SIAM J. COMPUT.
Vol. 16, No. 1, February 1987

SUBLINEAR PARALLEL ALGORITHM FOR COMPUTING THE GREATEST COMMON DIVISOR OF TWO INTEGERS*

RAVINDRAN KANNAN†, GARY MILLER‡ AND LARRY RUDOLPH§

Abstract. The paper presents a sublinear time parallel algorithm for computing the greatest common divisor of two integers. Its running time on two n bit integers is $O(n \log\log n/\log n)$ using the weak concurrent read concurrent write model.

Key words. parallel algorithm, greatest common divisor

AMS(MOS) subject classification. 68Q99

Introduction. The advent of practical parallel processors has caused a re-examination of many existing algorithms with the hope of discovering a parallel implementation. One of the oldest and best known algorithms is Euclid's algorithm for computing the greatest common divisor (GCD). In this paper we present a parallel algorithm to compute the GCD of two integers.

Although there have been results in the parallel computation of the GCD of polynomials (Borodin, von zur Gathen and Hopcroft [2]), the integer case still appeared to be inherently serial; indeed Brent and Kung [3] achieve a running time of $O(n)$ with n processors arranged in a systolic array, where n is the number of bits required to represent the larger of the two input numbers. Although their method is an improvement on the best known serial integer GCD algorithm $O(n \log^2 n \log\log n)$ by Schönhage [10], it still requires n iterations; the parallelism only reduces the bit operations per iteration.

In this paper we present a sublinear time parallel algorithm to compute the integer GCD of two numbers on a weak CRCW-PRAM model of parallel computation allowing concurrent reads but only concurrent writes of the same value. (This model is often called the *common* model in the literature.) The time bound is $O(n \log\log n/\log n)$ assuming there are $n^2(\log n)^2$ processors working in parallel. This is computed assuming unit time for each elementary *bit* operation. Note that since we require concurrent reads and concurrent writes, our result cannot be used to construct a sublinear depth, polynomially sized boolean circuit for computing the GCD. This remains an interesting open problem.

This is the final journal version of Kannan, Miller and Rudolph [7] which contains essentially the entire contents of this paper. Since the appearance of the earlier version of this paper, Chor and Goldreich [4] have improved the running time getting rid of the $\log\log n$ term.

1. An overview of the algorithm. Throughout we assume that A and B are the two (nonnegative) integers whose GCD we want to find; A is the greater of the two and is an n bit integer. The basic idea of the algorithm is as follows: in the classical Euclidean algorithm we replace A with $A - qB$, where q is the quotient when A is divided by B. Here, instead, we consider in parallel $pA - q_pB$ for $p = 0$ to n, where q_p

* Received by the editors September 23, 1985; accepted for publication (in revised form) May 26, 1986.

† Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213. The work of this author was supported in part by National Science Foundation grant DCR-8416190.

‡ Department of Computer Science, University of Southern California, Los Angeles, California 90024.

§ Department of Computer Science, Hebrew University, Jerusalem, Israel.

is the quotient when pA is divided by B. All these integers are between 0 and B. By the pigeon-hole principle, there are at least two that agree on leading $\log n$ bits. Thus their difference is a nonnegative integer with at most $(n - \log n)$ bits. If we can replace A by their difference then we would reduce the problem size by $\log n$ bits during each two iterations thus requiring $O(n/\log n)$ iterations. There are two problems with this idea: (i) The GCD of A and B may be changed by the replacement and (ii) it is unclear how to execute the whole process in sublinear number of bit operations—a naive implementation would require $O(\log n)$ time for each iteration! We solve both these problems by a more complicated scheme.

The first problem is fixed (Lemma 1 stated below) as follows: we show that if $\text{GCD}(A, B) = g$ and $\text{GCD}(pA - qB, B) = h$, then g divides h and furthermore, h/g divides p. Since p is at most n, the only extra factors that are introduced into the GCD when we replace A, B by $pA - qB, B$ are made up of powers of primes between 0 and n. At the outset of the algorithm, we remove all common prime factors of magnitude at most n between A and B (in $O(\log n)$ time) and then we run the entire algorithm. Now clearly, at the completion of the algorithm, the extra factors introduced in the GCD by the replacement can be removed quickly. Prime factors of magnitude at most n are referred to as "small" factors throughout.

