Parallel Algorithms – Initial Assignment

Alexandra Vegelien 2552901 Niels Wouda s2533561

8th October 2019

Contents

1 Introduction

Prime numbers find extensive use in the field of cryptography, where they form the building blocks of a number of public-key crypto-systems. RSA, in particular, is based on the difficulty of finding primes numbers that factor a large integer. Although a decryption method is known, its computational complexity is such that it is practically infeasible to find the decryption key. Furthermore, prime numbers underpin many implementations of hash tables, check-sum methods, and pseudo-random number generators. As such, computer scientist and mathematicians alike are eager to determine efficient algorithms for finding prime numbers.

In this paper we will develop a parallel algorithm for determining all prime numbers in a given half-open interval, and showcase some applications. Section 2 describes the problem formulation in more detail. Thereafter, in section 3, we will develop the sequential and parallel algorithms, discuss and implement various refinements, and provide theoretical cost estimates. In section 4, we show running times of our various implementations on a computer cluster, and compare our theoretical estimates with practice. In section 5, we showcase two applications: first, generating twin primes of the form $k \pm 1$, with k some suitable even, positive integer; second, we show how our parallel method may be used to verify Goldbach's conjecture. Finally, in section 6, we summarise our main findings, and offer directions for future study.

2 Problem definition

At first glance, computing prime numbers appears a simple enough task. A trivial algorithm successively steps through the positive integers k, starting from 2, and determines if any previous positive integers are amongst its proper divisors. Should no proper divisor—other than the trivial divisors 1 and k—exist, the number k must be prime. It is not difficult to see that such an algorithm runs in time $O(k^2)$. A further sophistication follows from the observation that any proper divisor beyond the square root $\sqrt{k}$ has divisors smaller than $\sqrt{k}$ (Riesel, 2012, 2–3). As such, an algorithm may terminate once no factor up to $\sqrt{k}$ properly divides k, thus reducing the time complexity to $O(k^{\frac{3}{2}})$.

For 'small' prime numbers up to several million, such methods may well suffice. For large numbers, not in the least primes of many thousands of digits, their super-linear time complexity makes them uniquely unsuited for the task. What if, instead of 'looking back' from a candidate integer to find its proper divisors, we 'look ahead' from the divisors and cross-out any multiples that are certainly not prime?

Practical considerations force such a sieving scheme to operate up to a set upper bound, say n. Many such sieves exist: the sieve of Eratosthenes (276–194 BCE) is a classic method, but more recently the sieve of Atkin attains an even better theoretical time complexity (Atkin and Bernstein, 2004). Below we will show the Eratosthenes' sieve attains a time complexity of $O(n \ln \ln n)$, which for the sieve of Atkin reduces to $O(n/\ln \ln n)$. The purpose of this study is firstly the implementation and refinement of a parallel prime number sieve, and only secondly to develop an efficient method for generating prime numbers. As such, we will work with the Eratosthenes' sieve rather than the sieve of Atkin: it is simple to explain, particularly trivial to implement in its sequential form, and allows for many sorts of refinements.

3 Solution method

In this section we will develop and refine a parallel method for sieving prime numbers up to some bound n. First, in section 3.1, we implement a sequential algorithm for the sieve of Eratosthenes. Although this might appear pointless at first glance, a sequential algorithm offers valuable insight into the problem at

hand, and illuminates opportunities for parallelisation. Indeed, as we will show in section 3.2, it even finds re-use in the parallel algorithm developed there.

3.1 Sequential sieve

The sieve of Eratosthenes is a sequential algorithm to "sieve" all prime numbers up to a given bound n. The algorithm works by initially marking all candidate integers $k \in [2, n)$ as prime. It then employs an iterative scheme to select a next integer marked prime, and crosses-out all its multiples $(2k, 3k, \ldots)$ up to n. Any un-marked integers upon termination are prime.

Observe that the multipliers $(2, 3, \ldots, k-1)$ must have been considered in iterations prior to k, such that these numbers multiplied by k are already crossed out! It follows that we only have to cross out any multiples of k starting at k^2, and thus, the algorithm may terminate as soon as $k^2 \geqslant n$ holds, or rather, we only need to iterate over the integers $k \in (2, 3, \ldots, \lfloor\sqrt{n}\rfloor)$. This version of the algorithm is given as algorithm 1. An implementation is given as `sieve.c` in appendix A.

Algorithm 1: Sequential prime number sieve.

Input : An upper-bound n.
Output: p_s, the set of primes in $[0, n)$.

```
1 Initialise the list n' as true for all elements in [2, n).
2 for k ∈ (2, 3, ..., ⌊√n⌋) do
3 |   if n'[k] is true then
4 |   |   Set each multiple of k in n' to false, starting at k^2.
5 p_s := {j ∈ [2, n) : n'[j] is true}.
```

3.1.1 BSP cost

We will provide an estimate for the expected amount of work. When an integer $k \in (2, 3, \ldots, \lfloor\sqrt{n}\rfloor)$ is prime, the sieve crosses out all multiples of k in the half-open interval $[k^2, n)$. There are about,

$$\frac{n - k^2}{k},$$

such multiples in the interval. The expected amount of cross-out operations W is thus,

$$\begin{aligned}
\mathbb{E}(W) &= \sum_{k=2}^{\lfloor\sqrt{n}\rfloor} \mathbb{P}(k \text{ is prime}) \left(\frac{n - k^2}{k}\right) \\
&\approx \sum_{k=2}^{\lfloor\sqrt{n}\rfloor} \left(\frac{1}{\ln k}\right) \left(\frac{n - k^2}{k}\right) \\
&= \sum_{k=2}^{\lfloor\sqrt{n}\rfloor} \frac{n - k^2}{k \ln k}. \qquad (1)
\end{aligned}$$

Using the logarithmic integral, we alternatively obtain the following estimate,

$$\mathbb{E}(W) \approx \int_2^{\lfloor\sqrt{n}\rfloor} \frac{n - t^2}{t \ln t} dt. \qquad (2)$$

