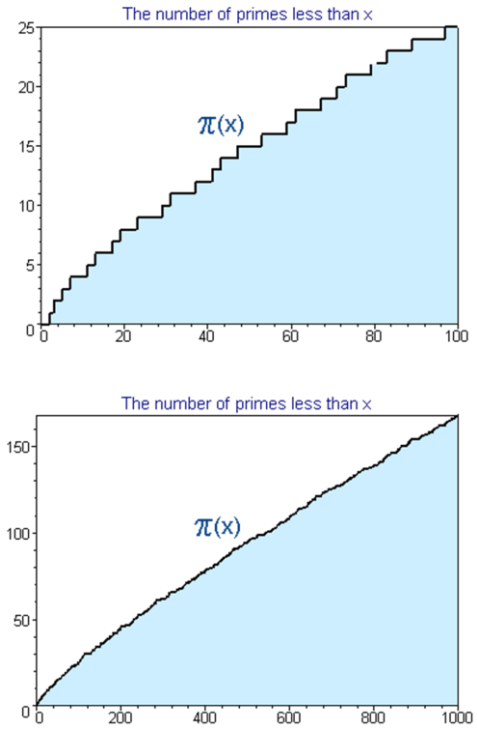

Fig. 1.Graph of function $\boldsymbol{\pi(x)}$ [3]

Before the invention of computers, mathematicians made a table of prime numbers. The most used one was D.N. Lehmer's[2] table of prime numbers (maximum value of x in table was 10,006,721).

[1]A prime number (prime) is a natural number greater than one that has no positive divisors other than one and itself.

[2] Derrick Norman Lehmer (1905.-1991.) was an American mathematician and number theorist.

	x	pi(x)
1	10	4
2	100	25
3	1,000	168
4	10,000	1,229
5	100,000	9,592
6	1,000,000	78,498
7	10,000,000	664,579
8	100,000,000	5,761,455
9	1,000,000,000	50,847,534
10	10,000,000,000	455,052,511
11	100,000,000,000	4,118,054,813
12	1,000,000,000,000	37,607,912,018
13	10,000,000,000,000	346,065,536,839
14	100,000,000,000,000	3,204,941,750,802
15	1,000,000,000,000,000	29,844,570,422,669
16	10,000,000,000,000,000	279,238,341,033,925
17	100,000,000,000,000,000	2,623,557,157,654,233
18	1,000,000,000,000,000,000	24,739,954,287,740,860
19	10,000,000,000,000,000,000	234,057,667,276,344,607
20	100,000,000,000,000,000,000	2,220,819,602,560,918,840
21	1,000,000,000,000,000,000,000	21,127,269,486,018,731,928
22	10,000,000,000,000,000,000,000	201,467,286,689,315,906,290
23	100,000,000,000,000,000,000,000	1,925,320,391,606,803,968,923
24	1,000,000,000,000,000,000,000,000	18,435,599,767,349,200,867,866
25	10,000,000,000,000,000,000,000,000	176,846,309,399,143,769,411,680

Fig. 2.Table of $\boldsymbol{\pi(x)}$ function values and corresponding x values [3]

Until 2007, mathematicians were manually finding a way of computing π(x) function. Xavier Gourdon was the first to write computer program for calculating π(x) function (maximum value of x was $\boldsymbol{4 * 10^{22}}$). The importance of the algorithm was in its use in hash functions and data encryption. Encryption is based on the theory of numbers, in which primes have important role. Some cryptographic algorithms such as RSA are affected by prime factorization. Practically, there is a public key which is in fact product of two large prime numbers and is used for encryption, and there is a secret key, that consists of prime numbers, for message decryption. Public key can be literally public, but only primes that are needed for decryption are known. It's almost impossible to guess those two prime numbers that form the public key. This algorithm is also used in integer and string hash functions. Both functions use prime number, which belongs to the specified interval.

