Parallel implementation of the Sieve of Eratosthenes

Torben Hansen, F120116

Utrecht University - Institute of Mathematics

This paper concludes the first project in the Mastermath course *Parallel algorithms* and concerns the Sieve of Eratosthenes. The basic sequential version of the Sieve of Eratosthenes is simple to implement and very efficient; complexity is $O(n \ln \ln n)$ with n being the input number. With various improvements the hidden constant factor can be made pretty small. Parallelising the Sieve of Eratosthenes adds intricacy to the code but may amount into a more efficient program. In this paper we will discuss and implement a parallel versions using the BSP scheme. The Parallel version presented have BSP cost

$$O\left(\frac{n \ln \ln n}{2p} + p + g(p-1)\frac{\sqrt{n}}{\ln(n)} + 4l\frac{\sqrt{n}}{\ln(n)}\right).$$

Experiments in section 3 show that for small bit size input there is no significant improvements to gain by increasing the number of processors. A detailed study of the case $p = 8$ shows that the expected execution times is almost at least 2.5 times bigger than the actual running time at Huygens for numbers that are smaller than 31 bit.

Throughout this paper we will use *SoE* as an abbreviation for *Sieve of Eratosthenes* and when we say that some data is located at a processor i we really mean that the data is located on the memory that processor i has direct access too.

Enjoy.

Who would have imagined that something as straightforward
as the natural numbers, could give birth to anything
so baffling as the prime numbers?
— Ian Stewart (1945–)

Contents

1 Sequential Sieve of Eratosthenes

We begin by introducing the sequential version of the SoE and analyse the complexity in the RAM model. We also present an implementation of the sequential SoE written in C.

1.1 Description sequential version

As mentioned the SoE is very simple in its basic form and works as following: Start with the integers from 2 to n. The number 2 is a prime; cross out all *larger* multiples. The next smallest remaining number is 3, which is a prime; cross out all multiples of 3. The next smallest remaining number is 5, which is prime; cross out all multiples... Etc. Algorithm 1.1.1 displays the algorithm in pseudo code.

Algorithm 1.1.1 Sieve of Eratosthenes

```
Input: Integer $n$ with $n \geq 2$.
Output: List $B$ with the primes from 2 to $n$.
  Let $A$ be a bit array with length $n$. Let $B$ be a list. Assume for simplicity that
  the index in $A$ starts at 1 (this is the only place we assume this. The assumption
  is easy fixed in practice!).
  Initialize all elements in $A$ to 1 except $A[1]$ which is set to 0.
  for $i = 2 \to \lfloor\sqrt{n}\rfloor$ do
    if $A[i] == 1$ then
      $B.add(i)$
      $p = i^2$
      while $p \leq n$ do
        $A[p] = 0$
        $p = p + i$
      end while
    end if
  end for
  for $i = \lfloor\sqrt{n}\rfloor + 1 \to n$ do
    if $A[i] == 1$ then
      $B.add(i)$.
    end if
  end for
```

Below we add additional points regarding algorithm 1.1.1.

- If one only want to count the number of primes in the interval $[2, n]$ one could do without the list B in the algorithm.

- We may stop sieving when we reach $\lfloor\sqrt{n}\rfloor$. This is an easy lemma: Assume not all composite numbers has been crossed out. Take one, say x, with $x \le n$. x must have at least two different primes, say p and q, that must satisfy $p, q > \sqrt{n}$ otherwise x would have been crossed out. But this implies there exists $x' \ge 1$ such that $x > x'\sqrt{n}\sqrt{n} = x'n$ - contradiction.

- We may start sieving with a new prime, say p, from the square of that prime because all numbers $2p, 3p, \ldots, \delta p$ with δ maximal integer such that $\delta p < p^2$, has been crossed out by other primes smaller than p (because $\delta < p$).

- An easy way to improve algorithm 1.1.1 is to only represent odd numbers. This can be done because the prime 2 will cross out all even numbers in the first iteration. This would save a factor of 2. We do this in our C implementation.

In appendix A there is C implementation of the sequential SoE. Essentially this implementation does not exactly output the same as algorithm 1.1.1 since dynamical array allocations is not a part of the standard library in C we instead represent 2 and all odd integers from 3 to n. Output is the number of primes below n. With this representation and the number of primes below n it is easy to allocate an array with the size of the number of primes and run through the array containing the numbers from 3 to n and pick out the primes. This wont change the asymptotic time of algorithm 1.1.1.

1.2 Complexity

The analysis of algorithm 1.1.1 is not that hard. We will use the following result, which proof has been omitted.

Proposition 1.2.1. *For the reciprocal prime sum we have* $\sum_{p\le n}\frac{1}{p} = \ln\ln n + O(1)$.

A proof strategy for proposition 1.2.1 could be first to show that $\sum_{p\le n}\frac{lnp}{p} = \ln n + O(1)$ and then apply partial summation using the function $f(t) = \frac{1}{\ln t}$ with coefficient $a_n = \frac{\ln q}{q}$ if n equals the a prime q and $a_n = 0$ otherwise (i.e. when n is not prime).

Algorithm 1.1.1 starts with an initialization which takes n steps to perform and the algorithm ends with a pass through the "rest" of array A. The last pass takes $n - \lfloor\sqrt{n}\rfloor$ because we are only doing constant work in each step. What is left, is to figure out the complexity of the main for-loop. For each i we proceed to the while-loop if and only if i is a prime. If i is prime we go trough the while-loop at most $\frac{n}{i}$ times. The amortized time for the main-loop is therefore

$$\begin{aligned} O\left(n-\pi(n)+\sum_{p\le\lfloor\sqrt{n}\rfloor}\frac{n}{p}\right) &= O\left(n-\pi(n)+n\sum_{p\le\lfloor\sqrt{n}\rfloor}\frac{1}{p}\right) \\ &= O\left(n-\pi(n)+n\cdot\ln\ln\lfloor\sqrt{n}\rfloor\right) \\ &= O\left(n\cdot\ln\ln n\right). \end{aligned}$$

using proposition 1.2.1. Letting T_{SoE} denote the running time for algorithm 1.1.1 we have

$$T_{SoE} = O\left(n + n\cdot\ln\ln n + n - \lfloor\sqrt{n}\rfloor\right) = O\left(n\cdot\ln\ln n\right).$$

Even though $\ln\ln n$ goes to infinity it does so extremely slow! Actually $\ln\ln n < 10$ for all $n < e^{e^{10}} \approx e^{2206}$ a huge number! Also notice that the amortized work per integer between 1 and n, is astonishing $O(\ln\ln n)$ - practically constant.