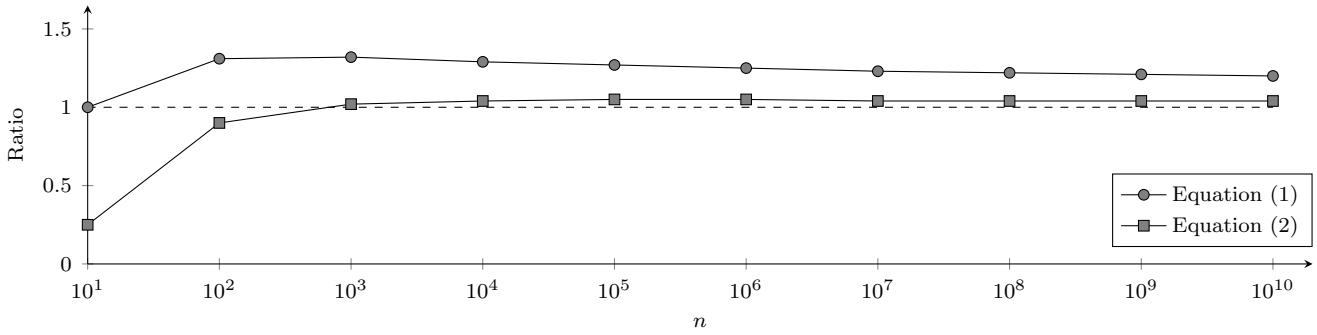

Figure 1: Ratios of the predicted number of operations over the actual number of operations for the sequential algorithm, for various values of n.

Both approximations rely on the prime-number theorem for an expression of the probability that a random positive integer is prime (see *e.g.* Cormen et al. (2009, 965–966)). Since there is no communication or synchronisation necessary for the sequential algorithm, we immediately find the BSP cost $T_{\text{seq sieve}}$ to equal $\mathbb{E}(W)$. We now add an operation counter to our programme, and compare the predicted (equations (1) and (2)) and actual operations for increasing values of n. The ratio of predicted over actual operations for both are given in figure 1.

Several conclusions follow. First, for $n \geqslant 10^3$, the predicted number of operations are pessimistic estimates of the actual number performed. This implies the algorithm is more efficient than our analyses would suggest. Second, the estimate offered by equation (2) is considerably tighter than the one provided by equation (1). Finally, the limited evidence obtained here suggests the cost estimates both approach the actual performed number of operations as n grows large.

3.2 Parallel approaches

Redeveloping the sieve as a parallel method is not straightforward. As we will see, much depends on the chosen data distribution, discussed first in section 3.2.1. After establishing a suitable distribution, we will give a basic parallel algorithm in algorithm 2, and estimate its expected cost in section 3.2.2. Finally, we present several refinements (in section 3.2.3) to both the sequential and parallel sieve that ought to increase their performance significantly in practice.

3.2.1 Data distribution strategies

For the sake of convenience, but w.l.o.g., assume the search space consists of the half-open interval $[0, n)$. A parallel algorithm based on the Eratosthenes' sieve must operate over the numbers up to $\lfloor\sqrt{n}\rfloor$, as we have shown above. When splitting this search space in some fashion, much depends on the manner in which it is split. Let $b = \lceil n/p \rceil$. Splitting the interval into p contiguous blocks would result in sub-intervals $\{[0, b), [b, 2b), \ldots, [(p-1)b, n)\}$, where primes in each sub-interval may be computed independently by each processor $P(s)$, $0 \leqslant s < p$. Note that the number of operations to be performed by processor $P(s)$ now depends on the upper bound of the assigned sub-interval, as $\lfloor\sqrt{(s+1)b}\rfloor$. It follows that each processor faces a different amount of work, which is undesirable as it might lead to underutilised machines—especially amongst those tasked with the 'lower' blocks.

Consider then a cyclic distribution. Here the problem dissipates, as each processor is responsible for a thinned slice of the entire interval $[0, n)$, and thus must operate over about $\lfloor\sqrt{n}\rfloor$ integers. But this is actually more work than the block distribution would entail, as now each processor must sieve all such

integers, whereas for the block distribution only the processor tasked with the 'highest' block would have to perform this much work. As such, there is no reason to expect a cyclic distribution to perform better than the block distribution, and in fact it may well perform worse.

We opt initially for a naïve block distribution, an algorithm for which is given in algorithm 2, where procedure `Sequential-Sieve` is as given in 3.1 (algorithm 1). The algorithm consists of two super-steps: the first, to communicate the sub-intervals to each processor; the second, to compute all primes in the assigned sub-interval. An implementation is given as `bspsieve.c` in appendix A.

Algorithm 2: Parallel block sieve for processor $P(s)$, with $0 \leqslant s < p$.

Input : A half-open interval $[0, n)$.
Output: p_s, the set of primes in $n_s = [sb,\ (s+1)b)$.

```
                                                          // Superstep 0
if s = 0 then
    for t from 0 to p − 1 do
        Compute bounds [tb, (t + 1)b) and send these to processor P(t).
                                                          // Superstep 1
Receive bounds from P(0).
Initialise the list n'_s as true for all elements in n_s.
for prime ∈ Sequential-Sieve(⌊√((s + 1)b)⌋) do
    Set each multiple of prime in n'_s to false.
p_s := {j ∈ n_s : n'_s[j] is true}.
```

3.2.2 BSP cost

In this subsection, we will analyse the BSP cost of algorithm 2 in order of the super-steps. In the first super-step, processor $P(0)$ determines the sub-interval bounds for each processor $P(t)$, with $0 \leqslant t < p$. It then transfers two data words: one for each of the lower and upper bound of the interval, respectively. It follows that h_s equals $2p$ for processor $P(0)$, and all processors receive two data words ($h_r = 2$). Hence, the communication step is a $2p$-relation, and completes with a global synchronisation at cost l.

