The Integer Greatest Common Divisor is NC

S. M. Sedjelmaci
LIPN, CNRS UPRES-A 7030
Université Paris-Nord,
Av. J.B.-Clément, 93430 Villetaneuse, France.
sms@lipn.univ-paris13.fr
March 2007.

Abstract: We prove that the Greatest Common Divisor (gcd) of two integers of less than n bits, can be computed in $O(\log^2 n)$ time with a polynomial number of processors. As a consequence, integer coprimality and modular inversion problems also belong to the NC class.

Keywords: Parallel algorithms, Parallel integer GCD, Complexity analysis; NC class Complexity, Number theory.

1 Introduction

A considerable progress have been made in many area of parallel algorithms. For many problems, NC algorithms have been found, among them matrix and polynomials computations. Perhaps one of the first important result is given in Csansky's paper [6], where he described a fast determinant and polynomial characteristic algorithms of a matrix over fields of characteristic zero. A generalization of these results was given by Borodin et al. [2], where they show that computing determinant and characteristic polynomial, over arbitrary fields are in $\mathtt{NC}^2$. Von zur Gathen [18] gives $\mathtt{NC}^2$ algorithms for computing polynomial GCD (greatest common divisor), LCM, squarefree decomposition over fields of charasteristic zero, and computing the extended Euclidean scheme of two polynomials over arbitrary field. On the other hand, for many other problems, proofs of P-completness have

been obtained such as the circuit value problem [11] or linear programming [7] for example. However, the situation is less satisfactory for some resistant problems. While NC algorithms have been discovered for the basic arithmetic operations, the parallel complexity of some fundamental problems as integer gcd, integer coprimality and modular inversion has remained open, since first being raised in a paper of Cook [5], more then 20 years ago. Many authors attempt to design fast parallel integer gcd algorithms, among them, we mention the first sublinear algorithm of Kannan-Miller-Rudolf [8] with $O(n \log\log n / \log n)$ parallel time with $O(n^2 \log^2 n)$ number of processors on PRAM models. In 1990, Chor and Goldreich [4] improves this parallel performance with $O(n/\log n)_\epsilon$ parallel time with $O(n^{1+\epsilon})$ number of processors, for any $\epsilon > 0$. Sorenson [16] and the author [14] also suggest other parallel algorithms with the same parallel performance. Since then, no major improvements have been made.

In this paper we discuss four basic problems: Integer gcd, Integer Coprimality, Extended Integer gcd and Modular Inversion. We prove that all these problems are in the NC class.

Our approach is based on a very slow sequential algorithm, since it computes the gcd of two integers in no less than $O(n^4)$, far from the fastest integer gcd known of Knuth-Schonhage [13] with $O(n \log^2 n \log\log n)$ time. Perhaps, our $O(n^4)$ sequential algorithm is the slowest GCD algorithm presently known. However, it is simpler than Stein's binary algorithm [15], since only adds and rightshifts are used, but its simplicity allow some mofications which lead to a simple parallelization.

The main ideas used to achieve our NC parallel algorithm are the following:

1. The basic idea for the whole theory is the result by Valiant-Skyum-Berkowitz-Rackoff [17]. It says that any sequential program computing a polynomial of degree $< d$ with C steps can be converted to a parallel program with parallel time $O(\ (\log d)\ (\log C + \log d)\ )$ using $O(\ (Cd)^\beta\ )$ processors, for an appropriate β.

2. A Fixed-point Loop Lemma which, under some conditions, avoids a `while` loop and replaces it by a `for` loop.

Throuhough this paper, we represent the input integers as formal strings of bits obtained from their binary expansions and ask for PRAM model [9] with bit instructions solving the problem. The paper is organized as follows: In Section 2, we describe our sequential and parallel algorithms. Section 3 is devoted to a

parallel extended GCD algorithm and its applications. We conclude with some remarks in Section 4.