The above argument is basically an estimate of how many cross outs we are doing during a run with the algorithm. Hence the estimate can be thought of as being an upper bound of how many cross outs we need doing a run and in this line of thinking the complexity relies on how good we are at avoiding multiple cross outs of numbers. Looking at the estimates above and the (rather) basic C implementation one see that a good implementation should do at most $\frac{1}{2}n\cdot\ln\ln\lfloor\sqrt{n}\rfloor$ cross outs.

2 Parallel Sieve of Eratosthenes

In this chapter we develop a parallel version of the SoE using the *Bulk Synchronous Parallel* (BSP) model. For the implementation we are using BSPlib with C. We assume that the result (either number of primes below some limit or the primes themselves below some limit) is needed by every processor.

2.1 Distribution and communication

The first thing we must settle on is how to distribute the sieving array over the processors. When doing so we must take into account which distribution entails the least amount of communication and the best work and data load between all processors. Also we want an efficient way of initializing the distribution. We consider two different ways of distributing the sieving array; cyclic and block distribution. To describe both, say we want to partition the array $A[0,1,\dots,n-1]$ with $A[i]=i+1$ for $0 \le i < n$, using p processors $P(0), P(1), \dots, P(p-1)$.

Cyclic This distribution is defined by the map $a_i \mapsto P(i \mod p)$ for $0 \le i < n$. That is, processor s contain indexes $s + jp$ for $j = 0, 1, 2, \dots, \lfloor \frac{n-s}{p} \rfloor$. Below is an example with $n = 20$ and $p = 3$.

1	4	7	10	13	16	19

2	5	8	11	14	17	20

3	6	9	12	15	18

where the array with $1, 4, 7, 10, 13, 16, 19$ is contained in processor 0 etc. The benefit of this distribution is that we get a perfect data load over all processors with almost no effort and it is easy to determine which processor is the owner of a specific element. The backside of this distribution is that the workload may be horrible. In the above example one may notice that processor 2 contains only 1 prime while processor 1 contain 4 primes. This work load bias can be much bigger when n is a bigger number. Another problem is the need for a larger communication when the next sieving prime is to be determined. Every processor needs to send his candidate of a sieving prime to all other processors.

Block The block distribution solves the two problems above but is instead more complicated to work with. There are several ways to deploy the block distribution. The naive way is to assign $\lceil \frac{n}{p} \rceil$ to each processor. But this may again give a load bias between the first $p-1$ processors and the last processor. Instead we use the following partition: Processor $P(s)$ contains indexes from $\lfloor s\frac{n}{p} \rfloor$ to $\lfloor (s+1)\frac{n}{p} \rfloor - 1$. In the example from above we get the partition

1	2	3	4	5	6

7	8	9	10	11	12	13

14	15	16	17	18	19	20

With this distribution we get block lengths of either $\lfloor \frac{n}{p} \rfloor$ or $\lfloor \frac{n}{p} \rfloor + 1$.

We also use the basic reduction of only representing odd numbers. That is, since 2 will cross out all even numbers we don't need to represent them. That is, for some limit n we only represent the odd numbers from 3 to n or $n-1$ depending on whether n is odd or even respectively. We therefore need space for only $\lfloor \frac{n-1}{p} \rfloor$ elements. Each local sieving array have size $\left\lfloor (s+1)\lfloor \frac{n-1}{2} \rfloor / p \right\rfloor - \left\lfloor s \lfloor \frac{n-1}{2} \rfloor / p \right\rfloor$.

When in processor s consider the array contained in there and pick an index i in that array. We must know which odd integer this index corresponds to. This is done by the map $(s, i) \mapsto \lfloor s\frac{n}{p} \rfloor + 2(i+1) + 1$. Given the processor number s we can also find the endpoints of the interval containing the odd integers on that particular processor.

$$\begin{aligned} Left\,endpoint\ &: \left\lfloor s\frac{\lfloor\frac{n-1}{2}\rfloor}{p}\right\rfloor \mapsto 2\left\lfloor s\frac{\lfloor\frac{n-1}{2}\rfloor}{p}\right\rfloor + 3 \\ Right\,endpoint\ &: \left\lfloor (s+1)\frac{\lfloor\frac{n-1}{2}\rfloor}{p}\right\rfloor - 1 \mapsto 2\left\lfloor (s+1)\frac{\lfloor\frac{n-1}{2}\rfloor}{p}\right\rfloor + 1 \end{aligned}$$

Compared to the naive method the former method provide a more smooth distribution but it does require some extra computing power to manage. Benefits are two-fold though; Lower memory use and faster execution.

Communication of sieving prime To avoid a lot of broadcasting between processors when finding the next sieving prime we enforce an invariant; processor 0 must always contain all elements up to at least $\sqrt{n}$. By doing this we can use processor 0 to find all sieving primes without the need of first finding a candidate in all processors and broadcast all candidates. This invariant might imply that we need to reduce the number of processors that can be used. Say the user request P processors. We are then after the maximal p' such that

$$1 \leq p' \leq P, \quad \lfloor\sqrt{n}\rfloor \leq \left\lfloor \frac{\lfloor\frac{n-1}{2}\rfloor}{p'} \right\rfloor - 1$$

Set $p' = \left\lfloor \lfloor\frac{n-1}{2}\rfloor / (\lfloor\sqrt{n}\rfloor + 1) \right\rfloor$. Then if $p' < P$ we need to reduce to only using p' processors, and if $p' \geq P$ we can stick with the choice made by the user. It is worth noticing that a situation where we need to restrict to a smaller number of processors is not likely to occur when n is big (which is the case most people deal with). For example when only $n > 10000$ we need more than 100 processors before any restriction will take place.

