Algorithmica (1990) 5: 1–10

Algorithmica
© 1990 Springer-Verlag New York Inc.

An Improved Parallel Algorithm for Integer GCD

Benny Chor[1,2] and Oded Goldreich[1,3]

Abstract. We present a simple parallel algorithm for computing the greatest common divisor (gcd) of two n-bit integers in the Common version of the CRCW model of computation. The run-time of the algorithm in terms of bit operations is $O(n/\log n)$, using $n^{1+\varepsilon}$ processors, where ε is any positive constant. This improves on the algorithm of Kannan, Miller, and Rudolph, the only sublinear algorithm known previously, both in run time and in number of processors; they require $O(n \log\log n/\log n)$, $n^2 \log^2 n$, respectively, in the same CRCW model.

We give an alternative implementation of our algorithm in the CREW model. Its run-time is $O(n \log\log n/\log n)$, using $n^{1+\varepsilon}$ processors. Both implementations can be modified to yield the extended gcd, within the same complexity bounds.

Key Words. Greatest common divisor, Parallel algorithms.

1. Introduction. The problem of computing the greatest common divisor (gcd) of two integers efficiently in parallel is one of the outstanding open problems in the theory of parallel computation. While a serial solution to the problem has been known for thousands of years (Euclid's algorithm), no significantly more efficient parallel algorithm has been found.

Euclid's algorithm is based on magnitude tests ("is $a > b$?") and on gcd preserving transformations ($\gcd(a, b) = \gcd(a, b-a)$). These are performed in a $\Theta(n)$ long sequence of iterations (where n denotes the number of bits in the two inputs). The next iteration cannot start before the previous iteration has terminated. Thus it is not obvious whether any attempt to parallelize Euclid's algorithm could achieve $o(n)$ parallel time.

Kannan *et al.* [5] presented the first sublinear-time parallel algorithm for integer gcd. They came up with a method to "pack" $O(\log n)$ iterations[4] of Euclid's algorithm into one parallel phase. The transformations applied in this packing are "almost" gcd preserving. After $O(n/\log n)$ phases, the process terminates with an answer which is "close enough" to the correct gcd to actually find it. Each phase is implemented in $O(\log\log n)$ parallel time, and thus the overall

[1] Laboratory for Computer Science, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139, USA.

[2] Supported in part by an IBM Graduate Fellowship and a Bantrell Postdoctoral Fellowship. Present address: Department of Computer Science, Technion, Haifa 32000, Israel.

[3] Supported in part by a Weizmann Postdoctoral Fellowship. Present address: Department of Computer Science, Technion, Haifa 32000, Israel.

[4] All logarithms are to base 2.

Received January, 1986; revised February, 1987, and December, 1987. Communicated by F. Thomson Leighton.

time is $O(n \log\log n/\log n)$. This is achieved using $n^2 \log^2 n$ processors in the concurrent read concurrent write (CRCW) model of bit processors, assuming unit cost for each elementary bit operation. The specific variant of resolving write conflicts in the Kannan-Miller-Rudolph algorithm is the Common CRCW model. In this variant, processors who concurrently write to the same memory location are required to write the same value. In general, this conflict resolution variant can be simulated without any loss of efficiency by the more powerful Arbitrary or Priority variants [4], [7], [10]. It should be noted that when the values written are bit values, the Arbitrary variant can be simulated on the Common variant while increasing the run-time by at most a factor of 2 (this is done by alternating writes of "1" and of "0").

The algorithm we present here exibits a tradeoff between parallel time and number of processors. For any k in the range $1 \le k \le n/\log n$, it achieves $O(n/k)$ parallel time, using $n2^{2k+1} + k^3 2^{3k}$ processors (in the same Common variant of the CRCW model). For the value $k = \varepsilon \log n/2$, the parallel time is $O(n/\log n)$, using $n^{1+\varepsilon}$ processors. In addition to the improved complexity, the algorithm is much simpler than the Kannan-Miller-Rudolph algorithm. While they modify Euclid's algorithm, we have as our starting point a different sequential algorithm—the Brent-Kung *plus-minus* gcd algorithm [1]. In contrast to Euclid's algorithm, which operates on the most-significant bits first, the Brent-Kung algorithm operates on the least-significant bits first. Unlike the classical gcd algorithm, the tests used in the Brent-Kung algorithm are parity tests ("is b even?"), and the gcd preserving transformations are different ($\gcd(a, b) = \gcd(a, (a \pm b)/2)$ for a, b odd, and $\gcd(a, b) = \gcd(a, b/2)$ for a odd, b even). While the number of iterations is still $O(n)$, the advantage of this approach is that the next iteration can start as soon as the least-significant bits from the previous iteration are known. In particular, this gives a linear-time implementation on a systolic array [1].

