THE K-ARY GCD ALGORITHM

by

Jonathan Sorenson

Computer Sciences Technical Report #979

November 1990

The k-ary GCD Algorithm

Jonathan Sorenson*
Computer Sciences Department
University of Wisconsin-Madison
1210 West Dayton Street
Madison, WI 53706
USA
sorenson@cs.wisc.edu

November 13, 1990

Abstract

In this paper we introduce the new k-ary greatest common divisor (GCD) algorithm, a generalization of the binary algorithm. Interestingly, this new algorithm has both a sequential version that is very practical and parallel versions that rival the best previous parallel GCD algorithms.

We show that for k a prime power, the sequential k-ary GCD algorithm has a $\Theta(n^2)$ worst-case running time on n-bit inputs. However, in a multiple-precision implementation the k-ary algorithm outperforms several other GCD algorithms, including the binary algorithm by a factor of 2, and the Euclidean algorithm by a factor of 12.

Parallel versions of the algorithm include the following for the CRCW PRAM model: an $O(n/\log n)$ time algorithm using $O(n^{1+\epsilon})$ processors, and for $d \geq 1$, an $O(\log^d n)$ time algorithm using $\exp[O(n/\log^d n)]$ processors. The first algorithm matches the complexity of the best previous parallel GCD algorithm using a polynomial number of processors, and the second algorithm is the first deterministic polylog time GCD algorithm using a subexponential number of processors. We also give EREW and CREW PRAM versions of the algorithm.

University of Wisconsin-Madison Computer Sciences Technical Report #979
*Supported by NSF grant CCR-8552596

1 Introduction.

Given two positive integers u and v, the greatest common divisor (GCD) of u and v is the largest integer that evenly divides both u and v. In this paper, we present the k-ary GCD algorithm, a generalization of the binary algorithm of Stein. Interestingly, this new algorithm has both a sequential version that is very practical and parallel versions that rival the best previous parallel GCD algorithms.

Applications for greatest common divisor algorithms are numerous, including computer algebra systems, symbolic computation, cryptography, and integer factoring. Perhaps the most famous and well-studied GCD algorithm is the classical algorithm of Euclid. Others include the least-remainder Euclidean algorithm, the binary algorithm of Stein (see Brent [Bre76]), the binary algorithm using left shifts [Bre76], Purdy's algorithm which exploits carry-free arithmetic [Pur83], Norton's shift-remainder algorithm [Nor87], and Schönhage's algorithm [Sch71]. In addition, there are variants of many of these algorithms; for instance Lehmer designed a variant of the Euclidean algorithm for multiple precision inputs [Leh38]. Most of these algorithm have an $O(n^2)$ worst-case running time on n-bit inputs. For multiple precision inputs, many consider the binary and left shift binary algorithms to be the best in practice. For general references on GCD algorithms, see Knuth [Knu81] and Bach and Shallit [BS90].

In this paper, we show that the k-ary GCD algorithm also has an $O(n^2)$ worst-case running time when k is a prime power. More importantly, we demonstrate that this new algorithm is very practical. We implemented a multiple-precision version of the algorithm, and for k carefully chosen, the k-ary GCD algorithm outperformed several other algorithms, including the Euclidean algorithm by a factor of 12, the binary algorithm by a factor of 2, and the left shift binary algorithm by 35%.

The parallel complexity of computing greatest common divisors is a long-standing open problem. Deng [Den89] showed that certain GCD related problems are $\mathcal{NC}$-equivalent to finding the optimal solution of a two-variable linear program. Although there are several sublinear time parallel GCD algorithms, no $\mathcal{NC}$ algorithm is known, and computing GCD's is not known to be $\mathcal{P}$-complete.

In this paper we discuss parallel algorithms in terms of the parallel random access machine (PRAM) model. Previous parallel GCD algorithms include those by Kannan, Miller, and Rudolph [KMR87], who gave the first sublinear time algorithm, Adleman and Kompella [AK88], who gave a polylog time randomized algorithm using a subexponential but super-polynomial number of processors, and Chor and Goldreich [CG90], who currently have the fastest parallel algorithms using only a polynomial number of processors. Chor and Goldreich's algorithms are based on the linear time algorithm of Brent and Kung [BK83] for systolic arrays. For a general reference on the PRAM model of computation and parallel algorithms, see Karp and Ramachandran [KR88].

We give four new parallel algorithms based on the k-ary method:

- An EREW PRAM algorithm with a running time of $O(n)$ using $O(n \log n \log \log n)$ processors.
- A CRCW PRAM algorithm with a running time of $O(n/\log n)$ and a CREW PRAM

algorithm with a running time of $O(n \log \log n / \log n)$, both using $O(n^{1+\epsilon})$ processors. These algorithms match the complexity bounds obtained by Chor and Goldreich.

- A CRCW PRAM algorithm with a running time of $O(\log^d n)$ using $\exp[O(n/\log^d n)]$ processors for any $d \geq 1$. This is the first deterministic, polylog time parallel GCD algorithm using a subexponential number of processors. Adleman and Kompella's algorithm has a running time of $O(\log^2 n)$ using $\exp[O(\sqrt{n \log n})]$ processors, but requires randomness.

Finally, we point out that although the k-ary GCD algorithm is more complicated than, say, the Euclidean or the binary algorithms, the k-ary algorithm is much easier to parallelize.

The rest of this paper is organized as follows: In the next section, we present the k-ary GCD algorithm in detail. In section 3, we discuss the sequential version of the algorithm, proving a quadratic worst-case running time when k is a prime power and presenting the implementation timing results. In section 4, we discuss the four parallel algorithms. We conclude with some open problems.

2 A Description of the Algorithm.

In this section we describe the k-ary GCD algorithm. Although our point of view is sequential in nature, the ideas presented here apply to parallel versions of the algorithm as well.

We begin by presenting the binary algorithm. From that we generalize to the k-ary algorithm, which we describe in detail. We then analyze the number of iterations required by the algorithm.

