J. Symbolic Computation (1998) **26**, 509–523
Article No. sy980226

Efficient Algorithms for Computing the Jacobi Symbol[†]

SHAWNA MEYER EIKENBERRY[‡] AND JONATHAN P. SORENSON[§]

Department of Mathematics and Computer Science, Butler University, U.S.A.

We present two new algorithms for computing the Jacobi Symbol: the right-shift and left-shift k-ary algorithms. For inputs of at most n bits in length, both algorithms take $O(n^2/\log n)$ time and $O(n)$ space. This is asymptotically faster than the traditional algorithm, which is based in Euclid's algorithm for computing greatest common divisors. In practice, we found our new algorithms to be about twice as fast for inputs of 100 to 1000 decimal digits in length. We also present parallel versions of both algorithms for the CRCW PRAM. One version takes $O_\epsilon(n/\log\log n)$ time using $O(n^{1+\epsilon})$ processors, giving the first sublinear parallel algorithms for this problem. The other version takes polylog time using a subexponential number of processors.

1. Introduction

In this paper, we present two new algorithms for computing the Jacobi symbol (see Section 2 for a definition). After a brief discussion of some applications, we review the previous work on Jacobi symbol algorithms, including both sequential and parallel computation models, and then we summarize our results.

Solovay and Strassen (1977) observed that one may use the Jacobi symbol to probabilistically test for primality. Specifically, to test the integer m for primality, choose an integer $a \in [2, m-1]$ uniformly at random, and compare (a/m) to $a^{(m-1)/2} \bmod m$. If these do not match (modulo m), then m is composite. Otherwise, m *might* be prime; the probability of a composite number passing as a prime is at most $1/2$. This test can be repeated to reduce the chance of error.

Perhaps the most important application for the Jacobi symbol is in finding quadratic nonresidues. Nonresidues are used in computing square roots modulo a prime (see Bach (1990b, 1991) and Peralta (1986)), in writing a prime as a sum of two squares (Shallit and Rabin, 1986), and in several cryptography schemes that are based on the difficulty of computing square roots modulo a composite number (see, for example, McCurley (1990), Williams (1980, 1986), and Menezes *et al.* (1997)).

[†]Computing equipment provided through a grant from the Holcomb Research Institute. A preliminary version of this paper was presented on November 3, 1995 at the AMS meeting at Kent State University, Kent, Ohio. An extended abstract appeared as Meyer and Sorenson (1996).
[‡]Supported by the Butler Summer Institute.
[§]Supported by a Butler University Faculty Research Fellowship. E-mail: `sorenson@butler.edu`, URL: `http://www.butler.edu/~sorenson/`

0747–7171/98/100509 + 15 $30.00/0

As there are $(p-1)/2$ quadratic nonresidues modulo any odd prime p, to find a nonresidue one simply chooses integers a at random until $(a/p) = -1$. Under the assumption of the Extended Riemann Hypothesis (ERH), the Ankeny–Bach theorem states that there exists a nonresidue a satisfying $a \leq 2\log^2 p$ (Ankeny, 1952; Bach, 1990a). Thus, one may eliminate randomness by assuming the validity of the unproven ERH.

There are several algorithms for computing the Jacobi symbol, including the ordinary algorithm (based on Euclid's GCD algorithm), Eisenstein's algorithm, and Lebesgue's algorithm (based on the least-remainder GCD algorithm); see Shallit (1990) for detailed analyses of these. The ordinary algorithm and Lebesgue's algorithm take $O(\log x \log y)$ bit operations to compute (x/y). Eisenstein's algorithm has an exponential worst-case running time. The more recent binary algorithm (Shallit and Sorenson, 1993) takes $O(\log^2(xy))$ bit operations; this algorithm is probably the most efficient in practice. The asymptotically fastest Jacobi symbol algorithm involves computing the continued fraction expansion of x/y using Schönhage's GCD algorithm (Schönhage, 1971) and extracting the Jacobi symbol from this information (see Gauss (1870), Bach (1990b)). This method takes only $O(n\log^2 n \log\log n)$ bit operations, but is not considered practical. For algorithms that compute cubic and higher residuosity, see Scheidler and Williams (1995).

Work on parallel Jacobi symbol algorithms is not as advanced. A straight-forward parallelization of the binary algorithm yields an $O(n)$ time parallel algorithm for the EREW PRAM. The only known $\mathcal{NC}$ algorithm for evaluating quadratic residuosity (Fich and Tompa, 1988) works only in finite fields, with the additional restriction that the characteristic be bounded by a polynomial in the input size. No $\mathcal{NC}$ algorithm is known for computing the Jacobi symbol, or even for computing GCDs; the question of the existence of an $\mathcal{NC}$ algorithm for GCDs is a well-known open problem in parallel complexity (Greenlaw *et al.*, 1995). See Adleman and Kompella (1988), Chor and Goldreich (1990), Kannan *et al.* (1987), and Sorenson (1994) for sublinear time and polylog time/subexponential processor parallel GCD algorithms.

In this paper, we present the right-shift and left-shift k-ary Jacobi symbol algorithms, which are based on the k-ary GCD algorithms (Jebelean, 1993; Sorenson, 1994). We obtain the following results:

(1) Both algorithms use at most $O(\log(xy)/\log k)$ iterations of their main loop to compute (x/y). See Sections 3 and 4.

