Universidade do Porto

Faculdade de Engenharia

FEUP

Parallelization of the Sieve of Eratosthenes Algorithm

Mestrado Integrado de Engenharia Informática e Computação

Parallel Computing

João Carlos Eusébio Almeida - up201306301
João Gabriel Marques Costa - up201304197
Nuno Gonçalo Neto Silva - up201200642

May 17, 2017

Contents

1 Introduction

For the Parallel Computing course at the Faculty of Engineering at the University of Porto, the students were proposed the development of a study regarding the parallelization of the Sieve of Eratosthenes algorithm, using the "OpenMP" and "OpenMPI" libraries to aid said study.

In this article, the proposed algorithm as well as the several approaches used for the parallelization of the algorithm will be presented, alongside the measures of performance for these approaches obtained through the conduction of different experiments.

2 Problem Description

The main problem in analysis in this article refers to the calculation of prime numbers in the $[2, n] : \forall n \in \mathbb{N} \backslash \{1\}$ range [1]. A number of i value is considered a prime number if it is an integer greater than 1 and divisible only by two unique natural divisors: 1 and itself (i).

3 The Sieve of Eratosthenes Algorithm

One of the existing algorithms that provides a solution for the described problem is the Sieve of Eratosthenes algorithm. This algorithm, programmatically speaking, operates over a list which contains all the integers in the $[2, n]$ range, and consists in marking every multiple of a prime number which is inferior or equal to the $\sqrt{n}$ value. At the end of the algorithm's execution, every number without a marking in the list totals the existing prime numbers in the $[2, n]$ range.

The algorithm presents a temporal complexity of $\mathcal{O}(n \log \log n)$ and a spacial complexity of $\mathcal{O}(n)$ [2].

The pseudo-code for the algorithm as well as a visual representation of it are as follows [3]:

Algorithm 1 The Sieve of Eratosthenes

```
1:  // Initialization of the array of integers in the [2, n] range
2:  for i:- 2 to n do
3:      numbers[i − 1] ← true
4:  end for
5:  k ← 2
6:  while k² ⩽ n do
7:      // The multiples of k are marked in the [k², n] range
8:      i ← k²
9:      while i ⩽ n do
10:         numbers[i − 1] ← false
11:         i ← i + k
12:         // k is assigned the value of the next non-marked integer superior to k
13:         k ← k + 1
14:         while numbers[k − 1] = false do
15:             k ← k + 1
16:         end while
17:     end while
18: end while
```

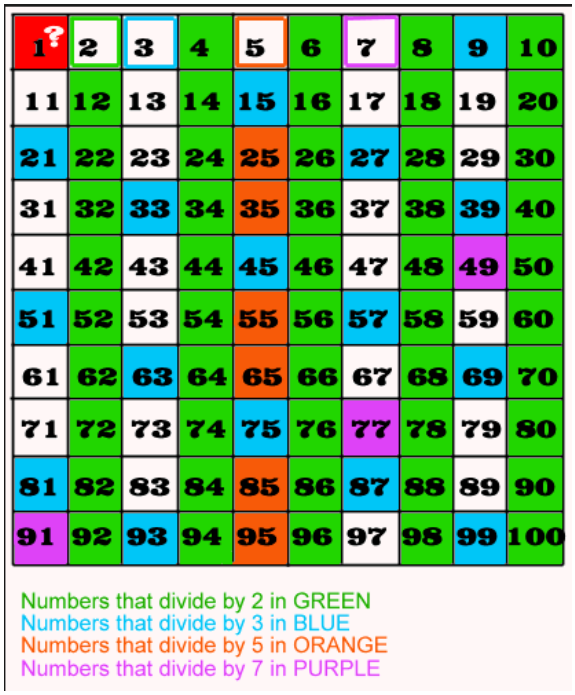

Figure 1: A visual representation of the final state of the Sieve of Eratosthenes algorithm in the $[2, 100]$ range. The numbers in white correspond to the prime numbers in the range.

4 Algorithm Implementation

In this section of the article, several implementations of the algorithm are described, all developed in the C++11 language.

4.1 Sequential Algorithm

The sequential algorithm version is a direct implementation of the Algorithm 1 pseudo-code presented in the previous section.

4.2 Parallelizing the Algorithm

In order to test the performance as well as the scalability of the algorithm, multiple versions of the algorithm implementing different memory models were developed, namely: A version implementing a shared memory model, a version implementing a distributed memory model and a version implementing a hybrid of the two previous memory models.