2.1 The Binary Algorithm

Since the k-ary GCD algorithm is a generalization of the binary algorithm, we begin with a review of the latter. Let u and v be the inputs to the algorithm, and assume they are both odd. Then the binary algorithm consists of the following steps:

```
while $u \neq 0$ and $v \neq 0$ do:
    if $u$ is even, $u := u/2$
    else if $v$ is even, $v := v/2$
    else
        $t := |u - v|/2$;
        if $u > v$ then $u := t$ else $v := t$;
if $u = 0$ then $t := v$ else $t := u$;
Output($t$);
```

Notice that division by 2 does not affect the $\gcd(u, v)$, as u and v are both initially odd. Since each iteration reduces the product uv by a factor of 2 or more, the number of iterations of the algorithm is at most $\log_2(uv) + O(1)$.

The binary algorithm is very practical because a multiple precision division subroutine is not required. In addition, the divisions by 2 can often be implemented with a shift operation.

For multiple precision inputs, this makes the binary algorithm significantly faster than the Euclidean algorithm on many computers.

There is much literature devoted to the analysis of the binary algorithm, and it has many variations. For more, see Brent [Bre76], Knuth [Knu81], or Bach and Shallit [BS90]. For an extended version of the binary algorithm, see Norton [Nor85] or Knuth [Knu81].

2.2 The Main Ideas

We now wish to generalize the binary algorithm to use any positive integer k in place of 2. Let u and v be the inputs to the algorithm, and assume that they are relatively prime to k. Then the *main loop* of the algorithm is as follows:

```
while $u \neq 0$ and $v \neq 0$ do:
    if $\gcd(u,k) > 1$, $u := u/\gcd(u,k)$
    else if $\gcd(v,k) > 1$, $v := v/\gcd(v,k)$
    else
        find nonzero integers $a, b$ satisfying $au + bv \equiv 0 \pmod{k}$;
        $t := |au + bv|/k$;
        if $u > v$ then $u := t$ else $v := t$;
```

Notice that if we set $k = 2$, $a = 1$, and $b = -1$ we have the binary algorithm as a special case.

We remark that the integer pairs a, b can be precomputed and stored in a table, as can the $\gcd(x, k)$ for $0 \leq x < k$. The tables are then indexed using $u \bmod k$ and $v \bmod k$. We will postpone the discussion of how these tables are constructed.

If we are to build a GCD algorithm around our main loop, we must address the following two questions:

1. Can the main loop be modified to correctly compute the $\gcd(u, v)$?
2. How large are the values of a and b, and can they be found efficiently?

In answer to the first question, notice that initially u and v are relatively prime to k, so divisions by divisors of k will not affect $\gcd(u, v)$. However, replacing (u, v) by $(\min\{u, v\}, |au + bv|/k)$ does not preserve the greatest common divisor of u and v in general. But we prove the following:

Lemma 1 *For u, v, a, and b integers, if $g = \gcd(u, v)$ and $h = \gcd(v, au + bv)$, then $g \mid h$ and $h/g \mid a$.*

Proof: The lemma follows by noticing $h = \gcd(v, au)$. □

For the k-ary GCD algorithm to work, then, *before* the main loop we must remove and save common divisors of u and v that may divide possible values of a and b, and *after* the main loop, we must remove the factors introduced by the divisors of a and b. For this to be practical, we must show that a and b are always small, which leads us to the second question above and its answer, the following lemma:

Lemma 2 *Let $k > 1$ be an integer. For every pair of integers (u, v), there exists a pair of integers (a, b) with $0 < |a| + |b| \leq 2\lceil\sqrt{k}\rceil$ such that $au + bv \equiv 0 \pmod{k}$.*

Proof: We use a simple pigeon-hole argument. Consider the mapping $(a, b) \mapsto au + bv \bmod k$. There can only be k distinct images to this map. If we let a and b range over $1 \leq a, b \leq \lceil\sqrt{k}\rceil + 1$, we have more than k pairs in the domain. Thus there must be two pairs (a_1, b_1) and (a_2, b_2), with $a_1 \neq a_2$ or $b_1 \neq b_2$, that map to the same residue class modulo k. This means the pair $(a_1 - a_2, b_1 - b_2)$ maps to 0, and further it satisfies $0 < |a_1 - a_2| + |b_1 - b_2| \leq 2\lceil\sqrt{k}\rceil$. □

Kannan, Miller, and Rudolf [KMR87] prove results similar to Lemmas 1 and 2. Their algorithm searches in parallel for an integer $a \leq n$ such that $au - qv = O(u/n)$, where $q = \lfloor au/v \rfloor$.

Lemma 2 implies that for k a prime, uv is reduced by a factor of $\Omega(\sqrt{k})$ each time through the main loop, which implies only $O(n/\log k)$ iterations are needed.

The complete algorithm will consist of four phases:

1. The precomputation phase constructs a table of a and b values needed in the main loop and a table containing the $\gcd(x, k)$ for $0 \leq x < k$.

2. The first trial division phase removes and saves common divisors d of u and v satisfying $d \leq \sqrt{k} + 1$ or $d \mid k$.

3. The main loop, which is described above.

4. The second trial division phase removes divisors $d \leq \sqrt{k} + 1$ introduced by the main loop and then restores the divisors saved in phase 2.

We give more details in the next subsection. At this point an example is in order.

An Example

Let $u = 263$, $v = 151$, and $k = 7$. We assume the precomputation phase has been performed, and the first trial division phase finds no common divisors. We proceed to the main loop:

1. We find that $u \equiv v \equiv 4 \pmod{7}$, so we use $a = 1$ and $b = -1$, and calculate $|au + bv|/7 = |263 - 151|/7 = 16$ and set $u = 16$ since u was larger.

2. We have $u = 16 \equiv 2$ and $v = 151 \equiv 4 \pmod{7}$, so we use $a = 2$ and $b = -1$ to give $|au + bv|/7 = |32 - 151|/7 = 17$, which we assign to v since v was larger.