The second problem involves more complications. We want to find p, q_p with p between $-n$ and n such that $pA - q_pB$ is an integer with at most $(n - \log n)$ bits. We thus need $pA - q_pB$ to satisfy

$$0 \leqq pA - q_pB < \min(B, 2^{(n - \log n)}).$$

We make an important observation: we need only to know the $O(\log n)$ most significant bits of A, B to find a p and a q_p that "nearly" satisfy these conditions. Moreover, p and q_p will be no larger than n in absolute value. Thus it appears that we need to deal only with $O(\log n)$ bit numbers at each iteration. But if we update at the end of each iteration, the update will cost $O(\log n)$ parallel time which is too expensive. To avoid this, the algorithm will proceed in $O(n/(\log n)^2)$ phases. Each phase will consist of $O(\log n)$ iterations. During each phase we deal only with the leading $O((\log n)^2)$ bits of A and B. We record in a unimodular 2×2 matrix T the transformations we have performed during the phase on the vector (AB). At the end of the phase the entries of the matrix T are $O((\log n)^2)$ bit numbers. We now perform the transformation given by T on the actual vector (AB) considering all the bits of the integers and then proceed to the next phase. The updates at the end of every phase cost $O(\log n)$ time (since we deal with the whole n bit integers), but since there are only $O(n/(\log n)^2)$ phases this costs only $O(n/\log n)$ time on the whole. Inside each phase, we deal only with $O((\log n)^2)$ bit numbers and thus arithmetic operations take $O(\log \log n)$ time. Thus we spend $O(\log \log n)$ time to chop off each $O(\log n)$ bits, giving the overall time bound of $O(n \log \log n/\log n)$.

2. A sublinear-time, parallel integer GCD algorithm. In this section we present the theoretical foundations as well as some of the algorithmic details of the GCD algorithm. We let A and B denote the two integers whose GCD we wish to compute. Throughout this paper, we ensure that $A \geqq B > 0$. At the outset we set n equal to the number of bits required to represent A, the larger of the two input numbers, and keep it fixed.

This section begins with three lemmas. The first characterizes how the GCD of A and B changes when A is replaced by the value $(pA - qB)$. The second lemma gives the details of our application of the pigeon-hole principle to reduce the number of

bits during an iteration. A third lemma is concerned with the situation in which A is much larger than B.

LEMMA 1. *If* $g = \mathrm{GCD}(A, B)$; $h = \mathrm{GCD}(pA - qB, B)$ *then* g *divides* h *and* (h/g) *divides* p.

Proof. Since g divides A and B, it clearly divides h. Let

$$h' = \mathrm{GCD}\left(\frac{pA}{g} - \frac{qB}{g}, \frac{B}{g}\right),$$

then

$$h' = \mathrm{GCD}\left(\frac{pA}{g}, \frac{B}{g}\right).$$

But A/g and B/g have no common factors. So $h' = \mathrm{GCD}(p, B/g)$. Hence $h' | p$ and further $h' = h/g$. □

LEMMA 2. *If* a, b *and* n *are positive integers and* $a \leqq bn$ *then there exist integers* p *and* q *not both zero such that* $|p| \leqq nb/a$, $|q| \leqq 2n$ *and* $0 \leqq pa - qb \leqq a/n$.

Proof. Consider all the integers p satisfying $0 \leqq p \leqq nb/a$. For each such p, there obviously exists a unique q such that $0 \leqq pa - qb < b$ with $|q| \leqq n$. (q is of course the quotient when pa is divided by b.)

There are $\lfloor bn/a \rfloor + 1$ distinct such pairs (p, q). In addition, take the pair $p = 0$, $q = -1$ which is distinct from the rest and satisfies $0 \leqq pa - qb \leqq b$. This makes a total of $\lfloor bn/a \rfloor + 2$ pairs p, q with $0 \leqq pa - qb \leqq b$. Thus, by the pigeon-hole principle, there are two values $p_1 a - q_1 b$ and $p_2 a - q_2 b$ such that

$$0 \leqq (p_1 a - q_1 b) - (p_2 a - q_2 b) \leqq \frac{b}{\lfloor bn/a \rfloor + 1} \leqq \frac{b}{bn/a} = \frac{a}{n},$$

where $p_1 \neq p_2$. The combination

$$(p_1 - p_2)a - (q_1 - q_2)b$$

satisfies $|p_1 - p_2| \leqq nb/a$ and $|q_1 - q_2| \leqq 2n$ and is the desired result. □

Let $\#X$ indicate the number of significant bits of X, not counting leading zeros.