There is a way of avoiding all communication during the sieving. That is to use a the sequential SoE on input $\sqrt{n}$ finding all primes from 2 to $\sqrt{n}$ and thereafter sieve using the primes just found. The advantage of this approach is, that one don't need to start up communication $\pi(n)$ times but can do with one start up. The downside is that you know need to do some more initial work in the beginning.

2.2 Description parallel version

Is this section we present the implementation. We give the pseudo code which we in section 2.3 will use to analyse the cost with respect to the BSP cost model. For the full implementation we refer to appendix B. Below we describe more detailed how we set of the distribution

Setup We imagine we have a bit array A of length $\lfloor \frac{n-1}{p} \rfloor$, representing the odd integers from 3 to n or $n-1$ depending on whether n is odd or even. A is initialized with ones. Let i be an index of A. Then we use the map $i \mapsto 2(i+1)+1$ to map each index in A to an unique odd integer. On each processor $P(s)$ we allocate an array as described in section 2.1. For any processors s the index in each allocated array now uniquely corresponds to one of the odd integers from 3 to n (or $n-1$) and using the formulas in section 2.1 this number can be computed. This finishes the setup.

Sieving Fix a processor s. The sieving consist of a number of equal steps. Each step is performed as follows. We start by picking the next sieving prime p. This is exclusively done in processor 0 and thereafter the index of p is broadcast i.e. the unique i such that $i \mapsto 2(i+1)+1 = p$ which is the element we find when searching the sieving array. When s has received the next sieving prime it calls a separate method that runs a sieving on the array B that reside on s. To sieve with a prime, one now only need to calculate the offset into B where the sieving should start i.e. the index in B that maps to the smallest odd integer there is a multiple of p. Algorithm 2.2.1 calculates the offset and takes as input the prime p (for which the index i map to) and endpoints of array B. To see that we need to be pay

Algorithm 2.2.1 calcOffset

```
Input: prime $p$, left endpoint $leftEnd$ and right endpoint $rightEnd$ both of array
  $B$ located at a particular processor.
Output: Smallest index in $B$ such that the corresponding odd integer is a multiple
  of $p$. If this is not possible to find such an index output is -1.
  if $leftEnd \mod p == 0$ then
    return 0
  end if
  $res = -(leftEnd \mod p) + p$
  if $res + leftEnd > rightEnd$ then
    return -1
  end if
  if $(res + leftEnd) \mod 2 == 0$ then
    return $\frac{res+p}{2}$
  else
    return $\frac{res}{2}$
  end if
```

attention to the possibility of completely missing a local sieving array consider the example with $n = 25$ and $p = 4$. This partition the array into the following local arrays: $[3,5,7]$, $[9,11,13]$, $[15,17,19]$, $[21,23,25]$. When sieving with prime 5, we totally misses the local array located at processor 1. To do the sieving we use the algorithm presented in algorithm 2.2.2. This is essentially the same sieving process

Algorithm 2.2.2 Soe

```
Input: Local sieving array $B$, $primeIndex$ (index from processer 0) of prime $p$, left endpoint $leftEnd$ and right endpoint $rightEnd$ of array $B$ and the processor number, say $s$.
  if $s == 0$ then
    $offset = 2 * primeIndex * (primeIndex + 3) + 3$
  else
    $offset = calcOffset(2 * primeIndex + 3, leftEnd, rightEnd)$
  end if
  if $offset \neq -1$ then
    while $offset < |B|$ do
      $B[offset] = 0$
      $offset = offset + prime$
    end while
  end if
```

as in the sequential algorithm - therefore the name.

Pseudo code for the parallel version We now give the pseudo code of the parallel version. We assume that the output is only going to be the number of primes below some limit n. The code is displayed as algorithm 2.3.3.

Improvements Of course numerous improvements exist. We mention some of them now. If one has a slow communication one may consider sending two primes during the sieving instead of just one. Adjusting the code for this entails some extra checks but if the communication is slow, this should provide a measurable improvement. Of course one can also deploy some of the standard improvements for the sequential sieve. This could be implementation of a wheel that will reduce some redundant cross outs or reorganization of the (essentially) two loops, that perform the sieving, such that we reduce the number of cache misses. Especially the last modification may give a significant speed up. If the program is to be used as an auxiliary method one may also consider using the sequential sieve for small inputs; see section 3 for why.

A further research into improving our current scheme for the parallel version can probably be done in the communication since we during the sieving only communicate from processor 0. Could one for example obtain an improvement by also sending some information from the other processors?

Algorithm 2.2.3 Parallel Soe

```
Input: Limit $n$ and number of processors $p$. Array $A$ distributed by the block
  distribution from section 2.1 using the setup from section 2.2. let $A_s$ be the local
  array.
Output: $numberOfPrimes$: Number of primes below $n$ (including).
  $primeIndex = 0$
  $leftEnd = 2\left\lfloor s\frac{\lfloor\frac{n-1}{2}\rfloor}{p}\right\rfloor + 3$
  $rightEnd = 2\left\lfloor (s+1)\frac{\lfloor\frac{n-1}{2}\rfloor}{p}\right\rfloor + 1$
  while $primeIndex < \frac{\sqrt{n}-3}{2} + 1$ do
    (0)
    $Soe(s, leftEnd, rightEnd, |A_s|, A_s)$
    if $s == 0$ then
      repeat
        $primeIndex = primeIndex + 1$
      until $A_s[primeIndex] == 1$ or $primeIndex == \frac{\sqrt{n}-3}{2}$
    end if
    (1)
    for $i = 1 \to p-1$ do
      put $primeIndex$ in $P(s,*)$
    end for
  end while

  (2)
  for $i = 0 \to |A_s| - 1$ do
    if $A_s[0] == 1$ then
      $numberOfPrimes_s = numberOfPrimes_s + 1$
    end if
  end for
  if $s == 0$ then
    $numberOfPrimes_s = numberOfPrimes_s + 1$
  end if
  (3)
  put $numberOfPrimes_s$ in $P(s,*)$.
  (4)
  $numberOfPrimes = 0$
  for $i = 0 \to p-1$ do
    $numberOfPrimes = numberOfPrimes + numberOfPrimes_i$
  end for
```

#Superstep	Cost
0+1	$\frac{n \ln \ln n}{2p} + \pi(\sqrt{n})6 + \pi(\sqrt{n})(p-1)g + 2\pi(\sqrt{n})l$
2	$\pi(\mu) + 1 + l$
3	$(p-1)g + l$
4	$p + l$

Table 1: BSP cost for algorithm 2.2.3.