The basic idea in our algorithm is to pack k consecutive transformations of the Brent-Kung algorithm into one parallel phase. The number of phases is thus $O(n/k)$. It should be noted that a naive implementation of a phase might cost k time units per phase, thus rendering the whole procedure useless. Fortunately, more efficient implementations can be found. By preparing multiplication tables for k-bit integers in advance, and using them to multiply n-bit integers by k-bit integers, a phase can be implemented in $O(1)$ time.

We give two alternative implementations of our gcd algorithm in the CRCW- and CREW-PRAM models. The complexity of these implementations (including the preprocessing stage) is summarized in Table 1.

Table 1

Model	Time	Number of Processors	Time $(k = \varepsilon \log n/2)$	Number of processors $(k = \varepsilon \log n/2)$
CRCW	$O(n/k + \log k)$	$n2^{2k+1} + k^3 2^{3k+6}$	$O(n/\log n)$	$n^{1+\varepsilon}$
CREW	$O(n \log k/k + \log^2 k)$	$n2^{2k+1} + k^3 2^{3k+6}$	$O(n \log\log n/\log n)$	$n^{1+\varepsilon}$

We remark that an alternative way of getting an improved parallel gcd algorithm is to simulate the Brent–Kung systolic array algorithm by a PRAM. (We would like to stress that this alternative improvement was not previously known.) To do this, we first apply a general simulation of systolic arrays with bit processors by systolic arrays with k-bit processors (such simulation is implicit in [3] and [9]). Starting with $O(n)$ systolic arrays with bit processors, we get $O(n/k)$ systolic arrays with k-bit processors. Next, we simulate each k-bit systolic processor by $k \cdot 2^{3k}$ processors which implement a table look-up for each step of the k-bit processor (recall that each bit operation is unit cost on such processors, and see Section 3 for the number of processors required to perform a table look-up). The look-up table can be constructed in $O(k)$ time using $k \cdot 2^{3k}$ processors by a straightforward simulation. This way we can achieve $O(n/k+k)$ run-time using $O(n \cdot 2^{3k})$ processors in CRCW. We prefer not to use this approach since it is not as efficient as our first suggestion, and does not use any insights gained from the special structure of the Brent–Kung algorithm.

The rest of this paper is organized as follows. In Section 2 we describe the Brent–Kung algorithm and its run-time analysis. In Section 3 we give an overview of our parallel algorithm. Section 4 describes the preprocessing stage. Section 5 contains the modifications needed in the extended gcd. In Section 6 we present the CREW implementation, and Section 7 contains some concluding remarks.

2. The Brent–Kung GCD Algorithm.

Euclid's algorithm finds the gcd of two n-bit integers in $O(n)$ iterations. In every iteration, both arguments are reduced, using gcd preserving operations. To find the gcd, tests of the form "is $b < c$" are required. In contrast, the *plus–minus* algorithm of Brent and Kung tests only the two least-significant bits of each argument. The transformations applied are addition/subtraction and division by 2. The code for the Brent–Kung gcd algorithm is as follows:

```
procedure PLUS MINUS gcd:
   INPUT: $A$, $B$
$a, b \leftarrow A, B$
{ $a$ is odd, $b \neq 0$, $|a|, |b| \leq 2^n$ }
$\alpha \leftarrow n$
$\beta \leftarrow n$
$\delta \leftarrow 0$
      repeat
         while parity$(b) = 0$ do
            { apply transformation I—divide by 2 }
            $b \leftarrow b/2$
            $\beta \leftarrow \beta - 1$
            $\delta \leftarrow \delta + 1$
         od
         { $a$ and $b$ are odd, $|b| \leq 2^{\beta}$, $\delta = \alpha - \beta$ }
         if $\delta > 0$ then
```

```
            { apply transformation II—swap }
13.         swap($a, b$), swap($\alpha, \beta$), $\delta \leftarrow -\delta$
          { $\delta = \alpha - \beta$, $\alpha \leq \beta$, $|a| \leq 2^\alpha$, $|b| \leq 2^\beta$; a and b are odd }
14.       if parity($(a+b)/2$) = 0
15.         then { apply transformation III—add and divide by 2 }
            $b \leftarrow (a+b)/2$
16.         else { apply transformation IV—subtract and divide by 2 }
            $b \leftarrow (a-b)/2$
          { b is even, $|b| \leq 2^\beta$ }
17.     until $b = 0$
18. return gcd $= a$
```

