Parallel Implementation of the Accelerated Integer GCD Algorithm

KENNETH WEBER†
Instituto de Matemática, Universidade Federal do Rio Grande do Sul, Brazil‡

(Received 19 May 1995)

The accelerated integer greatest common divisor (GCD) algorithm has been shown to be one of the most efficient in practice. This paper describes a parallel implementation of the accelerated algorithm for the Sequent Balance, a shared-memory multiprocessor. For input of roughly 10000 digits, it displays speed-ups of 1.6, 2.5, 3.4 and 4.0 using 2, 4, 8 and 16 processors, respectively.

© 1996 Academic Press Limited

1. Introduction

The greatest common divisor (GCD) of two integers is one of the most common and most important arithmetic operations. As the power of computer hardware and software and our understanding of computational mathematics has grown, the size of the integers for which the GCD must be computed has also grown. For example, the multivariate polynomial GCD algorithm GCDHEU, of Char et al. (1989), may require the calculation of the greatest common divisor of two integers that are thousands of digits long when computing the GCD of two “industrial-size” input polynomials.

The accelerated integer GCD algorithm, a derivative of the right-shift \( k \)-ary GCD algorithm of Sorenson (1994) has been shown to be very good for computing the GCD of moderate- to large-sized integers sequentially. In Weber (1995), it has been shown to be twice as fast, for 50 digits, and six times as fast, for 2000 digits, as the well-known GNU MP implementation of the binary GCD algorithm (Granlund, 1991). This paper describes the first implementation of a straightforward parallelization of the accelerated algorithm on a shared-memory multiprocessor—the Sequent Balance. An independent development by Jebelean (1993) of nearly the same algorithm was designed for parallel implementation on a pipelined systolic device proposed as a rational arithmetic co-processor.

Sorenson’s \( k \)-ary reduction, which is the heart of the accelerated algorithm, is more amenable to straightforward parallelization than any of the reductions used by previous algorithms. The bulk of the work required by the reduction is a linear combination of the form

\[
\frac{(au - bv)}{2^r},
\]

(1.1)

† E-mail: kweber@mat.ufrgs.br
‡ Formerly Department of Mathematics and Computer Science, Kent State University, U.S.A.
Input: \( u_0, v_0 > 0 \) odd, with \( \ell(u_0) \geq \ell(v_0) \)
Output: \( \text{gcd}(u_0, v_0) \)

AGCD1: [Save initial values]
\[ u \leftarrow u_0, v \leftarrow v_0 \]

AGCD2: [Acceleration loop]
\[ \text{while } v \neq 0 \text{ do} \]
\[ \text{if } \ell(u) - \ell(v) > s \text{ then} \]
\[ u \leftarrow \text{bmod}(u, v) \]
\[ \text{else [k-ary reduction]} \]
\[ (b, a) \leftarrow \text{ReducedRatMod}(u, v, 2^{t_2}) \]
\[ u \leftarrow |au - bv|/2^{t_2} \]
\[ \text{RemoveFactors}(u, 2) \]
\[ \text{swap}(u, v) \]

AGCD3: [Remove spurious factors]
\[ \text{return } \text{bmodgcd}(u_0, \text{bmodgcd}(v_0, u)) \]

Figure 1. Accelerated GCD algorithm.

where \( a \) and \( b \) are much smaller than \( u \) and \( v \). The parallel implementation described below is based on a straightforward parallelization of this linear combination. The “factoring” technique, described by Hummel et al. (1992), was used to schedule the parallel subtasks that work together to compute Formula 1.1. For input of roughly 10,000 digits, the parallel implementation displays speed-ups of 1.6, 2.5, 3.4 and 4.0 using 2, 4, 8 and 16 processors, respectively. For 40,000 digits it displays a speed-up of 5.8 for 16 processors. Thus, a parallel implementation of the accelerated algorithm seems practical on a shared-memory multiprocessor only for large input, such as that produced by the GCDHEU algorithm mentioned above. A more complicated parallelization of an asymptotically fast GCD algorithm due to Schönhage (1971) may provide better results.

An overview of the accelerated algorithm and asymptotic parallel complexity follow in the next two sections. The parallel implementation is described in detail in Section 4, and results from timing the implementation are presented in Section 5. The paper ends with a summary and some ideas for further development.

