Parallel Algorithms: The Sieve of Eratosthenes

Kleopatra Pirpinia

Abstract

In this report we will discuss a parallel implementation of the sieve of Eratosthenes within the framework of the BSP model, we will estimate its BSP cost and present the results of various numerical experiments.We will also add some minor modifications in order to check the Goldbach conjecture as well as to generate pairs of twin prime numbers.

Contents

1 Sieve of Eratosthenes

In mathematics, the sieve of Eratosthenes is a simple, ancient algorithm for finding all prime numbers up to any given number n. It does so by iteratively marking as composite (i.e. not prime) the multiples of each prime, starting with the multiples of 2.

1.1 The sequential algorithm

The sequential algorithm is the following:

```
Input: $n \in \mathbb{N}$
Output: A list with all the primes up to $n$.

  if $n$ is odd then
    Create the list of numbers $l = [2, 3, 5, 7, \ldots, n]$.
  else
    Create the list of numbers $l = [2, 3, 5, 7, \ldots, n-1]$.
  end if
  for $p = 3, \ldots, \sqrt{n}$ do
    Remove all the multiples of $p$ from $l$
    $p$ is the next element of $l$
  end for
  return $l$
```

Firstly, note that we will consider only the odd numbers (and 2 of course, which is a prime) up to n, which reduces the number of operations we have to perform.

Furthermore, let us notice that it is sufficient to start crossing out the multiples of p at p^2,since all the smaller multiples will have already been crossed out by that time. Therefore, when p^2 becomes larger than n, the algorithm must stop. In other words, it suffices that p goes only up to $\sqrt{n}$.

1.1.1 The implementation

We implement the above algorithm creating firstly the array l of length $\frac{n}{2}$.

In order to do this, we need to find a convenient correspondence between the elements of the array and their indices.

In our case it is straightforward that a nice correspondence is $l[i] = 2i + 1$, with $i \geq 1$. The first element of the array (in the position 0) is set to be always 2.

Furthermore, we want to start crossing out the multiples of every element, and we do this by considering $l[i]$ and doing a cycle with step $2i + 1$.

For example, for $l[1] = 3$, we start crossing out its multiples at $l[i+2i+1] = l[4] = 9$ and so on. Doing this we get straight to the multiples without controlling every number, and we cross them out instantly setting them equal to 0.

It is then easy to go through l and count the number of primes, and create a new list l' without the 0's.

1.1.2 The cost

It is given that the probability of an arbitrary integer $x \geq 2$ to be prime is $\frac{1}{\ln x}$.
We also know that the number of multiples that we cross out for each number is

$$\frac{n}{p}$$

Combining this with the fact that

$$\sum_{p \leq n} \frac{1}{p}$$

goes asymptotically to $\ln \ln n$ we get that the cost of our algorithm is $O(n \ln \ln n)$. It does not matter asymptotically if we omit the even numbers from the initial array.

1.2 The parallel algorithm

For the parallel algorithm, the fact that we only need the primes up to $\sqrt{n}$ in order to cross out all the non primes up to n is fundamental. The idea is that if all the processors have the list of primes up to $\sqrt{n}$, then we can distribute the numbers from $\sqrt{n}$ to n and they'll have all the information they need in order to cross out the multiples in parallel.

1.2.1 The implementation

The parallel algorithm is given below:

Input: $n \in \mathbb{N}$
Output: The total number of primes below n

```
(0)  Compute sequentially the primes up to $\sqrt{n}$ and store it in the array
     initialPrimes.

     Create the array local of size $b := \left\lceil \frac{n - \sqrt{n}}{2p} \right\rceil$ using block distribu-
     tion, starting from number $\sqrt{n} + 2sb$.

     for $z \in$ initialPrimes do
       $i :=$ first index such that local[i] $= kz$ for some $k \in \mathbb{N}$
       while $i < b$ do
         flag local[i] as non-prime
         $i = i + z$
       end while
     end for

     $count_s := 0$
     for $x \in local$ do
       if $x$ is not flagged then
         $count_s = count_s + 1$
       end if
     end for
(1)  for $t := 0, \ldots, p - 1$ do
       Put $count_s$ in $P(t)$
     end for
(2)  $finalCount := 0$
     for $t = 0, \ldots, p - 1$ do
       $finalCount = finalCount + count_s$.
     end for
```