2.3 BSP cost

For the analysis of the parallel version we assume that flagging an element in a sieving array takes one flop.

Observe that the total cost of superstep 0 and 1 basically is the cost of running the sequential Soe, cost of communicating and the fixed cost of 6 flop for computing the offset into each array. Since the size of the global array is $\delta := \lfloor \frac{n-1}{2} \rfloor$ we have that the total number of flops is approximate $\delta \ln \ln \delta + p\pi(\sqrt{n})6$. Since we have p processors we get the total approximate number of flop for superstep 0 is approximate $\frac{\delta \ln \ln \delta + p\pi(\sqrt{n})6}{p} \approx \frac{n \ln \ln n}{2p} + \pi(\sqrt{n})6$ which imply that superstep 0 and 1 have the approximate cost $\frac{n \ln \ln n}{2p} + \pi(\sqrt{n})6 + \pi(\sqrt{n})(p-1)g + 2\pi(\sqrt{n})l$. The convention in analysing the BSP cost is that we sum over the maximal number of flops by any processor in each iteration of superstep 0. But this will require us to determine, in each iteration, which local array imply the lowest offset of the sieving prime. Instead we have calculated the amortized number of flops and made the assumption that this is uniformly distributed over all processors.

Superstep 2 is a computational superstep which clearly obtain it's maximal flops on processor 0. It calculates the local number of primes. The cost is $\pi(\mu) + 1 + l$ where μ is the size of the local array on processor 0. We assume here that the density of primes decreases.

Superstep 3 is a communication superstep that simply broadcast one data word to each processor. The cost is $(p-1)g + l$.

Superstep 4 calculates the number of primes by adding up all local counters. The cost is $p + l$.

Table 1 sum up the cost of each superstep If we let T_{bspSoe} be the BSP cost of the parallel version we have

$$\begin{aligned}
T_{bspSoe} &= \frac{n \ln \ln n}{2p} + 6\pi(\sqrt{n})(p-1)g + 2\pi(\sqrt{n})l + \pi(\mu) + 1 + l + (p-1)g + l + p + l \\
&= \frac{n \ln \ln n}{2p} + 12\frac{\sqrt{n}}{\ln(n)} + 1 + p + \frac{\frac{\left\lfloor \frac{n-1}{2} \right\rfloor}{p} + 1}{\ln\left(\left\lfloor \frac{n-1}{2} \right\rfloor + p\right) - \ln(p)} \\
&\quad + g(p-1)\left(2\frac{\sqrt{n}}{\ln n} + 1\right) + l\left(4\frac{\sqrt{n}}{\ln n} + 3\right) \\
&\approx \frac{n \ln \ln n}{2p} + p + g(p-1)\frac{\sqrt{n}}{\ln(n)} + 4l\frac{\sqrt{n}}{\ln(n)}
\end{aligned}$$

where we have used the approximation $\pi(n) = O\left(\frac{n}{\ln(n)}\right)$.

3 Benchmark tests

In this chapter we test our implementation of the parallel SoE discussed in the last chapter. Our strategy is first to benchmark the computer we are using and then comparing empirical data with the expected theoretical performance. We will be testing the implementation on the supercomputer *Huygens*.

3.1 Benchmark

To do the benchmark we use a DAXPY operation. We use the already implemented version from the *BSPedupack* but also present a benchmark test in which the $O(n)$ *bsp_ put* address calculations is replaced with $O(n)$ *bsp_ get* address calculations. Table 2 and 3 shows the benchmark specifications and defines the BSP computer we are working with. One imidiately observe that g and l are higher for the *bsp_ get*. This reflects the fact that *bsp_put* is more efficient then *bsp_get*. Also observe that the increase in l reflect the fact that with more processors there is more to do at each synchronization. The variance in r must be due to inaccuracy in the benchmarking method.

3.2 Results/experiments

Here we present our results. Experiments have been done with the following method: We consider a bit b between 2 and 31 and pick a uniform randomly chosen number n such that $n \in [2^{b-1} + 1, 2^b]$. With this number we measure the execution time for

Table 2: Benchmark *Huygens* with *bsp_put*.

p	r (Mflop/s)	g	l
1	195.715	51.9	530.8
2	195.715	54.5	2109.5
4	193.256	54.7	4138.7
8	195.698	57	7634

Table 3: Benchmark *Huygens* with *bsp_get*.

p	r (Mflop/s)	g	l
1	195.715	82	747.3
2	182.929	80.5	2759.6
4	195.587	90.6	6317.7
8	192.932	74.4	16842.2

$p = 1, 2, 4, 8$ at *Huygens*. The results are displayed in the graphs below. We have divided the result into several plots to give a better view and comparison basis and also with the goal of showing where the parallel advantage kick in. As BSP library we have used BSPonMPI.

A more serious study would have involved picking (uniformly) a much bigger sample space for each bit, say 500-1000 numbers (but depending on the variance of the results), and take the average over a run with numbers in the sample space. Also a more universal study would have involved calculating the number of flops and a comparison with the result to the expected number of flops. In addition one could also compute the utilization of the CPU with respect to flop/s.

Comments From the charts the first thing one may notice is the lack of improvements in the execution time for small bit size input. this reflect that fact that more processors imply more communication. Therefore to make use of more processors n must be an appropriate bit size.