2. Overview of Algorithm

The accelerated integer GCD algorithm is described in detail in Weber (1995). In this section we give a brief overview of the algorithm, displayed in Figure 1. It is assumed that each integer is represented in base 2, as a sequence of \( W \)-bit unsigned integers, which we call limbs, borrowing terminology from Granlund (1991). The notation \( \ell(u) \) represents the number of bits in the binary representation of \( u \).

Many GCD algorithms work by replacing the original pair of input values with a pair of smaller integers that has the same GCD. This is done repeatedly, until one of the pair is zero. The non-zero value is then the GCD of the original input pair. The accelerated algorithm replaces the original pair by a smaller pair whose GCD is a multiple of the GCD of the original pair. When one of the values in the pair reaches zero, the spurious factors are eliminated from the non-zero value (step AGCD3).

In Step AGCD2, two different reduction strategies are employed, depending on the difference in the sizes of \( u \) and \( v \). If the difference is relatively small, \( u \) is replaced by \( \text{bmod}(u, v) \), defined as

\[ \text{bmod}(u, v) = |u - (u/v \mod 2^{c})v|/2^{c} \]
Input: \(u, v > 0, k > 1\) with \(\gcd(u, k) = \gcd(v, k) = 1\)
Output: \(f = (n, d)\) such that \(0 < |n|, |d| < \sqrt{k}\) and \(nv \equiv du \pmod{k}\)

Figure 2. The ReducedRatMod algorithm.

where \(c = \ell(u) - \ell(v) + 1\). If the difference in the sizes of \(u\) and \(v\) is large, the \(k\)-ary reduction, proposed by Sorenson (1994) is used, with \(k = 2^{2^t}\). First, ReducedRatMod (see Figure 2) is used to compute \(a\) and \(b\) such that \(au - bv \equiv 0 \pmod{2^{2t}}\) and \(|a|, |b| < 2^t\). Then \(u\) is replaced by the smaller number \((au - bv)/2^{2t}\). Although the bmod reduction preserves the greatest common divisor, the \(k\)-ary reduction does not; the greatest common divisor of the new pair is a multiple of that of the old.

Once the reduction is made, any powers of two are removed from the result by RemoveFactors, by shifting the result right until there are no low 0-bits.

The parameters \(s\) and \(t\) are used to tune the algorithm. (Although in the implementation described below they are constant, it is possible to let them grow smaller as \(u\) and \(v\) themselves shrink.) The value \(s\) represents how close in size \(u\) and \(v\) must be in order for a \(k\)-ary reduction to be used and \(t - 1\) is the minimum number of bits by which \(u\) is reduced in a \(k\)-ary reduction.

The elimination of the spurious factors from the result of the reductions is done by computing the greatest common divisor of the result, \(u\) and \(v\), using bmodgcd, an algorithm similar to the Euclidean algorithm (p. 320, Knuth, 1981), except that the bmod operation is used instead of the mod operation.

Consider the following example; \(u = 70211\) and \(v = 55433\), and we may choose \(s\) to be 4 and \(t\) to be 4, so \(k = 2^{2^4} = 256\). These numbers are quite appropriate when the machine’s word size is 8 bits. The accelerated algorithm computes the following sequence of values for \(v\) at the while loop test.

\[
\begin{align*}
55433 \\
2425 &= (12 \cdot 70211 - 4 \cdot 55433)/2^8 \\
111 &= (55433 - 17 \cdot 2425)/2^{4+5} \\
1 &= (2425 - 23 \cdot 111)/2^{1+6} \\
0 &= (111 - 111 \cdot 1)/2^7.
\end{align*}
\]

It is unnecessary to eliminate the spurious factors from the final value of \(u = 1\), since it is as small as possible already.
3. Asymptotic Complexity

In this section we state a conservative bound on the time required by the parallel version of the accelerated algorithm, and compare it with the fastest-known sequential algorithm.

It seems best to provide a conservative asymptotic analysis considering that our objective is to study efficient implementations of parallel GCD algorithms, rather than algorithms which might prove the fastest in the limit but inefficient in practice. This analysis is conservative in two ways; first, the computational model (EREW-PRAM) is the most restrictive version of the PRAM family (Karp and Ramachandran, 1990), and seems to fit more closely a wide range of architectures in use. Second, although it is possible to implement some of the subalgorithms much more efficiently, (Sorenson, 1994), we did not do so, because the implementations require special hardware or huge tables.