In the second super-step, the amount of operations a processor $P(s)$, $0 \leqslant s < p$, must perform depends on its block size, as we have shown previously in section 3.2.1. The block sieve consists of two parts: first, primes in the candidate region $(2, 3, \ldots, \lfloor\sqrt{(s+1)b}\rfloor)$ are determined, and second, their multiples in the block $[sb, (s+1)b)$ are crossed-out. Any numbers not crossed-out following this second step must be themselves prime. By equations (1) and (2) and figure 1, we have a good approximation of the cost associated with the first step. When the set of candidate primes $\mathcal{P}$ is computed, the second step consists of about,

$$\sum_{k \in \mathcal{P}} \frac{b}{k}, \tag{3}$$

crossing-out operations, as there are b integers in each block, and thus about b/k multiples of k. We postulate that combining equations (1), (2) and (3) for an estimate of the work W_s facing processor $P(s)$

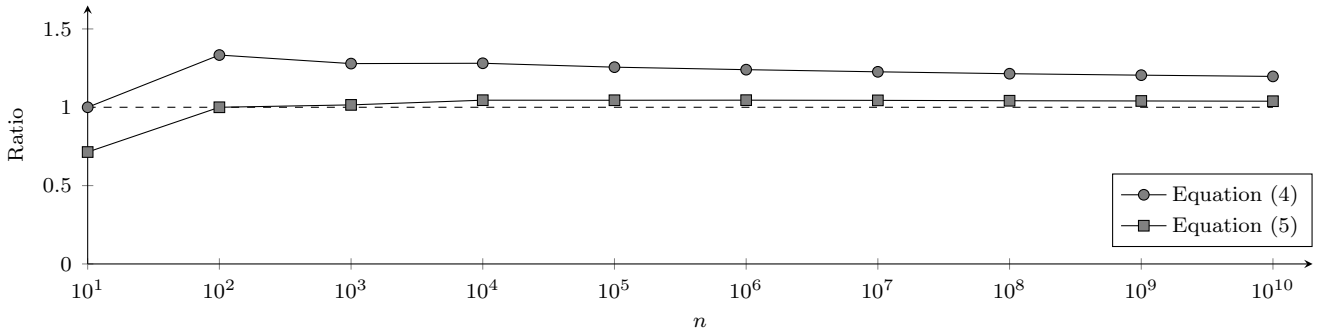

Figure 2: Ratios of the predicted number of operations over the actual number of operations for algorithm 2, for various values of n and $p = 1$.

as,

$$\begin{aligned}\mathbb{E}(W_s) &\approx \sum_{k=2}^{\lfloor\sqrt{(s+1)b}\rfloor} \left(\frac{1}{\ln k}\right)\left(\frac{b}{k}\right) \\ &= \sum_{k=2}^{\lfloor\sqrt{(s+1)b}\rfloor} \frac{b}{k \ln k} \end{aligned} \tag{4}$$

is indeed valid. Its continuous counterpart follows as,

$$\begin{aligned}\mathbb{E}(W_s) &\approx \int_2^{\lfloor\sqrt{(s+1)b}\rfloor} \frac{b}{t \ln t} dt \\ &\approx b\left(\ln\ln\sqrt{(s+1)b}\right). \end{aligned} \tag{5}$$

Setting $p = 1$, the original formulations in equations (1) and (2) obtain, albeit without the optimisation of crossing-out from k^2 for each candidate k. Analogous to figure 1, we perform a similar experiment here, the results of which may be found in figure 2 (since $p = 1$, it follows that $n = b$). We verified the estimate also for larger p up to four, and never found it to be off more than 10%.

Again, equation (5) provides a tighter bound than equation (4), and both appear to converge quickly to the actual number of operations. As such, the postulate that equations (4) and (5) provide a good estimate for the amount of crossing-out operations is validated in light of the data. Accounting for two counting and prime identification steps (`countPrimes`, respectively `getPrimes`; see `boundedsieve.c` in appendix A), the BSP cost of algorithm 2 for a general processor $P(s)$ follows as,

$$\begin{aligned}T_{\text{par sieve}} &= \mathbb{E}(W_s) + 2\sqrt{(s+1)b} + 2b + 2pg + 2l \\ &\approx b\left(\ln\ln\sqrt{(s+1)b}\right) + 2\left(\sqrt{(s+1)b} + b + pg + l\right) \\ &\approx b\left(\ln\ln\sqrt{(s+1)b}\right) + 2\left(pg + l\right) \\ &\leqslant b\left(\ln\ln\sqrt{n}\right) + 2\left(pg + l\right). \end{aligned} \tag{6}$$

Although the initial sequential-sieving part in algorithm 2 introduces a term in $\sqrt{n}$, we consider this a lower-order variable: except for trivial n or extremely large p, $b \gg \sqrt{n}$. The final inequality is motivated by the observation that each processor must wait until *all* are finished with their compute task, before

synchronisation may complete. For processor $P(p-1)$, this comes out to $\lfloor\sqrt{n}\rfloor$, which governs the overall computation cost. A similar argument explains the communication cost $2pg$, although only $P(0)$ actually sends this many data words.

What may be expected in terms of a speed-up, now that we know the BSP costs for both the parallel and sequential algorithms? Clearly, the speed-up $S_p(n)$ of the parallel approach must be,

$$S_p(n) = \frac{T_{\text{seq sieve}}}{T_{\text{par sieve}}} \overset{(\triangle)}{\approx} \frac{n \ln\ln\sqrt{n}}{b \ln\ln\sqrt{n}} = \frac{n}{b} = p. \tag{7}$$

In $(\triangle)$ we conveniently ignore the k^2 offset optimisation in the sequential algorithm, and any communication costs in the parallel case, such that the derivation works out nicely. This estimate is itself only an approximation, and even builds on earlier approximations, and as such does not perfectly replicate differences between the actual algorithms. Nonetheless, we have every reason for optimism: the derivation suggests computational speed-up scales linearly with the amount of processors available!