We are now going to compare the experiment with $p = 8$ with the theoretical execution time. For a bit b the theoretical execution time may be calculated by the formula

$$\left(\frac{2^b \ln\ln 2^b}{2p} + p + g(p-1)\frac{\sqrt{2^b}}{\ln(2^b)} + 4l\frac{\sqrt{2^b}}{\ln(2^b)}\right)\frac{1}{r}.$$

Notice that this will give an upper bound on the theoretical execution time for a number with bit size b.We will use the BSP computer from table 2 because we

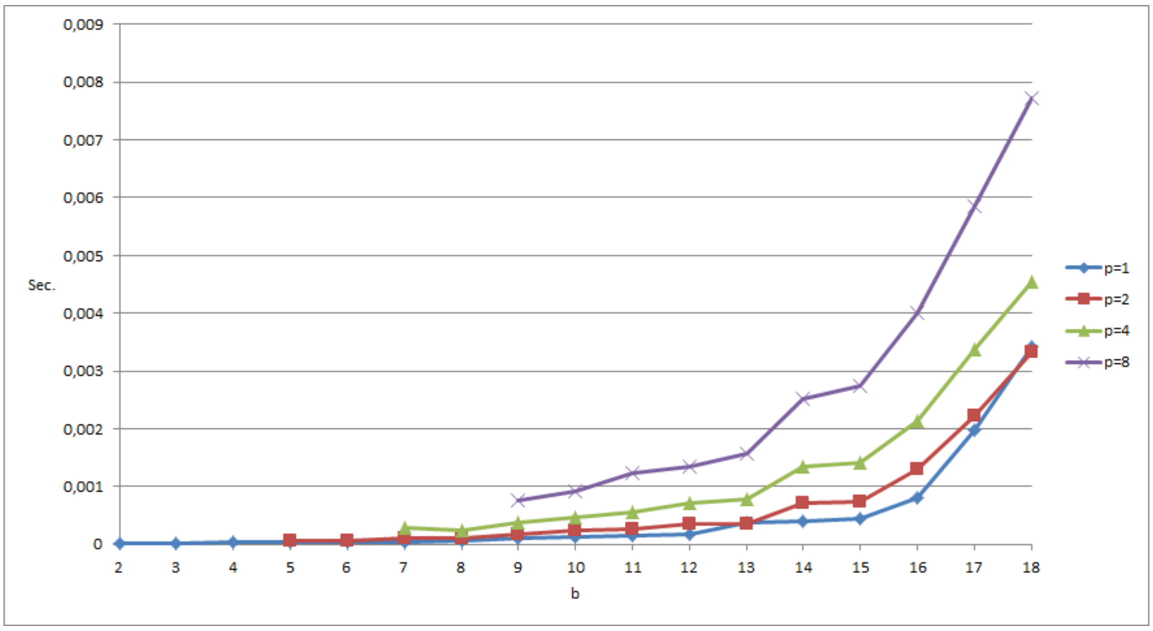

Figure 1: Small bit size numbers has poor advantage of parallelization.

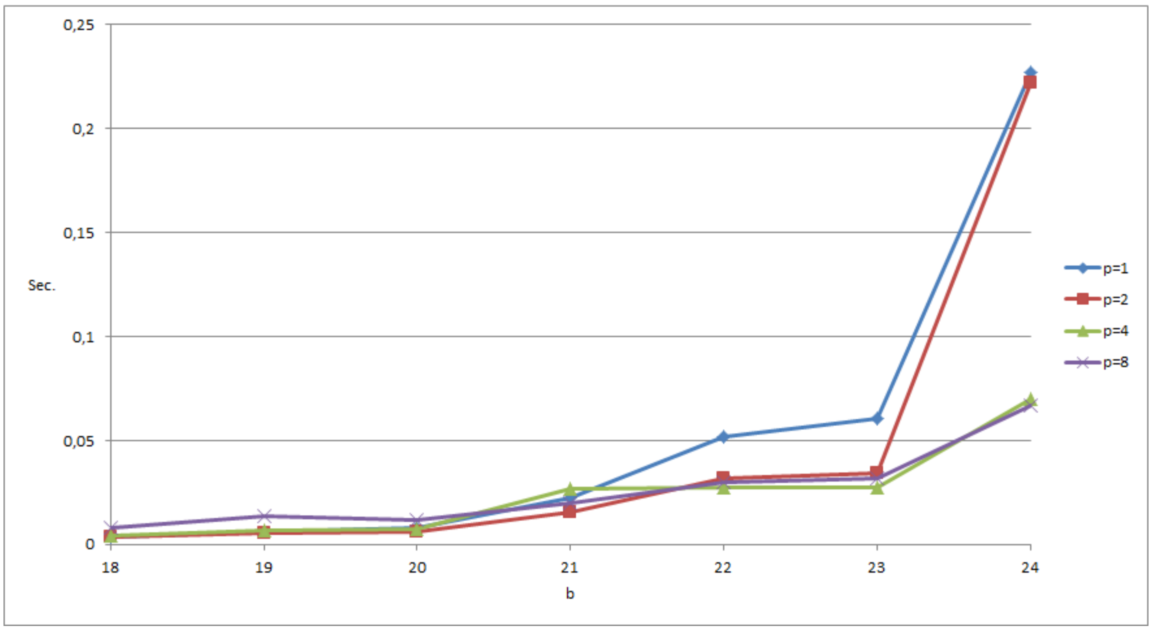

Figure 2: For numbers with bit size around 21-22 we begin to see the first signs of improvements by using more processors.