In the following discussion we let \( n = \ell(u) \).

**Theorem 3.1.** The accelerated integer GCD algorithm takes \( O(n^2/p + n \log p) \) time when implemented on an EREW PRAM with \( p \) processors.

**Proof.** Consider each step of the algorithm separately (see Figure 1). Step AGCD1 requires \( O(n/p) \) operations.

Step AGCD2 is the main loop; first we analyse the cost of each iteration. ReducedRatMod takes \( O(t^2) = O(\log^2 n) \) operations. RemoveFactors takes \( O(n/p) \) operations. The linear combination and right-shift required in both the bmod and \( k \)-ary reductions require \( O(n/p) \) arithmetic operations, plus an extra \( O(\log p) \) on an EREW PRAM for propagation of the carry bit in the final addition (see Section 4 for more details). Thus the time spent in one iteration is dominated by the time it takes to perform all the linear combinations needed. Each \( k \)-ary reduction requires one linear combination, but each bmod reduction requires a number of combinations proportional to the number of bits by which \( u \) is reduced. [See Section 3 of Weber (1995) for a description of an algorithm for bmod.] Let \( \lambda_i \) be the number of bits by which \( u \) is reduced in the \( i \)th iteration; then \( \sum_i \lambda_i \leq n \), and the cost of one iteration is \( O(\lambda_i(n/p + \log p)) \). We can now give the cost of the main loop as

\[
O\left(\sum_i \lambda_i(n/p + \log p)\right) = O(n^2/p + n \log p).
\]

A similar argument shows that the bmodgcd algorithm used in step AGCD3 takes \( O(n^2/p + n \log p) \) time, thus establishing the validity of the theorem. \( \square \)

If \( n \approx p \), the time complexity becomes \( O(n \log n) \). If \( n \gg p \), we can think of the time-complexity as \( O(n^2/p) \).

The fastest known sequential algorithm is due to Schönhage (1971). It can compute the GCD of two integers in \( O(n \log^2 n \log \log n) \) time. Thus it is clear that the parallel implementation of the accelerated algorithm should prove faster than an implementation of Schönhage’s sequential algorithm, given enough processors. It is not clear, however, which algorithm to use when only a few processors are available, and whether a parallel implementation of Schönhage’s algorithm would prove better. The main body of the algorithm is recursive, and appears to be inherently sequential. It does, however, use multiplication of large integers as a subalgorithm. Many parallel integer multiplication algorithms have been described in the literature; Schönhage and Strassen (1971) show...
that multiplication can be performed in \( \log n \) time given enough processors. In any case, a parallel implementation of Schönhage's algorithm and comparison to the one presented here is best left for another study.

4. Parallel Implementation

In this section we describe the parallel implementation of the accelerated algorithm for a Sequent Balance shared-memory multiprocessor. The basis for the parallel implementation is the sequential implementation of the algorithm (Weber, 1995). The parallel implementation is written in C and calls on routines from the GNU Multiple Precision Arithmetic Library (Granlund, 1991). GMP uses 32-bit limbs when representing multi-precision integers on the Sequent Balance; that is, \( W = 32 \). The value used for \( t \) is 32; this allows the \( a \) and \( b \) values in the \( k \)-ary reduction to fit into one limb. By experimentation we settled on \( s = 10 \).

The GNU C Compiler (Stallman, 1991) was used to compile the parallel implementation. Dynix system calls provided for process management and shared memory allocation was performed by `shmalloc` and `shfree` from the Sequent Parallel Programming Library (Osterhaug, 1989).

Experience (Weber, 1990) suggests that when implementing a parallel program on a shared memory multiprocessor, it is important to identify the parts of the algorithm that allow for the maximum amount of independent, parallel execution. Only these parts are performed in parallel; the rest are computed sequentially. In the accelerated algorithm, the linear combination and right shift of Formula 1.1 (where \( 2^r \) exactly divides the numerator) is used both in the \( k \)-ary and the \texttt{bmod} reduction steps. It provides the most opportunity for speedup from parallel execution. We focused our attention on this operation, converting to parallel execution only those routines from the sequential implementation that perform the linear combination.