3. We now have $u = 16 \equiv 2$ and $v = 17 \equiv 3 \equiv -4 \pmod{7}$. So we use $a = 2$ and $b = 1$ this time and calculate $|au + bv|/7 = |32 + 17|/7 = 7$ which we assign to v again since it was larger.

4. Now v is divisible by $k = 7$, so we divide to get $v = 1$.

5. We have $u = 16 \equiv 2$ and $v = 1 \equiv 1 \pmod{7}$, so we use $a = 1$ and $b = -2$ to give $|au + bv|/7 = |16 - 2|/7 = 2$ which we assign to u.

6. For the last step we have $u = 2$ and $v = 1$ so $a = 1$ and $b = -2$ again to give $|au + bv|/7 = 0$, which we assign to u, and we are done since u is zero.

Since v has no divisors, the final trial division phase has no effect, and we conclude that the GCD of 263 and 151 is 1, the value of v. As both the inputs are in fact prime, this is correct.

2.3 The Details

Let u and v be the inputs to the k-ary GCD algorithm. We now give a detailed description of the algorithm by phase.

Phase I: Precomputation

The goal in the precomputation phase is to construct a table A of pairs (a, b) such that, when given integers u and v, we can quickly look in the table to find a and b such that $au + bv \equiv 0 \pmod{k}$ with $|a| + |b|$ minimized. We will also compute a table P of prime divisors used in the trial division phases, a table G of GCDs, and a table I of inverses modulo k. All these tables can be implemented using one-dimensional arrays with a total of $O(k)$ storage locations or $O(k \log k)$ bits.

Tables I and G:

We store $\gcd(x, k)$ in $G[x]$, and if $G[x] = 1$ we store $1/x \bmod k$ in $I[x]$, for $0 \leq x < k$. Both can be computed using the extended Euclidean algorithm in $O(k \log k)$ arithmetic operations. Alternatively, GCD computations can be avoided as follows: Using the sieve of Eratosthenes, factor all integers up to k and compute the entries for table G from this. Table I can be computed using the identity $x^{-1} \equiv x^{\phi(k)-1} \pmod{k}$.

Table A:

Given u and v relatively prime to k, let $x = u/v \pmod{k}$. Then $u - xv \equiv 0 \pmod{k}$. If we can find a value of a such that both $|a|$ and $|ax|$ are small, we then use $b = -ax \pmod{k}$.

In the main loop we can easily compute $x \equiv u/v$ using table I described above. So in table A we store the optimal value of a for each x.

Table A can be constructed using exhaustive search in $O(k\sqrt{k})$ arithmetic operations, since by Lemma 2 the largest value of a we ever need is $O(\sqrt{k})$. The table may also be constructed as follows: (1) Loop over all values of a and b up to $\sqrt{k} + 1$. (2) For each pair a, b find all solutions x to the equation $b \equiv -ax \pmod{k}$. Note that not every pair a, b has a solution x. (3) For every solution x with $\gcd(x, k) = 1$, store a in $A[x]$. This requires only $O(k \log k)$ arithmetic operations using an extended GCD algorithm.

Table P:

The prime divisors needed in phases 2 and 4 are stored in table P. These are just the primes below $\sqrt{k} + 1$ and the prime divisors of k. Using trial division and the Sieve of Eratosthenes, all can be found in $O(\sqrt{k} \log k)$ arithmetic operations.

For example, for $k = 7$ we have the following tables:

	x:	0	1	2	3	4	5	6
Table I	x^{-1}:		1	-3	-2	2	3	-1
Table A	a:		1	1	2	2	1	1
Table G	$\gcd(x,k)$:	7	1	1	1	1	1	1
Table P		2	3	7				

Suppose $u \equiv -3$ and $v \equiv 2 \pmod 7$. We begin by using table I to find $v^{-1} \equiv -3$. We then compute $x = u/v \equiv 2$. Using table A we find $a = 1$, and we compute $b = -ax = -2$. Sure enough, $au + bv \equiv 1 \times (-3) + (-2) \times 2 \equiv 0 \pmod 7$.

Note that if the k-ary GCD algorithm is to be used more than once, the precomputation phase need not be repeated, as the tables constructed depend only on k and not on the inputs.

Phase II: Trial Division

The purpose of the first trial division phase is to remove all common divisors d of u and v where $d \le \sqrt{k} + 1$ or $d \mid k$. Possible values of d are listed in table P. We save the common divisors found in g.

```
$g := 1$;
for every $d$ in table $P$ do:
    while $d \mid u$ and $d \mid v$ do:
        $u := u/d$; $v := v/d$; $g := g \times d$;
```

Phase III: Main Loop

The main loop of the k-ary GCD algorithm was discussed earlier. Below we show how the tables are used to compute GCDs and find the a, b pairs:

```
while $u \neq 0$ and $v \neq 0$ do:
    $u' := u \bmod k$; $v' := v \bmod k$;
    if $G[u'] > 1$ then $u := u/G[u']$
    else if $G[v'] > 1$ then $v := v/G[v']$
    else
        $x := u' \times I[v'] \bmod k$;
        $a := A[x]$; $b := -a \times x \bmod k$;
        $t := |au + bv|/k$;
        if $u > v$ then $u := t$ else $v := t$;
```

Phase IV: Trial Division

The final trial division phase removes the extraneous divisors introduced during the main loop, and restores those removed in the first trial division phase.

```
if $v = 0$ then $t := u$ else $t := v$;
for every $d$ in table $P$ do:
    while $d \mid t$, $t := t/d$;
$g := t \times g$;
Output($g$);
```

2.4 Counting Iterations

We conclude this section by proving upper and lower bounds on the number of iterations of the main loop for all values of $k > 1$. The difficulty here is to bound the number of iterations which perform the transformation $u := u/\gcd(u,k)$. The idea is to show if we repeat this transformation twice, the first time we must get a large GCD. We begin with some definitions.