3.2.3 Enhancements

While the sequential and parallel algorithms of the previous sections are already efficient in terms of their asymptotic complexity, there are several practical refinements we consider below.

Start from a k^2 offset This refinement was already discussed for the sequential algorithm in section 3.1, but it also works for the block sieve. Observe that the expected gain here is modest: for any particular block assigned to processor $P(s)$ $(0 \leqslant s < p)$, the primes k such that $k^2 \in [sb, (s+1)b)$ are likely only a (small) minority of all primes in $[2, \lfloor\sqrt{(s+1)b}\rfloor]$.

Consider only odd candidates Observe that no primes $k > 2$ are even numbers, as two would divide all these. It follows that only odd numbers are candidate primes, and we can exploit this in both the sequential and parallel algorithms by storing only these candidates. As an implementation detail, arrays are zero-indexed and have a unit stride, and we thus need a mapping from candidate numbers to indices, and an inverse from indices back to candidate primes. Let us term these mappings h and h^{-1}, respectively. Then, we propose,

$$h(n) = n \text{ div } 2, \quad h^{-1}(i) = 2i + 1,$$

where *div* represents integer division, for $n \geqslant 3$ and $i > 0$. The actual implementation is more involved, but this mapping is at its heart. Since this observation halves the search-space, we may expect a two times speed-up over the general case estimated in equation (6).

Consider only candidates of the form $6k \pm 1$ Decoupling indices from numbers offers a fruitful abstraction, that may readily be exploited further. In particular, we claim all primes other than $\{2, 3\}$ must be of the form $6k \pm 1$, with $k \in \{1, 2, \ldots\}$.

It is not difficult to see why this must be the case. All positive integers $n \geqslant 6$ can be written as $6k + i$, with k as before and $i \in \{0, 1, \ldots, 5\}$. This follows directly from a simple application of modular arithmetic. Now, we know $6k + 0$, $6k + 2$, and $6k + 4$ are even numbers, and two divides them. Hence, they cannot be prime. Observe furthermore that $6k + 3$ divides by three, and thus cannot be prime either. This leaves only $6k + 1$ and $6k + 5 = 6(k + 1) - 1$ as candidates, which is equivalent to $6k \pm 1$ in the general case.

As for the odd integers above, we need a mapping and its inverse. Re-using h and h^{-1} for convenience, we propose,

$$h(n) = 2\left[(n+1) \text{ div } 6\right] - 1_{\{n \bmod 6 = 5\}}, \quad h^{-1}(i) = 6[(i+1) \text{ div } 2] - 2[i \bmod 2] + 1,$$

Table 1: Performance (seconds) of the base parallel sieve computation super-step, as given in algorithm 2, for various combinations of n and p.

p \ n	1	$\times 10^4$	1	$\times 10^5$	1	$\times 10^6$	1	$\times 10^7$	1	$\times 10^8$	1	$\times 10^9$	1	$\times 10^{10}$
1	9.50×10^{-5}		1.04×10^{-3}		1.22×10^{-2}		9.31×10^{-2}		1.15		1.26×10^{1}		1.38×10^{2}	
2	4.95×10^{-5}		4.99×10^{-4}		5.78×10^{-3}		4.28×10^{-2}		5.93×10^{-1}		6.58		7.22×10^{1}	
4	3.38×10^{-5}		2.16×10^{-4}		2.91×10^{-3}		2.35×10^{-2}		3.68×10^{-1}		4.16		4.53×10^{1}	
8	2.33×10^{-5}		1.19×10^{-4}		1.57×10^{-3}		1.36×10^{-2}		1.72×10^{-1}		2.09		2.27×10^{1}	
16	1.99×10^{-5}		7.97×10^{-5}		8.81×10^{-4}		8.20×10^{-3}		1.20×10^{-1}		1.69		1.86×10^{1}	
24	1.70×10^{-5}		9.04×10^{-5}		6.22×10^{-4}		6.75×10^{-3}		1.12×10^{-1}		1.60		1.77×10^{1}	

Table 2: Performance (seconds) of the base parallel sieve computation super-step with k^2 enhancement, for various combinations of n and p.

p \ n	1	$\times 10^4$	1	$\times 10^5$	1	$\times 10^6$	1	$\times 10^7$	1	$\times 10^8$	1	$\times 10^9$	1	$\times 10^{10}$
1	1.17×10^{-4}		1.07×10^{-3}		1.26×10^{-2}		8.89×10^{-2}		1.15		1.26×10^{1}		1.83×10^{2}	
2	6.40×10^{-5}		4.92×10^{-4}		5.92×10^{-3}		4.16×10^{-2}		5.89×10^{-1}		6.54		9.24×10^{1}	
4	3.30×10^{-5}		2.25×10^{-4}		2.96×10^{-3}		2.30×10^{-2}		3.64×10^{-1}		4.13		5.43×10^{1}	
8	2.39×10^{-5}		1.20×10^{-4}		1.54×10^{-3}		1.35×10^{-2}		1.71×10^{-1}		2.08		2.58×10^{1}	
16	1.87×10^{-5}		7.06×10^{-5}		8.43×10^{-4}		8.13×10^{-3}		1.20×10^{-1}		1.69		2.00×10^{1}	
24	1.58×10^{-5}		8.77×10^{-5}		6.02×10^{-4}		6.06×10^{-3}		1.11×10^{-1}		1.61		1.87×10^{1}	

where *mod* represents the modulo operator, for $n \geqslant 5$, and $i > 0$. Here we use the 'trick' that any number $(6k \pm 1) + 1$ integer divides to k, regardless of the sign on ± 1. Again, the actual implementation is more involved, but this mapping captures its essence. Compared to the odd candidates refinement, what speed-up may be expected? Observe that all odd integers are of the form $6k + \{1, 3, 5\}$. We have established that $6k + 3$ cannot be prime, which reduces the search-space by about one-third. We thus expect a speed-up of about one-and-a-half times compared to the odd candidate enhancement.