(2) Sequentially, both algorithms take at most $O(\log^2(xy)/\log k)$ bit operations and at most $O(\log(xy) + k^2 \log k)$ space when $k \leq (\log(xy))^{1/2-\epsilon}$. By setting $n = \log(xy)$ and $k = 2^{\lfloor 0.4 \log n \rfloor}$, we obtain an $O(n^2/\log n)$ running time using $O(n)$ space. See Section 5.

(3) In practice, we found our new algorithms to be approximately two to three times as fast as previous algorithms, including the binary algorithm, on inputs of 100–1000 decimal digits in length. See Section 6.

(4) By choosing $k = 2^{\lfloor \epsilon \log n \rfloor}$, both algorithms take $O_\epsilon(n/\log\log n)$ time using $n^{1+\epsilon}$ processors under the Common CRCW PRAM model of parallel computation. This gives the first sublinear parallel algorithms for computing the Jacobi symbol. We also obtain polylog time, subexponential processor algorithms. See Section 7.

Both of our new algorithms can be readily modified to compute the Kronecker symbol.

2. Notation and Background

We begin by reviewing the definition of the Jacobi symbol.

2.1. DEFINITIONS

Let a be a positive integer and let p be an odd prime. Then a is a *quadratic residue* (or simply *residue*) modulo p if $\gcd(a, p) = 1$ and there exists an integer x such that $x^2 \equiv a \pmod{p}$. If a is not a quadratic residue and $\gcd(a, p) = 1$, then we say that a is a *quadratic nonresidue* (or *nonresidue*). The *Legendre symbol* (a/p) has the value 1 if a is a residue, -1 if a is a nonresidue, and 0 if $p \mid a$. The Legendre symbol can be computed using $(a/p) \equiv a^{(p-1)/2} \pmod{p}$.

Let m be an odd, composite integer with prime factorization $m = p_1 p_2 \cdots p_l$. The *Jacobi symbol*, a generalization of the Legendre symbol, is defined by $(a/m) = (a/p_1)(a/p_2) \cdots (a/p_l)$; it matches the Legendre symbol whenever m is prime. If $(a/m) = -1$, then a is a nonresidue modulo m, and if $\gcd(a, m) \neq 1$ then $(a/m) = 0$. However, if $(a/m) = 1$, a may be either a residue or a nonresidue.

2.2. JACOBI SYMBOL IDENTITIES

The Jacobi symbol satisfies the following identities which we will utilize (see Hardy and Wright (1979), Ireland and Rosen (1990) for proofs). Assume a, b are integers and n, m are odd, positive integers throughout.

$$\left(\frac{ab}{n}\right) = \left(\frac{a}{n}\right)\left(\frac{b}{n}\right); \tag{2.1}$$

$$\left(\frac{a}{nm}\right) = \left(\frac{a}{n}\right)\left(\frac{a}{m}\right); \tag{2.2}$$

$$\left(\frac{-1}{n}\right) = (-1)^{(n-1)/2}; \tag{2.3}$$

$$\left(\frac{2}{n}\right) = (-1)^{(n^2-1)/8}; \tag{2.4}$$

$$\left(\frac{a}{n}\right) = \left(\frac{a \pm bn}{n}\right); \tag{2.5}$$

$$\left(\frac{m}{n}\right) = \left(\frac{n}{m}\right)(-1)^{(m-1)/2 \cdot (n-1)/2}. \tag{2.6}$$

For simplicity, we define $(0/1) = 1$, but $(0/n) = 0$ for any integer $n > 1$. Note that (2.5) implies that $(a/n) = (a \bmod n/n)$.

2.3. JACOBI SYMBOL ALGORITHM OUTLINE

Assume u, v are integers, with v odd and $|u| \geq v > 0$. Most Jacobi symbol algorithms compute (u/v) by doing the following:

(1) Initialize a variable $t := 1$; this keeps track of the value of the Jacobi symbol as the algorithm progresses.
(2) If $u < 0$, apply (2.3) above; that is, set $u := -u$ and if $v \bmod 4 = 3$, set $t := -t$.
(3) Remove all factors of 2 from u and adjust t appropriately using (2.1) and (2.4).

(4) If $u < v$, then apply (2.6), thereby interchanging u and v; adjust t as required.
(5) Apply (2.5) in some form to make u smaller.
(6) If $u = 0$, then stop. The answer is t if $v = 1$, and the answer is 0 otherwise. If $u \neq 0$, then go back to Step 2.

Steps 2–6 are repeated until $u = 0$, which must eventually occur because one of u, v must decrease in absolute value every iteration.

The difference between the ordinary, Eisenstein, Lebesgue, and binary algorithms lies in how Step 5 is performed. In the ordinary algorithm, $(u/v) = (u \bmod v/v)$ is used. In Eisenstein's algorithm, $(u/v) = (u - bv/v)$ is used, where b is the even integer nearest to u/v. In Lebesgue's algorithm, $(u/v) = (u - bv/v)$ is used, where b is u/v rounded to the nearest integer. In the binary algorithm, $(u/v) = ([(u - v)/2]/v)(2/v)$ is used. Because of how Step 5 is done, the number of iterations through this process is at least linear in $\log v$ in the worst case for all four algorithms.

Our new algorithms have a parameter k which is a power of 2. In the following section we show how to perform Step 5 so that u is reduced by a factor proportional to $\sqrt{k}$. The result is an algorithm requiring only $O(\log(uv)/\log k)$ iterations in the worst case.