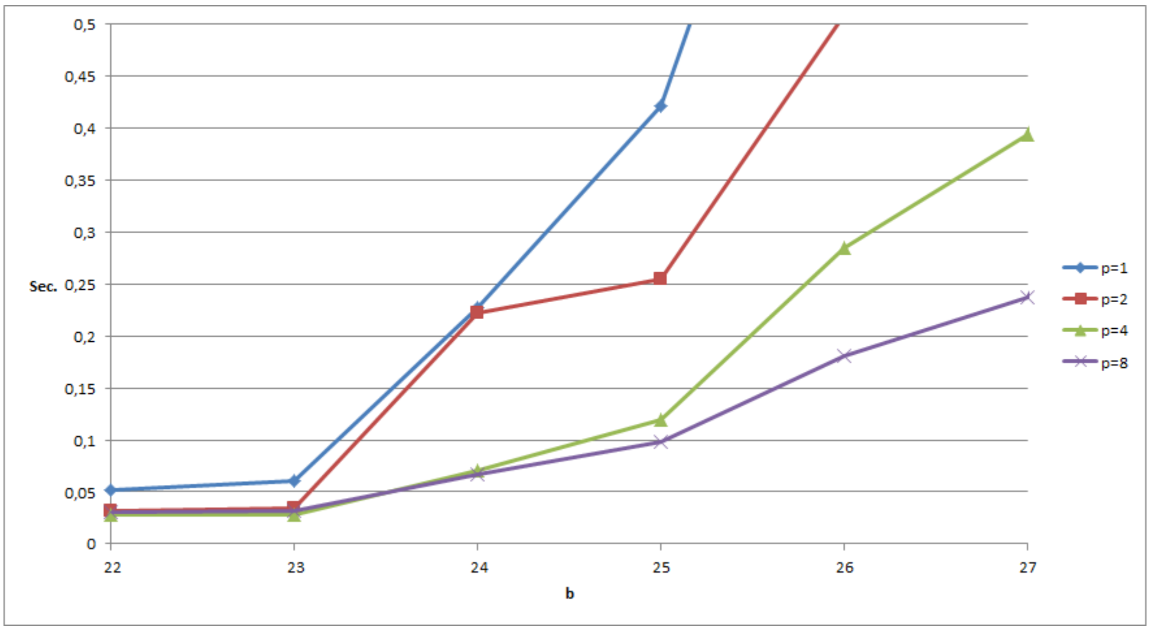

Figure 3: Improvements becomes clearer.

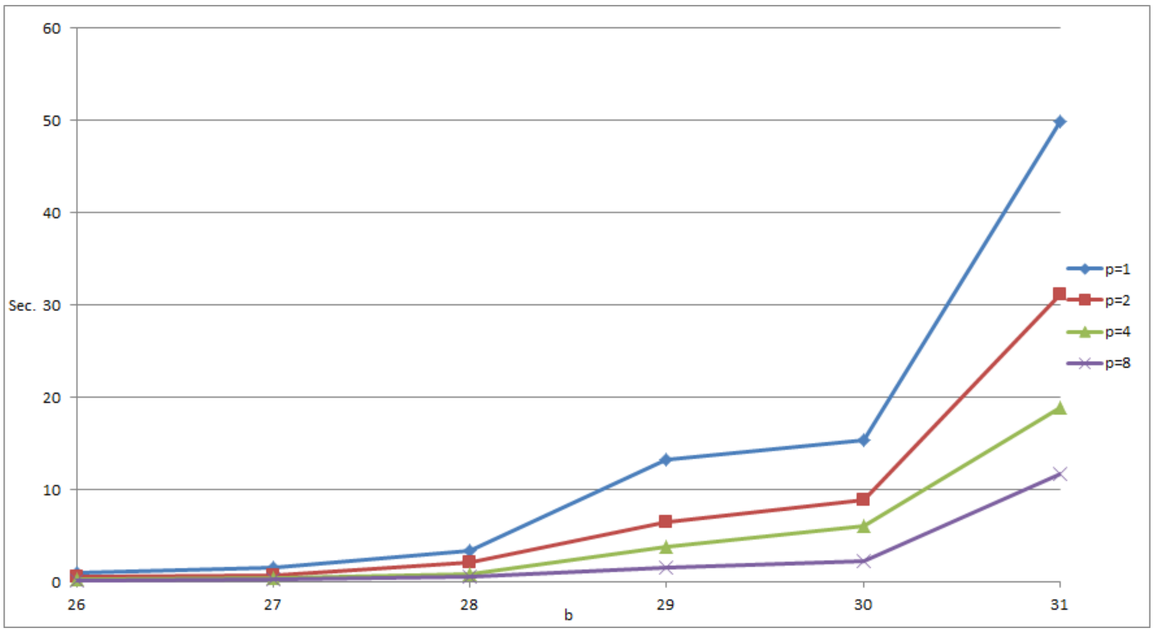

Figure 4: Nearing the limit of *int* the advantage of using 8 processors to 1 processors is approximate a factor of 4.

Table 4: Result from experiment with $p = 8$ and the expected execution time. We start from $b = 9$ because we can't run with 8 processors on smaller input.

b	experiment (sec.)	expected (sec.)
9	0.000762	0.005737
10	0.000927	0.007304
11	0.001239	0.009396
12	0.001355	0.012191
13	0.001566	0.015935
14	0.002519	0.02097
15	0.002749	0.027766
16	0.004004	0.036992
17	0.00585	0.0496
18	0.007717	0.066982
19	0.01363	0.091226
20	0.011955	0.125569
21	0.019891	0.175202
22	0.030168	0.248794
23	0.031679	0.361364
24	0.066814	0.539859
25	0.097834	0.834079
26	0.180812	1.338
27	0.237409	2.234
28	0.542927	3.876
29	1.538	6.967
30	2.305	12.897
31	11.618	24.447

almost exclusively use *bsp_put* in the implementation. In table 4 we have put the experimental execution time and the expected execution time. The higher expected execution times may be due to several things. First of all we use $n = 2^b$ which will give en upper bound. Also the formula used is again an approximation e.g. we have approximated using the formula $\pi(n) = O\left(\frac{n}{\ln(n)}\right)$. Looking closer at the table reveal that the ratio between the expected execution time and the experimental execution time is decreasing towards 1. This may reflect the approximative nature of the formula used. When the SoE is used as an auxiliary method (in a factorization algorithm as the elliptic curve method, where one need prime up to a certain limit) this range of bit sizes is fairly common and in that range our implementation out beats the expected time with at least a factor 2.5.