2 The Greatest Common Divisor

2.1 The Fixed Point Lemma

Before describing our sequential gcd algorithm, we propose a lemma which is a basic tool for avoiding conditional loop as **While** Condition(X) **do**, or **Repeat** Instructions(X) **Until** Condition(X), where Condition(X) is a boolean condition on the variable X.

Lemma: Let F be a discrete function defined on vectors (or a set of ordered list) of n integers. We assume that, for a given such vector X of integers, $F(X)$ is computed by the following while loop (the repeat-until case is similar) :

$X \leftarrow X_0$;
While Condition(X) **do**
    $X \leftarrow F(X)$;
EndWhile
Return X.

If the final value X^* is a fixed point of F, i.e.: $F(X^*) = X^*$, after no more than $N_{max} = n^{O(1)}$ iterations, then the computation of X^* can be done in a free conditional loop way (N is any integer such that $N \geq N_{max}$):

$X \leftarrow X_0$;
For $i = 1$ to $N \geq N_{max}$ **do**
    $X \leftarrow F(X)$;
EndFor
Return X.

Proof: If the while loop terminates with the value X^* after N_1 iterations with $N_1 \leq N_{max}$, so is in the for loop. The for loop continues and gives, in the next iteration $N_1 + 1$, the value $F(X^*) = X^*$, since X^* is a fixed point of F, and so on until iteration N, hence the result.

Now, let us consider the following algorithm:

Input: $x,\ y > 0$ odds ;
Output: $\gcd(x, y)$;

$$\begin{pmatrix} u \\ v \end{pmatrix} \leftarrow \begin{pmatrix} x \\ y \end{pmatrix} ;$$

While ($u \neq v$)

$$\begin{pmatrix} u \\ v \end{pmatrix} \leftarrow \begin{pmatrix} v \\ (u+v)/2^t \end{pmatrix};\ \text{such that } (u+v)/2^t \text{ is odd.}$$

EndWhile

Return u.

Fig. 1. `The While-GCD` Algorithm

Example: Let $(x, y) = (35, 19)$ we obtain in turn:

$$\begin{pmatrix} 35 \\ 19 \end{pmatrix} \to \begin{pmatrix} 19 \\ 27 \end{pmatrix} \to \begin{pmatrix} 27 \\ 23 \end{pmatrix} \to \begin{pmatrix} 23 \\ 25 \end{pmatrix} \to \begin{pmatrix} 25 \\ 3 \end{pmatrix}$$

$$\to \begin{pmatrix} 3 \\ 7 \end{pmatrix} \to \begin{pmatrix} 7 \\ 5 \end{pmatrix} \to \begin{pmatrix} 5 \\ 3 \end{pmatrix} \to \begin{pmatrix} 3 \\ 1 \end{pmatrix} \to \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

Proposition: Let $x, y \geq 1$ be two odd integers. The transformation $(u, v) \to (v, (u+v)/2^t)$, with $(u+v)/2^t$ odd, preserves the gcd, i.e.: $\gcd(v, (u+v)/2^t) = \gcd(u, v)$.

Proof: Since u and v are both odd, we have $\gcd(v, (u+v)/2^t) = \gcd(v, u+v) = \gcd(u, v)$.

Theorem: Let $u, v \geq 1$ be two odd integers such that $|u - v| = r2^t > 1$, with $r \geq 1$ odd, and $t \geq 1$. Let (u_k, v_k) be the sequence of consecutive values of u and v, obtained in the `While-GCD` algorithm. Then

i) $\max\{u_{t+1}, v_{t+1}\} < (3/4)\ \max\{u, v\}$.

ii) The algorithm terminates after at most $O(n^2)$ iterations and returns $\gcd(x, y)$.