3. The Right-shift Algorithm

The central idea behind the right-shift k-ary algorithm is the use of something of the following form in Step 5 above:

$$\left(\frac{u}{v}\right) = h \cdot \left(\frac{(au - bv)/k}{v}\right) \tag{3.1}$$

where h is whatever is needed $(0, \pm 1)$ to make this true. Note that when $a = 1$, $b = -1$, and $k = 2$, we obtain the binary algorithm as a special case. For simplicity we will assume henceforth that k is an even power of two.

The following lemma shows that we can always choose a, b so that $|(au + bv)/k| = O(u/\sqrt{k})$.

LEMMA 3.1. *Let u, v, k be positive integers with u, v both relatively prime to k. Then there exist integers a, b with $a > 0$, $a, |b| \leq \sqrt{k} + 1$, such that $au + bv \equiv 0 \pmod{k}$.*

PROOF. See Sorenson (1994). □

In fact, (3.1) as written above is not always feasible; any common factors of a and v will cause the algorithm to output 0, which is not always correct. However, the following lemma gives a corrected version of (3.1).

LEMMA 3.2. *Let u, v, r be positive integers, with v odd and $k = 2^{2r}$, a square. Let a, b be nonzero integers such that $au + bv \equiv 0 \pmod{k}$. Let $d = \gcd(a, v)$, giving $a' = a/d$, $v' = v/d$. Then we have*

$$\left(\frac{u}{v}\right) = \left(\frac{u}{d}\right)\left(\frac{a'}{v'}\right)\left(\frac{(a'u + bv')/k}{v'}\right).$$

PROOF. We have

$$\left(\frac{u}{v}\right) = \left(\frac{u}{dv'}\right) = \left(\frac{u}{d}\right)\left(\frac{u}{v'}\right) = \left(\frac{u}{d}\right)\left(\frac{a'}{v'}\right)^2\left(\frac{u}{v'}\right)$$
$$= \left(\frac{u}{d}\right)\left(\frac{a'}{v'}\right)\left(\frac{a'u}{v'}\right) = \left(\frac{u}{d}\right)\left(\frac{a'}{v'}\right)\left(\frac{a'u+bv'}{v'}\right).$$

Because k is a square and v' is odd, this completes the proof. □

Note that this can be generalized to when k is an odd power of 2; simply include an additional factor of $(2/v')$.

Combining these ideas, we have the right-shift k-ary Jacobi symbol algorithm, which we present in Pascal-like pseudocode below. It is written with a sequential implementation in mind, although we will parallelize this algorithm in Section 7. As a convention, we use uppercase letters to denote multiprecision integers, and lowercase letters to denote single-precision integers (that is, integers bounded by k in absolute value).

RS *k*-ary Jacobi Symbol Algorithm

INPUTS: Positive integers U, V, r with V odd, and $k = 2^{2r}$.
OUTPUT: (U/V)

```
t := 1;
While U ≠ 0 do:
    If U < 0 then
        U := −U;
        If V mod 4 = 3 then t := −t;
    End if;
    (U, t) :=oddify(U, V, t, k);
    If U < V then
        (U, V) := (V, U); { Interchange U and V }
        If V mod 4 = 3 and U mod 4 = 3 then t := −t;
    End if;
    (a, b) :=Rfind(U mod k, V mod k, k);
        { Returns nonzero a, b such that aU + bV ≡ 0 (mod k)}
    d := gcd(V mod a, a);
    a := a/d; V := V/d;
    t := t·Jacobi(U mod d, d)·Jacobi2(a, V);
    If t = 0 then Return(0);
    U := (aU + bV)/k;
End while;
If V = 1 then Return(t) else Return(0);
```

The **oddify** function removes factors of 2 from U while adjusting t as necessary. Our nonstandard implementation will be justified later.

Function oddify(U, V, t, k)

INPUTS: Positive integers U, V, k with V odd, and $t = \pm 1$.
OUTPUT: (U, t) with U odd and t adjusted using equation (2.3).

```
Repeat
    u := U mod k;
    If u = 0 then
        e = log_2 k; { Here e is even }
    Else
        e := 0;
        While u mod 2 = 0 do:
            e := e + 1; u := u/2;
        End while;
    End if;
    U := U/2^e; { Performed using a bit shift }
Until u ≠ 0;
If e mod 2 ≠ 0 and V mod 8 = 3 or 5 then t := −t;
Return((U, t));
```

The **Rfind** function is essentially an implementation of Lemma 3.1. This algorithm is an adaptation of an extended version of Euclid's algorithm as given by Weber (1995).

