INFORMATION AND COMPUTATION 114, 115–130 (1994)

# Two Fast Parallel Prime Number Sieves

JONATHAN SORENSON*

*Computer Sciences Department, University of Wisconsin at Madison, Madison, Wisconsin 53706*

AND

IAN PARBERRY†

*Center for Research in Parallel and Distributed Computing, Department of Computer Sciences, University of North Texas, Denton, Texas 76203-3886*

A *prime number sieve* is an algorithm that lists all prime numbers up to a given bound $n$. Two parallel prime number sieves for an algebraic EREW PRAM model of computation are presented and analyzed. The first sieve runs in $O(\log n)$ time using $O(n/(\log n \log\log n))$ processors, and the second sieve runs in $O(\sqrt{n})$ time using $O(\sqrt{n})$ processors. The first sieve is optimal in the sense that it performs work $O(n/\log\log n)$, which is within a constant factor of the number of arithmetic operations used by the fastest known sequential prime number sieves. However, when both sieves are analyzed on the Block PRAM model as defined by Aggarwal, Chandra, and Snir, it is found that the second sieve is more work-efficient when communication latency is significant. 

## 1. INTRODUCTION

A *prime number sieve* is an algorithm that lists all prime numbers up to a bound $n$. The best-known prime number sieve, attributed to Eratosthenes, finds the primes up to $n$ using $O(n \log\log n)$ arithmetic operations on "small numbers." Currently, the fastest known prime number sieves use only $O(n/\log\log n)$ arithmetic operations (see Mairson (1977) and Pritchard (1981)). Parallel implementations of prime number sieves have also been investigated (see Bokhari (1987) and Hikita and Kawai (1980)), but little rigorous theoretical analysis has been done. We present and analyze two parallel prime number sieves; one that performs best when

* Sponsored by NSF Grant CCR-8552596. Author's current address: Department of Mathematics and Computer Science, Butler University, 4600 Sunset Avenue, Indianapolis, Indiana 46208. E-mail address: sorenson@butler.edu.

† E-mail address: ian@ponder.csci.unt.edu.



0890-5401/94 $6.00

communication latency is low, and one that performs best when communication latency is high.

Our model of computation is the standard algebraic parallel random access machine (PRAM) model with bounded word size, in which each processor may read and write to a shared memory and perform arithmetic operations on $O(\log n)$-bit integers in one parallel step (we use the term *time* for the number of parallel steps). The *work* performed by a parallel algorithm is the product of the number of processors and the running time. We say that a parallel algorithm is *optimal* if it satisfies two conditions: (1) its running time is bounded above by a polynomial in $\log n$, and (2) it performs work proportional to the running time of the fastest known sequential algorithm for the same problem. For more on parallel models of computation and the PRAM, see Parberry (1987) and Karp and Ramachandran (1990).

Our first sieve is a parallelization of a modified version of a linear sieve due to Pritchard (1987, Algorithm 3.3). This algorithm runs in $O(\log n)$ time using $O(n/(\log n \log\log n))$ processors on an exclusive-read, exclusive-write (EREW) PRAM. Since the fastest known sequential sieves use $\Theta(n/\log\log n)$ arithmetic operations, this parallel sieve is optimal. Our second sieve is a parallelization of another linear sieve due to Pritchard (1983) that only uses $O(\sqrt{n})$ space. This algorithm runs in $O(\sqrt{n})$ time using $O(\sqrt{n})$ processors on an EREW PRAM, and appeared in Parberry (1981).

Although our first sieve is more work-efficient than our second sieve ($O(n/\log\log n)$ compared to $O(n)$), we believe that our second sieve is more efficient in practice. To support this, we analyzed both parallel sieves according to a more "realistic" model of parallel computation, the Block PRAM (BPRAM) of Aggarwal *et al.* (1989). In addition to the parameters $n$ (the input size) and $p$ (the number of processors), the BPRAM has a parameter $l$ which is a measure of communication latency. A running time cost of $l$ is charged for copying a block of memory from the shared global memory to private memory or vice versa. We show that if $l = \Omega(\log\log n)$, then on the BPRAM, our second sieve performs no more work than our first sieve. If $l$ is much larger (say $l = \log n$), then our second sieve is in fact superior. This phenomenon is due to the very fine grained parallelism used by our first prime number sieve.

The remainder of this paper is divided into five sections. After we cover some preliminaries in Section 2, we discuss our two parallel prime number sieves in Sections 3 and 4. We then analyze the sieves on the BPRAM model in Section 5, and conclude with a few remarks in Section 6.

## 2. PRELIMINARIES

Our algorithms use the algebraic PRAM model of computation with an $O(\log n)$ bound on the word size. Essentially, this model consists of a potentially infinite number of synchronous processors, each of which can read and write into a shared memory, perform indirect addressing, and compute the sum, product, difference, or integer quotient of two $O(\log n)$-bit integers in constant time. There are various conventions for how simultaneous accesses to the individual words of shared memory are handled. In the Exclusive Read, Exclusive Write (EREW) model, simultaneous access to a single word of memory is not permitted. In the Concurrent Read, Exclusive Write (CREW) model, simultaneous reads are permitted, but simultaneous writes are not. In the Concurrent Read, Concurrent Write (CRCW) model, both simultaneous reads and writes are permitted. In the latter case, two of the popular conventions for dealing with write conflicts are the *common* CRCW PRAM, in which concurrent writes are allowed only if all processors that write to a cell write the same value, and the *priority* CRCW PRAM, in which the processor with the lowest processor identification number has its value written to the contested cell. Each processor knows its own identification number.

