

Two fast parallel GCD algorithms of many integers

S.M. Sedjelmaci
LIPN CNRS UMR 7030
Université Paris-Nord
Paris, France
sms@lipn.univ-paris13.fr

ABSTRACT

We present two new parallel algorithms which compute the GCD of n integers of $O(n)$ bits in $O(n/\log n)$ time with $O(n^{2+\epsilon})$ processors in the worst case, for any $\epsilon > 0$ in CRCW PRAM model. More generally, we prove that computing the GCD of m integers of $O(n)$ bits can be achieved in $O(n/\log n)$ parallel time with $O(mn^{1+\epsilon})$ processors, for any $2 \leq m \leq n^{3/2}/\log n$, i.e. the parallel time does not depend on the number m of integers considered in this range. We suggest an extended GCD version for many integers as well as an algorithm to solve linear Diophantine equations.

KEYWORDS

GCD of many integers; Parallel algorithms; Parallel Complexity of GCD; Complexity analysis

ACM Reference format:

S.M. Sedjelmaci. 2017. Two fast parallel GCD algorithms of many integers. In *Proceedings of ACM ISSAC conference, Kaiserslautern, Germany, July 25–28, 2017 (ISSAC’17)*, 8 pages. <https://doi.org/10.1145/3087604.3087610>

1 INTRODUCTION

The computation of the GCD of two integers is not known to be in the NC class, nor it is known to be P-complete [2]. The best parallel performance with deterministic algorithms was first obtained by Chor and Goldreich [5], then by Sorenson [18] and Sedjelmaci [15] since they propose, with different approaches, parallel integer GCD algorithms which can be achieved in $O(n/\log n)$ time with $O(n^{1+\epsilon})$ number of processors, for any $\epsilon > 0$, in Concurrent Read Concurrent Write (CRCW) PRAM model (see [10] for the PRAM model of computation). Table 1 summarizes some parallel GCD algorithms for two integers.

The GCD computation for more than two integers is important in many applications, for example in computing canonical normal forms of integer matrices [7]. There are several papers dealing with sequential algorithms computing the GCD of many integers (see [1, 3, 4, 9, 12, 13, 20]).

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ISSAC’17, July 25–28, 2017, Kaiserslautern, Germany
© 2017 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-5064-8/17/07...\$15.00
<https://doi.org/10.1145/3087604.3087610>

This paper deals with fast parallel algorithms for computing the GCD of many integers. Probabilistic approaches are given in [6] and [8]. In [6], the authors show how the GCD computation of n integers can be reduced to the case of two integers, by means of random linear combinations, with probability $> 1/2$. This gives an efficient parallel algorithm with an extra $O(\log n)$ time. However this algorithm is randomized.

The parallel deterministic case does not seem to be addressed. A naive approach, using a binary tree computation to compute the GCD of n integers of $O(n)$ bits would require $O(n)$ parallel time with $O(n^{2+\epsilon})$ processors, for any $\epsilon > 0$. One may also use the existing parallel GCD algorithms of two integers and try to adapt them to reach a fast parallel GCD algorithm of many integers. However, it is not obvious how to preserve the same $O(n/\log n)$ parallel time as for the GCD of two integers with $O(n^{2+\epsilon})$ processors, which is roughly the bit-size of all the n input integers.

In [17] a first attempt to achieve a parallel algorithm with this parallel performance was presented. The main idea is to use under some conditions the pigeonhole principle on the n input integers of the vector $A = (a_0, \dots, a_{n-1})$. There exists two integers say (a_i, a_j) that match in their $O(\log n)$ most significant bits. Their difference $\alpha = |a_i - a_j|$ is in fact $O(\log n)$ bits smaller w.r.t. the larger size of input integers a_i 's. Then we reduce all the other components of A modulo α to obtain a new vector A' where all its components are $O(\log n)$ bits smaller. The algorithm just iterates this process.

However, there are some drawbacks to this algorithm: The number of distinct integers is important. For example if there are only $O(\log n)$ distinct integers then the pigeonhole technique will reduce the bit size of the integers by $O(\log \log n)$ (see Corollary 1) and the number of iteration in the while loop will be $O(n/\log \log n)$ which is expensive. So we cannot reach the $O(n/\log n)$ parallel time. Moreover, comparing the integers to zero and/or counting the number of distinct integers will cost $O(\log n)$ parallel time. All these operations must be done at each iteration and will cost an extra $O(\log n)$ parallel time with a polynomial number so that the time complexity of the algorithm will be $O(n)$ in parallel with polynomial number of processors. So two questions remain unclear:

- Q1: What happens if $\alpha = 0$? For example, if $n = 8$ and $A = (255, 255, 193, 161, 129, 97, 65, 65)$, then there are only two pairs of integers that match in their 3 most significant bits, namely (255, 255) and (65, 65). Unfortunately, in both cases $\alpha = 0$. This is important since we must reduce modulo α , so how to do it if no such $\alpha > 0$ exists ?

- Q2: Can we find such a small $\alpha > 0$ in $O(1)$ parallel time with only $O(n^{2+\epsilon})$ processors, which is roughly the bit-size of all the n input integers ?

In this paper we present two new algorithms. We show how to fix these two issues by adding a new transformation to the algorithm presented in [17]. This new transformation is based on the best rational approximations (continued fractions). The second algorithm is much simpler than the first one since it is only based on this new transformation.