Function Rfind(u, v, k)

INPUTS: Positive integers u, v, k with $\gcd(u, k) = \gcd(v, k) = 1$.
OUTPUT: (a, b) according to Lemma 3.1.

```
w := uv^{-1} mod k; { v^{-1} mod k is found via an extended GCD algorithm }
(x_1, x_2) := (k, 0);
(y_1, y_2) := (w, 1);
While y_1 ≥ √k do:
    q := ⌊x_1/y_1⌋;
    (z_1, z_2) := (x_1, x_2) − q · (y_1, y_2);
    (x_1, x_2) := (y_1, y_2);
    (y_1, y_2) := (z_1, z_2);
End while;
If y2 > 0 then
    a := y_2; b := −y_1;
Else
    a := −y_2; b := y_1;
End if;
Return((a, b));
```

The **Jacobi**(u, v) function computes (u/v) for single-precision integers u, v with v odd. It can be implemented using the ordinary, Lebesgue, or binary algorithms. The **Jacobi2** function allows its second argument to be multiprecision, and can be implemented as follows:

Function Jacobi2(u, V)

INPUTS: Positive integers u, V with V odd.
OUTPUT: (u/V).

```
s := 1;
```

While $u \bmod 2 = 0$ do:
    $u := u/2$;
    If $V \bmod 8 = 3$ or 5 then $s := -s$;
End while;
If $u \bmod 4 = 3$ and $V \bmod 4 = 3$ then $s := -s$;
Return($s \cdot$**Jacobi**$(V \bmod u, u)$);

Note that in all these algorithms and functions, divisions by k and other powers of 2 can be implemented using bit shifts. Also, computing remainders modulo powers of 2 can be done using bit extraction.

THEOREM 3.1. *Let U, V, r be positive integers with V odd, and let $k = 2^{2r}$. The right-shift k-ary Jacobi symbol algorithm computes (U/V) using at most*

$$O\left(\frac{\log(UV)}{\log k}\right)$$

iterations of its main loop.

PROOF. Correctness follows from our discussion above and Lemma 3.2. The bound on the number of iterations follows from Lemma 3.1. □

These results can be generalized to arbitrary $k > 1$. We chose not to present the full generality for two reasons. First, both the algorithm and the equivalent of Theorem 3.1 become more involved and difficult (see Sorenson (1994)). Second, in practice, the best value to use for k is an even power of two, and so there is no demand for the more general theory.

4. The Left-shift Algorithm

The left-shift version of the k-ary Jacobi symbol algorithm is based on the following lemma.

LEMMA 4.1. *Let u, v, k be positive integers, with $u \geq v > u/k$. Then there exist nonzero integers a, b with $|a|, |b| \leq k$ such that $|au + bv| \leq u/(k+1)$.*

PROOF. This follows immediately from Theorem 36 of Hardy and Wright (1979). □

Using this, one can insure that U will decrease by a factor of at least $k+1$ every iteration of the algorithm. Note that we may assume $\gcd(a, b) = 1$ will always hold.

In order to apply Lemma 4.1, we require that $V > U/k$. This requirement is met by "shifting V to the left" first. In other words, an integer e is computed such that $k^{e+1}V > U \geq k^e V$.

The following lemma shows how Lemma 4.1 can be applied to compute the Jacobi symbol.

LEMMA 4.2. *Let u, v, k be positive integers, with v odd, and let e be a nonnegative integer. Let a, b be nonzero integers. Let $d = \gcd(a, v)$, giving $a' = a/d$, $v' = v/d$. Then we have*

$$\left(\frac{u}{v}\right) = \left(\frac{u}{d}\right)\left(\frac{a'}{v'}\right)\left(\frac{a'u + bk^e v'}{v'}\right).$$

PROOF. The proof is similar to that of Lemma 3.2. □

Combining these two lemmas, we obtain the following algorithm. Our notation remains consistent with that of the last section, with the exception that single-precision integers are those with absolute value bounded by k^2.

LS *k*-ary Jacobi Symbol Algorithm

INPUTS: Positive integers U, V, r with V odd, and $k = 2^r$.
OUTPUT: (U/V)

$t := 1$;
While $U \neq 0$ do:
  If $U < 0$ then
    $U := -U$;
    If $V \bmod 4 = 3$ then $t := -t$;
  End if;
  $(U, t) :=$**oddify**(U, V, t, k);
  If $U < V$ then
    $(U, V) := (V, U)$; { Interchange U and V }
    If $V \bmod 4 = 3$ and $U \bmod 4 = 3$ then $t := -t$;
  End if;
  $e := \lfloor (\lfloor \log_2 U \rfloor - \lfloor \log_2 V \rfloor)/r \rfloor$;
  $T := k^e V$;
  If $T > U$ then $T := T/k$; $e := e - 1$;
  $h := \lfloor \log_2 U \rfloor + 1 - 2r$;
  $(a, b) :=$**Lfind**$(\lfloor U/2^h \rfloor, \lfloor T/2^h \rfloor, k)$;
    { Returns nonzero a, b such that $|aU + bk^e V| \leq U/(k+1)$}
  $d := \gcd(V \bmod a, a)$;
  $a := a/d$; $V := V/d$; $T := T/d$;
  $t := t \cdot$**Jacobi**$(U \bmod d, d) \cdot$**Jacobi2**(a, V);
  If $t = 0$ then Return(0);
  $U := aU + bT$;
End while;
If $V = 1$ then Return(t) else Return(0);

Note that $\lfloor U/2^h \rfloor$ and $\lfloor T/2^h \rfloor$ are single-precision integers, and they can be computed by extracting the leading bits of U and T; no division is required. Also, **oddify** must be slightly modified because we no longer assume that k is an even power of 2; we leave this detail to the reader.

Function **Lfind** is an implementation of Lemma 4.1, and like its counterpart **Rfind**, is based on the extended GCD algorithm of Euclid.

Function Lfind(u, v, k)

INPUTS: Positive integers u, v, k.
OUTPUT: (a, b) according to Lemma 4.1.