II. ALGORITHM DESCRIPTION

The algorithm calculates number of primes in the interval from 1 to a given value. Time complexity of sequential program is $\boldsymbol{O(n^2)}$. Objective is to achieve speedup by parallelizing the algorithm. Speedup is achieved by parallelizing the parts of the program that can be parallelized, i.e. parts that are independent from each other and can be executed in parallel. OpenMP[3] is used as an application programming interface for parallelization. Optimal speedup achieved by executing a parallel algorithm on n processors is equal to:

$$\boldsymbol{optimal\ speedup = \frac{sequential\ execution\ time}{number\ of\ processors}}$$

In this case, the optimal speedup would be equivalent to:

$$\boldsymbol{optimal\ speedup = \frac{O(n^2)}{n}}$$

It should be noted that the performance of the program execution is affected by hardware architecture on which the program runs. In this experiment are used Intel Core i3-380M and Intel Core i7-3610QM processors. The processor i3-380M has 2 cores, operating at base frequency of 2.53 GHz, while i7-3610QM has 4 cores, operating at base frequency of 2.3 GHz. Both processors support Hyper-Threading technology, so each core of i3-380M and i7-3610QM can run two threads. Each core of i3-380M has 64KB of L1 cache, 256KB of L2 cache and 3MB of shared L3 cache, while each core of i7-3610QM has 64KB of L1 cache, 256KB of L2 cache and 6MB of L3 cache. Here after is listed pseudocode of the algorithm that calculates number of prime numbers.

Pseudocode :

```
n← inputNumber
prime← 1
total← 0
for(i=0 to n )
{
        prime← 1
        for( j=0 to i )
        {
                if(i mod j is 0)
                {
                        prime← 0
                        break
                }
        }
        total = total + prime
}
```

[3]OpenMP (**O**pen **M**ulti-**P**rocessing) is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran.

III. ALGORITHM PARALLELIZATION

As we can see from the pseudocode, it's necessary to check, for every number in the range of 1 and that number, if it's prime number, in order to define final number of primes in the range (the complexity of checking whether the number is prime or not is O(n)). Complexity of the algorithm shown in pseudocode above is O(n^2).).

We saved time by dividing independent jobs in for loop and assigning them to different threads. OpenMP, as a tool, gives various options for parallelization, if the used architecture meets certain requirements. ***#pragma omp parallel*** directive is used to indicate block that suggests parallelization. After that, variable that represents shared variable within threads is defined with ***shared***(n) directive, and in particular, with ***private***($i, j, prime$) directive it's emphasized that variables which iterate through the loop (i and j), as well as flag variable (that tells if number is prime or not), are private variables for every thread. Finally, in order to accomplish our parallelization goal, one more, specific directive ***#pragma omp for reduction***(+:***total***) is defined. It allows us to really execute parts of for loop on different threads, with an extra requirement that the total variable is special for each of the threads, but its value is added to the final value when all of the threads finish their execution.

Additional time savings could be achieved by parallelization of the inside for loop, however, this for loop manipulates the variable 'prime' which would be, in that case, shared within all the threads. This would mean that, if one thread changed the value of 'prime' variable, all the other threads should stop executing. This would cause loss of time spent on communication between threads more than achieving savings by dividing jobs on threads, so this part stayed sequential.

Algorithm with OpenMP directives as well as the optimized algorithm are given in addendums A and B, respectively.

IV. MAIN RESULTS

Various amounts of input data are used for algorithm testing; it's used from 100,000 to one million natural numbers for the search of prime numbers. Ten tests are performed in the specified range with a step of 100,000 (start with 100,000 and then go to 200,000, 300,000, …,1 million).

As it was expected, the sequential execution was the slowest, while the parallelized algorithm gave much better results. There are two parallel versions of the algorithm, one is more optimized, so instead of one variable, there is an array, with four or eight elements (depending on the number of threads), each storing results for different thread. In that case, time spent on synchronization is decreased or the need for a barrier is removed. There is an additional part that adds partial sum (4/8 array elements) and that part is exclusively sequential.

The following two tables show test results on i3-380M and i7-3610QM. They consist of sequential algorithm execution time, parallel algorithm with shared variable execution time and parallel algorithm with partial sums execution time columns. Additionally, results of parallel algorithm with shared variable execution on i3-380M, but with 2 threads instead of 4, are also included in the table below.