We make use of some well-known results from the literature (for a survey and detailed references, see Karp and Ramachandran (1990) and Parberry (1987)). These include the following:

1. Any CRCW PRAM algorithm using $p(n)$ processors may be simulated by an EREW PRAM using the same number of processors but with a multiplicative increase of $O(\log p(n))$ in the running time.

2. Any PRAM algorithm with $t(n)$ running time and $p(n)$ processors may use fewer processors while performing the same amount of work (simply have each real processor simulate multiple virtual processors). If only $f(n)$ processors are available, the running time increases to $O(t(n)\, p(n)/f(n))$.

3. If "$\circ$" is an associative binary operation, the *parallel prefix problem* is the following. Given $x_1, ..., x_n$, compute $y_1, ..., y_n$, where $y_1 = x_1$ and for $2 \leqslant i \leqslant n$, $y_i = y_{i-1} \circ x_i$. The parallel prefix problem can be solved in time $O(\log n)$ on an EREW PRAM with $O(n/\log n)$ processors.

4. If $\Omega(\sqrt{n})$ processors are available, then $\lfloor \sqrt{n} \rfloor$ can be computed in time $O(1)$ by a CREW PRAM (and hence in time $O(\log n)$ on an EREW PRAM).

For most of the steps in our algorithms, it is a simple matter for a processor to determine what task it is to perform. Usually, this involves determining an array location or integer value as a function of the

processor's identification number. However, occasionally the association of a processor with its task (processor allocation) is not obvious. In this case, we explicitly include a step in the algorithm for the allocation of processors. This is done by specifying how many processors must be allocated to an array location or an integer value. With this information, a table is constructed using parallel prefix computations that allows each processor to determine its task. We leave the details to the reader.

If $r$ is a real number, let $\lfloor r \rfloor$ denote the largest integer not exceeding $r$. Let $a$ and $b$ be positive integers. We define $a \bmod b$ to be the remainder when $a$ is divided by $b$, or $a \bmod b = a - \lfloor a/b \rfloor \cdot b$. We say that $b \mid a$ ($b$ divides $a$) if $a \bmod b = 0$. An integer $p > 1$ is *prime* if the only positive integers that divide $p$ are itself and 1. The letters $p$ and $q$ typically denote primes, and $p_k$ denotes the $k$th prime, with $p_1 = 2$. (We occasionally use $p$ to denote the number of processors; these instances are clearly noted, and there should be no confusion.)

We use the following results from number theory (see Hardy and Wright (1979)). Let $\pi(x)$ denote the number of primes $\leqslant x$. Let $\log x$ denote the natural logarithm of $x$. The Prime Number Theorem states that $\pi(x) \sim x/\log x$. As corollaries, we have the following estimates involving primes (the last estimate is Mertens' Theorem):

$$\sum_{p \leqslant x} \log p \sim x,$$

$$\sum_{p \leqslant x} \frac{1}{p} = \log \log x + O(1),$$

$$\prod_{p \leqslant x} \left(1 - \frac{1}{p}\right) = O\left(\frac{1}{\log x}\right).$$

Two positive integers $a$ and $b$ are *relatively prime* if the largest integer that divides both $a$ and $b$ is 1. This is written $(a, b) = 1$. Euler's totient function $\phi(m)$ counts the number of positive integers $\leqslant m$ that are relatively prime to $m$, and satisfies the following identity:

$$\phi(m) = m \cdot \prod_{p \mid m} \left(1 - \frac{1}{p}\right).$$

Note that if $m$ is the product of the first $k$ primes, then $\log m = \sum_{p \leqslant p_k} \log p \sim p_k$ and so, using Mertens' Theorem, $\phi(m) = O(m/\log p_k) = O(m/\log \log m)$.

We make use of the following elementary parallel algorithm.

THEOREM 2.1. *On input a positive integer $n$, a list of primes up to $n$ can be computed on an EREW PRAM in $O(\log n)$ time using $O(n^{3/2})$ processors.*

*Proof.* For each $x \leqslant n$ in parallel, simply trial divide $x$ in parallel by $2, 3, \ldots, \min\{x-1, \lfloor\sqrt{n}\rfloor\}$. If a divisor is found, then $x$ is not prime. This takes $O(1)$ time and $O(n^{3/2})$ processors (one for each divisor for each candidate) on a CRCW PRAM, and $O(\log n)$ time on an EREW PRAM with the same number of processors. ∎

It is a simple matter to reduce the processor bound to $O(n^{3/2}/\log n)$ by trial dividing by primes only, but Theorem 2.1 is sufficient for our purposes. Also note that the set of primes up to $n$ can be computed sequentially using trial division, in $O(\log n)$ space on a Turing machine or RAM or any other reasonable model of sequential computation.

We conclude our preliminary material with a discussion of the *wheel*, a simple but useful data structure for prime number sieves pioneered for sequential sieving algorithms by Pritchard (1981, 1982, 1983). It is used in manipulating the set of integers relatively prime to the first $k$ primes. Let $m$ be the product of the first $k$ primes (for example, if $k=3$ then $m = 2 \cdot 3 \cdot 5 = 30$). Then the $k$th wheel $W_k[\ ]$ is an array of length $m$ (indexed by $0 \cdots m-1$), where each array element consists of four fields:

$W_k[x].rp$: This is 1 if $x$ is relatively prime to $m$, and 0 otherwise.

$W_k[x].dist$: This is $d$, where $x+d$ is the smallest integer $>x$ that is relatively prime to $m$.