Proof: First, note that if $u = v$, then the algorithm terminates and return $\gcd(u, v) = u$. Now we assume that $u \neq v$ and $(u, v) = (v_0 + r2^t, v_0)$, the case $(u, v) = (u_0, u_0 + r2^t)$ is similar. We have

$$\begin{pmatrix} u_0 = v_0 + r2^t \\ v_0 \end{pmatrix} \to \begin{pmatrix} v_0 \\ v_0 + r2^{t-1} \end{pmatrix} \to \begin{pmatrix} v_0 + r2^{t-1} \\ v_0 + r2^{t-2} \end{pmatrix}$$

$$\rightarrow \begin{pmatrix} v_0 + r2^{t-2} \\ v_0 + r2^{t-2} + r2^{t-3} \end{pmatrix} \rightarrow \cdots \rightarrow \begin{pmatrix} v_0 + r2^{t-2} + \ldots + 2r \\ 1/2^m\{v_0 + r2^{t-2} + r2^{t-3} + \ldots r\} \end{pmatrix}$$

After t iterations, the integer $2^m v_t = v_0 + r2^{t-2} + r2^{t-3} + \ldots r$ is even. So $v_t < (1/2)u_0$ and $u_t < u_0$. Then, after $t+1$ iterations, we have $u_{t+1} = v_k < (1/2)u_0$ and $v_{t+1} \leq (1/2)(u_t + v_t) < (3/4)u_0$. The case $(u_0, u_0 + r2^t)$ is similar and gives the same upper bound. So, after $t+1$ iterations, we have $\max\{u_t, v_t\} < (3/4)\max\{u, v\}$. Similarly, if $u_{t+1} = v_{t+1}$, we stops and return the result: $u_{t+1} = \gcd(u, v)$. Otherwise $|u_{t+1} - v_{t+1}| = r_2 2^{t_2} > 1$, then we repeat the same process to the pair (u_{t+1}, v_{t+1}), and obtain: after $t_2 + 1$ more iterations

$$\max\{u_{t_2}, v_{t_2}\} < (3/4)\ \max\{u_t, v_t\} < (3/4)^2 \max\{u, v\},$$

and so on until we reach a pair (u_m, v_m), with $u_m = v_m$. Moreover, we observe that $t = t_1 < n$, $t_2 < n$ and so on: $t = t_1 < n, \cdots, t_i < n, \cdots, t_p < n$, for $p \geq 1$. So, $(t_1 + 1) + (t_2 + 1) + \cdots, (t_p + 1) \leq pn$, and after pn iterations we have

$$1 \leq \max\{u_{pn}, v_{pn}\} < (3/4)^p \max\{u, v\} < (3/4)^p 2^n.$$

Hence, $p < \frac{n}{\log_2(4/3)} \sim (2.169)\ n$ and the `While-GCD` algorithm terminates after at most $pn < (2.17)\ n^2$ iterations.

We prove in the next algorithm that we can avoid the while loop tests:

Input: $x,\ y > 0$ odds, $\max(x, y) < 2^n$.
Output: $\gcd(x, y)$;

$$\begin{pmatrix} u \\ v \end{pmatrix} \leftarrow \begin{pmatrix} x \\ y \end{pmatrix} \ ;$$

For $i = 1$ **to** $3n^2$ **Do**

$$\begin{pmatrix} u \\ v \end{pmatrix} \leftarrow \begin{pmatrix} v \\ (u+v)/2^t \end{pmatrix} \ ;\ \text{such that } (u+v)/2^t \text{ is odd.}$$

EndFor
Return u.

Fig. 2. *The* `For-GCD` *Algorithm*

Theorem: The `For-GCD` algorithm terminates after the $3n^2$ iterations and returns $\gcd(x, y)$. Therefore, its worst-case bit complexity is $O(n^3)$.

Proof: Straightforward from the fixed point Lemma. However, we prefer to give the proof in details for lazy readers. The algorithms `For-GCD` and `While-GCD` give the same pair (u_i, v_i) until we reach a pair (u_k, v_k) such that $u_k = v_k$, with $k \leq \lfloor n^2/(2 - \log_2 3) \rfloor$. At this point, the `While-GCD` algorithm terminates and returns u_k, and the `For-GCD` algorithm loops with the same consecutive pair (u_k, v_k), until the $(3n^2)$th iteration, i.e.:

$$\forall i, \quad 0 \leq i \leq 3n^2 - k \quad (u_{k+i}, v_{k+i}) = (u_k, v_k) = (u_k, u_k).$$

Hence the result.