As mentioned before, we begin the implementation with a sequential part, using a function $seq(n)$ (which uses the algorithm described in the previous Section) that computes the number of primes up to n. So, we make all p processors find the primes up to $\sqrt{n}$ and place them in an array which we call *initialPrimes.*

Then the parallel part starts: the remaining $n - \sqrt{n}$ numbers are distributed among the processors in blocks, thus there are $\frac{n-\sqrt{n}}{2p}$ numbers on every processor, rounded up (remember that we consider only the odd numbers between 2 and n, thus the 2 of the denominator).

The block distribution was chosen over the cyclic since it permits us to locate easily the multiples: if k is a multiple of z then also $k+z$ is; moreover, as we will see later on, it allows us to find nicely the pairs of twin primes - almost all primes that are twins are adjacent to each other!

Now, every processor starts crossing out the multiples of the initially given primes. However this time we choose to not set them equal to zero, but flag them in a different way instead - we just change the sign and set them negative. This way the value of each element is not lost, and it can still be checked if the flagged number is a multiple of another prime, in order to start "jumping" from one multiple to the next, thus saving some extra operations.

After that every processor counts the number of non-flagged entries, and broadcasts this number to all the others; finally all of them then compute the final number of primes.

1.2.2 The BSP cost

Based on the remarks we made above, we can give an estimation of the BSP cost. For the sequential part we have

$$\lfloor\sqrt{n}\rfloor \ln\ln\lfloor\sqrt{n}\rfloor$$

For the flagging of multiples in parallel we have

$$\frac{n}{p}\ln\ln n + l$$

Then we have a $(p-1)$-relation in the communication superstep, so the cost is

$$(p-1)g + l$$

Finally we have p additions in order to compute the final sum. Therefore the overall cost is

$$T = \lfloor\sqrt{n}\rfloor \ln\ln\lfloor\sqrt{n}\rfloor + \frac{n}{p}\ln\ln n + (p-1)g + p + 3l$$

2 Twin primes

A *twin prime* is a prime number that differs from another prime by two.

```
Input: $n \in \mathbb{N}$ and the arrays as they are at the end of the previous algorithm
Output: The number of twin primes below $n$

(0)   $nLocalTwins_s := 0$

      if $s = 0$ then
         for $i = 0, \ldots, size($initialPrimes$) - 1$ do
            if initialPrimes[i+1]$=$ initialPrimes[i]$+2$ then
               $nLocalTwins_s = nLocalTwins_s + 1.$
            end if
         end for
         if local[0] is prime  $\wedge$  initialPrimes[-1] is prime then
            $nLocalTwins_s = nLocalTwins_s + 1.$
         end if
      end if

      for $i = 0, \ldots, b - 1$ do
         if local[i] is prime  $\wedge$  local[i+1] is prime then
            $nLocalTwins_s = nLocalTwins_s + 1.$
         end if
      end for

      $flag := 0$
      if local[b-1] is prime then
         $flag = 1$
      end if
(1)   if $s < p - 1$ then
         Put $flag$ in $P(s + 1)$ into the variable $prevFlag$
      end if
(2)   if $prevFlag = 1$  $\wedge$  local[0] is prime then
         $nLocalTwins_s = nLocalTwins_s + 1.$
      end if
(3)   for $t := 0, \ldots, p - 1$ do
         Put $nLocalTwins_s$ in $P(t)$
      end for
(4)   $totalTwins := 0$
      for $t = 0, \ldots, p - 1$ do
         $totalTwins = totalTwins + nLocalTwins_s.$
      end for
```

The above is the algorithm used to compute the number of twin primes from

2 to n, after having used the parallel sieve described in the previous section. Therefore we refer to the same arrays as before and these operations are intended to be executed right after the previous ones, still in the SPMD part.

Note that because we are only considering arrays with odd numbers and we are using the block distribution, twin primes are adjacent to each other. The only case in which this is not true is when this pair is split among two different processors.