$W_k[x].pos$: If $x$ is relatively prime to $m$, this is the number of integers $\leqslant x$ that are relatively prime to $m$. Otherwise, it is 0.

$W_k[x].inv$: If $x=0$, this is $-1$. If $1 \leqslant x < \phi(m)$, it is the $x$th integer relatively prime to $m$. If $x \geqslant \phi(m)$, it is 0.

For example, Table 1 shows the second wheel ($m=6$), and Table 2 shows the third wheel ($m=30$).

The $k$th wheel has the following two useful properties:

- $x$ is relatively prime to $m$ (and hence the first $k$ primes) iff $W_k[x \bmod m].rp = 1$.

TABLE 1

The Second Wheel, $W_2$, for Which $m=6$

| $W_2[\ ]$: | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| *rp* | 0 | 1 | 0 | 0 | 0 | 1 |
| *dist* | 1 | 4 | 3 | 2 | 1 | 2 |
| *pos* | 0 | 1 | 0 | 0 | 0 | 2 |
| *inv* | −1 | 1 | 0 | 0 | 0 | 0 |

TABLE 2

The Third Wheel, $W_3$, for Which $m = 30$

| $W_3[\ ]$: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *rp* | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| *dist* | 1 | 6 | 5 | 4 | 3 | 2 | 1 | 4 | 3 | 2 | 1 | 2 | 1 | 4 | 3 |
| *pos* | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 3 | 0 | 4 | 0 |
| *inv* | −1 | 1 | 7 | 11 | 13 | 17 | 19 | 23 | | | | | | | |
| $W_3[\ ]$: | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 |
| *rp* | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| *dist* | 2 | 1 | 2 | 1 | 4 | 3 | 2 | 1 | 6 | 5 | 4 | 3 | 2 | 1 | 2 |
| *pos* | 0 | 0 | 5 | 0 | 6 | 0 | 0 | 0 | 7 | 0 | 0 | 0 | 0 | 0 | 8 |

• The smallest integer larger than $x$ relatively prime to $m$ is $x + W_k[x \bmod m].dist$.

Let $t_k(x)$ be the number of integers $\leqslant x$ that are relatively prime to $m$. Then if $(x, m) = 1$ we have

$$t_k(x) = \phi(m) \cdot \lfloor x/m \rfloor + W_k[x \bmod m].pos,$$

and in general,

$$t_k(x) = \phi(m) \cdot \lfloor x/m \rfloor + W_k[(x + W_k[x \bmod m].dist) \bmod m].pos - 1.$$

Note that $t_k$ is invertible on the set of integers relatively prime to $m$. $t_k$'s inverse, $t_k^{-1}(x)$, gives the $x$th positive integer relatively prime to $m$:

$$t_k^{-1}(x) = m \cdot \lfloor x/\phi(m) \rfloor + W_k[x \bmod \phi(m)].inv.$$

Both $t_k$ and $t_k^{-1}$ are computable in constant time, and may be used to reduce the storage requirement of a prime number sieve as follows. Let $s[\ ]$ be a bit vector of length $h = t_k(n) = O((\phi(m)/m) \cdot n)$. Then $s[\ ]$ can represent a subset $S$ of the integers $\leqslant n$ that are relatively prime to $m$ by setting $s[t_k(x)]$ to 1 iff $x \in S$ for $(x, m) = 1$.

If $S$ is the set of primes up to $n$ that are relatively prime to $m$, then a list of all primes up to $n$ can be constructed by using a parallel prefix computation, applying $t_k^{-1}$, and inserting the first $k$ primes. Assuming that the $k$th wheel and the first $k$ primes are available, this implies that if any of the following three problems can be solved in $O(\log n)$ time using $O(h/\log h)$ processors on an EREW PRAM, then they all can:

(1) List all the prime numbers $\leqslant n$.

(2) Construct a bit string of length $h$ representing the set of prime numbers $\leqslant n$ relatively prime to $m$.

(3) Compute $\pi(x)$ for all integers $x$, with $1 \leqslant x < n$ and $(x, m) = 1$.

We leave verification of this to the reader.

The following result explains how to compute wheels efficiently in parallel:

LEMMA 2.2. *The first $k$ primes, $m = p_1 p_2 \cdots p_k$, and the $k$th wheel $W_k[\ ]$ can be constructed by an EREW PRAM algorithm in $O(\log m)$ time using $O(m \log m / \log\log m)$ processors.*

*Proof.* Recall that $\log m \sim p_k$, and note that by the prime number theorem, $p_k = \Theta(k \log k)$, so $k = O(\log m / \log\log m)$. Computing the first $k$ primes takes $O(\log k)$ time using $k^{O(1)}$ processors (by Theorem 2.1). Then $m$ can be computed in $O(\log k)$ time using $O(k)$ processors in a binary tree arrangement.

To compute the *rp* field, first write a 1 in each array location in parallel. Then for each $x < m$ in parallel, divide $x$ by each of the first $k$ primes. If a divisor is found, write a 0 in position $x$. This takes $O(1)$ time using $O(mk)$ processors on a common CRCW PRAM. Simulating this on an EREW PRAM takes $O(\log mk) = O(\log m)$ time with the same processor count.

Using parallel prefix, set the *pos* field in position $x$ to the sum of the *rp* fields of all positions $\leqslant x$. At this point, $W_k[x].pos$ gives the number of integers $\leqslant x$ that are relatively prime to $m$, for all $x < m$. (This field is set to 0 for those $x$ with $(x, m) > 1$ at a later step.) This takes $O(\log m)$ time using $O(m/\log m)$ processors.