Algorithm (*s*) / Num. of elem.	**Sequential**	**Parallel with shared variable (2 threads)**	**Parallel with shared variable (4 threads)**	**Parallel with partial sum (4 threads)**
100 000	2.329	1.823	1.305	1.233
200 000	8.369	6.652	5.01	4.506
300 000	18.287	14.005	10.994	9.491
400 000	32.542	23.725	18.49	16.383
500 000	49.039	37.239	28.455	27.052
600 000	69.242	54.059	41.287	37.117
700 000	90.302	70.098	52.999	48.293
800 000	119.85	92.559	68.156	62.492
900 000	149.131	116.439	89.792	76.856
1 000 000	185.042	142.912	111.376	95.048

TABLE I. Optimisation results on i3-380M (execution time)

Results of parallel algorithm with shared variable execution on i7-3610QM, but with 2 and 4threads instead of 8, are also included in the table below. To save space, table shows results with a step of 100,000.

Algorithm (*s*) / Num. of elem.	**Sequential**	**Parallel with shared variable (2 threads)**	**Parallel with shared variable (4 threads)**	**Parallel with shared variable (8 threads)**	**Parallel with partial sum (8 threads)**
100 000	1.622	1.138	0.782	0.499	0.468
200 000	5.976	4.431	2.839	1.747	1.576
300 000	13.217	9.61	6.131	3.604	3.37
400 000	22.372	16.554	10.234	6.365	5.865
500 000	34.752	25.709	16.13	9.672	8.892
600 000	49.912	36.505	22.452	13.683	13.335
700 000	67.618	48.786	30.969	18.674	16.578
800 000	86.861	62.218	39.769	24.164	21.468
900 000	109.484	78.247	49.405	29.734	27.156
1 000 000	133.801	95.335	57.335	35.928	33.546

TABLE II. Optimisation results on i7-3610QM(execution time)

Achieved speedup of the fastest parallel code in relation to sequential achieved on i3 is satisfying and is about 1.94 for million elements. While achieved acceleration on i7 is about 3.98 for same number of elements. Regarding to the acceleration of parallel code, which is achieved by removing barriers (reduction of time for synchronization), code is faster for about 16 seconds for one million elements and achieved acceleration is 1.17 on i3, while on i7 code is faster for about 2 seconds and achieved acceleration is 1.07. The reason for this is presence of parts of the sequential algorithm that cannot be parallelized (like inside for loop and sum of partial results). It should be noted that for small inputs (numbers smaller than 100) sequential code is slightly faster than parallel version of code. This happens due to the additional overhead that occurs because of job distribution and synchronization between threads.

Speedup ratio, S, and parallel efficiency, E, may be used to provide an estimate for how well a code sped up if it was parallelized. For example, if f = 0.1 the speedup bound above predicts a 10 fold speedup in the limit. On the other hand, a code that is 50% parallelizable will at best see a factor of 2 speedup.[4]

Efficiency E in this case for i3 processor (N=4 threads) is

$$E = S/N = 1.94/4 = 0.485 \approx 0.5$$

and, for i7 processor (N=8 threads), efficiency is almost the same, as it's shown below:

$$E = S/N = 3.98/8 = 0.497 \approx 0.5$$

Program that scales linearly (S=N) has parallel efficiency 1. A task-parallel program is usually more efficient than a data-parallel program.[4]

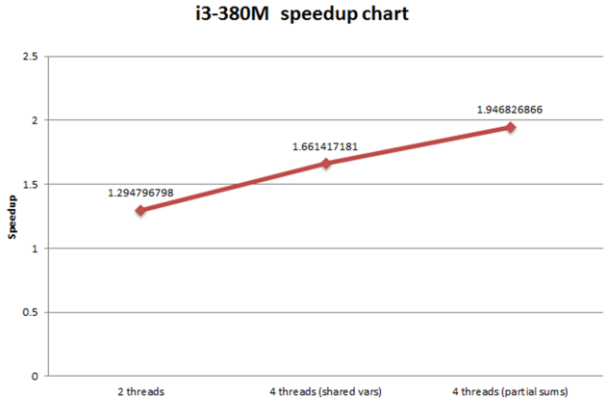

Fig. 3. Intel i3 processor speedup graph (considering n=1000000)