A Sequential Soe

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<math.h>
#include<time.h>

/*
* Input: The program will ask for input.
* Output: Number of primes below input (including the input)
* and execution time.
*
* The program is meant as a auxilliary method that can be
* used in several ways; number of primes below a limit,
* expanded to return an array of the primes between 2 and the
* input.
* We only represent odd numbers using the representation
* i |-> 2*i+1 where i is the index in the sieving_array with
* the convention that 0 |-> 2. With that convention we have
* index 2*i+1 is a prime if and only if sieving_array[i]=1,
* when we exit the method.
*/
main(){
  clock_t start = clock(); //start time

  long n;
  long sieving_array_size = 0;
  char *sieving_array;
  int number_of_primes_below_n = 1; //2 is prime

  printf("Input limit:\n");
  scanf("%d",&n);

  if(n<2){
    printf("No primes below 2\n");
    exit(0);
  }
  else{
    sieving_array_size = (n+1)/2;
    sieving_array = (char*)malloc(sizeof(char) * sieving_array_size);
    if (!sieving_array) {
    /* If data == 0 after the call to malloc,
    allocation failed for some reason */
      perror("Error allocating memory\n");
      exit(0);
    }
    memset(sieving_array, 1, sizeof(char)*sieving_array_size);
  }

  int i;
  int loop = 1;
  int prime;
  while(loop < (int)(sqrt(n)-1)/2+2 ){
    if(sieving_array[loop] == 1){
      number_of_primes_below_n++;
      prime = 2*loop+1; //Sieving with this prime
      for(i = loop*(2*loop+2); i < sieving_array_size; i = i + prime){
        /* We start at prime^2=2*(2*loop^2+2*loop)+1 and since we do not
        *  need to sieve the even primes we sieve at every double
        *  multiple of prime. */
        sieving_array[i] = 0;
      }
    }
    loop++;
  }
  while(loop < sieving_array_size+1){
    if(sieving_array[loop] == 1)
      number_of_primes_below_n++;
    loop++;
  }
```

```
  printf("Number of primes below %d is %d\n",n,number_of_primes_below_n);

  clock_t end = clock();
  printf("Time: %f sec.\n", (float)(end - start)/CLOCKS_PER_SEC);
  return 0;
}
```

B Parallel Soe

```
#include "bspedupack.h"

#define SZLONG (sizeof(long))

/*  This program computes the number of primes betwen 2 and n (including).
    In addition the user may chose to save the primes on each processor
    or both save and print out (standard output) all primes.

    Options:
      0 - Ordinary parallel sieve that count primes below n.
      1 - Store all primes found, on each processor.
      2 - Same as 1 but includes a printout of all primes found.
          This is done by processor 0.

    Notes:
      - We only represent odd numbers.
      - Block distribution. Invariant: Processor 0 contains all odd
        numbers from 3 to at least sqrt(n).
        Block size on processor s:
        floor((s+1)*(floor((n-1)/2) / p)) - floor(s*(floor((n-1)/2) / p))
      - Processor zero takes care of finding the next sieving prime.
*/

/* Global fields */
int P; /* number of processors requested */
long n; /* limit */
```

```
char flagOption; /* Self explaining */

void bspSoe(){

  void Soe(int s, int leftIndex, int rightIndex, int szArray,
                        int primeIndex, char sievingArray[]);
  int szLocalSievingArray = 0; /* Local sieving array size */
  char *sievingArray; /* Local sieving array */
  int *Primes; /* Stores primes - if option is flagged */
  int *Accumulate; /* To distribute results later */
  //long *Multiples; /* For becnhmarking */
  long number_of_flops=0; /* Local number of flops */
  int i, j, p, primeIndex, s, leftEnd, rightEnd, number_of_primes_below_n = 0;
  double time0, time1;

  bsp_begin(P);
  p= bsp_nprocs(); /* Number of processors obtained */
  s= bsp_pid();    /* Processor number */

  /* Make sure every processor knows everything */
  bsp_push_reg(&n,SZLONG);
  bsp_push_reg(&flagOption,sizeof(char));
  bsp_push_reg(&primeIndex,SZINT);
  bsp_sync();

  bsp_get(0,&n,0,&n,SZLONG);
  bsp_get(0,&flagOption,0,&flagOption,sizeof(char));
  bsp_sync();
  bsp_pop_reg(&n);
  bsp_pop_reg(&flagOption);

  /* Define size of arrays for distributing results */
  Accumulate = vecalloci(p);
  //Multiples = (long *)malloc(sizeof(long) * p);
  bsp_push_reg(Accumulate, p*SZINT);
  //bsp_push_reg(Multiples, p*SZLONG);
```

```
if(s==0)
  printf("Setting up distribution\n"); fflush(stdout);

/* Setup distribution. We might need to leave out some
   processors because we enforce that processor 0 contain
   odd integers from 3 up to at least sqrt(n). 'p' will
   be the number of processors used. This situation will
   not happen very often. Infact if n>10000 we need more
   than 100 processers before any restriction will take place.
*/

/* Avoid multiple computations */
int szGlobalArray = floor((n-1)/2);
leftEnd = 2*s*(szGlobalArray / p)+3;
rightEnd = 2*(s+1)*(szGlobalArray / p)+1;

/* Allocate sieving arrays */
szLocalSievingArray = floor((s+1)*(szGlobalArray / p)) -
                                floor(s*(szGlobalArray / p));
sievingArray = (char*)malloc(sizeof(char) * szLocalSievingArray);
if (!sievingArray && n > 2) {
  /* If sievingArray == 0 after the call to malloc,
  allocation failed for some reason */
  bsp_abort("Error allocating memory\n");
}
/* Set all cells to 1 */
memset(sievingArray, 1, sizeof(char)*szLocalSievingArray);

if(s==0)
  printf("Sieving\n"); fflush(stdout);

primeIndex = 0; /* First sieving prime(3) index  */

time0 = bsp_time();
```

```
/* Below is the main sieving.
   Benchmark: Each time we hit a prime, we do at most
   12 flops to calculate the offset and initialize
   the sieving plus the number of multiples we need
   to calculate for the prime.
*/
while(primeIndex < (int)((sqrt(n)-3)/2)+1){
  //number_of_flops += (rightEnd % (2*primeIndex+3))+12 -
                                (leftEnd % (2*primeIndex+3));
  Soe(s, leftEnd, rightEnd, szLocalSievingArray, primeIndex, sievingArray);
  if(s==0){ /* Find next sieving prime. This is done in processor 0. */
    do{
      primeIndex++;
    }
    while(sievingArray[primeIndex] == 0 && primeIndex < (int)((sqrt(n)-3)/2)+1);
  }
  bsp_sync(); /* To enforce the BSP scheme. */
  if(s==0){
    for(i=1; i<p; i++){
      bsp_put(i,&primeIndex,&primeIndex,0,SZINT);
    }
  }
  bsp_sync();
}
bsp_pop_reg(&primeIndex);

if(s==0)
  printf("Counting primes\n"); fflush(stdout);