Let p be a prime, and n a positive integer. We say $p^e \parallel n$ if $p^e \mid n$ and $p^{e+1} \nmid n$.

Define $Q(n) = \min\{p^e \ : \ p^e \parallel n$, and p is prime $\}$. For example, if n is a prime power, $Q(n) = n$. Also $Q(40) = 5$ and $Q(20) = 4$.

Although not important for our application, note that computing the value of $Q(n)$ is polynomial-time equivalent to factoring n. We also have $Q(n) < \sqrt{n}$ unless n is a prime power, $(1/x)\sum_{n\leq x} Q(n) \sim x/(2\log x)$, and $\#\{n \leq x : Q(n) \leq 23\} > x/2$ for x sufficiently large.

Interestingly, the number of iterations of the main loop of the k-ary GCD algorithm depends on $Q(k)$.

Theorem 3 *If $k > 1$, $k = o(\log^2(uv))$, and $q = Q(k)$, then on positive integers u and v as input, the k-ary GCD algorithm computes $g = \gcd(u,v)$, and in the worst case the number of iterations is $\Theta(\log(uv)/\log q)$.*

We prove this using the following lemmas:

Lemma 4 *Let u and k be positive integers, p a prime, and $e > 0$ such that $p^e \parallel k$. Also let $g = \gcd(u,k)$ and $h = \gcd(u/g,k)$. If $p \mid h$ then $p^e \mid g$.*

Proof: Suppose $p \mid h$ and $p^f \parallel g$ with $f < e$. Since $h \mid u/g$, $gh \mid u$ and $p^{f+1} \mid u$. But $p^{f+1} \mid k$ as well, implying $p^{f+1} \mid g$ since $g = \gcd(u,k)$, a contradiction. □

Lemma 5 *If $k > 1$ and $q = Q(k)$ then the number of iterations of the main loop of the the k-ary GCD algorithm is $O(\log(uv)/\log q)$.*

Proof: Consider any two consecutive iterations of the main loop of the algorithm. It suffices to show that the product uv decreases by a factor d with $\log d = \Omega(\log q)$.

Case 1:

Suppose one of these two iterations performs the transformation

$$(u,v) \to (\min\{u,v\}, |au+bv|/k).$$

By Lemma 2, this reduces the product uv by a factor of $d \geq k/(2\lceil\sqrt{k}\rceil) > \sqrt{k}/2 - 1/2$. This gives $\log d = \Omega(\log k)$, and $k \geq q$.

We assumed that $\sqrt{k}/2 - 1/2 > 1$, or $k > 9$. However, one can show only $O(\log(uv))$ iterations are needed for all values of k from 2 to 9; simply construct the tables of a and b values to see that the product uv decreases by a factor larger than 1 at each iteration.

Case 2:

Alternatively, suppose both iterations perform divisions by $\gcd(u, k)$ or $\gcd(v, k)$. Since one of u and v must be relatively prime to k at each iteration, without loss of generality we assume $\gcd(v, k) = 1$. Let $g = \gcd(u, k)$, the divisor used for the first of the two iterations, and $h = \gcd(u/g, k)$, the second divisor used. Let p be a prime divisor of h; p must exist, for $g,h > 1$. By Lemma 4, $p^e \mid g$, where $p^e \parallel k$, and so $d = gh \geq p^{e+1} > q = Q(k)$. Thus $\log d \geq \log q$, as desired. □

Lemma 6 *Let $k > 1$ and $q = Q(k)$. There exist infinitely many inputs u, v such that the number of iterations of the main loop of the k-ary GCD algorithm is $\Omega(\log(uv)/\log q)$.*

Proof: Let f be a positive integer, let l be the product of all primes below $\sqrt{k}+1$, and let $u = q^f kl + 1$ and $v = 1$. If f is sufficiently large, we may assume $k = o(\log^2(uv))$. By the prime number theorem (see Hardy and Wright [HW79]), $\log l \sim \sqrt{k} = o(\log(uv))$. Since f is at least $(1/\log q)(\log u - \log l - \log k - 1)$, this means $f = \Omega(\log(uv)/\log q)$. We will show that the algorithm requires at least f iterations.

As u and v are both relatively prime to k and all primes below $\sqrt{k}$, the trial division phase does not alter u or v. The first iteration of the main loop will assign $a = 1$ and $b = -1$, changing u to $|u - v|/k = q^f l$. The next f iterations will each remove a factor q from u, plus common divisors of k and l if there are any. A full f iterations are needed because the GCD computations cannot remove a divisor of q^e larger than q at each iteration as $q = Q(k) \parallel k$. □

Proof of Theorem 3: Lemma 5 implies the algorithm halts, and so correctness follows from Lemma 1. The number of iterations follows from Lemmas 5 and 6. That completes the proof. □

We also give a brief heuristic average case analysis of the k-ary GCD algorithm. Let $B(k)$ denote the average number of bits removed from the product uv during each iteration of the main loop of the algorithm. If we assume that the residues of u and v modulo k are evenly distributed over the interval $0 \ldots k-1$ at each iteration, noticing that only one of u or v changes at each iteration, and if we further assume the values of a and b are evenly distributed, we reach the estimate

$$B(k) = \frac{1}{k}\left(\phi(k)\log_2(2\sqrt{k}) + \sum_{d|k}\phi(k/d)\log_2 d\right).$$

This implies that the average number of iterations is $\log_2(uv)/B(k)$. However, when compared with the implementation results in the next section, we find this consistently overestimates by roughly 25%, yet it preserves the relative ordering between different values of k.

We believe this is due to an uneven distribution in the values of a and b used from table A; smaller values of a occur more frequently than larger ones in practice. Clearly a more detailed average case analysis is called for.