LEMMA 3. *Suppose* $\#B = l$ *and* $\#A = k + l$ *where* k *is positive. Then the quotient when* A *is divided by* B *can be found approximately (with an error of at most* ± 3*) using only the leading* $\min(2k, k + l)$ *bits of* A *and the leading* $\min(k, l)$ *bits of* B.

Proof. The case when $l \leqq k$ is trivial because in this case the lemma allows us to consider all the bits of A and B. So assume that $l > k$.

Let

$$B = 2^{l-k} B' + B''$$

where $0 \leqq B'' < 2^{l-k}$ and B' comprises the leading k bits of B.

Also let

$$A = 2^{l-k} A' + A''$$

where $0 \leqq A'' < 2^{l-k}$ and A' comprises the leading $2k$ bits of A.

Suppose q' is the quotient and r' the remainder when B' divides A'. Then

$$A' = q'B' + r', \qquad 0 \leqq r' \leqq B' - 1 < 2^k - 1.$$

Then

(*)
$$2^{l-k} A' = q'(2^{l-k} B') + 2^{l-k} r',$$
$$A' < 2^{2k}, \qquad B' \geqq 2^{k-1}.$$

Thus $q' < 2^{2k-k+1} = 2^{k+1}$. From (*),

$$A = 2^{l-k}A' + A'' = q'(2^{l-k}B' + B'') + 2^{l-k}r' + A'' - q'B'',$$

i.e., $A = q'B + r$ where $-2^{l+1} < r < 2^{l-k}(2^k - 1) + 2^{l-k} = 2^l$.

Since $B \geqq 2^{l-1}$,

$$-4B < r < 2B.$$

Thus $(A - q'B)$ is between $-4B$ and $+2B$. Since we keep all numbers positive, we replace r by $r+4B$ if it is negative. Thus at the end $0 \leqq r \leqq 2B$, i.e., $\#r \leqq \#B + 1$. □

Note that we can find q' and r in the proof of the lemma in parallel time $O(\log^2 k)$ by long division since we have to deal only with $O(k)$ bit integers. In the algorithm, we will apply such a long division whenever $\#A - \#B = k$ exceeds $(\log n)^2 + 1$. Then we replace A, B by B, r in the proof of the lemma. Thus we have chopped off at least $(\log n)^2$ bits off A in $O(\log \log^2 n)$ time which preserves our ratio of time to number of bits chopped off to at most $1/(O(\log n))$.

The remainder of this section presents the algorithmic details. The main program, "IntegerGCD," is first presented and followed by the required procedures. We will use two low-level subroutines: one for normal division and the other for selecting out of a set elements one that has a particular property. Since parallel algorithms for both these problems are well known in the literature, we postpone their explanations to the end.

Given two integers, A, B each at most n bits long, this algorithm computes their GCD *using at most* $(n \log n)^2$ *processors executing in parallel. We make use of the notation*:

- $\#X$ indicates the number of significant bits of X, not counting leading zeros. If X is a matrix, then it indicates the number of bits in the largest entry of the matrix.
- $X[h:l]$ indicates the $h-l+1$ bits of X from the low order lth bit to the high order hth bit. That is, $\lfloor X/2^l \rfloor \bmod 2^{h+1}$. The least significant bit is the zeroth bit.

PROGRAM IntegerGCD(A, B):

The procedure takes as input two **positive** integers A, B, ensures that $A \geqq B$ at the start and maintains these conditions until it has found the GCD of A and B. Once the numbers get small, we run the ordinary (serial) Euclidean algorithm on them which takes time bounded by a polynomial in the number of bits. We assume that such an algorithm is available and it treats the case when the two integers are nonnegative—one or both of them could be zero.

```
1:   If A < B then swap them.
2:   n ← the number of bits of A  /*n is never changed*/
3:   If #B ≦ 2(log n)^2 then begin
4:            Find C = A(mod B) by long division
                  *Now C and B are both 2(log n)^2 bit integers*
5:            Find by usual serial Euclidean algorithm the
                GCD of C and B and return this.
6:            Exit
7:            end
8:   Remove common small factors from A and B,
       and call the product of these factors SF
9:   Remove small factors from A and from B.
10:    repeat
11:      DoAPhase (A, B)
```