This algorithm returns the gcd of A and B, provided that A is odd. Its correctness follows from the fact that if b is even and a is odd, then $\gcd(a, b) = \gcd(a, b/2)$, while if both b and a are odd then $\gcd(a, b) = \gcd(a, (a+b)/2) = \gcd(a, (a-b)/2)$. The assertions in braces simplify the run-time analysis: the variable α (resp. β) is an upper bound on the number of bits in $|a|$ (resp. $|b|$), and δ is their difference. In every repeat loop (lines 7-16) except possibly the first one, the sum $\alpha + \beta$ decreases by at least 1. Therefore the number of loops is at most $2n = O(n)$. In fact, α and β are not necessary for the actual execution of the algorithm (they only simplify the analysis). Instead, it suffices to keep δ around, and remember that $\delta = \alpha - \beta$ holds throughout the execution.

3. The Parallel GCD Algorithm.

The main observation we use is that at any point during the execution of the Brent-Kung algorithm, the next k transformations applied to a, b, and δ are totally determined by δ and the $k+1$ least-significant bits of a and b. In fact, not the entire δ is needed, but only the signs of δ during the next k transformations. For these, in turn, it suffices to keep the information of the sign of δ, a bit specifying whether $\delta \in [-k, k]$, and, in case δ is in this interval, its value. Each such k-transformation can be described by

$$(a, b) \leftarrow \frac{1}{2^k}(a, b)\begin{pmatrix} c & d \\ e & f \end{pmatrix},$$

$$\delta \leftarrow \sigma \cdot \delta + g,$$

where c, d, e, f are all integers in the range $[-2^k, 2^k]$, $-k \leq g \leq k$, and $\sigma \in \{-1, 1\}$.

We can precompute a 2^{k+1}-by-2^{k+1}-by-$4k+4$ transformation table T which specifies, for every possible configuration of the $k+1$ least-significant bits in a and b and the value of δ, the four entries of the matrix together with the δ modification. Let $\mathbf{a}$ denote the $k+1$ least-significant bits of a, and $\mathbf{b}$ denote the $k+1$ least-significant bits of b. The string $\boldsymbol{\delta}$ is the concatenation of the sign bit of δ, a bit specifying whether $\delta \in [-k, k]$, and, in case δ is in this interval, its absolute value. (Overall, there are $4k+4$ possibilities for $\boldsymbol{\delta}$.) The entry $T[\mathbf{a}, \mathbf{b}, \boldsymbol{\delta}]$ contains the k-transformation which should be applied to a, b, and δ.

After the preprocessing stage is completed, the algorithm proceeds in phases, each phase performing the task of k consecutive transformations in the Brent-Kung algorithm. Given a, b, and δ, the value of $\boldsymbol{\delta}$ is computed, and the entry $T[\mathbf{a}, \mathbf{b}, \boldsymbol{\delta}]$ corresponding to them is found. The pair (a, b) is multiplied by the corresponding matrix, and the modification to δ is performed. From the analysis in Section 2, it follows that the overall number of phases is at most $2n/k$.

The transformation table used by the algorithm is a look-up table with $(4k+4) \cdot 2^{2k+2}$ entries, each storing no more than $5k$ bits (for $k \geq 5$). We use implicitly the fact that an L-bit entry from a table with S entries can be fetched in four parallel time units by $S \cdot \max(L, \log S)$ bit processors in the CRCW model. (For each of the S entries, $\log S$ processors are needed to identify the address of the desired entry, and L processors are needed to fetch the L bits.) Thus accessing an entry in the transformation table requires fewer than $5k \cdot (4k+4)2^{2k+2} < k^2 \cdot 2^{2k+7}$ bit processors.