We prove, in the following, how to compute rightshifts without division and branching.

First, we assume that the constant $t \geq 1$ is known. From now on, we consider integers as ordered lists of bits, obtained from their binary expansion. Let A be an integer formally represented by its ordered list of n bits. We can easily do the t rightshifts: $A \rightarrow A/2^t$ odd, as follows:

Input: $t \geq 1$, an ordered list of n bits $A = (a_n, a_{n-1}, \cdots, a_1)$.
Output: A list $B = (b_n, b_{n-1}, \cdots, b_1)$ obtained by t rightshift the list A ;

For $i = 1$ **to** $(n - t)$ **Do** $b_i = a_{i+t}$;
For $i = n - t + 1$ **to** n **Do** $b_i = 0$;
Return B.

The `Rightshift`(A, n, t) procedure.

Since it might not always be clear how to pick the constants t required for this shift, the conversion may become non-uniform. However, rather than computing t, we find easier and helpful to compute $2^t - 1$ without division and branching, as follows:

Input: An ordered list of n bits $A = (a_n, a_{n-1}, \cdots, a_1)$, s.t.: $A = 2^t r$, with r odd.
Output: A ordered list $B = (b_n, b_{n-1}, \cdots, b_1)$, of n bits s.t.: $B = 2^t - 1$.

$b_1 = 1 - a_1$;
For $i = 1$ **to** $n - 1$ **Do**
    $b_{i+1} = (1 - a_i)\ .\ b_i$;
EndFor
Return B.

The `Modulo2`(A) procedure.

The basic idea is that starting from the first rightmost $a_i = 1$, all the values of b_i become 0, i.e.: if $i \leq j \leq n$, we have $b_j = 0$.

Example If $A = 100$, then $A = (1, 1, 0, 0, 1, 0, 0)_2$ and we obtain $B = (0, 0, 0, 0, 0, 1, 1)_2$. The successive values of a_i and b_i are given in Table 1.

i	a_i	b_i
1	0	1
2	0	1
3	1	0
4	0	0
5	0	0
6	1	0
7	1	0

Table 1: Successive values of a_i and b_i.

Combining the previous ideas, rightshifts can be straightforward computed in the following `Makeodd`(A) procedure:

Input: An integer $A \geq 1$, with $A = (a_n, a_{n-1}, \cdots, a_1)_2$.
Output: An odd integer $C \geq 1$, with $C = (c_{n+1}, c_n, c_{n-1}, \cdots, c_1)_2$
such that, as integers, $C = A/2^t$ is odd.

$B \leftarrow$ `Modulo2`(A) ; $c_{n+1} = 0$;
For $k = 1$ **to** n **Do**
For $i = 1$ **to** $(n - b_k)$ **Do** $c_i = b_k \,.\, a_{i+1} + (1 - b_k).a_i$;
For $i = (n - b_k + 1)$ **to** $n + 1$ **Do** $c_i = 0$;
EndFor
Return $C = (c_{n+1}, c_n, c_{n-1}, \cdots, c_1)_2$.

The `Makeodd`(A) procedure.

Observations:

1) In order to make valid the instruction "**For** $i = n - b_k + 1$ **to** $n + 1$ **Do** $c_i = 0$ " in case $b_k = 0$, we have added an extra bit $c_{n+1} = 0$. This does not affect the final value of integer C.

2) Note that if $b_k = 0$, then $c_i = a_i$, for $i = 1, 2, \cdots, n$, and $c_{n+1} = 0$ i.e.: $C = A$. Moreover, if $b_k = 1$, then C is obtained by one rightshift A, i.e.: $c_i = a_{i+1}$, for $i = 1, 2, \cdots, n-1$ and $c_n = c_{n+1} = 0$. So, as far as $b_k = 1$, we rightshift A and finally obtain, step by step, the expected $C = A/2^t$. For the previous example, $C = A/2^2$.

3) The `Makeodd`(A) procedure returns A if A is odd since $B = 0$.