$(x_1, x_2, x_3) := (u, 1, 0)$;

$(y_1, y_2, y_3) := (v, 0, 1)$;
While $y_1 > u/(k+1)$ do:
    $q := \lfloor x_1/y_1 \rfloor$;
    $(z_1, z_2, z_3) := (x_1, x_2, x_3) - q \cdot (y_1, y_2, y_3)$;
    $(x_1, x_2, x_3) := (y_1, y_2, y_3)$;
    $(y_1, y_2, y_3) := (z_1, z_2, z_3)$;
End while;
$a := y_2$; $b := y_3$;
Return$((a, b))$;

As in the previous section, we have the following.

THEOREM 4.1. *Let U, V, r be positive integers with V odd, and let $k = 2^r$. The left-shift k-ary Jacobi symbol algorithm computes (U/V) using at most*

$$O\left(\frac{\log(UV)}{\log k}\right)$$

iterations of its main loop.

5. Sequential Complexity

In this section we prove subquadratic running times for both k-ary Jacobi symbol algorithms. We begin with a discussion of our model of computation and a lemma on arithmetic with small integers. We then present our sequential complexity results.

Our model of computation is a RAM with potentially infinite memory that is addressable at the bit level (sometimes called the *naive bit complexity* model). Any basic operation on one or two bits takes constant time, as does indirect addressing and any basic flow of control operations. Let x, y be integers with $y \neq 0$. To compute $x \pm y$ or compare x to y takes $O(\log x + \log y)$ time, xy takes $O(\log x \log y)$ time, and $\lfloor x/y \rfloor$ and $x \bmod y$ take $O(\log(x/y + 1) \log y)$ time.

The following lemma shows that arithmetic operations where one of the operands is "single-precision" take essentially linear time. Thus, this lemma provides a theoretical foundation for differentiating between single and multiple precision numbers in our algorithms. In practice, single-precision arithmetic is performed in hardware, and so the benefits of this lemma happen "automatically".

LEMMA 5.1. *Let X, y, and $k = 2^r$ be positive integers with $y \leq k$. Then Xy, X/y, and $X \bmod y$ can be computed in $O(\log X)$ bit operations using a precomputed table of size $O(k^2 \log k)$ bits. It requires $O(k^2 \log^2 k)$ bit operations to construct this table.*

PROOF. The idea is to precompute a table containing the product, quotient, and remainder of all pairs of positive integers bounded by k. Then operations with multiple precision numbers are performed by manipulating the multiple precision integers in base 2^r.

For further discussion, see Sorenson (1994). □

With this lemma in hand, we are now prepared to prove the following theorems.

THEOREM 5.1. *The right-shift k-ary Jacobi symbol algorithm computes (U/V) using at most*

$$O\left(\frac{\log^2(UV)}{\log k} + \log(UV)\log k + (k\log k)^2\right)$$

bit operations and $O(\log U + k^2 \log k)$ space.

PROOF. Let U and V denote the initial values of these variables in the algorithm. WLOG, we will assume that $U > V$. Then, ignoring time spent in functions **oddify**, **Rfind**, **Jacobi**, **Jacobi2**, or to compute $\gcd(V \bmod a, a)$, one pass through the main loop takes no more than $O(\log U)$ time by Lemma 5.1. **Rfind**, **Jacobi**, and the gcd can all be computed in $O(\log^2 k)$ time. **Jacobi2** takes no more than $O(\log V + \log^2 k) = O(\log U + \log^2 k)$ time. Thus, excluding the **oddify** function, the time for one loop iteration is $O(\log U + \log^2 k)$. Applying Theorem 3.1 we obtain the bound of $O(\log^2 UV/\log k + \log(UV)\log k)$.

One pass through the repeat loop in function **oddify** takes $O(\log^2 k + \log U)$ time. If the repeat loop executes more than once, then every pass except the last reduces U by a factor of k. The number of final passes is bounded by the number of times **oddify** is called, which is at most $O(\log(UV)/\log k)$ by Theorem 3.1. There are at most $O(\log U/\log k)$ of passes that are not the last. Thus, the total number of passes is $O(\log(UV)/\log k)$. Multiplying by the time for one pass, we obtain an $O(\log^2(UV)/\log k + \log(UV)\log k)$ bound for the total time spend in function **oddify**.

By Lemma 5.1, at most $O(k^2 \log^2 k)$ time is spent in precomputation. □

THEOREM 5.2. *The left-shift k-ary Jacobi symbol algorithm computes (U/V) using at most*

$$O\left(\frac{\log^2(UV)}{\log k} + \log(UV)\log k + (k\log k)^2\right)$$

bit operations and $O(\log U + k^2 \log k)$ space.

PROOF. The proof follows the same lines as the previous theorem; we leave it as an exercise for the reader. □

6. Implementation Results

In this section, we present results of timing experiments we conducted to determine how well the k-ary Jacobi symbol algorithms perform in practice.

Note that the data presented below depend not just on the algorithms used, but also on the programmer, the compiler and programming language, and the platform. The reader should keep this in mind before drawing any conclusions based on our data.

We implemented the two new algorithms along with the ordinary, Lebesgue, and binary algorithms in C++ using a common multiprecision library (this library was also used in Shallit and Sorenson (1993, 1994) and Sorenson (1994, 1995)). We used the Gnu `g++` compiler based on `gcc` Version 2.6.3, with standard level optimization. Our platform was a Hewlett-Packard 9000 series 715/75 workstation running HP-UX Version 9.01.