12: **until** $\#B < 2(\log n)^2$
13: Remove small factors from A and from B.
14: Find $C = A \pmod B$ by usual long division.
15: Run the serial Euclidean algorithm on C, B to get g'.
16: GCD $\leftarrow SF * g'$
END OF GCD

PROCEDURE DoAPhase (A, B)

Given A and B, each with at least $2(\log n)^2$ bits, with $A \geqq B > 0$ this procedure will return two new values for A and B, say A_{new} and B_{new}, with the properties that (i) the sum of the number of bits of A_{new} and B_{new} is at least $(\log n)^2$ less than the sum of those of A and B and that (ii) the GCD of A and B differs from the GCD of A_{new} and B_{new} by at most some small prime factors.

If the magnitudes of A and B differ by a lot then procedure LongDivide is called. Otherwise the procedure DoAnIteration is called at most $\log n$ times with the matrix T used to record the corresponding transformations to A and B.

1: $m \leftarrow \#A$; $s \leftarrow 2(\log n)^2$
2: **if** $\#A - \#B > (\log n)^2 + 1$
3: **then** LongDivide (A, B)
4: **else**
5: $a \leftarrow A[m : m - s + 1]$
6: $b \leftarrow B[m : m - s + 1]$
7: $T \leftarrow$ Identity Matrix
8: $endsize \leftarrow \#a + \#b - (\log n)^2$
9: **repeat**
10: DoAnIteration (a, b, T)
11: **until** $(\#a + \#b < endsize)$
12: $\begin{pmatrix} A \\ B \end{pmatrix} \leftarrow T \begin{pmatrix} A \\ B \end{pmatrix}$
13: Replace A, B by their absolute values.
14: **if** $A < B$ **then** swap their values.
/*Steps 12 through 14 are the only places A, B are changed*/
/*We obviously have $A \geqq B$ at the end of step 14*/
END PROCEDURE

PROCEDURE LongDivide (A, B)

Instead of performing a full division of A by B, we only divide the most significant $2k$ bits of A by the most significant k bits of B (see Lemma 3).

Assume $\#B + (\log n)^2 + 1 < \#A$.

0: $l \leftarrow \#B$
1: $k \leftarrow \#A - \#B$
2: $a \leftarrow$ most significant min $(2k, k + l)$ bits of A
3: $b \leftarrow$ most significant min (k, l) bits of B
4: $q \leftarrow \lfloor a/b \rfloor$ /*by long division*/
5: $C \leftarrow A - qB$
6: **if** $C < 0$ then $C \leftarrow C + 4B$
7: $A \leftarrow B$
8: $B \leftarrow C$
END PROCEDURE

PROCEDURE DoAnIteration (a, b, T)

Given a and b that are both $O((\log n)^2)$ bits and each at least $(\log n)^2$ bits, this procedure returns two new values for a and b whose sum of significant bits is at least $\log n$ less than it was upon entry. The transformations to a and b are recorded in the matrix T.

1: **if** $a/b \geqq n$
   /*NOTE: actual test is $a \geqq nb$*/
2: **then** Find a q such that $q = \lfloor a/b \rfloor$
3: $p \leftarrow 1$
4: **else** Find a pair (p, q), where $|p| \leqq nb/a$
   and $|q| \leqq 2n$, such that
   $0 \leqq pa - qb \leqq a/nm$ /*cf. Lemma 2*/
5: $T \leftarrow \begin{pmatrix} 0 & 1 \\ p & -q \end{pmatrix} T$
6: $\begin{pmatrix} a \\ b \end{pmatrix} \leftarrow \begin{pmatrix} 0 & 1 \\ p & -q \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix}$

END PROCEDURE

3. Analysis of the algorithm. The bulk of the algorithm is performed by DoAnIteration. In analyzing both the overall running time of the algorithm and its correctness we need to estimate how much smaller a and b are after each call of DoAnIteration as well as how much larger are the entries of T. Let $\# T$ equal the maximum number of bits in any entry of T. We next state some properties of the functioning of the procedure DoAnIteration.

LEMMA 4. *Let n be the number of bits of the larger of the two positive integers at the start of the algorithm. When each call of DoAnIteration is made, $a \geqq b \geqq 2^{(\log n)^2 - 1}$ and the call*

(i) *decreases the sum of their number of bits, i.e. $\# a + \# b$, by at least $\log n - O(1)$, and*

(ii) *increases the sum $\# T + \# a + \# b$ by at most $O(1)$.*

Proof. Note that if C and D are integers then $\#(CD) \leqq \# C + \# D$ and if they are 2×2 matrices then $\#(CD) \leqq \# C + \# D + 1$.