4) The `Makeodd`(A) procedure is division and branching free.

5) It is worth to note that `Makeodd`(A) procedure is computed with $O(n^2)$ time in bit-complexity, slower than the $O(n)$ time for the `Rightshift`(A, n, t) procedure. However, `Rightshift`(A, n, t) procedure needs the knowledge of the parameter t, in advance, i.e.:, how many rightshifts we must do, at each iteration, so parity tests are needed. By contrast, no such tests are needed in `Makeodd`(A) procedure, thus, this allow uniform computations and justifies its extra amount of time.

Now the `For-GCD` algorithm becomes:

Input: $x,\ y > 0$ odds, $\max(x, y) < 2^n$.
Output: $\gcd(x, y)$;

$$\begin{pmatrix} u \\ v \end{pmatrix} \leftarrow \begin{pmatrix} x \\ y \end{pmatrix} \ ;$$

For $i = 1$ **to** $3n^2$ **Do**

$$\begin{pmatrix} u \\ v \end{pmatrix} \leftarrow \begin{pmatrix} v \\ \texttt{makeodd}(u + v) \end{pmatrix} \ ;$$

endFor
return u.

Fig. 3. *The* `Add-and-Shift-GCD` *Algorithm*

2.2 How to add integers

Although obvious, we describe here how to add two non-negative integers $u = (u_n, \cdots, u_1)$ and $v = (v_n, \cdots, v_1)$. Let $w = (w_{n+1}, \cdots, w_1)$ such that $w = u + v$. Then w can be obtained as follows:

$w_{n+1} \leftarrow 0$;
For $i = 1$ **to** n **Do**
$\begin{pmatrix} w_i \\ w_{n+1} \end{pmatrix} \leftarrow \texttt{FA}(u_i, v_i, w_{n+1})$;
EndFor
Return $w = (w_{n+1}, \cdots, w_1)$.

Where `FA` is the well known Full Adder cell. If $(x, y, \texttt{Carry1})$ are 3 bits inputs, then the `FA` outputs are 2 bits $(w, \texttt{Carry2})$ such that $w = (x + y + \texttt{Carry1}) \pmod 2$ and $\texttt{Carry2} = (x + y + \texttt{Carry1}) \text{ div } 2$, or $x + y + \texttt{Carry1} = 2\,\texttt{Carry2} + w$.

The following Lemma shows how Full adders `FA` can be computed:

Lemma 2.1 *Let $x, y, z, c_1, c_2 \in \{0, 1\}$ such that $x + y + c_1 = 2c_2 + z$. Let $F = F(x, y) = (1 - x)y + (1 - y)x$ and let $G = G(x, y) = xy$, then*

$$z = c_1(1 - F) + (1 - c_1)F \quad \text{and} \quad c_2 = (1 - c_1)G + c_1(F + G).$$

Moreover, given 3 bits x, y, c_1, then the bits c_2 and z can be computed with at most 25 basic operations $+(a, b)$, $-(a, b)$ and $\times(a, b)$, for $a, b \in \{0, 1\}$.

Proof: We observe that F and G are two bits satisfying $x + y = 2G + F$. Denoting basic operations $+(a, b)$, $-(a, b)$ and $\times(a, b)$ by *unit cost*. The bits F and G can be computed with respectively 5 and 1 units, so that and z and c_2 costs respectively 14 and 11 units. Hence the result. Note that $-a$ can be simulated by $\times(-1, a)$.

Theorem: The `Add-and-Shift-GCD` algorithm can be converted to a parallel algorithm computing the GCD of two integers, in $O(\log^2 n)$ time

with a polynomial numbers of processors. Therfore, the INTEGER GCD problem is NC.