The *inv* field is computed as follows. For each $x < m$, initialize $W_k[x].inv$ to 0, and set $W_k[0].inv$ to $-1$. Then, for each $x \leqslant m-1$, if $W_k[x].rp$ is 1, then write $x$ in position $W_k[x].pos$. ($W_k[m-1].inv$ is set to 0 in the following step.)

To compute the *dist* field, $W_k[m-1].dist$ is set to 2, and for every $x < m-1$, set $W_k[x].dist$ to $W_k[W_k[x].pos+1].inv - x$. Finally, if $W_k[x].rp = 0$, set $W_k[x].pos$ to 0, and set $W_k[m-1].inv$ to 0. This takes $O(1)$ time using $O(m)$ processors on a common CRCW PRAM. Simulating this on an EREW PRAM takes $O(\log m)$ time with the same processor count. ∎

## 3. AN OPTIMAL PARALLEL ALGORITHM

In this section we describe the first of our two parallel prime number sieves. This parallel sieve is optimal (as defined in Section 2), and it is

based on a sequential sieve due to Pritchard. We begin by reviewing Pritchard's sieve.

### 3.1. *Pritchard's Linear Sieve*

Pritchard's linear sieve (1987, Algorithm 3.3) uses $O(n)$ arithmetic operations to compute a bit vector $s[\ ]$ that represents the set of primes up to $n$. This linear sieve exploits the fact that every integer $x$ can be uniquely represented in the form $x = pf$, where $p$ is the *least prime factor* of $x$ and $f = x/p$. We write $\text{lpf}(x)$ for the least prime factor of $x$. Note that $\text{lpf}(x) = x$ if and only if $x$ is prime, and if $x$ is composite then $\text{lpf}(x) \leqslant \sqrt{x}$. The algorithm proceeds by first initializing $s[\ ]$ to all 1's, and then every unique representation $pf$ up to $n$ is generated and $s[pf]$ is set to 0. This is done by looping over all valid $f$'s, and for each $f$, looping over all primes $p$ such that $p \leqslant \text{lpf}(f)$ and $p \leqslant n/f$. Algorithm P below gives the details.

ALGORITHM P.
  Input: a positive integer $n$.
    Find the primes up to $\lfloor \sqrt{n} \rfloor$;
    For $x := 2$ to $n$ do:
      $s[x] := 1$;
    For $f := 2$ to $n/2$ do:
      For each prime $p \leqslant \text{lpf}(f)$ with $pf \leqslant n$ do:
        $s[pf] := 0$;

Note that to determine the primes $p \leqslant \text{lpf}(f)$ it is not necessary to explicitly compute $\text{lpf}(f)$; simply use the primes $p = 2, 3, \ldots$ until $p \mid f$. For details, see the paper by Pritchard (1987).

Since each composite integer is "crossed off" the bit vector $s[\ ]$ exactly once, and the number of composite integers up to $n$ is at most $n$, the running time of Algorithm P is $O(n)$. By using a wheel of size $m = n^c$ where $0 < c \leqslant \frac{1}{2}$, this can be reduced to $O(n/\log\log n)$. Below is Algorithm $\text{P}_W$, which is Algorithm P using a wheel of size $m \approx n^{1/4}$.

ALGORITHM $\text{P}_W$.
  Input: a positive integer $n$.
    Find the primes up to $\lfloor \sqrt{n} \rfloor$;
    Choose $k$ maximal such that $p_k \leqslant (\log n)/4$;
    Compute $m = p_1 p_2 \cdots p_k = n^{1/4 + o(1)}$ and the $k$th wheel $W_k[\ ]$;
    For $x := p_{k+1}$ to $n$ step $W_k[x \bmod m].dist$ do:
      $s[t_k(x)] := 1$;
    For $f := p_{k+1}$ to $n/p_{k+1}$ step $W_k[f \bmod m].dist$ do:
      For each prime $p$, $p_k < p \leqslant \text{lpf}(f)$ with $pf \leqslant n$ do:
        $s[t_k(pf)] := 0$;

At the conclusion of Algorithm $P_W$, the vector $s[\ ]$ represents the set of primes up to $n$ that are relatively prime to $m$ (see Section 2). To list all the primes up to $n$, the vector $s[\ ]$ must be converted to a list of primes and the first $k$ primes must be added. We leave this detail to the reader.

Algorithm $P_W$ uses $O((\phi(m)/m)\cdot n)$ arithmetic operations. By the choice of $k$, this is $O(n/\log\log n)$ (again, see Section 2).

### 3.2. *A Parallel Version*

The main difficulty in parallelizing Pritchard's algorithm lies in determining a list of primes for each value of $f$ in the main loop. It is possible to overcome this difficulty by first computing $\mathrm{lpf}(f)$ for all $f \leqslant n/p_{k+1}$.

As a first step we present Algorithm 3.1 which computes $\mathrm{lpf}(x)$ for all $x \leqslant n$. The algorithm stores these values in an array $a[\ ]$ of length $n$. The method used is essentially a parallel version of the Sieve of Eratosthenes.

ALGORITHM 3.1.

Input: a positive integer $n$.