The algorithm, substantially, works as follows:

Processor 0 initially does the following operations:

- it looks for twin primes in the array `initialPrimes` which contains exclusively the primes up to $\sqrt{n}$
- it checks whether a pair of twin primes was split between `initialPrimes` and its local list (note that this processor gets the first chunk of all the numbers to sieve).

Then, every processor does the following:

- it looks for twin prims in its local list
- it checks whether the last element of the local list is prime, if this is the case the flag is set to 1
- it sends to the next one the information about the last element of the local list
- it checks whether there was a pair of twin primes split between him and the previous one
- it broadcasts the number of twin primes found to all the other processors
- computes the final number of twin primes

3 Goldbach's Conjecture

Goldbach's conjecture, one of the most famous problems of number theory, states that any given even integer greater than 2 can be expressed as a sum of 2 primes.

The following is the algorithm used to check its validity up to a given number n.

```
Input: $n \in \mathbb{N}$ and the arrays as they are at the end of the previous algorithm
Output: The number of non-Goldbach primes below $n$

(0)  if $s = 0$ then
       localPrimes$_s$ := (initialPrimes, primes in local)
     else
       localPrimes$_s$ := local without the non-primes
     end if

(1)  for $t := 0, \dots, p-1$ do
       Put localPrimes$_s$ in $P(t)$.
     end for

(2)  create the list allPrimes by ordered concatenation of localPrimes$_s$.
     if $n$ is even then
       $largestEven := n$
     else
       $largestEven := n - 1$
     end if

     $r := \left\lceil \frac{largestEven - 4}{2p} \right\rceil$

     Create the array listEven of size $r$ using block distribution, starting
     from number $4 + 2rs$

     for $i = 0, \dots, r-1$ do
       for $j = 0, \dots, finalCount - 1$ do
         for $k = j, \dots, finalCount - 1$ do
           if allPrimes[j]+allPrimes[k]=listEven[i] then
             flag listEven[i] as satisfied
             exit from the loops on $j$ and $k$ and continue the one on $i$
           end if
         end for
       end for
     end for

     $numberNonGoldbach$ := amount of number of listEven not satisfied
```

Remark.

Ideally, the algorithm should continue with the communication superstep in which each processor lets the other know its value for *numberNonGoldbach* and then they should compute the total amount, as done similarly for the amount of primes and pairs of twins; however, we cheated a little bit because we already know that the Goldbach Conjecture has been verified until 4×10^{18} (so we know that for any number that we might test it would return 0) and because the algorithm just given is already quite expensive, we decided to remove the "unnecessary" work.

4 Benchmarking

Now, we wanted to measure the performance of our computers in terms of the computing rate r, synchronization cost l and communication parameter g, running the program `bspbench`.

We performed this benchmark on the Dutch supercomputer Huygens, a clustered SMP with a total of 3456 cores, and on a MacBook Air with 2 Intel i5 cores @1.7 GHz.

Machine	Processors	r (Mflop/s)	g	l
Huygens	1	195.693	51.7	550.6
	2	195.715	55.5	1945.7
	4	195.321	57.8	4051.5
	8	195.480	57.4	7791.0
	16	195.321	60.0	15552.7
	32	195.210	61.2	32539.8
	64	195.487	63.2	100744.5
	128	195.049	116.0	213201.8
	256	195.487		
Macbook Air	1	251.080	50.3	663.2
	2	253.730	75.2	4409.5
	4	285.837	232.2	15470.0

Table 1: Value for r, g and l with the benchmark.

Remarks. We notice that g is always almost the same except in the case when we use 128 processors. This can be explained looking at the structure of Huygens: the fact that there are nodes of 32 cores, makes the communication inside it very fast, and even when there in one adjacent to it it is still ok (this is the case of 64 cores) but when we need more of them, the communication can become slower, since they might be not so near.