Proof: The `Add-and-Shift-GCD` algorithm returns the $\gcd(x, y)$ after in $O(n^4)$ time in bit-complexity. During the whole `Add-and-Shift-GCD` algorithm, only bits of integers inputs are formally manipulated with only $+, -$ and $\times$ as basic bit operations. Thus our algorithm can be considered as a *program*, in the sens defined by Valiant-Skyum-Berkowitz-Rackoff [17], computing the gcd of two n bits integers u and v in $O(n^4)$ steps. Therefore, we can now apply the parallelization method of Valiant-Skyum-Berkowitz-Rackoff [17]. The `Add-and-Shift-GCD` algorithm can be converted to a parallel algorithm for computing the GCD of two integers, in $O(\log^2 n)$ time and a polynomial number of processors. Neither division nor branching occurs.

Corollary: The COPRIMALITY problem is NC.

3 Parallel Extended GCD

For given two n-bits integers u, v, the EXTENDED GCD problem is to find integers a, b, and d, such that

$$au + bv = d,$$

where $d = \gcd(u, v)$. A important application is that, in case $d = 1$, we can straightforward compute the inverse of u modulo v, since $au \equiv 1 \pmod{v}$.

Theorem: Given two n-bits integers u and v, there exists a parallel extended gcd algorithm which compute the pair of integers (a, b) in $O(\log^2 n)$ time with a polynomial numbers of processors. Therfore, the EXTENDED GCD problem is NC.

Proof: It is well know that Stein's binary [15] and Brent-Kung's Plus-Minus algorithms [3] can be converted in extented gcd version. As for our algorithm `Add-and-Shift-GCD`, both these algorithms are only based on adds/subtractions and rights-schifts. However, it also deals with conditional

loops as while or repeat-until, and parity tests. Although we have seen that rights-schifts is not actually an issue, we cannot straightforward follow these methods without showing how to avoid conditional loops and parity tests. We give below the main ideas to solve these issues.

Let $\begin{pmatrix} u_k \\ v_k \end{pmatrix}_{k\geq 0}$ be the sequence of consecutive values of $(u,v)^T$ obtained in algorithm `Add-and-Shift-GCD`. So we have

$$\begin{pmatrix} u_{k+1} \\ v_{k+1} \end{pmatrix} \leftarrow M \times \begin{pmatrix} u_k \\ v_k \end{pmatrix},$$

where the matrix M is either A_1 (adding) or A_2 (shifting), with

$$A_1 = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \qquad \text{and} \qquad A_2 = \begin{pmatrix} 1 & 0 \\ 0 & 1/2 \end{pmatrix},$$

and, if t_i is the number of right-shifts at each iteration i, then:

$$M_i = (A_2)^{t_i} A_1.$$

However the coefficients of the matrix A_2 are not integers, but we can replace A_2 by the following matrix N:

$$N = \begin{pmatrix} 1 & 0 \\ v/2 & (1-u)/2 \end{pmatrix}.$$

The coefficients of N are integers and

$$N \begin{pmatrix} u \\ v \end{pmatrix} = A_2 \begin{pmatrix} u \\ v \end{pmatrix},$$

and we can substitute, at each iteration i, M_i by $S_i = (N)^{t_i} A_1$. Finally, at the end of the algorithm, we obtain

$$\Big(\prod_{i=1}^{3n^2} S_i \Big) \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \quad (1).$$

Moreover, the t_i's can be stored in an array T as follows:

For $i = 1$ **to** $3n^2$ **Do**
$\begin{pmatrix} u \\ v \end{pmatrix} = A_1 \begin{pmatrix} u \\ v \end{pmatrix}$;
$B \leftarrow$ `Modulo2`(v) ;
$t_i = 0$;
For $j = 1$ **to** n **Do** $t_i = t_i + b_j$;
$T[i] = t_i$;
`Makeodd`(v) ;
EndFor

or by storing the matrices N with their respective multiplicity t_i in an array R, as follows (I is the 2×2 identity matrix):

For $i = 1$ **to** $3n^2$ **Do**
$\begin{pmatrix} u \\ v \end{pmatrix} = A_1 \begin{pmatrix} u \\ v \end{pmatrix}$;
$B \leftarrow$ `Modulo2`(v) ;
$t_i = 0$;
For $j = 1$ **to** n **Do**
$R[i; j] = b_i N + (1 - b_i) I$;
`Makeodd`(v) ;
EndFor

Therfore, there exists an $O(n^4)$ algorithm which computes all the required matrices to obtain relation (1). Now we can apply the parallelization method of Valiant-Skyum-Berkowitz-Rackoff [17]. Hence, there exists a parallel algorithm for computing these matrices, in $O(\log^2 n)$ time and a polynomial number of processors. Neither division nor branching occurs. Moreover their product can be achieved in a same time bound $O(\log^2 n)$ and $O(n^3 \log n)$ processors, with a binary tree computations.