To perform the linear combination in parallel, the limbs in the base-\( 2^W \) representations of \( u \) and \( v \) are partitioned in such a way that \( u \) and \( v \) can be considered to be represented in mixed-radix form (pp. 192–193, Knuth, 1981). Thus the parallel calculation of the linear combination becomes

\[
a u + b v = a \sum_{i=0}^{n-1} u_i \prod_{j=0}^{i-1} B_j + b \sum_{i=0}^{n-1} v_i \prod_{j=0}^{i-1} B_j = \sum_{i=0}^{n} (a u_i + b v_i) \prod_{j=0}^{i-1} B_j
\]

where \( B_i \) is a power of \( 2^W \) (perhaps different for each \( i \), and the “super-digits” \( u_i, v_i \) are themselves multi-precision integers in the range 0 to \( B_i - 1 \). Each \( a u_i + b v_i \) is computed in parallel, producing a super-digit \( s_i \) and a carry/borrow out \( c_i \), with \( 0 \leq s_i < B_i \) and \( 0 \leq c_i < 2^W + 1 \). By propagating the carry/borrow, the result is obtained as groups of limbs.

We were also able to perform the division by \( 2^r \) (i.e., right-shift by \( r \)) in the same parallel phase. It is necessary to compute \( r = r_1 W + r_0 \) (where \( r_0 < W \)), by finding the lowest non-zero limb of the result, before partitioning \( u \) and \( v \) and going into parallel mode. Division by \( 2^r \) can be handled in the partitioning, so that the parallel phase also must perform a right-shift by \( r_0 \) bits. In order to do this, each process participating in the parallel computation must have access to the low limb of the next higher super-digit.

The final carry/borrow propagation in the linear combination could also be done in parallel, but since it does not speed up the computation enough to justify the additional
work needed, we chose to perform the propagation sequentially. In fact, sequential propagation may be significantly faster than its parallel counterpart for the average case; the carry out of a group of limbs operated on by some job will most likely be absorbed into the next highest limb or two. It is highly unlikely that a carry will go beyond two limbs. The carry/borrow out of a group of limbs is never more than $2^{W+1} - 1$, so the probability that a carry will propagate out of the second limb is about 1 in $2^W$, since a carry out will only occur if there is a carry into the limb and that limb is equal to $2^W - 1$.

Besides the parallel computation of the linear combination, it is also possible to overlap the sequential computation of the next $a$ and $b$ values with the calculation of the current linear combination. This would require a major reorganization of the current implementation, and was not attempted. This possible overlap is one of the features of the accelerated algorithm that make it attractive for implementation on a systolic array (Jebelean, 1994).

The parallel implementation is organized around a job queue. At the start of the program, one main process is chosen to perform all the sequential code. The other worker processes poll the job queue, waiting for a job to do. When the main process comes to a parallel section, it partitions the overall task of computing the linear combination into independent jobs and places them on the queue. The main process then polls the queue along with the worker processes, until the entire collection of jobs for that task is finished. Then the main process continues executing sequentially.

We use the factoring method described in Hummel et al. (1992) to partition a task into jobs. The amount of work in a given task is approximated by the length of the input arrays, so partitioning the task into subtasks means, in this case, deciding the size of the super-digits $u_i$ and $v_i$ mentioned above. First, half of the work of the task is divided into $p$ nearly equal parts, where $p$ is the total number of processes participating in the parallel program. (We assume there are less processes than available processors.) Then half of the remaining work is subdivided into $p$ nearly equal parts, and so on, until some minimum job size is reached. The advantage in factoring over other scheduling algorithms is that the partitioning is not only easy to calculate but also effective in keeping all the processors involved in the computation busy.

5. Results

In this section we present and analyse actual timings of the implementation described above. The test programs were compiled with the GNU C Compiler, version 2.4.5, and linked with GMP-1.3 (see Section 4). The times are given in Table 1. For each input length $\ell$, given as a number of 32-bit limbs, 8 pairs of integers were selected pseudo-randomly. The average time required by the implementation to find the GCDs for all 8 pairs is given when using 1, 2, 4, 8 and 16 processors. Times are given in hundredths of a second. Parallel computation of the linear combination was not performed when the number of limbs dropped to 16 or less.