The code for the parallel gcd algorithm is as follows:

```
procedure PARALLEL gcd (with parameter $k$):
   INPUT: $A, B$
Preprocessing
compute multiplication table for all pairs of $k$ bit integers
compute transformation table $T$
Execution
$a, b \leftarrow A, B$
    repeat
        $\mathbf{a} \leftarrow k+1$ least-significant bits of $a$
        $\mathbf{b} \leftarrow k+1$ least-significant bits of $b$
        $\boldsymbol{\delta} \leftarrow$ sign-bit, interval-bit, and $\log k$ least-significant bits of $\delta$
        $\mathcal{T} \leftarrow T[\mathbf{a}, \mathbf{b}, \boldsymbol{\delta}]$
        apply transformation $\mathcal{T}$ to $a, b, \delta$
    until $b = 0$
return gcd $= a$
```

We now specify how a transformation $\mathcal{T}$ is applied to the triple a, b, δ. The new value assigned to δ is obtained by adding a $\log k$-bit integer to either δ or $-\delta$. Of more concern is the (a, b) transformation. Since division by 2^k just means a shift by k, it causes no problems. Thus we should just worry about the multiplications (of k-bit integers by n-bit integers), and the addition/subtraction of two n-bit integers. A naive implementation of these operations might cost k time units, thus rendering the whole procedure useless. As we show below, both operations can be implemented in $O(1)$ parallel time, so the overall run-time for $2n/k$ phases will be $O(n/k)$.

To multiply an n-bit integer a by a k-bit integer c, a is partitioned into n/k disjoint blocks of k-bits each. We then represent a as the sum of two n-bit integers, a_1 and a_2. The "odd" blocks of a are the only nonzero blocks of a_1, while the "even" blocks of a are the only nonzero blocks of a_2 (see Figure 1). When multiplying a_i $(i = 1, 2)$ by c (a k-bit integer), each block expands to a

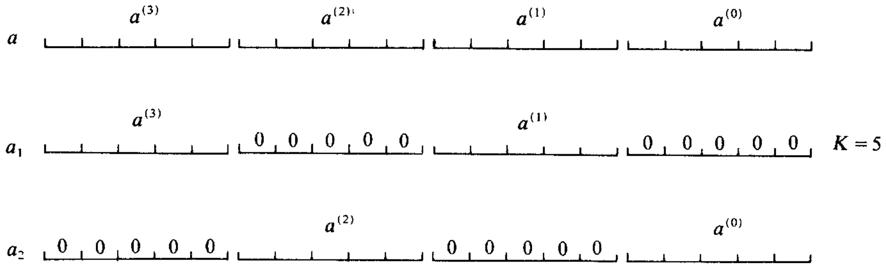

Fig. 1. Partition of a to odd and even blocks.

block of length $2k$. These $2k$-long blocks do not overlap. By precomputing all possible products of two k-bit integers, both $a_1 \cdot c$ and $a_2 \cdot c$ are computed in $O(1)$ parallel time by a table look-up. The number of processors required is $2^{2k} \cdot 2k$ per block. There are n/k blocks altogether, and thus $2n \cdot 2^{2k}$ processors are needed. (For $k < \sqrt{n}$, this is the bottleneck of our algorithm in terms of the number of processors, as we will shortly see.)

Finally, four $(n+k)$-bit integers should be added/subtracted (e.g., $a_1c+a_2c+b_1e+b_2e$). By combining a result of Chandra *et al.* [2] with the inequality $k \leq n$, such addition can be done in $O(1)$ parallel time using $n \cdot 2^k$ processors, assuming that k grows at least as the inverse of some primitive recursive function f satisfying $\lim_{n\to\infty} f(n) = \infty$.

The code for the application of a transformation $\mathcal{T}$ is as follows:

```
procedure APPLY TRANSFORMATION:
   INPUT: $a$, $b$, $\mathcal{T}$
   $c, d, e, f, g, \sigma \leftarrow$ corresponding elements of $\mathcal{T}$
      $\delta \leftarrow \sigma\delta + g$
      $a_1 \leftarrow \text{odd}(a)$, $a_2 \leftarrow \text{even}(a)$, $b_1 \leftarrow \text{odd}(b)$, $b_2 \leftarrow \text{even}(b)$
      using the multiplication table, compute $a_1c$, $a_2c$, $a_1d$, $a_2d$, $b_1e$, $b_2e$, $b_1f$, $b_2f$
      $(a, b) \leftarrow (a_1c + a_2c + b_1e + b_2e, a_1d + a_2d + b_1f + b_2f)$
return $a$, $b$, $\delta$
```

The overall time of a phase is essentially that of applying the transformation, which is $O(1)$. Hence running the whole algorithm (after the preprocessing stage) requires $O(n/k)$ parallel time, using $\max(2n2^{2k}, k^2 2^{2k+7})$ bit processors. To have *poly*(n) processors, k can be at most $O(\log n)$. For $k = \Theta(\log n)$, the resulting parallel time is $O(n/\log n)$.