Corollary: The Modular Inversion problem is `NC`.

4 Conclusion

Thanks to the parallelization method of Valiant-Skyum-Berkowitz-Rackoff, we have proved that the Integer Gcd problem as well its extended version are in NC class. Consequently, many others problems such as Integer Coprimality, Modular Inversion are also in NC class. We observe that most of gcd algorithms use preserving gcd transformations until zero is reached, the other non-zero integer is the expected gcd. However, because of the zero integer, the fixed point Lemma may not be easily applied. Finally, we hope that the fixed point Lemma should help to to provide division and branching free algorithms for others problems.

References

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman *The Design and Analysis of Computer Algorithms* (Addison Wesley, 1974).

[2] A. Borodin, J. von zur Gathen and J. Hopcroft, Fast parallel matrix and GCD computations, *Information and Control.* **52** 3 (1982) 241–256.

[3] R.P. Brent and H.T. Kung, Systolic VLSI arrays for linear-time GCD computation, in: Anceau and Aas eds., *Proceedings of VLSI'83* (1983) 145–154.

[4] B. Chor and O. Goldreich, An improved parallel algorithm for integer GCD, *Algorithmica.* **5** (1990).

[5] S. Cook, A Taxonomy of Problems with Fast Parallel Algorithms, *Information and Control.* **64** (1985) 2–22.

[6] L. Csanky, Fast parallel matrix inversion algorithms, *SIAM J.Comp.,* **5** (1976) 618–623.

[7] D. Dobkin, R. Lipton and S. Reiss, Linear Programming is logspace hard for P, *Information Processing Letters.,* **8** (1979) 96–97.

[8] R. Kannan, G. Miller and L. Rudolph, Sublinear Parallel Algorithm for Computing the Greatest Common Divisor of Two Integers, *SIAM J. on Computing.* **Vol 16, 1**, (1987) 7–16.

[9] R. Karp, V. Rammachandran, *Parallel Algorithms for Shared-memory Machines* in J.Van Leeuwen, Editor, Algorithms and Complexity., (Elsvier and MIT Press, 1990, Handbook of Theoretical Compuer Science) **Vol. A**.

[10] D.E. Knuth, *The Art of Computer Programming* (Addison Wesley, 3rd ed., 1998), **Vol. 2**.

[11] R. Ladner, The circuit value problem is logspace complete for P, *SIGACT News* **7, 1** (1975) 18–20.

[12] D.H. Lehmer, Euclid's algorithm for large numbers, *American Math. Monthly* **45** (1938) 227-233.

[13] A. Schönhage, Schnelle Berechnung von Kettenbruchentwicklugen, *Acta Informatica* **1** (1971) 139-144.

[14] M.S. Sedjelmaci, On a Parallel Lehmer-Euclid GCD Algorithm, *Proceedings of the International Symposium on Symbolic and Algebraic Computation ISSAC'2001* (2001) 303–308.

[15] J. Stein, Computational problems associated with Racah algebra, *J. of Comput. Phys.,* **1** (1967) 397–401.

[16] J. Sorenson, Two Fast GCD Algorithms, *J. of Algorithms* **16** (1994) 110–144.

[17] L.G. Valiant, S. Skyum, S. Berkowitz and C. Rackoff, Fast parallel computation of polynomials using few processors, *SIAM J.Computing* **12** No.4 (1983) 641–644.

[18] J. Von zur Gathen, Parallel algorithms for algebraic problems, *SIAM J.Computing* **13** No.4 (1984) 802–824.