Since the bmod operation is used in both the accelerated GCD and the bmodgcd algorithms, a parallel implementation of bmodgcd was produced as a side effect of preparing the parallel accelerated implementation from the sequential implementation. The times for this implementation are given in Table 2 for comparison.

The process initialization and shutdown times are not included in the times given in the above tables. They are given separately in Table 3. The first initialization/shutdown
in a program takes much longer. This extra time is probably the overhead incurred by initializing the shared memory pool.

The speed-up is defined as

\[ s = \frac{T(1)}{T(p)} \]

where \( T(p) \) is the amount of time an implementation takes to perform the algorithm on \( p \) processors. Speed-ups for both implementations are given in Figure 3. The times reported here for one processor are only about 3% larger than the times for completely sequential code, so the one processor parallel times reported above are used to compute the speed-ups.

The speed-ups for the accelerated algorithm are better. This is probably due to the fact that linear combinations in the accelerated algorithm produce more of a reduction in the size of the operands, resulting in a smaller number of iterations of the main loop, thus requiring less sequential execution per section of parallel execution.

The efficiency tells how much of the computer’s processing resources are needed to handle the overhead of managing extra processes (Karp and Flatt, 1990). It is defined by

\[ e = \frac{T(1)}{pT(p)} = \frac{s}{p} \]

Efficiency information for the two implementations is given in Figure 4. It appears that for all but the largest input it would be cost effective to use no more than 2 to 4 processors. Note that the efficiency increases as the length of the operands increases.
Table 3. Initialization/shutdown times (in 0.01 second units).

<table>
<thead>
<tr>
<th>Number of processors</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>First</td>
<td>15.1</td>
<td>19.3</td>
<td>25.4</td>
<td>35.3</td>
<td>44.5</td>
</tr>
<tr>
<td>Subsequent</td>
<td>0.1</td>
<td>5.4</td>
<td>13.3</td>
<td>21.6</td>
<td>30.2</td>
</tr>
</tbody>
</table>

Figure 3. Speed-ups for the accelgcd (left) and bmodgcd (right) algorithms.

The serial fraction,

\[ f = \frac{1/s - 1/p}{1 - 1/p} = \frac{p/s - 1}{p - 1} = \frac{pT(p)/T(1) - 1}{p - 1}, \]

is used to measure the proportion of the computation that is inherently sequential (Karp and Flatt, 1990). The serial fractions computed from the test data are given in Figure 5. They should remain relatively constant for a given input length no matter how many processors are used, but decrease as the input length increases (pp. 541, 543, Karp and Flatt, 1990). If we ignore the behavior for “small” input, this is exactly what happens. (For small input, we realize that the processors quickly run out of useful work, and the increase in the serial fraction is probably due to the synchronization overhead that comes from contention for the queue.) Thus the serial fraction detects no performance problems. However, it is rather high (even for 4096 limbs), suggesting that parallelization will only be useful for very large input. (In the conclusion we suggest measures to take to reduce the serial fraction.) Also notice that the serial fraction for bmodgcd is significantly higher than for the accelerated algorithm.

6. Conclusion

Although the speed-ups displayed by the shared memory multiprocessor implementation of the accelerated integer GCD algorithm are modest, they are probably as good as can be obtained when using algorithms that perform a linear combination reduction, as do all of the most commonly used ones; among these the accelerated algorithm affords the greatest reduction in the size of the operands for every linear combination performed. (The preceding section shows that it performs better than the bmodgcd algorithm, for
example.) In addition, an analysis of the serial fraction measure turns up no performance problems, at least for large input, providing evidence that the implementation is sound.

An improvement on the performance described above may be achieved by increasing the value of the $t$ parameter, which increases both the amount of work required for each linear combination and the amount by which the operands are reduced. This should have the effect of decreasing the serial fraction. It has been pointed out also that it is possible to overlap the calculation of the values of $a$ and $b$ for the next linear combination with the parallel computation of the current linear combination. Finally, better performance may be achieved on a different parallel architecture.

On the other hand, an algorithm due to Schönhage (1971) has a sequential complexity of $O(n \log^2 n \log \log n)$, close to the accelerated algorithm’s parallel complexity of $O(n \log n)$ on $n$ processors (see Section 3). It seems that a parallel version of this algorithm is difficult to obtain, but it is certainly worthwhile to explore that avenue in addition to the suggestions presented in the previous paragraph.

References