1. Compute $\lfloor\sqrt{n}\rfloor$ and a list of primes up to $\sqrt{n}$;
2. Allocate $\lfloor n/p\rfloor$ processors to each prime $p \leqslant \sqrt{n}$;
   (to be used in Step 4)
3. For $x := 1$ to $n$ in parallel do:
   $a[x] := x$;
4. For each prime $p \leqslant \sqrt{n}$ in parallel do:
   For $f := p$ to $\lfloor n/p\rfloor$ in parallel do:
   $a[pf] := p$ *with priority* $p$;

LEMMA 3.1. *Algorithm* 3.1 *computes* $a[x] = \mathrm{lpf}(x)$ *for all* $x \leqslant n$ *in* $O(\log n)$ *time using* $O(n \log\log n)$ *processors on an EREW PRAM.*

*Proof.* In Step 3 of the algorithm, $a[x]$ is set to $x$. If $x$ is prime, then $a[x]$ is not changed in Step 4, and so $a[x] = \mathrm{lpf}(x)$. If $x$ is composite, in Step 4 $a[x]$ is overwritten concurrently by each prime $p \leqslant \sqrt{n}$ dividing $x$, and since the smallest such $p$ has highest priority, we have $a[x] = \mathrm{lpf}(x)$.

Step 1 takes $O(\log n)$ time using $O(n^{3/4})$ processors (by Theorem 2.1). Step 2 can be done using parallel prefix computations in $O(\log n)$ time using $O(n/\log n)$ processors. Step 3 takes $O(1)$ time using $O(n)$ processors. Finally, Step 4 takes $O(1)$ time using $\sum_{p \leqslant \sqrt{n}} (n/p + O(1)) = O(n \log\log n)$ processors on a priority CRCW PRAM, which can be simulated on an EREW PRAM in $O(\log n)$ time with the same processor count (see Section 2). ∎

Next, we assume that $\mathrm{lpf}(x)$ is computed for all $x \leqslant n/p_{k+1}$. Algorithm 3.2 computes an array $b[\ ]$ of length $n$ such that $b[x] = \mathrm{lpf}(x)$ for all $x \leqslant n$ with $(x, m) = 1$. Using the invertible mapping $t_k$ discussed in Section 2,

$b[\ ]$ needs only $O((\phi(m)/m)\cdot n)$ memory cells. We assume that $t_k$ and its inverse are being used without being explicit.

ALGORITHM 3.2.

Input: positive integers $n$, $k$, and $\text{lpf}(x)$ for $x \leqslant n/p_{k+1}$ with $(x, m) = 1$.

1. Compute $m = p_1 p_2 \cdots p_k$ and the $k$th wheel $W_k[\ ]$;
2. Compute $\pi(x)$ for $2 \leqslant x \leqslant n/p_{k+1}$ and $(x, m) = 1$, and from this construct a list of primes up to $n/p_{k+1}$;
3. Allocate $\min\{\pi(\text{lpf}(f)), \pi(n/f)\}$ processors to each $f \leqslant \lfloor n/p_{k+1} \rfloor$ with $(f, m) = 1$;
   (to be used in Step 5)
4. For $x := 2$ to $n$ with $(x, m) = 1$ in parallel do:
   $b[x] := x$;
5. For $f := p_{k+1}$ to $\lfloor n/p_{k+1} \rfloor$ with $(f, m) = 1$ in parallel do:
   For each prime $p$, $p_k < p \leqslant \min\{\text{lpf}(f), \lfloor n/f \rfloor\}$ in parallel do:
   $b[pf] := p$;

LEMMA 3.2. *Assume $k$ is small enough so that $m = O(n/\log^3 n)$. On input positive integers $n$, $k$, and $\text{lpf}(x)$ for all $x \leqslant n/p_{k+1}$ with $(x, m) = 1$, Algorithm 3.2 computes $\text{lpf}(x)$ for all $x \leqslant n$ with $(x, m) = 1$ in $O(\log n)$ time using $O(n/(\log n \log p_k))$ processors on an EREW PRAM.*

*Proof.* After Step 4, we have $b[x] = x$ for all $x \leqslant n$ with $(x, m) = 1$. So if $x$ is prime, we have $b[x] = \text{lpf}(x)$. If $x$ is composite, then $x = pf$ where $p = \text{lpf}(x)$ and $f = x/p$. Since $x$ is relatively prime to $m$, $p$ and $f$ must be as well, and $p > p_k$. Therefore, in Step 5, for every composite integer $x \leqslant n$ with $(x, m) = 1$, $b[x]$ is set to $p$, where $p = \text{lpf}(x)$. This implies that the algorithm is correct.

Step 1 takes $O(\log m)$ time using $O(m \log m/\log\log m)$ processors by Lemma 2.2. By the bound on $m$, this is $O(\log n)$ time using $O(n/(\log^2 n \log\log n)) = O(n/\log^2 n)$ processors. Using parallel prefix computations, Steps 2 and 3 take $O(\log n)$ time using $O((\phi(m)/m)\cdot n/\log n)$ processors. Since $\phi(m)/m = O(1/\log p_k)$, this is $O(n/(\log n \log p_k))$ processors. Step 4 takes $O(1)$ time when using $O(n/\log p_k)$ processors. By using only $O(n/(\log n \log p_k))$ processors, this takes $O(\log n)$ time. Finally, Step 5 takes $O(1)$ time when using one processor for each composite integer of the form $pf$, where $(f, m) = 1$ and $p > p_k$, which is $O((\phi(m)/m)\cdot n) = O(n/\log p_k)$ processors. Again, by using only $O(n/(\log n \log p_k))$ processors, this takes $O(\log n)$ time. Combining the times and processor counts for all steps completes the proof. ∎

Our main result of this section is Algorithm 3.3, which combines Algorithms 3.1 and 3.2 to compute the set of primes up to $n$ in $O(\log n)$ time using only $O(n/(\log n \log\log n))$ processors.