We divide the proof into two cases depending on whether line (2) or line (4) of DoAnIteration is implemented in this call. Suppose it is line (2) which is implemented, then the decrease in $\# a + \# b$ will be

$$(\# a + \# b) - (\# b + \#(a - qb)) \geqq \# a - \# b \geqq \log n - O(1)$$

which gives (i). Since T will be multiplied by a matrix whose largest entry in absolute value is q we can conclude that $\# T_{\text{new}} \leqq \# T + \# q + O(1)$. Thus, $\# T$ increases by at most $\# q + O(1)$ while $\# a + \# b$ decreases by at least $\# q + O(1)$. This gives (ii) in the case when line (2) is implemented.

Suppose, instead, that line (4) is implemented. By Lemma 2, line (4) will find a $pa - qb$ that satisfies (i) and where p and q are no more than $\log n$ bits. So, looking at step 5 of DoAnIteration, we see that $\# T$ will be increased by at most $\log n + 1$ bits and so (ii) is satisfied. □

We next consider procedure DoAPhase. Due to the if statement in line 2 of DoAPhase, when line (12) in procedure DoAPhase is implemented, we know that

1) it has called DoAnIteration at most $\log n + O(1)$ times,

2) the entries in T are small, i.e.

$$\# T \leqq (\log n)^2,$$

as argued in the proof of Lemma 4,

3) and the sum of the bits in a and b has decreased by at least $(\log n)^2$ bits.

LEMMA 5. *Each execution of the procedure DoAPhase with parameters A and B with* $A \geqq B > 0$ *and* $\#B \geqq 2(\log n)^2$ *will decrease* $\#A + \#B$ *by at least* $(\log n)^2$.

Proof. If $\#A - \#B > (\log n)^2 + 1$ then LongDivide will be called and we get the desired decrease in bits. We may assume that $A \geqq B$ and $\#A - \#B \leqq (\log n)^2$. Note that the latter condition is equivalent to $(\log n)^2 \leqq \#b$ (see line (6) of DoAPhase). Let $m = \#A$ and $s = 2(\log n)^2$. Let a' and b' be the low order $m - s$ bits of A and B that remain after removing a and b, i.e. $a' = A[m-s:0]$ and $b' = B[m-s:0]$. Thus, $A = a2^{m-s} + a'$ and $B = b2^{m-s} + b'$. Let T be the matrix on line (12) and

$$\begin{pmatrix} a_{\text{new}} \\ b_{\text{new}} \end{pmatrix} = T \begin{pmatrix} a \\ b \end{pmatrix}.$$

Thus,

$$T\begin{pmatrix} A \\ B \end{pmatrix} = (2^{m-s})\begin{pmatrix} a_{\text{new}} \\ b_{\text{new}} \end{pmatrix} + T\begin{pmatrix} a' \\ b' \end{pmatrix}.$$

Since $\#T \leqq (\log n)^2$ and $\#a' = \#b' = m - s$, we have that

$$\#\left(T\begin{pmatrix} a' \\ b' \end{pmatrix}\right) \leqq m - s + (\log n)^2 + 1.$$

Thus, a' and b' only affect the lower half, $(\log n)^2 + 1$ bits, of the most significant $2(\log n)^2$ bits of A and B. In order to finish the proof we need to show that

$$(**) \qquad (\#a + \#b) - (m_a + m_b) \geqq (\log n)^2$$

where

$$m_a = \max(\#a_{\text{new}}, (\log n)^2),$$

$$m_b = \max(\#b_{\text{new}}, (\log n)^2).$$

We must use the maximum in the equation (**) since decreasing either $\#a_{\text{new}}$ or $\#b_{\text{new}}$ below $(\log n)^2$ will not in general decrease the number of bits in A_{new} or B_{new} because of the carry from the low order terms. Initially $\#a = 2(\log n)^2$ and $\#b$ may be smaller. But after the first call of DoAnIteration a will be replaced by an integer with $\leqq \#b$ bits. Thus $\#a_{\text{new}}, \#b_{\text{new}} \leqq \#b$. There are three cases to consider:

Case $a_{\text{new}} \leqq (\log n)^2$. Since $\#a = 2(\log n)^2$, we get:

$$(\#a + \#b) - (m_a + m_b) \geqq (\log n)^2 + (\#b - m_b)$$

using the fact that $\#b \geqq m_b$,

$$\geqq (\log n)^2.$$

Case $b_{\text{new}} \leqq (\log n)^2$. Similar to previous case.