4 Results and discussion

In this section, we will illustrate some run-times for the algorithms introduced above. We measure the performance as the average time in seconds each version of the parallel algorithm takes to complete com-

Table 3: Performance (seconds) of the base parallel sieve computation super-step with odd candidates and k^2 enhancements, for various combinations of n and p.

p \ n	1	$\times 10^4$	1	$\times 10^5$	1	$\times 10^6$	1	$\times 10^7$	1	$\times 10^8$	1	$\times 10^9$	1	$\times 10^{10}$
1	6.90×10^{-5}		6.86×10^{-4}		7.31×10^{-3}		5.58×10^{-2}		5.78×10^{-1}		6.48		8.38×10^{1}	
2	3.55×10^{-5}		3.13×10^{-4}		3.21×10^{-3}		2.68×10^{-2}		2.81×10^{-1}		3.37		4.32×10^{1}	
4	2.35×10^{-5}		1.53×10^{-4}		1.66×10^{-3}		1.51×10^{-2}		1.67×10^{-1}		2.11		2.58×10^{1}	
8	1.93×10^{-5}		8.13×10^{-5}		9.45×10^{-4}		8.42×10^{-3}		5.66×10^{-2}		1.06		1.22×10^{1}	
16	1.85×10^{-5}		5.21×10^{-5}		3.80×10^{-4}		5.10×10^{-3}		3.55×10^{-2}		8.40×10^{-1}		9.61	
24	1.35×10^{-5}		3.99×10^{-5}		3.30×10^{-4}		4.82×10^{-3}		2.70×10^{-2}		7.98×10^{-1}		8.99	

Table 4: Performance (seconds) of the base parallel sieve computation super-step with candidates of the form $6k \pm 1$ and k^2 enhancements, for various combinations of n and p.

p \ n	1	$\times 10^4$	1	$\times 10^5$	1	$\times 10^6$	1	$\times 10^7$	1	$\times 10^8$	1	$\times 10^9$	1	$\times 10^{10}$
1	6.00×10^{-5}		4.60×10^{-4}		5.27×10^{-3}		4.18×10^{-2}		2.71×10^{-1}		5.47		5.91×10^{1}	
2	3.50×10^{-5}		2.49×10^{-4}		2.61×10^{-3}		2.32×10^{-2}		1.50×10^{-1}		2.87		3.10×10^{1}	
4	2.48×10^{-5}		1.27×10^{-4}		1.46×10^{-3}		1.49×10^{-2}		1.14×10^{-1}		1.66		1.78×10^{1}	
8	1.76×10^{-5}		7.35×10^{-5}		7.15×10^{-4}		7.60×10^{-3}		4.57×10^{-2}		8.20×10^{-1}		8.90	
16	1.98×10^{-5}		4.64×10^{-5}		3.03×10^{-4}		4.65×10^{-3}		2.79×10^{-2}		6.45×10^{-1}		7.17	
24	1.48×10^{-5}		6.46×10^{-5}		2.52×10^{-4}		3.94×10^{-3}		2.22×10^{-2}		6.03×10^{-1}		6.72	

putation super-step (1), for $n = 10^i$, with $i \in \{4, 5, \ldots, 10\}$, and $p = 2, 4, 8, 16, 24$. Restricting our attention to the computation super-step is justified as it dominates the BSP cost, instead of the near-constant communication cost $2pg$ for all p considered here.

The performance of the base parallel sieve as given in algorithm 2 is given in table 1, the k^2 enhancement in table 2, the enhancement where only the odd numbers are considered in table 3 and the enhancement with only numbers of the form $6k \pm 1$ in table 4. As run-times are not a deterministic measure of algorithmic performance, these results are at best anecdotal evidence: ideally, the experiment would be repeated sufficiently many times to obtain statistical evidence of the various speed-ups. Nonetheless, in most cases the speed-ups are sufficiently pronounced even one experiment suffices.

The speed-ups of the various refinements discussed in section 3.2.3 relative to the base sequential algorithm are summarised in figure 3, where a perfect linear speed-up is given as a dashed line. We will discuss each modification in turn.

When we consider the k^2 enhancement, the results are not significantly different from those for the base parallel sieve algorithm 2. The average speed-up (where the speed-up is defined as the time taken by the basic parallel sieve over that taken by the enhanced algorithm, for all n and p considered) is only 0.98, which leads to a rather indecisive conclusion: this enhancement does not do all that much. This was partially to be expected, as the gains were theorised to be only modest.

For the odds enhancement, we do observe a clear improvement. The average speed-up is about 1.85, which is close to the expected factor of about two.

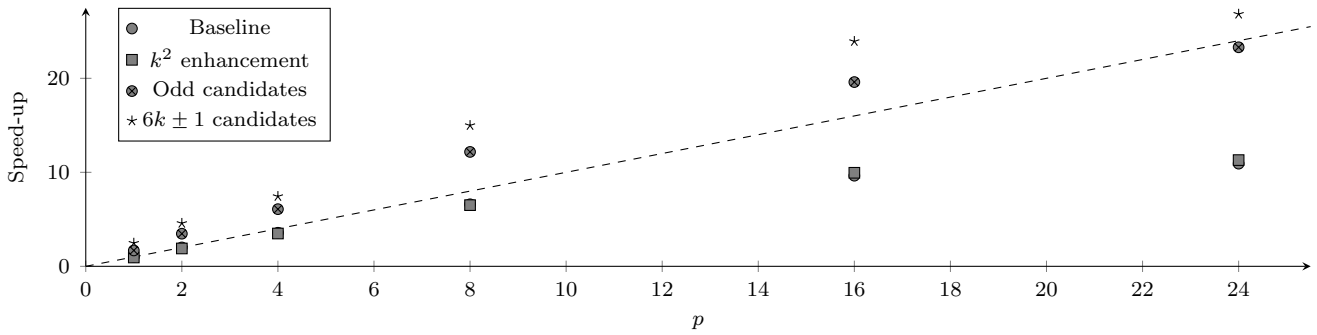

Figure 3: The average speed up of the proposed algorithms compared to the sequential baseline algorithm as a function of the number of processors. The dashed line represents $y = x$ as a reference for linearity.