ALGORITHM 3.3.

Input: positive integer $n$.

1. Choose $k$ maximal such that $p_k \leqslant (\log n)/4$;
   Let $m = p_1 p_2 \cdots p_k$;
2. Using Algorithm 3.1, compute $\text{lpf}(x)$ for all $x \leqslant n/p_{k+1}^2$;
3. Using the output from the previous step and Algorithm 3.2, compute $\text{lpf}(x)$ for all $x \leqslant n/p_{k+1}$ with $(x, m) = 1$;
4. Again, using the output from the previous step and Algorithm 3.2, compute $\text{lpf}(x)$ for all $x \leqslant n$ with $(x, m) = 1$;
   Mark $x$ as prime if $\text{lpf}(x) = x$;
5. Use parallel prefix to construct a list of primes up to $n$ that are relatively prime to $m$, and add $p_1, \ldots, p_k$ to this list;

THEOREM 3.3. *On input a positive integer $n$, Algorithm 3.3 computes a list of primes up to $n$ in $O(\log n)$ time using $O(n/(\log n \log\log n))$ processors on an EREW PRAM.*

*Proof.* Correctness follows from Lemmas 3.1 and 3.2.

Using Algorithm P to find the primes below $\log n$, Step 1 takes $O(\log n)$ time with one processor. Step 2 takes $O(\log n)$ time using $O((n/p_{k+1}^2) \log\log n) = O(n \log\log n/(\log n)^2)$ processors by Lemma 3.1. Steps 3 and 4 take $O(\log n)$ time using $O(n/(\log n \log p_k)) = O(n/(\log n \log\log n))$ processors. Step 5 takes $O(\log n)$ time using $O((\phi(m)/m) \cdot n/\log n) = O(n/(\log n \log\log n))$ processors. ∎

Note that an $O(\log n/\log\log n)$ time algorithm for the CRCW PRAM with the same work bound can be obtained by using the algorithm of Cole and Vishkin (1989) for the parallel prefix computations in Algorithms 3.1, 3.2, and 3.3.

## 4. A PRACTICAL PARALLEL ALGORITHM

In this section, we present a parallel prime number sieve with an $O(\sqrt{n})$ running time using $O(\sqrt{n})$ processors. This parallel algorithm is a straightforward parallelization of a segmented version of the Sieve of Eratosthenes using a wheel. This algorithm appeared previously without formal analysis in Parberry (1981).

The basic idea is as follows. The interval from 1 to $n$ is broken down into segments of length $\Delta = \lceil n/p \rceil$, where $p$ is the number of processors. Each processor finds all the primes in its assigned interval using the Sieve of Eratosthenes. With the addition of a wheel to speed the sieve, this takes

each processor $O(\Delta)$ time, assuming that the intervals are sufficiently large. Note that this same idea, when applied to sequential algorithms, leads to segmented sieves that use only $O(\sqrt{n})$ work space. Using the invertible map mentioned in Section 2, the work space can be further reduced to $O(\sqrt{n}/\log\log n)$. For sequential versions of this algorithm, we refer the reader to the papers of Bays and Hudson (1977) and Pritchard (1983).

The details are given in Algorithm 4.1 below. This algorithm uses a globally shared bit array $s[\ ]$ of length $n$, which at the conclusion of the algorithm represents the set of primes up to $n$. We leave the details involved in listing the primes up to $n$ using the vector $s[\ ]$ to the reader.

ALGORITHM 4.1.

Input: positive integers $k$, $n$.

Note: one processor executes the Precomputation and Cleanup phases; every processor executes the Main Loop in parallel.

*Precomputation (sequential)*:

1. Find the first $k$ primes $p_1, \ldots, p_k$, their product $m$, and compute the $k$th wheel $W_k[\ ]$;
2. Find the primes up to $\sqrt{n}$;
3. $\Delta := \lceil n/p \rceil$, where $p$ is the number of processors;

*Main Loop (parallel)*:

4. $l := \mathrm{id} \cdot \Delta + 1$ and $r := \min\{(\mathrm{id}+1)\Delta, n\}$, where id is the processor's identification number, $0 \leqslant \mathrm{id} < p$;
5. For $x := l$ to $r$ do:
   $s[x] := W_k[x \bmod m].rp$;
6. For each prime $q$, $p_k < q \leqslant \sqrt{n}$ do:
   *Compute the smallest $f$ so that $(qf, m) = 1$ and $qf \geqslant \max\{l, q^2\}$*:
   $f := \max\{q-1, \lceil l/q \rceil - 1\}$;
   $f := f + W_k[f \bmod m].dist$;
   *Remove the multiples of $q$*:
   While $qf \leqslant r$ do:
   $s[qf] := 0$;
   $f := f + W_k[f \bmod m].dist$;

*Cleanup (sequential)*:

7. For $i := 1$ to $k$ do:
   $s[p_i] := 1$;
8. $s[1] := 0$;

THEOREM 4.1. *Let $k$ be maximal so that $p_k \leqslant (\log n)/4$. Then on inputs $n$ and $k$, Algorithm 4.1 computes the set of primes up to $n$ in $O(n/p)$ time using $p = O(\sqrt{n})$ processors on an EREW PRAM.*