/* Count non-flagged elements - these are the primes */
for(i=0; i<szLocalSievingArray; i++){
  if(sievingArray[i]==1)
    number_of_primes_below_n += 1;
}
bsp_sync():
```

```
/* Distribute results */
for(i=0;i<p;i++){
  bsp_put(i, &number_of_primes_below_n, Accumulate, s*SZINT, SZINT);
  //bsp_put(i, &number_of_flops, Multiples, s*SZLONG, SZLONG);
}
bsp_sync();

/* Calculate results on each processor */
number_of_primes_below_n = 0;
//number_of_flops = 0;
for(i=0;i<p;i++){
  number_of_primes_below_n += Accumulate[i];
  //number_of_flops += Multiples[i];
}
number_of_primes_below_n += 1; //Two is a prime

/* If flagged: Store primes from 1 to n on all processors. */
if(flagOption > 0){
  if(s==0)
    printf("Storing primes\n"); fflush(stdout);

  Accumulate[0] -= 1; //Store 2 adhoc.
  int temp = 0;
  for(i=0;i<p;i++){
     temp += Accumulate[i];
     Accumulate[i] = -Accumulate[i]+temp;
  }

  Primes = vecalloci(number_of_primes_below_n);
  bsp_push_reg(Primes, (number_of_primes_below_n)*SZINT);
  bsp_sync();

  int index = 0, pr = 2, k;
  for(i=0;i<p;i++)
    bsp_put(i, &pr, Primes, 0, SZINT);
  for(j=0;j<szLocalSievingArray;j++){
    if(sievingArray[j]==1){
      pr=leftEnd+2*j;
      for(k=0;k<p;k++){
bsp_put(k,&pr,Primes, SZINT*(Accumulate[s]+index+1),SZINT);
      }
      index++;
    }
  }
  bsp_sync();

  /* If flagged: Print out primes to standard output */
  if(flagOption > 1){
    if(s==0){
      int size = 10;
      if(n > 1000000)
        size = 5;
      printf("Primes from 2 to %ld:",n); fflush(stdout);
      for(i=0;i<number_of_primes_below_n;i++){
        if(i % size == 0)
          printf("\n");
        printf("%d, ",Primes[i]); fflush(stdout);
      }
    }
    printf("The end.\n");
    bsp_sync();
  }
}

time1 = bsp_time();
printf("Processor %d: Number of primes from 2 to %ld is %d\n",
              s, n, number_of_primes_below_n); fflush(stdout);
if(s==0){
  printf("This took %.6lf seconds.\n",time1-time0);
 // printf("Number of FLOPS (approx): %ld\n", number_of_flops);
}
```

```
  bsp_end();
} /* End bspSoe*/

void Soe(int s, int leftEnd, int rightEnd, int szLocalSievingArray,
                         int primeIndex, char sievingArray[]){

  int offset, prime;
  int calcOffset(int prime, int leftEnd, int rightEnd);

  prime = 2*primeIndex+3; /* Sieving prime */
  if(s == 0)
    offset = 2*primeIndex*(primeIndex+3)+3;
  else /* Save some time in proc 0. Might be redundant if
                            the bottleneck is somewhere else */
    offset = calcOffset(prime, leftEnd, rightEnd);

  if(offset != -1) {
    while(offset < szLocalSievingArray){
      /* Since we do not need to sieve the even
         primes we sieve at every double multiple
         of the sieving prime.
      */
      sievingArray[offset] = 0;
      offset = offset + prime;
    }
  }
} /* End Soe */

/* Calculate offset into processer s with left endpoint 'leftEnd'
   (recall we only represent odd numbers). RETURN -1 if prime
   does not hit the interval represented in s.
*/
int calcOffset(int prime, int leftEnd, int rightEnd){
  if(leftEnd % prime == 0)
    return 0;
  int res = -(leftEnd % prime) + prime;
  int value = res + leftEnd;
  if(value > rightEnd)
    /* For small n we might skip a whole interval. e.g.
       when n=25, p=4 and when sieving with 5 we skip
       the second interval.
    */
    return -1;
  if(value % 2 == 0)
    /* We need this to be odd because we only represent odd numbers */
    return (res+prime)/2;
  else
    return res/2;
} /* End calcOffset */

int main(int argc, char **argv){

    int temp;

    bsp_init(&bspSoe, argc, argv);

    /* sequential part */
    printf("How many processors do you want to use?\n"); fflush(stdout);
    scanf("%d",&P);
    if (P > bsp_nprocs()){
        printf("Sorry, not enough processors available.\n"); fflush(stdout);
        exit(1);
    }

    printf("Please enter n:\n"); fflush(stdout);
    scanf("%ld",&n);
    if(n<2){
      printf("Error in input: No primes stricly smaller than 2!!\n");
      exit(1);
    }
    temp = (floor((n-1)/2)) / (floor(sqrt(n)));

    if(P > temp && n > 2){
      P = temp;
      printf("Can only use %d processor/s!\n", P); fflush(stdout);
    }

    printf("Options: 0 nothing, 1 store primes, 2 store and print out primes.\n");
    scanf("%d",&flagOption);
    if(flagOption < 0 || 2 < flagOption){
      printf("Error in input: Options are 0, 1 or 2 only!\n");
      exit(1);
    }

    /* SPMD part */
    bspSoe();

    /* sequential part */
    exit(0);
} /* end main */
```