We also notice that the $6k \pm 1$ enhancement shows a substantial improvement. The average speed-up is 2.33 where we expected a theoretical speed-up of 3. This is also a clear improvement relative to the odds enhancement, although a little smaller than theorised. We expect this may be due largely to the increased cost of mapping from and to the index array, especially when compared to the baseline implementation, or the odds enhancement—while determining these mappings remains a constant time operation, in practice, there are considerably more actions to perform, and the difference adds up.

Finally, a word about the hoped-for linearity in speed-up. While our algorithms do appear to achieve this for p up to about 8, thereafter we observe a small but noticeable slow-down. We do not know whether this is due to increased node utilisation on the cluster as we scale up the number of processors in use, a scaling factor overlooked in our cost estimate of equation (6), or simply a statistical artefact of the single run performed.

5 Applications

In this section, we investigate two applications of the parallel sieving method developed above. First, we will look into generating twin primes, and finally consider an application to verifying Goldbach's conjecture in parallel.

5.1 Twin primes

The twin prime conjecture which states that there are infinitely pairs of primes which differ by only two, which we call twin primes, has been one of the great unanswered questions in number theory. It has so far been proven that there are infinitely many pairs of primes which differ only 246 by the Polymath8b project.

In this subsection we want to show a straightforward application of the algorithms developed for sieving prime numbers: sieving twin primes. The enhancement of only going through the numbers of the form $6k \pm 1$ is especially useful for this application because if $(p, p+2)$ is a twin prime pair, then $p = 6k - 1$ and $p + 2 = 6k + 1$ for some k. Therefore, we will adapt this algorithm to generate the twin primes up to some bound n.

Two modifications to the algorithm must be made for it to generate twin primes. The first is that a prime should only be marked "twin prime" if it is prime and has a prime at a distance of two. This is an adaption in converting the list n'_s which contains whether a value is prime or not to the list of primes p_s, as shown in line (8) of algorithm 2. This line should now read,

$$p_s := \{j \in n_s \ : \ (n'[j] \text{ is } \texttt{true} \text{ and } (n'[j+2] \text{ is } \texttt{true} \text{ or } n'[j-2] \text{ is } \texttt{true})\}.$$

Observe that the equivalent change should not be made to algorithm 1, as we do still require all candidate primes be sieved.

The second modification has to do with a boundary case. If the bounded sieve is tasked with generating twin primes in some block $[b, 2b)$, and $2b$ mod 6 is either zero or one, then the interval may contain a prime at the boundary, but not its twin in another block. As $6k \pm 3$ is never prime, the modified interval $[b, 2b-3)$ ensures no twin primes are separated by the block boundary $2b$. An analogous check and potential remedy must be performed to the lower bound.

The BSP cost of the adapted algorithm remains unchanged as the only difference in operations stems from determining the bounds. This is a difference in $O(p)$, and since we generally assume $n \gg p$, this does not influence the order of the BSP cost.

5.2 Goldbach's conjecture

Goldbach's strong conjecture, as originally stated in a letter correspondence between Goldbach and Euler (Letter XLIII, 1742), reads as follows,

Conjecture 1 (Goldbach's) *Every even integer greater than* 2 *can be written as the sum of two primes.*

In recent years, the conjecture has been verified up to $4 \cdot 10^{18}$ (Oliveira e Silva et al., 2014), but no formal proof is as yet known.

For a parallel method to check the conjecture up to some bound n, let us take algorithm 2 as a starting point. After super-step (1), the primes up to some bound n are known in distributed form. To check whether any integers in the particular block assigned to processor $P(s)$ are prime, we need access to all primes up to $(s+1)b$. This motivates communication super-step (2), where each processor communicates its primes to all subsequent processors. Thereafter, each processor sorts the received primes and performs the Goldbach check over the range $[sb, (s+1)b)$: for each candidate k, we find two primes that together sum to k. Should no such primes exist, a message is printed. This algorithm is given as algorithm 3. We ran the algorithm for candidates up to 10^9, and all was as expected: no message was printed.

Algorithm 3: Parallel algorithm of the Goldbach conjecture for processor $P(s)$, with $0 \leqslant s < p$.

Input: A half-open interval $[0, n)$.

```
p_s := Parallel-Sieve([0, n)).
                                                        // Superstep 2
for t from s + 1 to p − 1 do
    Send p_s to processor P(t).
                                                        // Superstep 3
Receive all primes p_t into array primes from processors P(t), 0 ⩽ t ⩽ s.
Sort primes.
for all even integers k ⩾ 4 in [sb, (s + 1)b) do
    found := false.
    for first ∈ primes do
        second := k − first
        if first > second then
            break
        if second ∈ primes then
            found := true.
            break
    if not found then
        print "k is not the sum of two primes!"
```

What is the BSP cost of this method? For the first two super-steps (0) and (1), we may use equation (6) as a cost estimate. Observe that communication super-step (2) depends on the number of primes in each block: by the prime number theorem we have an approximation,

$$\hat{\pi}_{\text{interval}}([sb, (s+1)b)) = \hat{\pi}((s+1)b) - \hat{\pi}(sb) = \left\lfloor \frac{(s+1)b}{\ln\{(s+1)b\}} - \frac{sb}{\ln\{sb\}} \right\rfloor.$$

Since the logarithm is a monotonically increasing function, we have for subsequent blocks,

$$\hat{\pi}_{\text{interval}}([(s+1)b, (s+2)b)) \leqslant \hat{\pi}_{\text{interval}}([sb, (s+1)b)).$$

We thus expect the first block assigned to $P(0)$ to have the most primes, at about,

$$\hat{\pi}([0,b))_{\text{interval}} = \hat{\pi}(b) = \left\lfloor \frac{b}{\ln b} \right\rfloor.$$

Since these primes must be sent to all other processors, we have a h-relation of about $\hat{\pi}(b)p$. It follows that the BSP cost of this super-step is $\hat{\pi}(b)pg + l$.