Each algorithm was timed using a common set of 100 pseudo-random input pairs of each of sizes 100, 250, 500, and 1000 decimal digits in length. The average times are reported in Table 1.

Table 1. Average running times in CPU seconds.

	Input size			
Algorithm	100	250	500	1000
Ordinary Algorithm	0.0168	0.0783	0.276	0.997
Lebesgue's Algorithm	0.0170	0.0783	0.277	1.004
Binary Algorithm	0.0182	0.0775	0.264	0.933
RS k-ary Algorithm	0.0087	0.0315	0.0917	0.334
(when not zero)	0.0107	0.0394	0.125	0.405
LS k-ary Algorithm	0.0079	0.0308	0.0901	0.355
(when not zero)	0.0097	0.0382	0.122	0.430

Table 2. Average running times in CPU seconds (using pointers).

	Input size			
Algorithm	100	250	500	1000
Ordinary Algorithm	0.0154	0.0710	0.249	0.924
Lebesgue's Algorithm	0.0160	0.0735	0.260	0.968
Binary Algorithm	0.0165	0.0694	0.232	0.837
RS k-ary Algorithm	0.0082	0.0297	0.0849	0.324
(when not zero)	0.0100	0.0371	0.115	0.392
LS k-ary Algorithm	0.0074	0.0288	0.0868	0.346
(when not zero)	0.0088	0.0359	0.118	0.419

Table 3. Average number of main loop iterations.

	Input size			
Algorithm	100	250	500	1000
Ordinary Algorithm	121	306	613	1214
Lebesgue's Algorithm	98	244	484	966
Binary Algorithm	234	586	1173	2345
RS k-ary Algorithm	30	73	134	305
(when not zero)	37	92	185	370
LS k-ary Algorithm	31	77	141	318
(when not zero)	38	97	193	387

For the right-shift (RS) k-ary algorithm, we used $k = 2^{30}$; for the left-shift (LS) we used $k = 2^{15}$. As both new algorithms may "stop early" if a small common divisor is found, we also present the averages over only those inputs where the Jacobi symbol is nonzero.

In Table 1, when performing the equivalent of Step 4 (see Section 2), swaps were performed using three assignment statements (or copy operations). An alternative is to use pointers and then simply swap the pointers. This is more efficient if the inputs are sufficiently large, and it favors algorithms that perform more iterations. We present these results in Table 2.

Finally, we present the average number of main loop iterations performed by each algorithm in Table 3. Note that these data are independent of the particular implementation.

7. Parallel Complexity

In this section, we present two results on parallel algorithms for the Jacobi symbol: sublinear, polynomial processor algorithms and polylog time, subexponential processor algorithms.

Our model of computaton is the parallel random access machine (PRAM) where concurrent reads and writes are permitted (CRCW). Write conflicts are only allowed if the same value is being written (the Common CRCW PRAM). For more on parallel models of computation, see Greenlaw *et al.* (1995) and Karp and Ramachandran (1990).

Before giving our results for the Jacobi symbol, we need to address the cost of performing various arithmetic operations on the Common CRCW PRAM. Let x and y be integers of at most n bits in length, and define $M(n) := n \log n \log\log n$. Then

(1) computing $x \pm y$ and performing comparisons takes $O(1)$ time and $O(n \log\log n)$ processors (Chandra *et al.*, 1985);
(2) computing xy takes $O(\log n)$ time and $O(M(n))$ processors (Schönhage and Strassen, 1971);
(3) and computing $\lfloor x/y \rfloor$ and $x \bmod y$ takes $O(\log n \log\log n)$ time and $O(M(n))$ processors (Reif and Tate, 1989) or $O(\log n)$ time and $O(n^{1+\epsilon})$ processors (Beame *et al.*, 1986).

LEMMA 7.1. *Multiplication of an n-bit integer by an $O(r)$-bit integer (assuming $r = \Omega(\log\log n)$) takes $O(1)$ time using $O(n2^{2r})$ processors. This requires a precomputed table of size $O(r2^{2r})$ bits, which takes $O(\log r)$ time and $O(2^{2r}M(r))$ processors to construct.*

PROOF. (SKETCH) The idea is to precompute all products of pairs of integers bounded by 2^r, and then view the n-bit number in base 2^r. For details, see Chor and Goldreich (1990) or Lemma 6.2 from Sorenson (1994). □

LEMMA 7.2. *Division of an n-bit integer by an $O(r)$-bit integer takes $O(\log n/\log\log n)$ time using $O(n2^{2r})$ processors. Here we assume $r \le \log_2 n$ and $r = \Omega(\log\log n)$, and precomputation is required as in the previous lemma.*

PROOF. (SKETCH) We use Lemma 7.1 so that multiplication by an r-bit integer requires only $O(1)$ time.

The basis for our division algorithm is the $\mathcal{NC}^1$ division circuit of Beame, Cook, and Hoover for small integers (Beame *et al.*, 1986, Lemma 4.1). Their circuit can be mapped to an exclusive-read exlusive-write PRAM algorithm. When adapting their algorithm to the CRCW model, the time bottleneck is easily seen to be computing parallel prefix sums. We apply the algorithm of Cole and Vishkin (Cole and Vishkin, 1989; Vishkin, 1995) to obtain the necessary $\Theta(\log\log n)$ factor speedup. □

THEOREM 7.1. *The right and left-shift k-ary Jacobi symbol algorithms can be implemented in parallel so that they take $O_\epsilon(n/\log\log n)$ time using $O(n^{1+\epsilon})$ processors.*

PROOF. Let $\delta > 0$ with $\delta < \epsilon/4$, and choose $k = 2^{2\lfloor \delta \log n \rfloor}$ so that $\log k = \Theta_\epsilon(\log n)$ and $k^2 \le n^\epsilon$. We will use Lemma 7.1 with $r = (\log_2 k)$ (or equivalently, $2^r = k$).