3 A Sequential Version.

In this section we look at a sequential version of the k-ary GCD algorithm. We start by proving that, on n-bit inputs u and v, the algorithm has an $O(n^2 \log k / \log Q(k))$ worst-case running time according to the naive bit complexity model. If k is a prime power, this gives a quadratic running time. We next derive a heuristic estimate for the optimal value of k to use in the k-ary GCD algorithm as a function of the input size. Finally, we examine the results of a multiple-precision implementation that compares the k-ary GCD algorithm with several others, including the Euclidean and binary algorithms.

3.1 Complexity Results

Naive bit complexity assigns costs for basic arithmetic operations in terms of bit operations as follows:

- Addition/subtraction: To calculate $x+y$ or $x-y$ takes $O(\log x + \log y)$ bit operations.
- Multiplication/division: To calculate xy, x/y, or $x \bmod y$ takes $O((\log x)(\log y))$ bit operations.
- Comparison: To test if x is less than, equal to, or greater than y takes $O(\log x + \log y)$ bit operations.

Other operations such as flow of control, array indexing, and so on are assigned $O(1)$ cost. For more on naive bit complexity, see Bach and Shallit [BS90].

Theorem 7 *Let u and v be the inputs to the k-ary GCD algorithm, let $q = Q(k)$, and let $n = \log_2(uv)$. If $k = O((n/\log n)^2)$, then the algorithm takes at most $O(n^2 \log k / \log q)$ bit operations.*

Proof: We calculate the complexity of the algorithm by phase:

1. *Precomputation.* This phase can be done using $O(k \log^2 k)$ bit operations.
2. *Trial Division.* The two trial division phases have roughly the same cost. By the prime number theorem, there are at most $O(\sqrt{k}/\log k)$ divisions that give a non-zero remainder, one for each value of d, for a cost of $O(n\sqrt{k})$. The total cost for divisions with a remainder of zero is $O(n^2)$, for if $d_1 \ldots d_r$ are the associated divisors, $\sum_{i=1}^{r} \log d_i \leq n$. So the cost for both trial division phases is $O(n^2 + n\sqrt{k})$.
3. *Main Loop.* Each iteration costs at most $O(n \log k)$, and by Theorem 3, there are $O(n/\log q)$ iterations, giving a cost of $O(n^2 \log k / \log q)$.

Thus the total cost is $O(n^2 \log k / \log q + k \log^2 k + n\sqrt{k}) = O(n^2 \log k / \log q)$. □

So for k a prime power, since we have $Q(k) = k$, the algorithm has a quadratic running time.

3.2 Optimal k

Next, we address the question of what to choose for k.

Under the naive bit complexity model, we proved in Theorem 7 that if k is a prime power and not too large, the k-ary GCD algorithm has a quadratic running time independent of the precise value of k. However in practice the value of k is indeed important; we must balance the cost of precomputation and trial division when k is large with the gain in the number of iterations of the main loop. So we will discard the naive bit complexity model for now and view the complexity of the k-ary GCD algorithm from how it might behave in practice.

On an actual computer, all arithmetic operations on single-precision integers (ones that fit in one machine word) take $O(1)$ time, more or less independent of their size. This implies the cost of multiplying or dividing a multiple-precision integer x by a single-precision integer y takes $O(\log x)$ time. Let $n = \log(uv)$, where u and v are the inputs to the algorithm as before. Then if k is single-precision, the "actual" complexity of the k-ary algorithm is:

1. $O(k \log k)$ for precomputation,
2. $O(n\sqrt{k}/\log k + n^2/\log k)$ for trial division, assuming a list of primes is precomputed, and
3. $O(n^2/\log k)$ for the main loop (on the average).

This gives a total running time of

$$T(n,k) = O(k \log k + (\sqrt{k}/\log k)n + n^2/\log k).$$

To minimize T, we differentiate with respect to k and then set T' to zero giving the root $k_0 = \Theta((n/\log n)^2)$. This is the only positive zero of T', and is easily seen to be a relative minimum.

We call the k-ary GCD algorithm that uses $k = \lfloor c \cdot (n/\log n)^2 \rfloor$ for some positive constant c the *adaptive* k-ary GCD algorithm. Obviously, the best choice for c will depend on the relative costs of the various arithmetic operations for both single- and multiple-precision integers.

Finally, we remark that if k divides the multiple-precision base used, then calculating $u \bmod k$ and $v \bmod k$ become constant time operations, a desirable improvement.

3.3 Implementation Results

We will now discuss the results from a multiple-precision implementation of several GCD algorithms. In the following discussion, we will present four tables of timing results.

All four tables have the same format: The leftmost column gives the names of the algorithms. Across the top are the input sizes, which range from 10 to 1000 decimal digits in length. Each algorithm was run on 5 pseudo-random pairs of integers of each size. In the box indexed by the algorithm name and input size are two numbers. The top number gives the average time spent by that algorithm on inputs of that size in CPU seconds. The times for the k-ary algorithms include precomputation on every input. The bottom number is the average number of iterations of the main loop for that algorithm.

All the algorithms used the same library of multiple-precision arithmetic routines, written in Pascal, and run on a DECstation 3100 using the Ultrix operating system. The base used for multiple-precision integers was 32768.

Below are the results of timing trials for several previous GCD algorithms.

	Decimal Digits						
Algorithm	*10*	*25*	*50*	*100*	*250*	*500*	*1000*
Euclidean	0.005	0.019	0.049	0.178	1.070	4.972	25.948
Algorithm	18	49	90	195	480	967	1929
Least-Remainder	0.004	0.015	0.039	0.135	0.791	3.591	18.488
Euclidean Alg.	13	34	65	136	338	673	1340
Purdy's	0.005	0.016	0.044	0.130	0.652	2.389	9.156
Algorithm	88	225	477	938	2385	4824	9650
Binary	0.003	0.009	0.024	0.066	0.313	1.121	4.264
Algorithm	44	116	239	469	1167	2342	4709
Left-Shift	0.004	0.011	0.025	0.069	0.280	0.916	3.316
Binary Algorithm	22	55	105	223	551	1087	2196

If $u > v$, the left shift binary algorithm used here computes an integer e at each iteration such that $2^e v \leq u < 2^{e+1} v$ and then assigns to u the smaller of $u - 2^e v$ and $2^{e+1} v - u$.