Next, consider the computation super-step (3). Processor $P(s)$ will have some $\hat{\pi}_{\text{interval}}([0,(s+1)b))$ primes to store, which are to be sorted at cost $\hat{\pi}((s+1)b)\ln\hat{\pi}((s+1)b)$. It then iterates over the about $b/2$ even integers k in its assigned block, checking if for any prime up to $k/2$, another prime exists such that their sum equals k. Since we sorted the array, we may use a binary search here to determine if such a prime exists, at cost $\log_2 primes \approx \ln primes$. Using a hash set data structure, on the other hand, would obsolete the need for sorting *and* reduce the existence check to amortised constant time. This, however, is non-trivial to implement in languages that do not have such data structures built-in, so we forego these improvements here. The BSP cost of this step thus works out to be about,

$$\begin{aligned}
\mathbb{E}(W_g) &\approx \sum_{k=sb \text{ div } 2}^{(s+1)b \text{ div } 2} \hat{\pi}(k) \cdot \ln\hat{\pi}((s+1)b) + l \\
&\leqslant \frac{b}{2}\hat{\pi}((s+1)b/2) \cdot \ln\hat{\pi}((s+1)b) + l \\
&\leqslant \frac{b}{2}\hat{\pi}(n/2) \cdot \ln\hat{\pi}(n) + l \qquad (8) \\
&\overset{(\square)}{\approx} \frac{b}{2}\ln n \cdot \ln\hat{\pi}(n) + l. \qquad (9)
\end{aligned}$$

Where we again motivate inequality equation (8) through the cost of global synchronisation. Although in pathological cases the inner loop in algorithm 3 must iterate over all primes up to $k/2$, Oliveira e Silva et al. (2014) find empirically that the actual number of iterations rarely exceeds $\log k$. This underpins approximation ($\square$), which we verified to hold as a reasonably tight upper bound for our implementation: we find ratios stable at around 1.3 of predicted over actual operations, for various n up to 10^9. We finally obtain a total BSP cost of,

$$\begin{aligned}
T_{\text{Goldbach}} &= T_{\text{par sieve}} + \mathbb{E}(W_g) + \hat{\pi}(b)pg + l \\
&\approx b\left(\ln\ln\sqrt{n}\right) + 2\left(pg + l\right) + \frac{b}{2}\ln n \cdot \ln\hat{\pi}(n) + l + \hat{\pi}(b)pg + l \\
&= b\left(\ln\ln\sqrt{n}\right) + \frac{b}{2}\ln n \cdot \ln\hat{\pi}(n) + 2pg + \hat{\pi}(b)pg + 4l \\
&\approx b\left(\ln\ln\sqrt{n}\right) + \frac{b}{2}\ln n \cdot \ln\hat{\pi}(n) + \hat{\pi}(b)pg + 4l. \qquad (10)
\end{aligned}$$

6 Conclusion

As a brief summary, in the above we introduced a parallel algorithm to determine primes up to a given bound n, and several refinements. We gave theoretical cost estimates for each, and compared their running times in practice. The experimental results show an average speed-up of 0.98, 1.85, and 2.33, respectively, for the enhancements k^2, odds, and $6k \pm 1$. These speed-ups are considerable, especially for the latter two improvements. However, as the results are based on only a single run for each combination of n and p, we suggest counting operations, or repeated measurements to obtain a stronger condition on the speed-up in practice. This should also help decide whether the speed-up of parallelisation is perfectly linear.

Further investigations could study additional enhancements. Recall from section 3.2.1 that we expect the workload for the parallel algorithm to depend on the upper bound of the sub-interval (as also shown in *e.g.* equation (5)). A workload balancing method may be used to determine non-uniform sub-interval bounds, such that each processor has an equal expected amount of work. Especially for larger n and p, this might well make a substantial difference. Although this will not reduce the theoretical BSP cost as the upper bound remains in place, a prototype for this improvement (not implemented in the sieving algorithm) suggests gains of up to 20% are possible for even modest p.

A final consideration addresses an implementation detail. In our code, we use a mask array to track which integers are marked prime using Boolean variables. The `C` programming language defines Booleans as a special type of integer (namely, 0 and 1), which should be large enough to store these constants in memory. This implies at least one byte is used, although the specifics are compiler-dependent. Our Booleans, on the other hand, may be stored exactly in only a single bit, thus offering potential memory savings of at least `CHAR_BIT`! Indexing the mask array would be a little more involved, as it would need bit shifts to identify the correct Boolean value for a given number, but this might be worthwhile to reduce our programme's memory footprint, at the cost of a little performance overhead.

7 Author contributions

Alexandra wrote the introduction, the results and discussion text, developed an algorithm to generate twin primes, and wrote the conclusion. She implemented the $6k \pm 1$ optimisation and twin prime algorithm in computer code. Niels wrote the problem formulation, solution method, and developed an algorithm for the Goldbach conjecture. He implemented these algorithms in computer code and tested them on the cluster.

References

Atkin, A. O. L. and D. J. Bernstein (2004). Prime sieves using binary quadratic forms. *Mathematics of Computation 73*(246), 1023–1030.

Cormen, T. H., C. E. Leiserson, R. L. Rivest, and C. Stein (2009). *Introduction to Algorithms* (3 ed.). MIT Press.