Case $\#a_{\text{new}}$ *and* $\#b_{\text{new}} \geqq (\log n)^2$. Thus the maximums in equation (**) can be replaced with $\#a_{\text{new}}$ and $\#b_{\text{new}}$ reducing equation (**) to fact 3 (stated before Lemma 5).

4. Analysis of running time. We now show that the algorithm requires parallel time $O(n \log\log n / \log n)$. It is clear that the repeat loop of the main program is executed at most $n/(\log n)^2$ times, since each iteration removes at least $(\log n)^2$ bits from the sum of the bits in A and B (Lemma 5). We must show that each execution of the repeat loop takes no more than $O(\log n \log\log n)$ parallel time. This is done

by starting with the analysis of the innermost operations of DoAnIteration and then working our way out. The proofs of the following lemmas also indicate where the parallelism is needed.

We first review some of the known results for integer arithmetic operations performed in parallel. Integer addition of two n bit numbers can be performed in $O(\log n)$ time using n processors (Ladner and Fischer [8]). Integer multiplication also requires only $O(\log n)$ parallel time with $n \log\log n$ processors (Schönhage and Strassen [11]). Integer division too can be done in parallel time $O(\log n)$ with a polynomial number of processors, but the result is much harder (Beame, Cook and Hoover [1]). In fact, we do not need such fancy parellel division algorithms, we just require an $O((\log n)^2)$ algorithm using n^2 processors. This was earlier obtained by Cook [5].

At many points in the algorithm, we are required to choose a value or a pair of values from a set that satisfies a certain condition. It is easy to see that with enough processors, all the values in the set can be tested in parallel; however, in our weak model of parallel processing, simultaneous writes are allowed provided that all the processors write the same value. Thus if the set contains $O(n)$ items and many of the elements may satisfy some desired condition, $O(\log n)$ time may be required to choose any one such element. Fortunately, if there are n^2 processors then these selection processes can be done in the time it takes for a comparison as follows.

Briefly, the maximum of n can be chosen with n^2 processors by recording the results of all pairwise comparisons in a matrix: $M[i,j]$ is set to *true* if $a_i > a_j$ or if ($a_i = a_j$ and $i > j$). The row containing all *trues* corresponds to the maximum element. Since there will be only one such row, this row can be identified in constant time. In our GCD algorithm, we wish to select one of a set of elements in constant time. If a value satisfies our requirement, we tag it with a 1 otherwise with a 0. An element can then be chosen in constant time since the comparison will take only constant time (if $a_i = a_j$ then the processor "knows" if $i > j$). (Frieze and Rudolph [6] give a parallel algorithm that implements such a selection in constant time, with a high probability, with only n processors.)

LEMMA 6. *Each call to Procedure DoAnIteration can be executed in* $O(\log\log n)$ *parallel time using* $n^2(\log n)^2$ *processors.*

Proof. We assume that the two integers, a, b, contain at most $2(\log n)^2$ bits. Line (1) is accomplished by a multiplication of two numbers of no more than $2(\log n)^2$ bits, thus requiring no more than $O(\log\log n)$ time. Lines (2) or (4) can also be done in this time bound by assigning $(\log n)^2$ processors to each of the at most n^2 equations "$pa - qb$" in the case when line (4) is executed and to each of the n equations "$a - bq$" in the other case. The multiplication requires at most $O((\log n)^2)$ processors and can be computed in logarithmic time in the number of bits, i.e. $O(\log\log n)$ time. Only one (p, q) pair is needed; however, there may be many such pairs. For each p there will be only one q that satisfies the condition that $pa - qb < b$ and thus there are at most only $O(n)$ pairs to choose from. The selection can be done in constant time by the above outline with n^2 processors.

Lines (5) and (6) can be computed in $O(\log\log n)$ time since the entries in the matrices are no more than $O((\log n)^2)$ bits. □

LEMMA 7. *Procedure LongDivide can be executed in* $O(\log n)$ *time using no more than* $O(n)$ *processors.*

Proof. We show that each line can be executed within the required time bounds:

Line (1) can easily be computed using a binary fan-in tree and n processors in $O(\log n)$ time.