For the shared memory model implementation, different threads execute the code in parallel while sharing the same memory resources, while in the distributed memory model version, each of the processes has access to its own local memory resources, which are not shared with the other processes. In the hybrid memory model implementation, which uses principles of the two previously aforementioned memory models, the memory resources that are divided amongst the different processes, the way the data is communicated amongst these processes (e.g. if a process needs to access data stored in another process's memory resources) and how these are synchronized is left up to the programmer's own criteria.

4.2.1 Parallelization in Shared Memory

The version of the algorithm employing shared memory was implemented using the "OpenMP" library.

This version divides the iterations of the *for* cycle where the multiples of k are marked by a t number of threads. An excerpt of the source code of this implementation is as follows:

```
// Mark all the multiples of k between k*k and n
#pragma omp parallel for num_threads(n_threads)
for(long long i = k*k; i <=n; i += k) {
    nums[i-2] = false;
    }
```

Figure 2: Excerpt of the source code showing the *for* cycle where the parallelization of the shared memory version of the algorithm is done through the *omp parallel* instruction.

4.2.2 Parallelization in Distributed Memory

The version of the algorithm employing distributed memory was implemented using the "OpenMPI" library.

In this version, the array containing the integers in the $[2, k]$ range is divided by P processes, with each i process being attributed a data block where the first element is the element where the index is equal to:

$$first(i, n, P) = \frac{i * n}{P}$$

and the last element's index is:

$$last(i, n, P) = \frac{(i + 1) * n}{P}$$

Seeing as how, in order to mark the multiples of a k integer, only the $[k^2, n]$ range of integers is considered, each i process needs to evaluate whether the attributed block of integers needs any markings. If the i process determines that it's necessary to do any markings, the first multiple of the attributed block of integers needs to be calculated [4]. The pseudo-code for this calculation is as follows:

Algorithm 2 Algorithm that searches for the first multiple of k in the range of integers attributed to process i

```
1:  firstNum ← first(i, n, p) + 2
2:  lastNum ← last(i, n, p) + 2
3:  if k² ≤ firstNum then
4:      if (firstNum + 2) mod k = 0 then
5:          firstMult ← firstNum
6:      else
7:          firstMult ← firstNumb + (k − (firstNum mod k))
8:      end if
9:  else
10:     if k² ⩾ firstNum and k² ⩽ lastNum then
11:         firstMult ← k²
12:     else
13:         // Calculate/wait for next value of k
14:     end if
15: end if
```

The process $i = 0$ is the only one where, besides marking the multiples of the k integer value in its block of data, checks what will be the next value of k (by looking at the smallest non-marked number in its block of data) and broadcasts this value to the remaining processes.

4.2.3 Parallelization in Shared and Distributed Memory

The version of the algorithm employing hybrid memory was implemented with the use of both the "OpenMP" and "OpenMPI" library. This version is based on the implementation of the shared memory model version of the algorithm, with the use of distributed memory being the key difference, through the parallelization of the array containing the integers in the $[2, k]$ range in the same manner as on the excerpt of source code shown in Figure 2.

5 Experiences and Results Analysis

5.1 Experiences Description and Evaluation Methodology

In order to properly evaluate the different implemented versions of the Sieve of Eratosthenes algorithm described in the previous section, several experiments were conducted, with all the implemented versions being tested for $n = 2^i$, where the range of i corresponding to the integer range of $[25, 32]$. The implementation using shared memory was, additionally, tested using 2 to 8 threads while the distributed memory version was tested using 2 to 16 processes, using 4 computers connected in a computer network, with each computer running, at most, 4 processes (seeing as how they each had a quad-core processor).

In the case of the hybrid memory model implementation, 4 computers were used to conduct the experiments in the following combinations:

- 4 processes (1 for each computer), each using 2 to 8 threads;
- 8 processes (2 for each computer), each using 2 to 4 threads;
- 12 processes (3 for each computer), each using 2 threads;
- 16 processes (4 for each computer), each using 2 threads;

These combinations look to maximize the number of threads executing in each computer, seeing as how each one of the CPU cores allows for the start of 2 threads simultaneously, while dividing resources amongst processes in a "fair" way.

In order to measure the performance of these implementations of the algorithm, both the *Speedup* as well the number of operations per second being executed (MOP/s) were used. The equations for these metrics are as follows:

$$Speedup(i, v) = \frac{Ti, seq}{Ti, ver}$$

$$MOP/s(i, v) = \frac{2^i * \log * \log * 2^i}{Ti, ver * 10^6}$$

where Ti, seq is the time it takes the execute the sequential version for $n = 2^i$ and Ti, ver is the time it takes to execute a version of the algorithm ver for $n = 2^i$.