We now illustrate how the best value for k increases with the input size. In the table below we have results for the k-ary algorithm using the 10th, 25th, 50th, 100th, 250th, and 500th primes.

	Decimal Digits						
Algorithm	*10*	*25*	*50*	*100*	*250*	*500*	*1000*
29-ary	0.007	0.012	0.027	0.070	0.335	1.213	4.625
	15	38	77	151	378	754	1515
97-ary	0.014	0.018	0.030	0.066	0.285	1.008	3.827
	12	28	59	121	304	609	1213
229-ary	0.034	0.039	0.050	0.082	0.277	0.916	3.400
	10	27	55	107	268	530	1065
541-ary	0.079	0.083	0.092	0.122	0.296	0.888	3.139
	10	24	48	95	237	482	967
1583-ary	0.244	0.247	0.256	0.285	0.447	0.976	2.978
	9	21	43	87	217	430	854
3571-ary	0.557	0.560	0.568	0.598	0.752	1.241	3.107
	8	19	40	81	202	398	790

From this data, we decide that a reasonable value for k is roughly $(1/2) \cdot (n/\log n)^2$ where n is the length of the inputs in base 32768.

Next we investigate whether the running time of the k-ary algorithm is indeed sensative to the factorization pattern of k as Theorem 3 suggests. Below we have results for four values of k near 620, each with a different factorization pattern. We also list $Q(k)$ under the name of each algorithm in the table.

	Decimal Digits						
Algorithm	*10*	*25*	*50*	*100*	*250*	*500*	*1000*
619-ary	0.091	0.095	0.105	0.134	0.309	0.878	3.082
$Q(619) = 619$	10	23	47	92	236	473	945
621-ary	0.071	0.075	0.085	0.115	0.298	0.894	3.189
$Q(621) = 23$	11	28	54	112	279	565	1122
624-ary	0.058	0.062	0.073	0.106	0.301	0.947	3.440
$Q(624) = 3$	14	34	71	136	346	687	1375
625-ary	0.080	0.084	0.094	0.123	0.301	0.880	3.134
$Q(625) = 625$	10	25	51	103	264	518	1047

For larger inputs, the algorithms' running times are ordered according to the values of $Q(k)$, with the exception of 619 beating 625. In fact, this is what we expect; the 625-ary algorithm often performs divisions by 5.

Finally, we have two adaptive versions of the k-ary algorithm to match against the binary and left shift binary algorithms.

In light of the sensativity of the k-ary algorithm to the factorization of k, we had our adaptive algorithm choose k with few small prime divisors where possible. Though this does not insure $Q(k)$ is large, it does help.

We also use a *base-adaptive* algorithm, which is an adaptive algorithm that chooses a value of k of near-optimal size which also divides the multiple precision base. In this case this means choosing k a power of 2. The results are below.

	Decimal Digits						
Algorithm	*10*	*25*	*50*	*100*	*250*	*500*	*1000*
Binary	0.003	0.009	0.024	0.066	0.313	1.121	4.264
Algorithm	44	116	239	469	1167	2342	4709
Left-Shift	0.004	0.011	0.025	0.069	0.280	0.916	3.316
Binary Algorithm	22	55	105	223	551	1087	2196
Adaptive k-ary	0.004	0.011	0.026	0.065	0.273	0.879	2.985
	25	48	85	132	261	463	842
$k =$	5	11	27	61	271	841	2641
Base-Adaptive	0.004	0.009	0.021	0.051	0.202	0.617	2.086
k-ary	31	60	104	162	334	601	1045
$k =$	4	8	16	64	256	512	2048

Note that the k-ary algorithms in general are faster than previous algorithms, and adapting k to the multiple precision base is very beneficial.

We also implemented these algorithms on a VAXstation 3200 II running Unix. This machine has a slower processor with a much larger instruction set, and on this machine the k-ary algorithm does slightly better compared to previous algorithms than on the DECstation.

One unusual point is that on the VAX, the binary algorithm consistently outperformed the left shift binary algorithm.

We would like to mention one important application for GCD algorithms: factoring integers. Our k-ary GCD algorithm is especially well-suited for factoring algorithms that use many GCD calculations, for in this case the precomputation step need only be done once, and in many cases the first trial division phase can be skipped altogether. Examples of such factoring algorithms include the various cyclotomic methods [BS89, Pol74, Wil82] and Pollard's ρ-method [Pol75]. See also the paper by Montgomery [Mon87].

That concludes our discussion of sequential implementations for the k-ary GCD algorithm.

4 Some Parallel Algorithms.

As we mentioned in the introduction, in this section we give four different PRAM algorithms based on the k-ary method discussed in section 2. We begin by reviewing the parallel complexity of the basic arithmetic operations. Let n denote the length of the inputs in bits, and let $M(n) = n \log n \log\log n$.

- On a CRCW PRAM, addition and subtraction take $O(1)$ time using $O(n \log\log n)$ processors. See Chandra, Fortune, and Lipton [CFL85].
- On an EREW PRAM, addition and subtraction take $O(\log n)$ time and $O(n)$ processors.
- On an EREW PRAM, multiplication takes $O(\log n)$ time and $O(M(n))$ processors. See Schönhage and Strassen [SS71].
- On an EREW PRAM, division takes either:

  $O(\log n \log\log n)$ time and $O(M(n))$ processors (see Reif and Tate [RT89]), or

  $O(\log n)$ time and $n^{O(1)}$ processors (see Beame, Cooke, and Hoover [BCH86]).

The division algorithm of Beame, Cooke, and Hoover, when realized as a bounded fan-in boolean circuit, is not known to be logspace uniform. For a general reference, see Karp and Ramachandran [KR88].