Lines (2) and (3) take constant time to identify the appropriate bits.

Line (4), the division of a $2k$ bit number by a k bit number can be done within $O((\log k)^2)$ time with k processors.

Line (5) is simply a multiplication of a k bit number by an n bit number and this takes no more than $O(\log n)$ parallel time with n processors. The subtraction is also done within this time bound. □

LEMMA 8. *An execution of the procedure DoAPhase requires* $O(\log n \log\log n)$ *parallel time, using* $n^2(\log n)^2$ *processors.*

Proof. This procedure may invoke the procedure DoAnIteration or the procedure LongDivide and thus depends on the previous two lemmas. In a manner similar to the explanation in the previous proofs, line (2) requires only $O(\log n)$ time using n processors.

By Lemma 3 it is clear that procedure DoAnIteration is not called more than $\log n$ times.

Executing line (10) is also within the time bound since A and B have at most n bits each and the entries of T each are at most $(\log n)^2$ and thus the multiplication takes at most $O(\log n)$ parallel time using n processors. □

THEOREM 1. *The* GCD *of two integers, each representable in at most n bits requires parallel time* $O(n \log\log n/\log n)$ *using* $n^2(\log n)^2$ *processors.*

Proof. The time bound follows from the previous lemmas provided we can remove the common small factors in $O(n/\log n)$ parallel time. All small primes, i.e. less than n, can be identified quickly as follows: For each $p \leqq n$ assign $n \log n$ processors to check that no number up to $\sqrt{p}$ divides p. After all the small primes have been identified, all their powers, p^i for $i \leqq n$, are computed in parallel. Finally, for each power for each prime, p_j^i, we check to see if p_j^i is a factor. For each j the maximum i is then chosen as described above.

Since we can identify all small prime factors of an n bit number in $O((\log n)^2)$ time, the theorem follows. □

5. Conclusion. To summarize, the two salient features of the algorithm are: the observation based on the pigeon-hole principle that we can easily find an integer combination of the two integers A and B which has fewer bits than n and the idea of working in phases so as to perform arithmetics on n bit integers only once every phase, the more frequent operations being performed on $O((\log n)^2)$ bit integers. It appears that yet another approach is needed if the GCD is to be computed in poly-log parallel time.

REFERENCES

[1] P. W. BLAME, S. A. COOK AND H. J. HOOVER, *Log depth for division and related problems*, 25th Annual Symposium on the Foundations of Computer Science, October 1984, pp. 1-6.

[2] A. BORODIN, J. VON ZUR GATHEN AND J. HOPCROFT, *Fast parallel matrix and* GCD *computations*, 23rd Annual Symposium on Foundations of Computer Science, November 1982, pp. 65-71.

[3] R. P. BRENT AND H. T. KUNG, *Systolic* VLSI *arrays for linear time* GCD *computation*, VLSI 83, International Federation of Information Processing, 1983.

[4] B. CHOR AND O. GOLDREICH, *An Improved Parallel Algorithm for Integer* GCD, MIT Laboratory for Computer Science, Cambridge, MA, April 1985, to appear.

[5] S. A. COOK, *The classification of problems which have fast parallel algorithms*, Lecture Notes in Computer Science, Vol. 158, Springer-Verlag, New York-Berlin-Heidelberg, 1983.

[6] A. FRIEZE AND L. RUDOLPH, *A parallel algorithm for all pairs shortest paths in a random graph*, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1984.

[7] R. KANNAN, G. MILLER AND L. RUDOLPH, *Sublinear parallel algorithm for computing the greatest common divisor of two integers*, 25th Annual Symposium on Foundation of Computer Science, October 1984, pp. 7-11. © IEEE.

[8] R. E. LADNER AND M. J. FISCHER, *Parallel prefix computation*, J. Assoc. Comput. Mach., 27 (1980), pp. 831-838.

[9] J. REIF, *Logarithmic depth circuits for algebraic functions*, 24th Annual Symposium on Foundation of Computer Science, November 1983, pp. 138-145.

[10] A. SCHÖNHAGE, *Schnelle Berechnung von Kettenbruchententwicklungen*, Acta Inform., 1 (1971), pp. 139-144.

[11] A. SCHÖNHAGE AND V. STRASSEN, *Schnelle Multiplikation grosser Zahlen*, Computing, 7 (1971), pp. 281-292.