These experiments were conducted in desktops using a CPU Intel Core™i7-4790 with a 3.60GHz frequency (supporting "Turbo Boost" technology up to 4.00GHz) and 4 cores. The CPU also boasts 4 L1 data caches of 32KB, 4 L2 data/instruction caches of 256KB (one per core) as well as an L3 data/instruction cache of 8MB [5].

5.2 Results Analysis

5.2.1 Sequential Version Implementation

i	Execution Time (s)
25	0,253
26	0,538
27	1,145
28	2,512
29	5,109
30	10,575
31	21,799
32	44,991

Table 1: Table with the execution times, in seconds, of the sequential version of the algorithm, where $n = 2^i$.

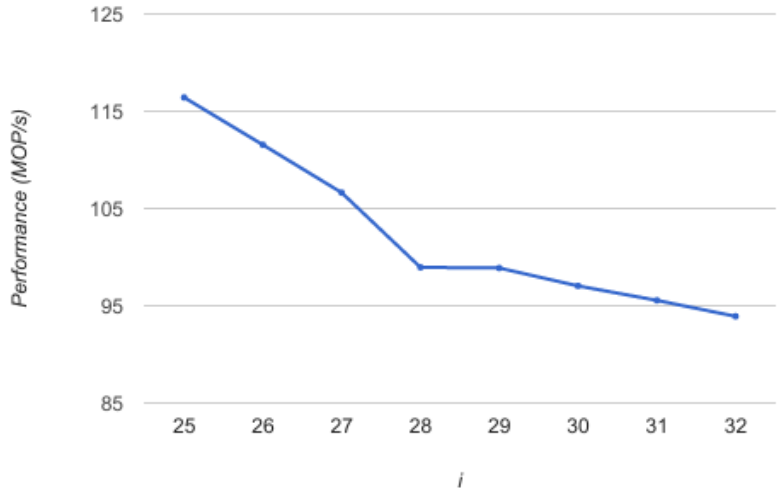

Figure 3: Performance of the sequential version of the algorithm, regarding the number of operations it's able to do per second.

Table 1 and the graphic in figure 3 show the different values for the execution times and performance of the sequential algorithm, respectively, for different sizes of the problem (range between $[25, 32]$ for i). It's possible to tell, from looking at the aforementioned graphic, that there's an overall drop off in performance with the size of the problem (fairly noticeable between $n = 2^27$ and $n = 2^28$), which could potentially be attributed to poor memory management in the version's implementation of the algorithm.

5.2.2 Shared Memory Version Parallelization

i	Execution Time (s)			
	$2T$	$4T$	$6T$	$8T$
25	0,196	0,195	0,202	0,204
26	0,438	0,428	0,456	0,460
27	0,948	0,943	0,961	1,006
28	1,995	2,008	2,024	2,043
29	4,379	4,348	4,378	4,410
30	9,128	8,392	9,226	9,440
31	17,902	17,618	17,836	17,910
32	36,437	37,241	36,952	36,604

Table 2: Table with the execution times, in seconds, of the shared memory version of the algorithm with a number of threads T in the range of $[2, 8]$, where $n = 2^i$.

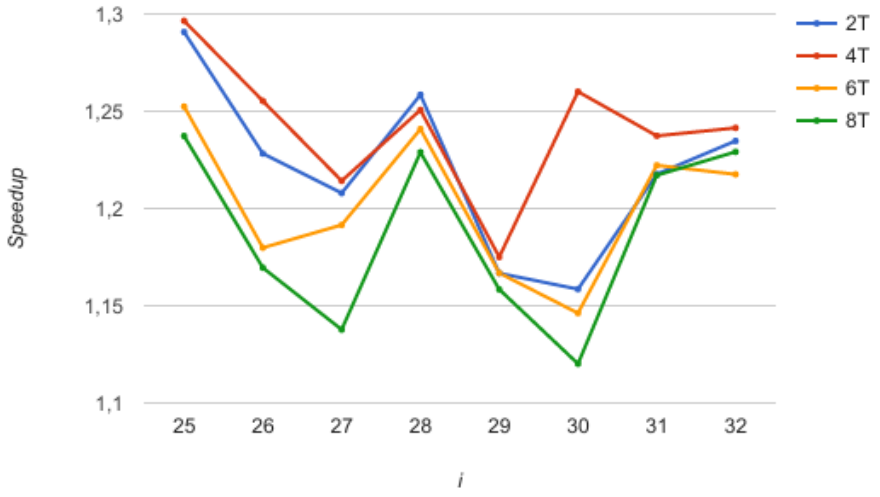

Figure 4: Performance of the shared memory implementation with a number of threads T in the range of $[2, 8]$.