We now prove two lemmas concerning the parallel complexity of the precomputation and trial division phases, and then give our PRAM algorithm results.

Lemma 8 *The preprocessing phase takes $O(\log k)$ time on an EREW PRAM and $O(\log\log k \log\log\log k)$ time on a CRCW PRAM, both using $O(k^2 M(\log k))$ processors.*

Proof: Begin by finding all the primes up to k using a parallel sieve. Next completely factor all the integers up to k in parallel. From this information tables G and P are easily computed. Table I is computed by multiplying together all pairs of integers up to k to see if the result is 1 mod k. Finally, table A is computed using exhaustive search. □

Lemma 9 *On an EREW PRAM, the trial division phases of the k-ary GCD algorithm take either*

- *$O(\sqrt{k}\log^3 n)$ time and $O(M(n))$ processors, or*
- *$O(\log n)$ time and $(nk)^{O(1)}$ processors.*

Proof: For the first one, we sequentially test each of the $O(\sqrt{k})$ divisors d by performing a binary search to compute the largest e such that $d^e \mid u$ or $d^e \mid v$. Each binary search will require $O(\log n)$ iterations, constructing d^e will take $O(\log^2 n)$ time, and the division takes $O(\log n \log\log n)$ time, all using at most $O(M(n))$ processors. That proves the first half of the lemma.

For the second one, we simply test all powers of all possible prime divisors in parallel using Beame, Cooke, and Hoover's division algorithm. We then in parallel find the largest power of each divisor that divided both u and v. These powers of divisors are then multiplied together using the iterated product $\mathcal{NC}^1$ reduction to division, and this product is divided out of both u and v. □

Theorem 10 *Let $\epsilon > 0$. If $k = 2^r$ with $r = \Omega(\log n)$ and $r \le (1/2 - \epsilon)\log n$, then the k-ary GCD algorithm has a running time of $O(n)$ using $O(M(n))$ processors on an EREW PRAM.*

Proof: By Theorem 3 there are $O(n/\log n)$ iterations of the main loop. Division by k takes $O(1)$ time and $O(n)$ processors, since integers are represented in binary. Thus each iteration of the main loop requires only $O(\log n)$ time and $O(M(n))$ processors. The main loop then takes $O(n)$ time and $O(M(n))$ processors, and so the result follows from Lemma 8 and the first part of Lemma 9. □

The author is unaware of any other EREW PRAM algorithm to compute GCDs in linear time using only $O(M(n))$ processors.

Theorem 11 *Let $\epsilon > 0$. If $k = 2^r$ with $r = \lfloor \epsilon \log n \rfloor + O(1)$ such that r is even, then the k-ary GCD algorithm has running times of $O(n/\log n)$ and $O(n \log\log n/\log n)$ on CRCW and CREW PRAMs respectively, using $O(n^{1+\epsilon})$ processors.*

Proof: By Theorem 3 there are $O(n/\log n)$ iterations of the main loop. We will show how each iteration takes $O(1)$ time on a CRCW PRAM. We use many ideas from Chor and Goldreich [CG90].

As in Theorem 10, division by k takes $O(1)$ time and $O(n)$ processors since integers are represented in binary. We also precompute a table of $O(k)$ entries to contain the products of all integers $\le O(\sqrt{k})$, allowing the product of integers with $O(\log k)$ bits to be computed in $O(1)$ time using a table lookup. This additional precomputation time is $O(\log\log k)$ using $O(kM(\log k))$ processors, easily within our claimed bound. We now can compute each iteration in $O(1)$ time and $O(M(n))$ processors aside from computing the products au and bv. For this we use the following idea in Chor and Goldreich.

To compute au, do the following. First write u in base $\sqrt{k} = 2^{r/2}$ as $u = \sum_{i=0}^{l} u_i(\sqrt{k})^i$ where l is roughly $2\log u/\log k$. Note that since r is even, the u_i can be computed in $O(1)$ time. Second, compute x and y such that $u = x + y$ where x is the sum of the odd terms in the above expansion for u, and y is the sum of the even terms. For example, if $\sqrt{k} = 10$ (for human readability) and $u = 123456$, then $x = 103050$ and $y = 020406$. The products ax and ay can then be computed in $O(1)$ time with table lookups using $O((n/\log\sqrt{k})\cdot k)$ processors, since there are no carries. We then compute $au = ax + ay$. Following our example, if $a = 5$ then $ax = 515250$ and $ay = 102030$, giving $au = 617280$. By our choice of k, this uses $O(n^{1+\epsilon})$ processors. We compute bv the same way.

Since $k = O(n^\epsilon)$, the theorem now follows from Lemma 8 and the first part of Lemma 9 for the CRCW case.

The CREW PRAM algorithm is much the same, however a multiplicative factor of $O(\log\log k) = O(\log\log n)$ time is needed for table lookups, and the additions must be pipelined. □

The results of this last theorem match those by Chor and Goldreich [CG90]. Note that they use a restricted CRCW PRAM; it allows concurrent writes only if the same value is being written. Theorem 11 can be easily modified to use the same restricted CRCW PRAM model.

Corollary 12 *Let $d \geq 1$. If $k = 2^r$ for $r = \lfloor n/\log^d n\rfloor$, then the k-ary GCD algorithm takes $O(\log^d n)$ time and $\exp[O(n/\log^d n)]$ processors on a CRCW PRAM.*

Proof: Follows from the proof Theorem 11, only use the second part of Lemma 9. □

The probabilistic CRCW PRAM algorithm of Adleman and Kompella [AK88] runs in time $O(\log^2 n)$ using $\exp[O(\sqrt{n\log n})]$ processors. Our algorithm runs faster (choose $d < 2$) but with far more processors; we can achieve the same processor bound at the cost of giving up the polylog running time. Also note that, unlike Adleman and Kompella's algorithm, ours uses no randomness.