Oliveira e Silva, T., S. Herzog, and S. Pardi (2014). Empirical verification of the even goldbach conjecture and computation of prime gaps up to $4 \cdot 10^{18}$. *Mathematics of Computation 83*(288), 2033–2060.

Riesel, H. (2012). *Prime numbers and computer methods for factorization* (2 ed.). Modern Birkhäuser Classics. Birkhäuser.

A Code

All code may be obtained from GitHub, alongside documentation and a selection of unit and regression tests. The base implementations of the sequential and parallel sieves are available in the `baseline` branch, and partially reproduced below. The k^2 and odd candidates refinements are implemented in the `odd-k2` branch. The twin primes code is available in `twin-primes`, while the Goldbach conjecture is implemented in `goldbach`. Finally, the 'lastest-and-greatest' with the $6k \pm 1$ enhancement is in `six-k`. Several of the baseline algorithms are reproduced below—for all details and helper methods, we strongly encourage the curious reader to consult our GitHub repository.

sieve.c

```c
#include <stddef.h>
#include <stdbool.h>
#include <stdlib.h>
#include <assert.h>

#include "primes.h"
#include "utils/utils.h"


size_t *sieve(bounds const *bounds, size_t *numPrimes)
{
    assert(bounds->upperBound >= 2);            // sanity checks.
    assert(bounds->lowerBound == 0);

    bool *isPrime = init(bounds);               // marks all numbers prime.

    for (size_t number = 2; number * number < bounds->upperBound; ++number)
        if (isPrime[number])
            // If this number is prime, then it is a proper divisor for any of
            // its multiples. Note that any multiples less than the square have
            // already been unmarked at this point.
            unmark(isPrime, bounds->upperBound, number * number, number);

    *numPrimes = countPrimes(isPrime, bounds->upperBound);
    size_t *result = getPrimes(isPrime, bounds, *numPrimes);

    free(isPrime);

    return result;
}
```

boundedsieve.c

```c
#include <stddef.h>
#include <stdbool.h>
#include <stdlib.h>
#include <assert.h>
#include <math.h>

#include "primes.h"
```

```
8  #include "utils/utils.h"
9
10
11 size_t *boundedSieve(bounds const *bounds, size_t *numPrimes)
12 {
13     if (bounds->lowerBound == 0)            // this is the 'regular' case.
14         return sieve(bounds, numPrimes);
15
16     assert(bounds->upperBound >= 2);        // sanity checks.
17     assert(bounds->upperBound > bounds->lowerBound);
18
19     bool *isPrime = init(bounds);           // marks all numbers prime.
20
21     // These are all primes in the candidate region [0, sqrt(upperBound) + 1),
22     // similar to the regular sequential algorithm.
23     size_t numCandPrimes = 0;
24     struct bounds const candidateBounds = {0, sqrt(bounds->upperBound) + 1};
25
26     size_t *candPrimes = sieve(&candidateBounds, &numCandPrimes);
27     size_t const range = bounds->upperBound - bounds->lowerBound;
28
29     // Equipped with these primes, we unmark all their multiples within the
30     // interval.
31     for (size_t idx = 0; idx != numCandPrimes; ++idx)
32     {
33         size_t const prime = candPrimes[idx];
34
35         // Since we unmark from the first multiple *after* the lower bound, we
36         // need to check the lower bound itself explicitly.
37         if (bounds->lowerBound % prime == 0)
38             isPrime[0] = false;
39
40         // Unmark all multiples of the given prime in the isPrime array. Note
41         // the unusual set-up as we translate from block range to index range,
42         // and compute the first multiple of prime after the lower bound.
43         unmark(isPrime, range, prime - bounds->lowerBound % prime, prime);
44     }
45
46     *numPrimes = countPrimes(isPrime, range);
47     size_t *result = getPrimes(isPrime, bounds, *numPrimes);
48
49     free(isPrime);
50     free(candPrimes);
51
52     return result;
53 }
```

bspsieve.c

```
1  #include <stdlib.h>
2
3  #include <bsp.h>
4
```

```
5  #include "primes.h"
6  #include "utils/utils.h"
7
8
9  extern long BSP_NUM_PROCS;
10 extern bounds const *BSP_BOUNDS;
11
12 static void stepAssignBounds();     // these neatly decouple the super-steps
13 static void stepComputePrimes();    // into separate functions.
14
15 void bspSieve()
16 {
17     bsp_begin(BSP_NUM_PROCS);
18
19     stepAssignBounds();     // First we compute the bounds for each processor
20     bsp_sync();             // task, and distribute those.
21
22     stepComputePrimes();    // After receiving the bounds, each processor
23     bsp_sync();             // computes primes within this sub-interval.
24
25     bsp_end();
26 }
27
28 static void stepAssignBounds()
29 {
30     if (bsp_pid() == 0)     // The first processor determines the bounds for
31     {                       // all others.
32         for (size_t processor = 0; processor != bsp_nprocs(); ++processor)
33         {
34             bounds const bounds = blockBounds(
35                 BSP_BOUNDS,
36                 BSP_NUM_PROCS,
37                 processor);
38
39             // Send *one* message to each processor, containing the lower
40             // and upper bound.
41             bsp_send(processor, NULL, &bounds, sizeof(struct bounds));
42         }
43     }
44 }
45
46 static void stepComputePrimes()
47 {
48     unsigned int messages;
49     bsp_qsize(&messages, NULL);
50
51     if (messages != 1)      // we expect *one* message containing the bounds.
52         bsp_abort("Processor %u did not receive bounds.", bsp_pid());
53
54     bounds bounds;          // Receive the given bounds..
55     bsp_move(&bounds, sizeof(struct bounds));
56
57     size_t numPrimes = 0;   // ..and compute primes.
58     size_t *primes = boundedSieve(&bounds, &numPrimes);
```

```
59
60      // TODO: Here you would, presumably, want to use these prime numbers.
61
62      free(primes);
63  }
```