4. The Preprocessing Stage. The preprocessing stage prepares the tables necessary for the algorithm. It does not depend on the actual inputs A, B but only on their size. Except for large values of k ($k > n/\log n$), the complexity of the preprocessing is dominated by the complexity of the main procedure.

The preprocessing stage has two parts. The first part is to compute in parallel all pairwise products of $(k/2)$-bit integers. The second part is to compute in parallel the tables corresponding to k consecutive iterations of the Brent–Kung gcd algorithm.

It is well known that the product to two m-bit integers can be computed in $O(\log m)$ parallel time using m^2 processors, even in the CREW model [8], [11]. This can be done by representing the product as the sum of m integers, each with at most $2m$ bits. In one time step, the sum of three integers is transformed into the sum of two integers—one representing the bitwise sum (with no carry) and the other representing the carries generated locally. By iterating such reductions $O(\log m)$ times, the original product is reduced into the sum of two $2m$-bit integers. Finally, these two integers are added in $\log m$ steps. Overall, with $2^{2m}m^2$ processors, the products of all pairs are computed in $O(\log m)$ parallel time. For $m = k/2$, we construct multiplication tables for m-bit integers using $k^2 2^k$ processors.

To compute the k-transformation table, we use recursion. Suppose that the l-transformation table was already computed. Namely, for each of the possible values for δ and every pair of $(l+1)$-bit integers (least-significant bits), the corresponding matrix $\begin{pmatrix} c & d \\ e & f \end{pmatrix}$ and the values g, σ $(-l \le g \le l, \sigma \in \{-1, 1\})$ which determine the δ transformation, are known. We now compute in parallel all entries in the $2l$-transformation table. Given δ and a pair of $(2l+1)$-bit integers a, b, we first find the l-transformation corresponding to δ and the $l+1$ least-significant bits of a, b . We then apply this l-transformation to δ, a, and b, getting the transformed values δ', a', and b'. These a', b' are $(l+1)$-bit integer themselves. Let $\begin{pmatrix} c' & d' \\ e' & f' \end{pmatrix}$ and g', σ' be the l-transformation corresponding to a', b', and δ'. The matrix product

$$\begin{pmatrix} c'' & d'' \\ e'' & f'' \end{pmatrix} = \begin{pmatrix} c & d \\ e & f \end{pmatrix} \cdot \begin{pmatrix} c' & d' \\ e' & f' \end{pmatrix},$$

whose entries are integers in the range $[-2^{2l}, 2^{2l}]$, is the matrix of the $2l$-transformation corresponding to a, b, and δ. Similarly, $\sigma'' = \sigma'\sigma$ and $g'' = \sigma' g + g'$ determine the new value of δ'' $(\delta'' = \sigma'\sigma\delta + \sigma' g + g')$. Both the matrix product and σ'', g'' can be computed in $O(1)$ parallel time (using the multiplication table for the matrix product). We notice that, throughout the recursion, it suffices to index the table by the string $\boldsymbol{\delta}$, rather than by the complete value of δ. (As before, $\boldsymbol{\delta}$ is the concatenation of the sign bit of δ, a bit specifying whether $\delta \in [-k, k]$, and, in case δ is in this interval, its absolute value.) The size of the table at the l-level (the l-transformation) is thus $(4k+4) \cdot 2^{2l}$. To compute each entry in the $2l$-transformation table, $5l \cdot (4k+4)2^{2l}$ processors are required in order to fetch $5l$-bit values from the l-transformation tables $(l > \log k)$, and $k \cdot 2^k$ processors are required in order to fetch values from the multiplication tables. Thus, to compute all entries in the $2l$-transformation table, we need $(4k+4)2^{4l} \cdot \max(5l \cdot (4k+4)2^{2l}, k2^k)$ processors. This term is maximized at the top level, where $k=2l$. This results in $40(k+1)^3 2^{3k} < k^3 2^{3k+6}$ CRCW bit-processors that are needed to compute the k-transformation table in $O(\log k)$ steps.