*Proof.* We begin by demonstrating the correctness of the algorithm. The proof is broken down according to whether or not $(x, m) = 1$. If $x$ is not relatively prime to $m$, then in Step 5, $s[x]$ is set to 0. If $x$ is not prime, $s[x]$ remains 0. If $x$ is prime, then $x$ must be one of the first $k$ primes, in which case $s[x]$ is set to 1 in Step 7. Thus the algorithm sets $s[x]$ correctly when $x$ is not relatively prime to $m$. If $(x, m) = 1$ and $x > 1$, then in Step 5, $s[x]$ is initialized to 1. If $x$ is not prime, then $x$ has a prime divisor $q$ with $p_k < q \leqslant \sqrt{n}$, and $x/q$ must be relatively prime to $m$. Hence in Step 6 an $f$ is found such that $qf = x$, and $s[x]$ is set to 0. If $x$ is prime, $s[x]$ remains 1. Thus the algorithm sets $s[x]$ correctly when $(x, m) = 1$, completing the proof that on termination of Algorithm 4.1, $s[x] = 1$ iff $x$ is prime, for all $2 \leqslant x \leqslant n$.

The time and processor requirements of the algorithm are as follows.

*Precomputation.* Using a linear sequential prime number sieve (such as Algorithm P) and Lemma 2.2 with one processor, this phase takes $O(\sqrt{n})$ time.

*Main Loop.* Step 4 takes $O(1)$ time, and Step 5 takes $O(\Delta) = O(n/p)$ time. For each prime $q$, the body of the loop in Step 6 takes time $O(1) + O((\phi(m)/m) \cdot \Delta/q)$. Summing over all the primes $\leqslant \sqrt{n}$, this gives a running time of $O(\sqrt{n}/\log n + (\phi(m)/m) \cdot \Delta \log\log n)$ (see Section 2). By our choice of $k$, $\phi(m)/m = O(1/\log\log n)$, giving a running time of $O(\sqrt{n}/\log n + n/p)$ for Step 6.

*Cleanup.* This takes $O(k) = O(\log m) = O(\log n)$ time.

The total running time for all phases is $O(n/p + \sqrt{n})$ which is $O(n/p)$ for $p = O(\sqrt{n})$. ∎

Note that if Step 2 were done in parallel (say using Theorem 2.1), then Algorithm 4.1's running time could be reduced to $O(\sqrt{n}/\log n)$ using $O(\sqrt{n}\log n)$ processors.

## 5. Analysis on the Block PRAM

When we compare Algorithm 3.3 to Algorithm 4.1, there are two things we observe:

1. Algorithm 3.3 is theoretically more efficient than Algorithm 4.1; it performs less work by a factor of $\Theta(\log\log n)$, and it permits more processors for a running time as low as $O(\log n)$.

2. Algorithm 3.3 uses very fine-grained parallelism in its main loop, whereas Algorithm 4.1 does not. Since the current generation of parallel

computers does not support complex, fine-grained parallelism efficiently, it follows that in practice Algorithm 3.3 requires significantly more overhead than Algorithm 4.1, and as a result Algorithm 4.1 may actually be more efficient.

In this section our goal is to formalize our second observation by analyzing both algorithms on a more "realistic" model of parallel computation: the Block PRAM (or BPRAM) of Aggarwal *et al.* (1989).

The BPRAM is an EREW PRAM with two modifications. The first modification is to memory: in addition to the infinite, globally shared memory of the standard PRAM, each processor has an infinite local memory which is only accessible to that processor. The only operations permitted on the global memory are copying blocks of global memory to local memory and vice versa. Each processor does all its work locally. The second modification is the addition of a parameter $l$ that models the communication latency inherent in a globally shared memory. For a single processor to copy a block of $n$ memory cells to or from global memory takes time $l+n$. Thus, the running time of a BPRAM algorithm depends on three parameters: the input size $n$, the number of processors $p$, and the communication latency parameter $l$.

The value of $l$ as a function of $p$ (or $n$) is implementation dependent. For example, if the access to the shared memory is provided by a shared bus, then $l=\Theta(p)$ is appropriate, while if access to the shared memory is by an interconnection network, then $l=\Theta(\log p)$ is appropriate. If $l=O(1)$, then the analysis on the BPRAM yields the same results as the standard PRAM. The analysis of Algorithms 3.3 and 4.1 on a BPRAM follows.

*Algorithm* 3.3. Algorithm 3.3's running time is $O((n/\log\log n)/p)$ for $p=O(n/(\log n \log\log n))$ on the standard EREW PRAM. In Step 6 of the second invocation of Algorithm 3.2, each processor writes to $\Theta((n/\log\log n)/p)$ global memory locations, no two of which are adjacent, so that the time for this step on the BPRAM is $\Theta(l(n/\log\log n)/p)$. Since a multiplicative factor of $l$ is the most an algorithm's running time can increase from the PRAM to BPRAM, the total time on the BPRAM is $\Theta(l(n/\log\log n)/p)$ using $p=O(n/(\log n \log\log n))$ processors.

When $l=\Theta(p)$, all parallel speedup is cancelled, giving a running time of $O(n/\log\log n)$ no matter how many processors are used. When $l=\Theta(\log p)=O(\log n)$, the running time is $O(n \log n/(p \log\log n))$.

*Algorithm* 4.1. On the standard EREW PRAM, Algorithm 4.1's running time is $O(n/p)$ for $p=O(\sqrt{n})$. If each processor copies its interval of length $\Delta$ into local memory before the main loop and back to global memory afterwards, the total number of accesses to global memory is constant for each processor. This gives a total running time of $O(n/p+l)$ for $p=O(\sqrt{n})$ processors.