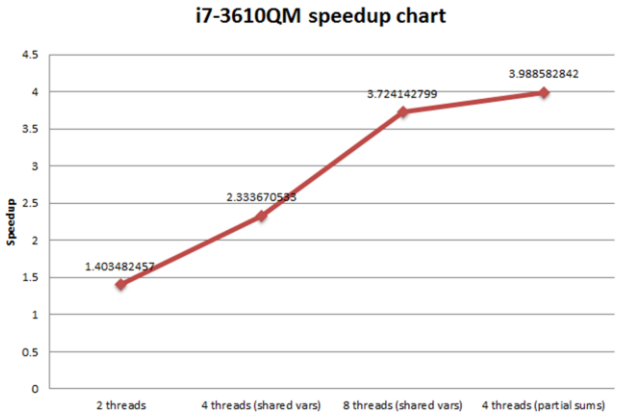

Fig. 4. Intel i7 processor speedup graph (considering n=1000000)

V. CONCLUSION

In this paper the focus is on comparison of sequential and parallel algorithm execution that counts number of primes from one to N (input value). Parallel execution is performed with different number of threads, with or without optimization. Conclusion is that it's possible to achieve speedup by parallelizing the algorithm. As it's shown in graphs above, the bigger the input number, the bigger is difference between sequential and parallel execution time. With big input values (bigger than 800000) and four threads, parallel execution time is almost two times smaller than sequential execution time. Additional time savings are made by eliminating the need for barrier (every thread got its variable) and avoiding the overhead caused by synchronization. This optimization is only visible for a very large input numbers. Great advantage of the chosen algorithm is independent task execution within a loop, which allows us to assign tasks to threads easier and also reduce a need for communication and synchronization between them.

VI. REFERENCES

[1] K. Mann, "The science of encryption: prime numbers and mod n arithmetic", Undated. [Online]. Available: https://math.berkeley.edu/~kpmann/encryption.pdf. [Accessed: 02- Nov-2015]

[2] People.sc.fsu.edu, "PRIME_OPENMP - Count Primes Using OpenMP", 2016. [Online]. Available: https://people.sc.fsu.edu/~jburkardt/c_src/prime_openmp/prime_openmp.html. [Accessed: 10-Nov-2015]

[3] C. Caldwell, "How many primes are there?",Primes.utm.edu, 2016. [Online]. Available: http://primes.utm.edu/howmany.html. [Accessed: 20-Dec-2015]

[4] Bu.edu, "Speedup Ratio and Parallel Efficiency : TechWeb : Boston University", 2016. [Online]. Available: http://www.bu.edu/tech/support/research/training-consulting/online-tutorials/matlab-pct/scalability/. [Accessed: 23- Jan- 2016].

A. ALGORITHM WITH OPENMP DIRECTIVES

```
int prime_number_par ( int n )
{
  int i,  j;
  int prime;
  int total = 0;
  int niz[4] = {0,0,0,0};

# pragma omp parallel \
  shared ( n ) \
  private ( i, j, prime )

# pragma omp for reduction ( + : total )
  for ( i = 2; i <= n; i++ )
  {
    prime = 1;

    for ( j = 2; j < i; j++ )
    {
      if ( i % j == 0 )
      {
        prime = 0;
        break;
      }
    }
    total = total + prime;
  }

  return total;
}
```

B. OPTIMIZED ALGORITHM

```
int prime_number_par ( int n )
{
  int i, j, k;
  int prime;
  int total = 0;
  int thread_num = omp_get_max_threads();
  int niz[thread_num];

  for ( k = 0; k <= thread_num; k++ ) { niz[k] = 0; }

# pragma omp parallel \
  shared ( n ) \
  private ( i, j, prime )

# pragma omp for
      for ( i = 2; i <= n; i++ ) {
        prime = 1;
for ( j = 2; j < i; j++ ) {
    if ( i % j == 0 )  {
      prime = 0;
      break;
     }
   }
    niz[omp_get_thread_num()] += prime;
  }

  for ( k = 0; k <= thread_num; k++ ) {
    total += niz[k];
  }
   return total;
}
```