5. Extended GCD.

The algorithm described so far outputs the gcd G of its two inputs A, B. In fact, it is possible to recover from it two integers s and t such that $G=(1/2^n)(sA+tB)$. However, certain applications (such as computing inverses modulo prime integers) requires a representation of the form $G=xA+yB$ where x and y are integers (and not just rationals). Such representation for the gcd is known as the extended gcd [6, p. 325]. We use the extended version of the binary gcd which Knuth attributes to Penk and Pratt [6, problem 35, pp. 339 and 599]. Essentially, the goal is to maintain representations of the current a and b as integer combinations of the original A and B:

$$a=lA+rB,$$

$$b=mA+sB.$$

It is easy to maintain such representation when the current a and b are either added, subtracted, or switched. The problem lies in division by 2, $b \leftarrow b/2$, and is solved by a case analysis. The easy case is when m and s are both even. Otherwise it turns out that both $m+B$ and $s-A$ are even (recall that A is odd). Thus we get

$$\frac{b}{2}=\frac{m+B}{2}\cdot A+\frac{s-A}{2}\cdot B.$$

We observe that the next k transformations depend only on the $k+1$ least-significant bits of a, b, A, B, l, r, m, and s and on δ. Using the above ideas, we get a simple modification of our parallel gcd algorithm, which computes the extended gcd within the same time and processor bounds (replacing k by $k/8$ will result in the same processor bound, while only increasing the run-time by a factor of 8).

6. Implementation in the CREW Model.

In this section we briefly sketch the changes necessary to implement the algorithm in a model which forbids concurrent writes but allows concurrent reads (the CREW model). In general, transforming an algorithm from CRCW to CREW might require an extra multiplicative factor in the run-time [10]. This factor is the logarithm of the number of possible concurrent writes to the same memory location. We have used concurrent writes in two places: accessing the look-up tables, and performing addition.

All look-up tables in our algorithm are indexed by $O(k)$ bits. Avoiding concurrent writes for this size table results in a factor of $O(\log k)$. The main difficulty lies in adding two n-bit integers. Unfortunately, the fan-in in the Chandra, Fortune, and Lipton addition algorithm is n^c (for $c>0$). Thus, a direct simulation

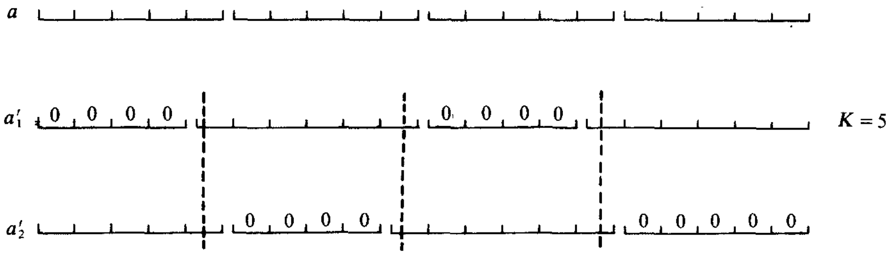

Fig. 2. Partition of a to slightly overlapping blocks.

would require $O(\log n)$ time per addition. We circumvent this by keeping a and b in the redundant representation $a = \pm a_1' \pm a_2'$, where

$$a_1' = \sum_{j=0}^{n/2k} (2^k)^{2j} \cdot \alpha_{2j},$$

$$a_2' = \sum_{j=0}^{n/2k} (2^k)^{2j+1} \cdot \alpha_{2j+1}$$

and $0 \le \alpha_i \le 2^k$, for every $0 \le i \le n/k$ (and similarly for $b = \pm b_1' \pm b_2'$). That is, a is represented as the sum of two integers with "almost disjoint block structure" (see Figure 2), with possible overlaps at locations ik $(i = 1, 2, \ldots, \lfloor n/k \rfloor)$, which correspond to powers of 2^k.