5 Conclusion.

We have show that the k-ary GCD algorithm gives both practical sequential algorithms and several interesting parallel algorithms. We close with some unresolved questions:

- We proved that the number of iterations of the main loop is $\Theta(n/\log Q(k))$, and we gave a heuristic estimate for the average case. Is it possible to show the average case is $O(n/\log k)$?

- Computing inverses looks difficult and inefficient using the k-ary GCD algorithm; the straightforward approach gives a cubic time sequential extended k-ary GCD algorithm. Is it possible to do better? Is there an efficient parallel version?

- In the worst case, the second trial division phase might involve the complete factorization of an integer with $\Omega(\log(uv))$ bits. Is this indeed true? Is this the average case, or can one show this integer is small on the average? If this number is large on the average, it may be that periodic trial division steps within the main loop to remove the extraneous divisors accumulated so far will speed the algorithm.

- Does anything interesting result if k is varied within the algorithm? If k is chosen a power of two, but allowed to vary, and if $a = 1$ is always used, the resulting algorithm resembles Norton's shift-remainder GCD algorithm [Nor87].
- The algorithm calculates the a, b pairs using precomputation. Is it possible to find optimal pairs efficiently without any precomputation? If so, it may be possible to design better parallel algorithms.
- D. H. Lehmer gave a multiple precision variant of the Euclidean algorithm [Leh38], and Knuth mentions a similar variant for the binary algorithm due to R. W. Gosper (see Knuth [Knu81]). Is such a variant possible for the k-ary algorithm?

Acknowledgements

I want to thank Eric Bach not only for his advice, encouragement, and support as my Ph. D. advisor, but also for his proof of Lemma 2. My thanks also go to Joao Meidanis for his assistance on a point regarding the preprocessing phase of the algorithm, and to Kevin McCurley and Carl Pomerance for their comments on an earlier draft of this paper. Finally I want to thank Jeff Shallit, who introduced me to the world of GCD algorithms through the course he taught with energy and enthusiasm on Number Theoretic Algorithms while visiting Wisconsin in the Fall of 1989, and whose suggestion to implement some of those algorithms led to this paper.

References

[AK88] L. M. Adleman and K. Kompella. Using smoothness to achieve parallelism. In *20th Annual ACM Symposium on Theory of Computing*, pages 528–538, 1988.

[BCH86] P. W. Beame, S. A. Cook, and H. J. Hoover. Log depth circuits for division and related problems. *SIAM J. Comput.*, 15:994–1003, 1986.

[BK83] R. P. Brent and H. T. Kung. Systolic VLSI arrays for linear-time GCD computation. In F. Anceau and E. J. Aas, editors, *Proceedings of VLSI '83*, pages 145–154. Elsevier, 1983.

[Bre76] R. P. Brent. Analysis of the binary Euclidean algorithm. In J. F. Traub, editor, *Algorithms and Complexity*, pages 321–355. Academic Press, 1976.

[BS89] E. Bach and J. Shallit. Factoring with cyclotomic polynomials. *Math. Comp.*, 52(185):201–219, 1989.

[BS90] E. Bach and J. Shallit. Algorithmic number theory. In preparation, 1990.

[CFL85] A. K. Chandra, S. Fortune, and R Lipton. Unbounded fan-in circuits and associative functions. *Journal of Computer and System Sciences*, 30, 1985.

[CG90] B. Chor and O. Goldreich. An improved parallel algorithm for integer GCD. *Algorithmica*, 5:1–10, 1990.

[Den89] X. Deng. On the parallel complexity of integer programming. In *1st Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 110–116, 1989.

[HW79] G. H. Hardy and E. M. Wright. *An Introduction to the Theory of Numbers.* Oxford University Press, 5th edition, 1979.

[KMR87] R. Kannan, G. Miller, and L. Rudolph. Sublinear parallel algorithm for computing the greatest common divisor of two integers. *SIAM J. Comput.*, 16(1):7–16, 1987.

[Knu81] D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison-Wesley, Reading, Mass., 2nd edition, 1981.

[KR88] R. Karp and V. Ramachandran. A survey of parallel algorithms for shared-memory machines. Technical Report UCB/CSD 88/408, Computer Science Division, University of California, 1988. To appear in *Handbook of Theoretical Computer Science*, North-Holland.

[Leh38] D. H. Lehmer. Euclid's algorithm for large numbers. *Amer. Math. Monthly*, 45:227–233, 1938.

[Mon87] P. L. Montgomery. Speeding the Pollard methods of factorization. *Math. Comp.*, 48(177):243–264, 1987.

[Nor85] G. Norton. Extending the binary GCD algorithm. In J. Calmet, editor, *Proceedings of the 3rd International Conference on Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, pages 363–372. Springer-Verlag, 1985. LNCS 229.

[Nor87] G. Norton. A shift-remainder GCD algorithm. In L. Huguet and A. Poli, editors, *Proceedings of the 5th International Conference on Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, pages 350–356. Springer-Verlag, 1987. LNCS 356.

[Pol74] J. M. Pollard. Theorems on factorization and primality testing. *Proc. Camb. Phil. Soc.*, 76:521–528, 1974.

[Pol75] J. M. Pollard. A Monte Carlo algorithm for factorization. *BIT*, 15:331–334, 1975.

[Pur83] G. B. Purdy. A carry-free algorithm for finding the greatest common divisor of two integers. *Comp. & Maths. with Appls.*, 9(2):311–316, 1983.

[RT89] J. H. Reif and S. R. Tate. Optimal size integer division circuits. In *21st Annual ACM Symposium on Theory of Computing*, pages 264–273, 1989.

[Sch71] A. Schönhage. Schnelle Berechnung von Kettenbruchentwicklungen. *Acta Informatica*, 1:139–144, 1971.

[SS71] A. Schönhage and V. Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7:281–292, 1971.

[Wil82] H. C. Williams. A $p+1$ method of factoring. *Math. Comp.*, 39(159):225–234, 1982.