When $l = \Theta(p)$, the running time is $O(n/p + p) = O(n/p)$ when $p = O(\sqrt{n})$. Thus, the running time of Algorithm 4.1 is not particularly sensitive to $l$.

Note that if $l = \Theta(\log \log n)$, then the total work done by both algorithms is the same, namely $O(n)$. If $l$ becomes significantly larger than this, then Algorithm 4.1 is more efficient in its total work. We conclude that Algorithm 4.1 is probably much more efficient in practice.

## 6. Conclusion and Open Problems

We have presented two parallel prime number sieves for the EREW PRAM: Algorithm 3.3 with optimal work $O(n/\log \log n)$ and a more practical Algorithm 4.1 with work $O(n)$. Algorithm 4.1 is in a sense more practical than Algorithm 3.3. A similar result is true about two closely related sequential algorithms (see Sorenson (1991)).

Our PRAM model used the *full arithmetic* instruction set of Parberry (1987). This could be viewed as too powerful an instruction set, were it not for the fact that we have limited the word size of the shared and local memory. (For more discussion of instruction sets on PRAMs with unbounded word size, see Parberry (1986, 1987), and Trahan *et al.* (1988, 1989)). It is common to define a PRAM with the fundamental operations at the bit level (see, for example, Goldschlager (1982)). Under this model, Algorithm 3.3 is no longer optimal; Pritchard gave a sequential prime number sieve which utilizes a dynamic wheel and takes only $O(n/\log \log n)$ additions and subtractions (see Pritchard (1981, 1982)). Algorithms $P_W$ and 3.3 use a large number of multiplications. It is an open problem whether it is possible to remove the multiplications from Algorithm 3.3 without a significant performance cost, or to parallelize Pritchard's dynamic wheel sieve efficiently to yield an optimal algorithm. One of the stumbling blocks in a parallelization of Pritchard's dynamic wheel sieve is that each iteration of the algorithm's main loop uses a data structure modified by every previous loop iteration.

Received August 7, 1991; final manuscript received April 28, 1992

## References

Aggarwal, A., Chandra, A. K., and Snir, M. (1989), On communication latency in PRAM computations, *in* "Proceedings, 1989 ACM Symposium on Parallel Algorithms and Architectures," pp. 11–21.

Bays, C., and Hudson, R. (1977), The segmented sieve of Eratosthenes and primes in arithmetic progressions to $10^{12}$, *BIT* **17**, 121–127.

BOKHARI, S. H. (1987), Multiprocessing the sieve of Eratosthenes, *IEEE Comput.* **20**(4), 50–58.

COLE, R., AND VISHKIN, U. (1989), Faster optimal parallel prefix sums and list ranking, *Inform. and Control* **81**, 334–352.

COSNARD, M., AND PHILIPPE, J.-L. (1989), Discovering new parallel algorithms: The sieve of Eratosthenes revisited, *in* "Computer Algebra and Parallelism" (J. D. Dora and J. Fitch, Eds.), pp. 1–18, Academic Press, San Diego.

GOLDSCHLAGER, L. M. (1982), A universal interconnection pattern for parallel computers, *J. Assoc. Comput. Mach.* **29**(4), 1073–1086.

HARDY, G. H., AND WRIGHT, E. M. (1979), "An Introduction to the Theory of Numbers," 5th ed., Oxford Univ. Press, London/New York.

HIKITA, T., AND KAWAI, S. (1980), Parallel sieve methods for generating prime numbers, *in* "Information Processing '80, Proceedings of the 6th International IFIP Computing Congress," pp. 256–262, North-Holland, Amsterdam.

KARP, R., AND RAMACHANDRAN, V. (1990), Parallel algorithms for shared-memory machines, *in* "Algorithms and Complexity" (J. van Leeuwen, Ed.), Handbook of Theoretical Computer Science, Vol. A, Elsevier, Amsterdam, and MIT Press, Cambridge, MA.

MAIRSON, H. G. (1977), Some new upper bounds on the generation of prime numbers, *Comm. ACM* **20**(9), 664–669.

PARBERRY, I. (1981), "Parallel Speedup of Sequential Prime Number Sieves," Technical Report TR30, University of Queensland.

PARBERRY, I. (1986), Parallel speedup of sequential machines: A defense of the parallel computation thesis, *SIGACT News* **18**(1), 54–67.

PARBERRY, I. (1987), "Parallel Complexity Theory," Research Notes in Theoretical Computer Science, Pitman, London.

PRITCHARD, P. (1981), A sublinear additive sieve for finding prime numbers, *Comm. ACM* **24**(1), 18–23, 772.

PRITCHARD, P. (1982), Explaining the wheel sieve, *Acta Informat.* **17**, 477–485.

PRITCHARD, P. (1983), Fast compact prime number sieves (among others), *J. Algorithms* **4**, 332–344.

PRITCHARD, P. (1987), Linear prime-number sieves: A family tree, *Sci. Comput. Programming* **9**, 17–35.

SORENSON, J. (1991), "An Analysis of Two Prime Number Sieves," Technical Report # 1028, Computer Science Department, University of Wisconsin at Madison.

TRAHAN, J. L., LOUI, M. C., AND RAMACHANDRAN, V. (1988), Multiplication, division, and shift instructions in parallel random access machines, *in* "Proceedings, 22nd Conference on Information Sciences and Systems," pp. 126–130.

TRAHAN, J. L., LOUI, M. C., AND RAMACHANDRAN, V. (1989), The power of parallel random access machines with augmented instruction sets, *in* "Proceedings, 4th Structure in Complexity Theory Conference," pp. 97–103.