Results for `bsp_put`

Here we present some (for sake of brevity) results we obtained after running **`bspbench`** on Huygens and on a Macbook Air:

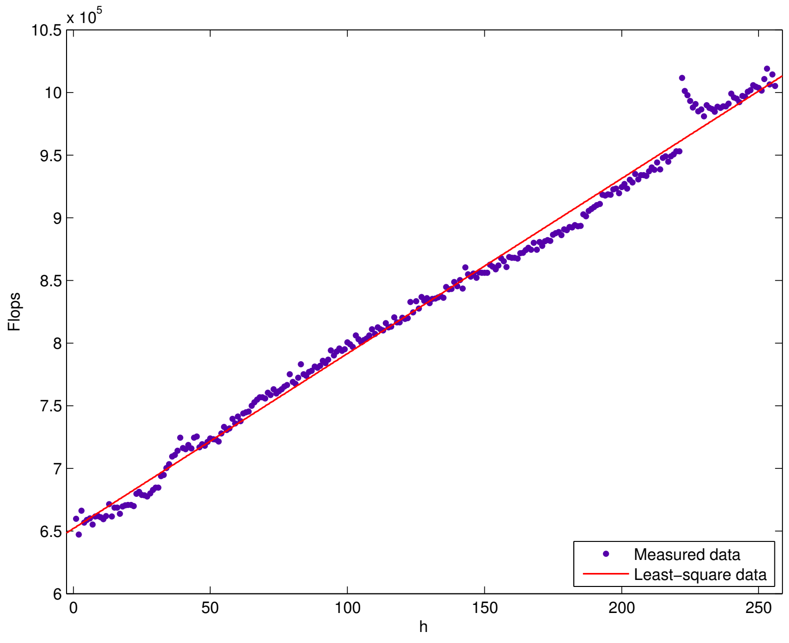

Figure 1: Huygens: $p = 32$

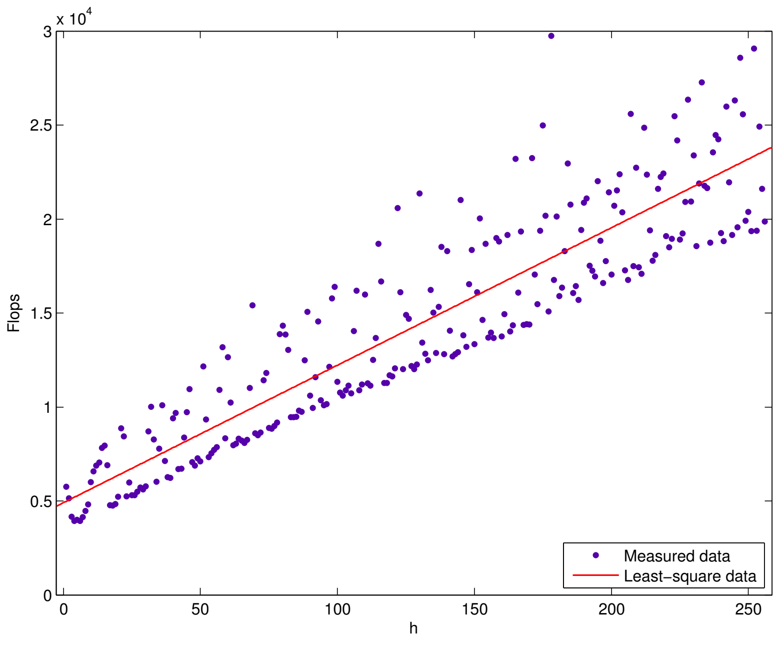

Figure 2: MacBook Air: $p = 2$

Results for `bsp_get`

We then modified our program to `bsp_get` and performed again some measurements:

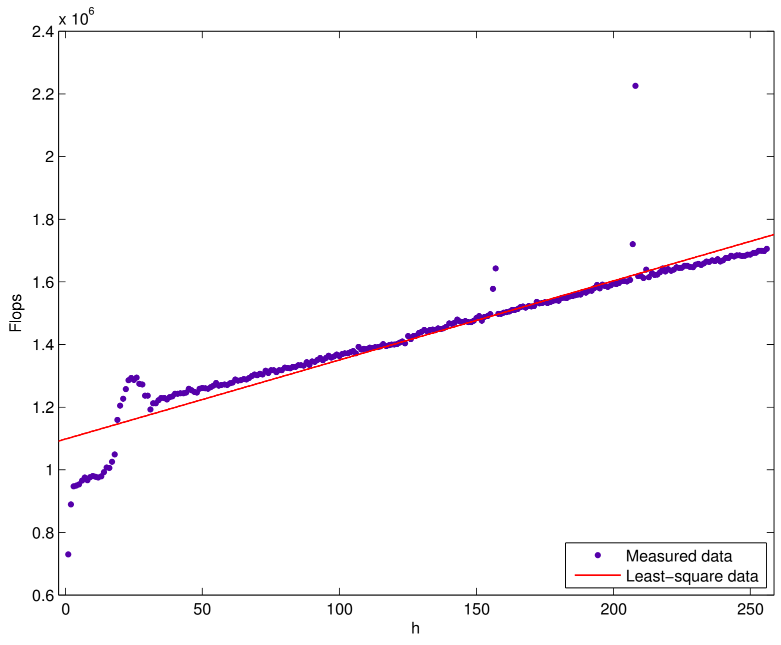

Figure 3: Huygens: $p = 32$ with `bsp_get`

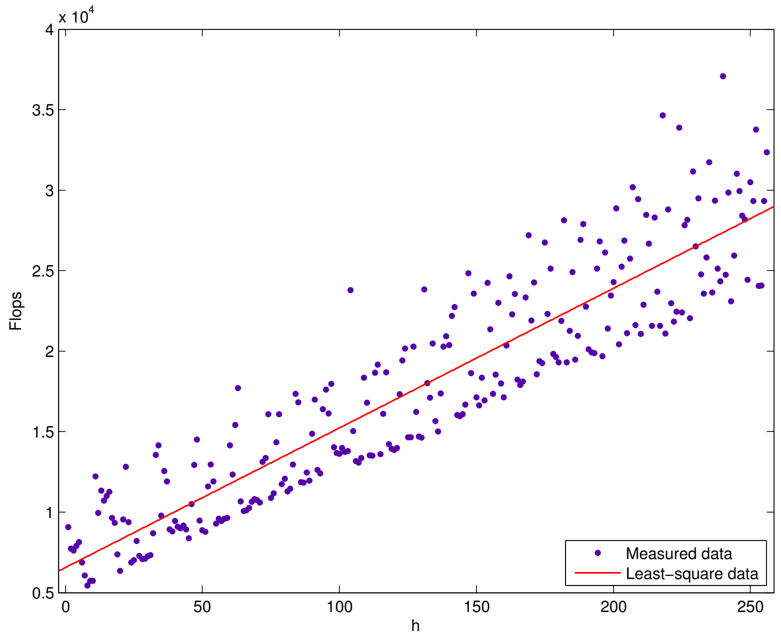

Figure 4: MacBook Air: $p = 2$ with `bsp_get`

5 Time measurements

In order to test the real effectiveness of our program, we decided to test it and compute the prime numbers up to 10^8 and 10^9.

5.1 Huygens

In the following Tables we give the results obtained from Huygens for $n = 10^8$:

p	t (sec)
1	7.6615
2	3.4998
4	1.3157
8	0.6531
16	0.2228
32	0.1258
64	0.0954
128	0.0876
256	0.1017

Table 2: Time measurements for $n = 10^8$ on Huygens

And the following are the ones for $n = 10^9$:

p	t (sec)
1	93.5070
2	46.4505
4	22.5715
8	11.6010
16	5.9265
32	3.0558
64	1.5035
128	0.8040
256	0.6616

Table 3: Time measurements for $n = 10^9$ on Huygens

In order to understand whether our parallelization was effective, we computed the speedup plots, which are shown below:

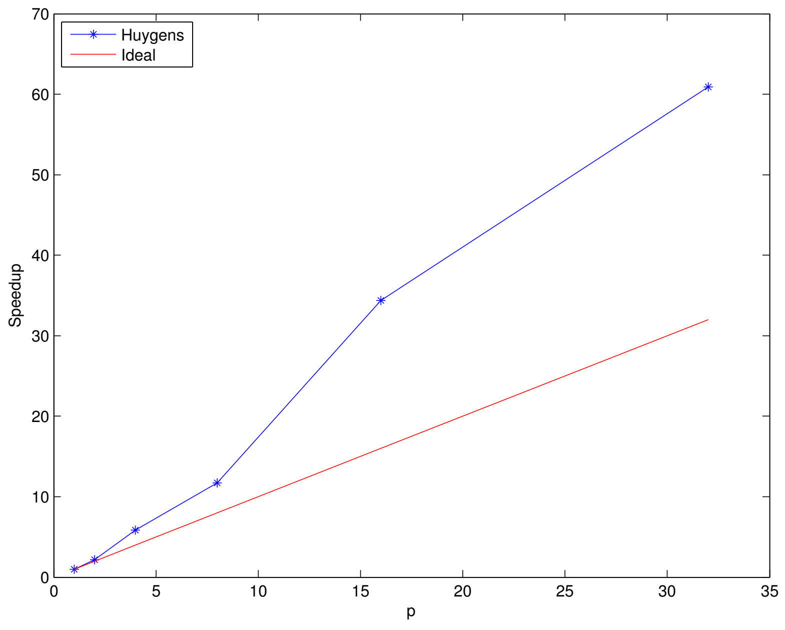

Figure 5: Speedup plot for Huygens with $n = 10^8$

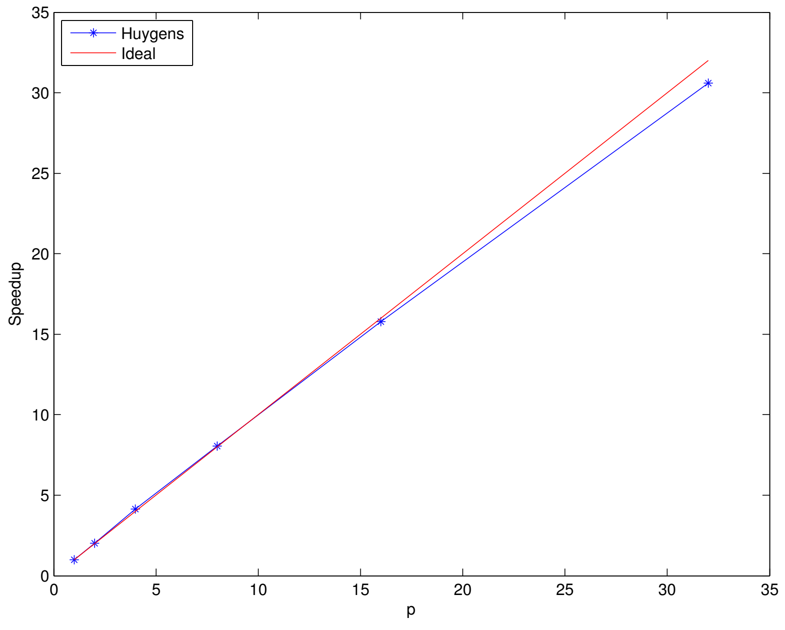

Figure 6: Speedup plot for Huygens with $n = 10^9$

The results are somewhat surprising, especially for the case 10^8: from the Figure it is clear that in that case parallelization was well worth the effort! We did not expect the program to improve so much, because parallelization often involves additional costs (communication, synchronization) that slightly worsen the performance. A possible explanation for this is that our initial result (with $p = 1$) was particularly bad, which made the other computation look very good. For the case 10^9 things look much more normal even if it is again a surprisingly good parallelization.

In order to see whether this is the case, we also computed the time with the purely sequential algorithm (described at the beginning of this report) as the $p = 1$ case, and we obtained the following speedup plots.