We will prove this theorem for the right-shift k-ary algorithm only. The proof for the

left-shift algorithm is very similar, only choose $\delta < \epsilon/8$ and use $2^r = k^2$ due to the different definition of "single-precision" integer.

(1) *Precomputation.* The precomputation for Lemma 7.1 takes $O(\log\log k)$ time and $O(k^2 M(\log k))$ processors.
We also need to precompute $\gcd(x, y)$, $x^{-1} \bmod k$, and (x/y) for all integers $0 < x, y \leq k$. We begin by finding the prime factorization of all integers up to k; by (Sorenson, 1994, Lemma 6.3), this takes $O((\log\log k)^2 \log\log\log k)$ time using $O(k^3 \log K)$ processors. From this information, $\gcd(x, y)$ is easily computed. To compute $x^{-1} \bmod k$, we try all possible inverses exhaustively. To compute (x/y), we use (2.2) and the prime factorization of y to reduce this to computing the Legendre symbol. To compute the Legendre symbol, we simply square all integers up to k to see if one is the square root. This takes $O(\log\log k)$ time and $O(k^3 M(\log k)\log k)$ processors.
Finally, we also precompute a table encoding the output of the **Rfind** function. This is done using exhaustive search in $O(1)$ time using $O(k^4 \log k)$ processors.
Thus, the total cost of precomputation is $O((\log\log k)^2 \log\log\log k)$ time and $O(k^4 \log k)$ processors.

(2) *The Main Loop.* First let us look at the time spent in the **oddify** function. Recall that division by a power of 2 takes $O(1)$ time using $O(n)$ processors. If $u = 0$ for an iteration of the repeat-loop, then the cost for that iteration is $O(1)$ time, $O(n)$ processors. If $u > 0$, then $\gcd(u, k)$ gives the value for e, and as this is precomputed, this also takes only $O(1)$ time, $O(n)$ processors. Thus, the total time spent in **oddify** over all iterations of the main loop of the algorithm is $O(n/\log k)$ using $O(n)$ processors.
Except for division by a and d, all other operations performed during an iteration of the main loop require $O(1)$ time using $O(nk^2)$ processors. Using Lemma 7.2, division by a and d takes $O(\log n/\log\log n)$ time and $O(nk^2)$ processors. This gives a total of $O(n\log n/(\log k \log\log n))$ time using $O(nk^2)$ processors for the main loop.

Combining the costs of precomputation and the main loop, we obtain a running time of $O((\log\log k)^2 \log\log\log k + n\log n/(\log k\log\log n))$. By our choice for k as a function of n, this is $O(n/\epsilon \log\log n)$. The number of processors is $O(k^4 \log k + nk^2) = O(n^{1+\epsilon})$. □

THEOREM 7.2. *Let $d \geq 1$ be an integer. The right-shift and left-shift k-ary Jacobi symbol algorithms can be implemented to take $O(\log^2 n \log\log n + \log^{d+1} n)$ time using $\exp[O(n/\log^d n)]$ processors.*

PROOF. If we substitute the use of Lemma 7.2 in the proof of the previous theorem with the division algorithm of Beame *et al.* (1986), we obtain a running time of $O((\log\log k)^2 \log\log\log k + n\log n/\log k)$ using $(nk)^{O(1)}$ processors. Choose $r = 2\lfloor n/\log^d n\rfloor$ so that $k = \exp[\Theta(n/\log^d n)]$. □

References

Adleman, L. M., Kompella, K. (1988). Using smoothness to achieve parallelism. In *20th Annual ACM Symposium on Theory of Computing*, pp. 528–538.
Ankeny, N. C. (1952). The least quadratic nonresidue. *Ann. Math.*, **55**, 65–72.