Regarding the parallelized version of the algorithm employing shared memory, we can see through the graph in Figure 4 that there is no performance improvement in using more than 4 threads, as the execution times and speedups obtained past this number of threads is slightly worse than using even 2 threads, accentuating for smaller values of i.

It's also possible to observe that there is a slight increase in performance between the values of 27 and 28 for the values of i, followed by a decrease when i equals 29 and an increase when i equals 30 (this last increase, however, only happened when 4 threads were running). There is also a slight increase in performance when 2, 6 and 8 threads are running between the values of 30 and 31 for the value of i. When i equals 28 and 29, the performance of the shared memory version is fairly similar across the board.

The use of 4 threads provides the overall more robust performance, with a maximum Speedup of 1,29 (when i equals 25), stabilizing around 1,25 when near 32 for the value of i.

5.2.3 Distributed Memory Version Implementation

i	Execution Times (s)							
	$2P$	$4P$	$6P$	$8P$	$10P$	$12P$	$14P$	$16P$
25	0,302	0,218	0,138	0,098	0,073	0,061	0,052	0,051
26	0,634	0,559	0,334	0,262	0,204	0,165	0,198	0,155
27	1,381	1,069	0,719	0,642	0,442	0,409	0,318	0,308
28	2,715	2,12	1,379	1,142	0,941	0,821	0,712	0,647
29	5,689	4,341	3,055	2,349	1,817	1,610	1,396	1,246
30	11,439	9,146	6,045	4,908	4,036	3,370	2,959	2,541
31	23,415	18,532	12,383	10,019	7,729	8,018	5,681	5,281
32	48,740	38,036	27,361	20,957	15,911	13,620	12,182	10,659

Table 3: Table with the execution times, in seconds, of the distributed memory version of the algorithm with a number of processes P in the range of $[2, 16]$, where $n = 2^i$.

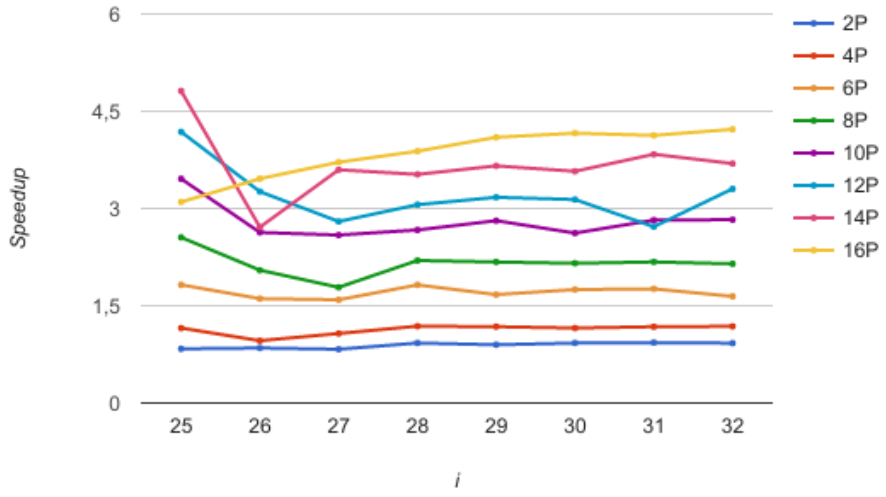

Figure 5: Performance of the distributed memory version of the algorithm with a number of processes P in the range of $[2, 16]$.

Looking at the results in Table 3 alongside the graph in Figure 5, we can see significant performance improvements over the versions in the two previous sections (sequential version and shared memory version) in the distributed memory implementation. In this implementation, generally speaking, the *Speedup* is greater for i the bigger the number of processes is running, achieving the most robust performance overall when the number of processes is equal to 16, stabilizing at around 4,2 as it gets closer to $n = 2^32$. In spite of this, the overall greatest *Speedup* achieved in this experiment was when $n = 2^25$ for 14 processes, at 4,8.

It's worth nothing that parallelizing the algorithm in a distributed memory fashion always obtains better results than in the shared memory implementation, even when the number of threads in the shared memory implementation is equal to the number of processes in the distributed memory implementation.