We now show how to maintain this representation under the k-transformation $(a, b) \leftarrow (ac + be, ad + bf)/2^k$ using $O(1)$ table look-ups. When multiplying a_i' $(i = 1, 2)$ by c (a k-bit integer), each block expands to length at most $2k$ $(\alpha_i' c \le 2^k \cdot (2^k - 1) < 2^{2k})$. These $2k$-long blocks do not overlap. So $ac + be$ is now represented as the sum of four "ordinary" n-bit integers. We partition each of these integers into k-bit blocks. First, corresponding blocks are added using a table look-up. Each sum, s_i, is written as a $(k+2)$-bit integer. Next, the two most-significant bits of s_i are added to the k least-significant bits of s_{i+1}, yielding s_{i+1}'. Each s_i' is now at most $(k+1)$-bits long. Finally, the most-significant bit of each s_i' is added to the k least-significant bits of s_{i+1}', resulting in a final α_{i+1} which is at most 2^k, and is "almost disjoint" from α_i.

From the above discussion it follows that our parallel gcd algorithm can be implemented in $O(n \cdot \log k/k + \log^2 k)$ time on a CREW PRAM with $O(k^3 2^{3k} + n 2^{2k})$ bit processors.

7. Concluding Remarks.

We did not explicitly specify which processor is assigned to what task at every step of the algorithm. The reader could easily check that this processor allocation can be handled uniformly (with a scheme that works for every n), and thus our results hold for the "uniform PRAM" model.

In all steps of the algorithm, each processor reads and possibly writes to a fixed memory location. Thus, by "hardwiring" the CRCW-PRAM algorithm, we get a Boolean circuit whose depth is the parallel time of the algorithm, and whose fan-in is bounded by the number of processors that can potentially access the same memory cell at the same step of the computation. In our case, this gives an unbounded fan-in circuit of depth $O(n/k)$, size $n2^{O(k)}$, and fan-in $n2^{O(k)}$. For $k = O(\log n)$, the resulting unbounded fan-in circuit can be easily transformed into a *bounded* fan-in circuit of *linear* depth and polynomial size. The question of finding a sublinear depth, polynomial size, bounded fan-in circuit for integer gcd, remains open. In particular, such a result would follow from any CRCW-PRAM algorithm with parallel time $o(n/\log n)$ and a polynomial number of processors.

Acknowledgments. We would like to thank Ming-Deh Huang, Erich Kaltofen, and Charles Leiserson for helpful suggestions.

References

[1] Brent, R. P., and H. T. Kung, Systolic VLSI arrays for linear time gcd computation, in *VLSI 83, IFIP*, F. Anceau and E. J. Aas (eds.), pp. 145-154, Elsevier, Amsterdam, 1983.

[2] Chandra, A. K., S. Fortune, and R. Lipton, Unbounded fan-in circuits and associative functions, *Proceedings of the Fifteenth Annual Symposium on Theory of Computing*, ACM, pp. 52-60, 1983.

[3] Cole, S. N., Real-time computation by n-dimensional iterative arrays of finite-state machines, *IEEE Transactions on Computers*, Vol. 18 (1969), pp. 349-365.

[4] Goldschlager, L., A unified approach to models of synchronous parallel machines, *Journal of the Association for Computing Machinery*, Vol. 29, No. 4 (1982), pp. 1073-1086.

[5] Kannan, R., G. Miller, and L. Rudolph, Sublinear parallel algorithm for computing the greatest common divisor of two integers, *SIAM Journal on Computing*, Vol. 16, No. 1 (1987), pp. 7-16.

[6] Knuth, D., *The Art of Computer Programming*, Vol. 2, second edition, Addison-Wesley, Reading, MA, 1981.

[7] Kucera, L., Parallel computation and conflicts in memory access, *Information Processing Letters*, Vol. 14, No. 2 (1982), pp. 93-96.

[8] Ofman, Y., On the algorithmic complexity of discrete functions, *Soviet Physics. Doklady*, Vol. 7, No. 7 (1963), pp. 589-591.

[9] Smith III, A. R., Cellular automata complexity trade-offs, *Information and Control*, Vol. 18 (1971), pp. 466-482.

[10] Vishkin, U., Implementation of simultaneous memory access in models that forbid it, Technical Report No. 210, Department of Computer Science, Technion, Haifa, 1981.

[11] Wallace, C. S., A suggestion for a fast multiplier, *IEEE Transactions on Electronic Computers*, (1964), pp. 14-17.