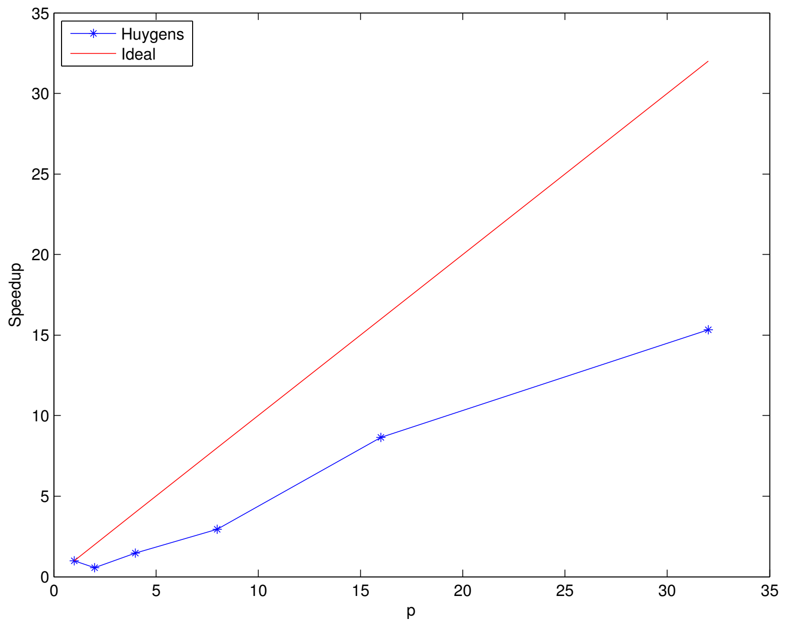

Figure 7: Speedup plot for Huygens with $n = 10^8$

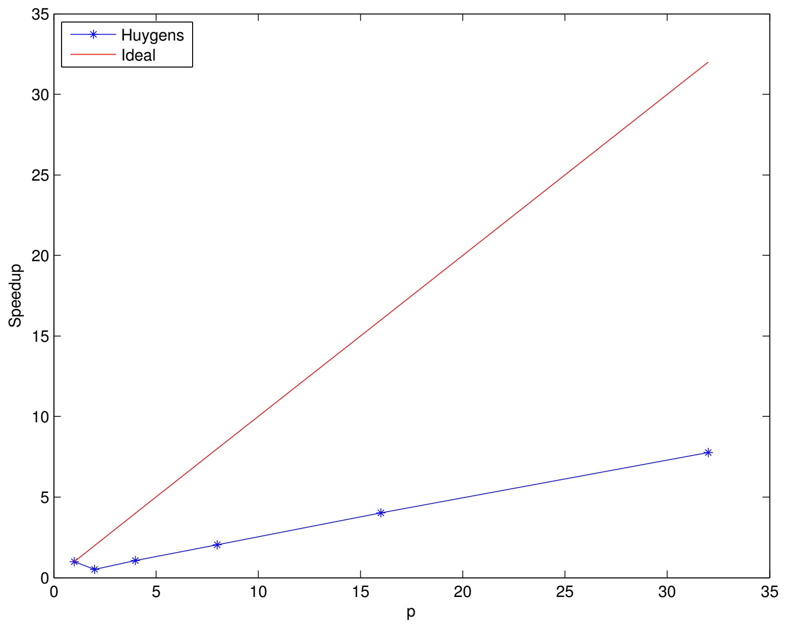

Figure 8: Speedup plot for Huygens with $n = 10^9$

So, it turns out that our sequential algorithm is much faster than the parallel version with $p = 1$, and that deeply affects the perspective on the subsequent measurements.

5.2 MacBook Air

In order to gain more insight on the parallelization even on a general purpose computer, we run our parallel sieve also on a MacBook Air.

In the following Tables we outline the results using both MulticoreBSP (denoted by m) and BSPonMPI (denoted by b) implementations of the BSP model.

n	p	time m	time b
10^8	1	1.150	1.3800
	2	0.9921	0.9789
	4	0.8375	0.8534
10^9	1	16.277	17.1801
	2	12.4026	11.8116
	4	11.0896	11.1477

Table 4: Time measurements for MulticoreBSP and BSPonMPI for both 10^8 and 10^9.