Bach, E. (1990a). Explicit bounds for primality testing and related problems. *Math. Comput.*, **55**, 355–380.
Bach, E. (1990b). A note on square roots in finite fields. *IEEE Trans. Infor. Theory*, **36**, 1494–1498.
Bach, E. (1991). Realistic analysis of some randomized algorithms. *J. Comput. Syst. Sci.*, **42**, 30–53.
Beame, P. W., Cook, S. A., Hoover, H. J. (1986). Log depth circuits for division and related problems. *SIAM J. Comput.*, **15**, 994–1003.
Chandra, A. K., Fortune, S., Lipton, R. (1985). Unbounded fan-in circuits and associative functions. *J. Comput. Syst. Sci.*, **30**, 222–234.
Chor, B., Goldreich, O. (1990). An improved parallel algorithm for integer GCD. *Algorithmica*, **5**, 1–10.
Cole, R., Vishkin, U. (1989). Faster optimal parallel prefix sums and list ranking. *Information and Control*, **81**, 334–352.
Collins, G. E., Loos, R. G. K. (1982). The Jacobi symbol algorithm. *SIGSAM Bull.*, **16**, 12–16.
Fich, F., Tompa, M. (1988). The parallel complexity of exponentiating polynomials over finite fields. *J. ACM*, **35**, 651–667.
Gauss, C. F. (1870). Theorematis fundamentalis in doctrina de residuis quadraticis demonstrationes et ampliationes novae. In *Werke*, Vol. 2, pp. 49–64. K. Göttingen, Gesellshaft der Wissenshaften.
Greenlaw, R., Hoover, H. J., Ruzzo, W. L. (1995). *Limits to Parallel Computation*. Oxford University Press.
Hardy, G. H., Wright, E. M. (1979). *An Introduction to the Theory of Numbers*, 5th edn. Oxford University Press.
Ireland, K., Rosen, M. (1990). *A Classical Introduction to Modern Number Theory*, 2nd edn. New York, Springer.
Jebelean, T. (1993). A generalization of the binary GCD algorithm. In Bronstein, M., ed., *Proceedings of the 1993 International Symposium on Symbolic and Algebraic Computation*, pp. 111–116. Kiev, Ukraine, ACM Press.
Kannan, R., Miller, G., Rudolph, L. (1987). Sublinear parallel algorithm for computing the greatest common divisor of two integers. *SIAM J. Computing*, **16**, 7–16.
Karp, R., Ramachandran, V. (1990). Parallel algorithms for shared-memory machines. In van Leeuwen, J., ed., *Algorithms and Complexity*. Elsevier and MIT Press. Handbook of Theoretical Computer Science, Vol. A.
Koblitz, N. (1994). *A Course in Number Theory and Cryptography*, 2nd edn. New York, Springer-Verlag.
McCurley, K. S. (1990). Odds and ends from cryptology and computational number theory. In *Cryptology and Computational Number Theory (Pomerance, 1990)*, pp. 145–166. Providence, RI, American Mathematical Society.
Menezes, A. J., van Oorschot, P. C., Vanstone, S. A. (1997). *Handbook of Applied Cryptography*. Boca Raton, FL, CRC Press.
Meyer, S. M., Sorenson, J. P. (1996). Efficient algorithms for computing the Jacobi symbol. In Cohen, H., ed., *Proceedings of the Second International Algorithmic Number Theory Symposium*, LNCS **1122**, pp. 225–239, Talence, France, Springer.
Peralta, R. (1986). A simple and fast probabilistic algorithm for computing square roots modulo a prime number. *IEEE Trans. Inf. Theory*, **32**, 846–847.
Pomerance, C., ed. (1990). *Cryptology and Computational Number Theory*, Vol. 42 of *Proceedings of Symposia in Applied Mathematics*. Providence, RI, American Mathematical Society.
Reif, J. H., ed. (1993). *Synthesis of Parallel Algorithms*. San Mateo, CA, Morgan Kaufman.
Reif, J. H., Tate, S. R. (1989). Optimal size integer division circuits. In *21st Annual ACM Symposium on Theory of Computing*, pp. 264–273.
Scheidler, R., Williams, H. C. (1995). A public-key cryptosystem utilizing cyclotomic fields. *Designs, Codes and Cryptography*, **6**, 117–131.
Schönhage, A. (1971). Schnelle Berechnung von Kettenbruchentwicklungen. *Acta Inf.*, **1**, 139–144.
Schönhage, A., Strassen, V. (1971). Schnelle Multiplikation großer Zahlen. *Computing*, **7**, 281–292.
Shallit, J. (1990). On the worst case of three algorithms for computing the Jacobi symbol. *J. Symb. Comput.*, **10**, 593–610.
Shallit, J. O., Rabin, M. O. (1986). Randomized algorithms in number theory. *Commun. Pure Appl. Math.*, **39**, 239–256.
Shallit, J. O., Sorenson, J. P. (1993). A binary algorithm for the Jacobi symbol. *SIGSAM Bulletin*, **27**, 4–11.
Shallit, J. O., Sorenson, J. P. (1994). Analysis of a left-shift binary GCD algorithm. *J. Symb. Comput.*, **17**, 473–486.
Solovay, R., Strassen, V. (1977). A fast Monte Carlo test for primality. *SIAM J. Comput.*, **6**, 84–85. Erratum in Vol. 7, p. 118, 1978.
Sorenson, J. P. (1994). Two fast GCD algorithms. *J. Algorithms*, **16**, 110–144.
Sorenson, J. P. (1995). An analysis of Lehmer's Euclidean GCD algorithm. In Levelt, A. H. M., ed., *Proceedings of the 1995 International Symposium on Symbolic and Algebraic Computation*, pp. 254–258. Montreal, Canada, ACM Press.

Vishkin, U. (1995). Advanced parallel prefix-sums, list ranking and connectivity. In *Synthesis of Parallel Algorithms(Reif, 1993)*, pp. 215–258. Oxford University Press.
Weber, K. (1995). The accelerated integer GCD algorithm. *ACM Trans. Math. Software*, **21**, 111–122.
Williams, H. C. (1980). A modification of the RSA public-key encryption procedure. *IEEE Trans. Inf. Theory*, **IT-26**, 358–368.
Williams, H. C. (1986). An m^3 public-key encryption scheme. In *Advances in Cryptology – CRYPTO'85 Proceedings*, pp. 358–368. Berlin, Springer.

Originally received 3 September 1996
Accepted 23 April 1998