5.2.4 Shared and Distributed Memory Version Parallelization

i	Execution Times (s)							
	4P/2T	4P/4T	4P/6T	4P/8T	8P/2T	8P/4T	12P/2T	16P/2T
25	0,071	0,049	0,042	0,045	0,041	0,049	0,040	0,070
26	0,179	0,135	0,139	0,138	0,149	0,122	0,133	0,126
27	0,401	0,309	0,298	0,303	0,299	0,274	0,288	0,259
28	0,745	0,654	0,651	0,671	0,611	0,592	0,545	0,544
29	1,518	1,396	1,358	1,375	1,174	1,205	1,168	1,137
30	3,111	2,859	2,807	2,713	2,484	2,437	2,345	2,323
31	6,368	5,785	5,932	5,524	5,027	5,147	4,839	4,754
32	13,048	11,674	12,063	11,257	10,382	10,695	10,163	9,827

Table 4: Table with the execution times, in seconds, of the hybrid memory version of the algorithm with a number of processes P in the range of $[2, 16]$ and threads T in the range of $[2, 8]$, where $n = 2^i$.

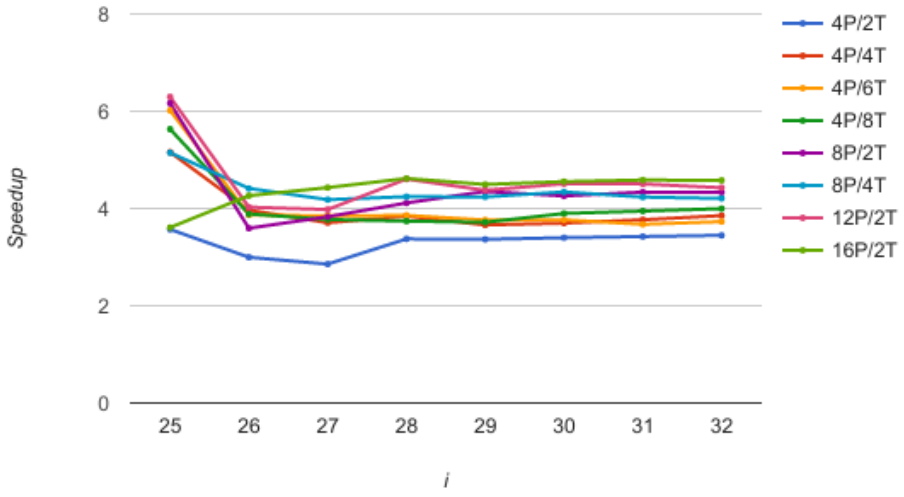

Figure 6: Performance of the hybrid memory version with a number of processes P in the range of $[2, 16]$ and a number of threads T in the range of $[2, 8]$.

Using a hybrid memory model (employing both shared memory and distributed memory), it's possible to observe through the results in Table 4 alongside the graph in Figure 6 that the results obtained are slightly better than those in the distributed memory version of the algorithm, although it's important to note that the *Speedup* for $n = 2^25$ is considerably higher in this version for most of the executions (with the exceptions being for the combination of 16 processes and 2 threads as well as 4 processes and 2 threads), achieving a maximum of 6,29 (for the combination of 12 processes and 2 threads). However, as i increases and nears $n = 2^32$, the *Speedup* tends do stabilize around 4 to 4,5.

6 Conclusions

In this article, 3 different approaches to the parallelization of the Sieve of Eratosthenes algorithm were analyzed, allowing for the further exploration of the concepts behind the shared memory, distributed memory and hybrid memory models using the "OpenMP" and "OpenMPI" libraries.

Analyzing the results shown in the previous section, it's important to note that the approaches based on a distributed memory model were able to improve performance up to 400% compared to a shared memory model approach. As such, dividing the problem of calculating prime numbers using this algorithm by different processes and processing units (having access to non-shared local resources) is the better approach, as opposed to parallelizing the problem with a single processing unit sharing local resources.

The approach that seems to combine both these aspects, however, using the "OpenMPI" library in order to divide the problem into different remote processes and parallelizing at a local level (using the "OpenMP" library) was the best performer, even if by a small margin. As such, using a hybrid memory model implementation seems to be the best way to maximize performance in this problem.

References

[1] Michael J. Quinn, *Parallel Programming in C with MPI and OpenMP*, International Edition, McGraw-Hill Professional, 2003.

[2] Wikipedia: Sieve of Eratosthenes, `https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes`, [Online; accessed 05-May-2017].

[3] Algorithmist: Prime Sieve of Eratosthenes, `http://www.algorithmist.com/index.php/Prime_Sieve_of_Eratosthenes`, [Online: accessed 05-May-2017].

[4] Wirian, D.J., *Parallel prime sieve: Finding prime numbers*, Institute of Information and Mathematical Sciences, Massey University at Albany, Auckland, New Zealand 1 (2009).

[5] CPU-World: Intel Core i7-4790 specifications, `http://www.cpu-world.com/CPUs/Core_i7/Intel-Core%20i7-4790.html`, [Online: accessed 05-May-2017].