The speedup plots are shown below:

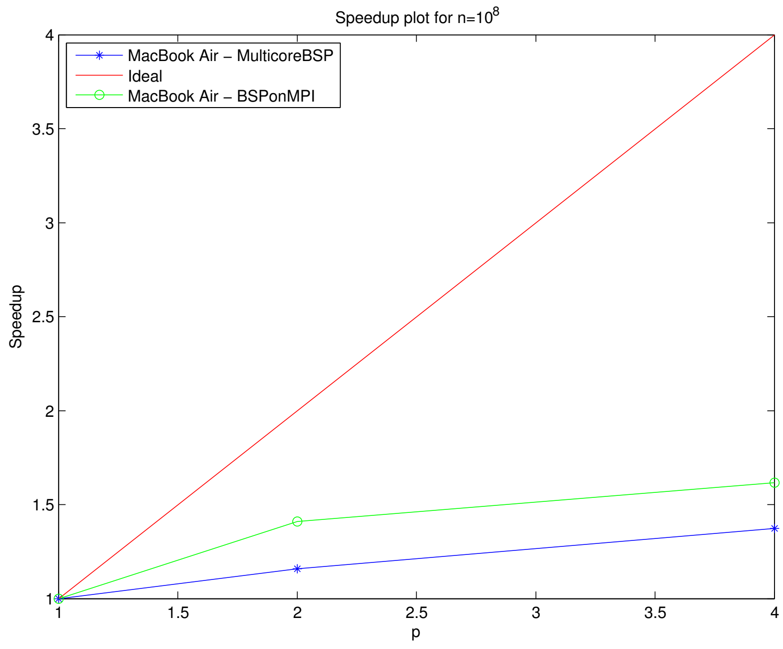

Figure 9: Speedup plot for MacBook Air with $n = 10^8$

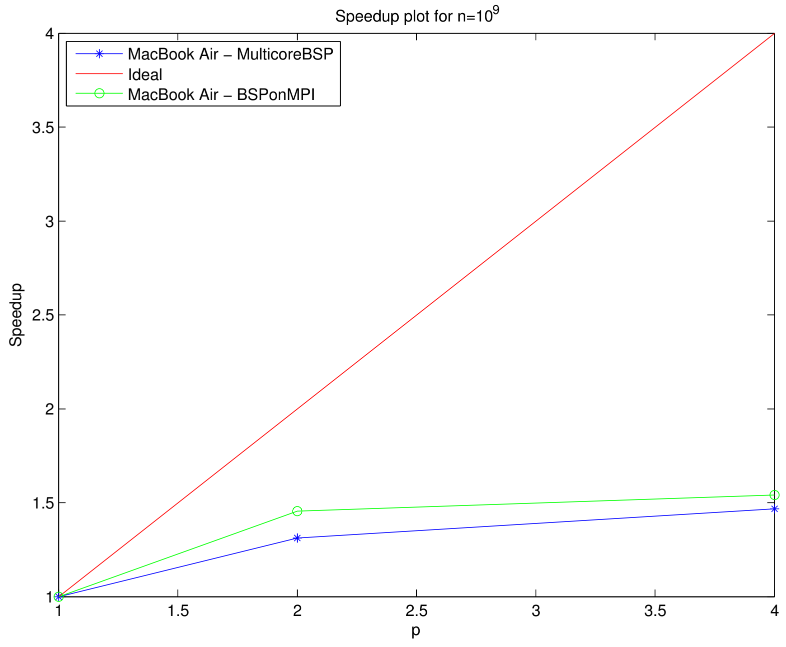

Figure 10: Speedup plot for MacBook Air with $n = 10^9$

We can see that BSPonMPI and MulticoreBSP have a very similar behavior, but the latter has the advantage of easier debugging, since we can use the tools normally used for C programming language.

6 Conclusions

The whole purpose of this report was to investigate whether the parallelization of the Eratosthenes' Sieve is convenient: from the results we were able to see a significant improvement in the speed of computation of all primes up to the desired n, which suggests that it is indeed a very useful approach.

Moreover, we gained more insight on the properties of a supercomputer, being able to look for primes with even 256 processors, and also a new perspective on common laptops, where one might forget that several cores can give a signficant improvement (provided that the program scales well!).