The main results of the paper are summarized below :

- The GCD computation of n integers of $O(n)$ bits can be achieved in $O(n/\log n)$ parallel time with $O(n^{2+\epsilon})$ processors, for any $\epsilon > 0$ in CRCW PRAM model, in the worst case.
- More generally, the GCD computation of m integers of $O(n)$ bits can be achieved in $O(n/\log n)$ parallel time with $O(mn^{1+\epsilon})$ processors, for any $2 \leq m \leq n^{3/2}/\log n$, i.e. the parallel time does not depend on the number m of integers considered in this range. To our knowledge, it is the first time that we find deterministic algorithms which compute the GCD of many integers with this parallel performance and polynomial work.
- We suggest an extended GCD version for many integers as well as an algorithm to solve linear Diophantine equations.

We first restrict our study to the case of n integers of $O(n)$ bits. The general case of m integers of $O(n)$ bits is similar. It is addressed in Section 5. We recall the Δ -GCD algorithm presented in [17] in Section 2. A modified version of Δ -GCD algorithm as well as a new parallel GCD algorithm are presented in Section 3. Section 4 is devoted to the correctness of our new algorithms. Section 5 deals with the complexity analysis of both algorithms. We suggest an extended GCD version as well as an application to solve linear Diophantine equations in Section 6.

2 THE Δ -GCD ALGORITHM

2.1 Notations

Throughout the paper, A is a vector of n integers $A = (a_0, a_1, \dots, a_{n-1})$, with $a_i \geq 0$, $n \geq 4$ (unless in Section 5.2 where A may have m integers of $O(n)$ bits). We use in Section 3 an integer parameter k satisfying $\log k = \theta(\log n)$. We note $\gcd(A) = \gcd(a_0, a_1, \dots, a_{n-1})$ and t_i is the integer formed by the $O(\log n)$ most significant bits of a_i , i.e.: $t_i = a_i \operatorname{div} 2^{n - \lceil \log n \rceil}$ if n is the bit size of the largest integer a_i of A , $0 \leq i \leq n - 1$. The integers of the input vector A have $O(n)$ bits. We note $\gcd(0, 0) = 0$.

We use the PRAM (Parallel Random Access Machine) model of computation and CRCW PRAM (Concurrent Read Concurrent Write) sub-model (see [10, 18] for more details on PRAM model of computation). If many processors are in write concurrency then we use the *Arbitrary* sub-model of CRCW-PRAM (see [10] for more details). With this sub-model, an arbitrary one of the multiple writes to the same

location succeeds. In this parallel model, the addition of two $O(n)$ bit integers can be achieved in $O(1)$ parallel time using $O(n^{1+\epsilon})$, for any $\epsilon > 0$. Moreover, thanks to pre-computed look-up tables, all the arithmetic operations of two $O(\log n)$ bits integers can be achieved in $O(1)$ parallel time with $O(n^{1+\epsilon})$ processors (see [18] for more details).

2.2 Basic results

The Δ -GCD algorithm is based on the following results which are variant forms of the pigeonhole principle:

Lemma 1: Let $A = \{a_1, a_2, \dots, a_n\}$ be a set of n distinct positive integers, such that $n \geq 2$ and $a_n/n < a_1 < a_2 < \dots < a_n$. Then $\exists i \in \{1, 2, \dots, n-1\}$ s.t.: $a_{i+1} - a_i < a_n/n$.

Proof:

By contradiction. If we assume that $\forall i, a_{i+1} - a_i \geq \frac{a_n}{n}$, then $a_n - a_1 = (a_n - a_{n-1}) + \dots + (a_2 - a_1) \geq (n-1)\frac{a_n}{n} = a_n - \frac{a_n}{n}$. So $a_1 \leq a_n/n$: a contradiction with $a_1 > a_n/n$. \square

A straightforward consequence is the following:

Corollary 1: Let $A = \{a_1, a_2, \dots, a_n\}$ be a set of n distinct positive integers, with $n \geq 2$, then $\min\{a_k, |a_i - a_j| > 0\} \leq \max\{a_i\}/n$, where $1 \leq k, i, j \leq n$. We recall below the Δ -GCD algorithm presented in [17]:

Input: A vector $A = (a_0, a_1, \dots, a_{n-1})$ of n positive integers, $n \geq 4$ and $\max\{a_i\} < 2^n$.

Output: $\gcd(a_0, a_1, \dots, a_{n-1})$.

If $A = (a_0, a_0, \dots, a_0)$ **then Return** a_0 ;
/* A is a scalar vector */

$\alpha := a_0$; $I := 0$; $p := n$;

While $(\alpha > 1)$ **Do**

For $(i = 0)$ **to** $(n - 1)$ **ParDo**

If $(0 < a_i \leq 2^n/p)$ **then** $\{\alpha := a_i ; I := i\}$

Endfor

If $(\alpha > 2^n/p)$ **then**

 /* Compute in parallel I, J and α */

$\alpha := \min\{|a_i - a_j| > 0\} = a_I - a_J ; a_I := \alpha ;$

Endif /* Here $\alpha \leq 2^n/p$ */

For $(i = 0)$ **to** $(n - 1)$ **ParDo**

 /* Reduce all the a_i 's */

If $(i \neq I)$ **then** $a_i := a_i \operatorname{mod} \alpha ;$

Endfor /* $\forall i \neq I, 0 \leq a_i < \alpha$ */

If $(\forall i \neq I, a_i = 0)$ **then Return** $\alpha ;$

$p := np$; /* p is $O(\log n)$ bits larger */

Endwhile

Return α .

THE Δ -GCD ALGORITHM.

Remarks:

1) If the result α is given after k iterations then $\alpha \leq 2^n/n^k$, with $0 \leq k \leq n/\log n$.

2) A weak version of the function \min is used based on the pigeonhole principle, where only the $O(\log n)$ most significant bits of the integers are considered in [17].

Table 1: Some Parallel GCD algorithms

Authors	Time	Nb. of proc.	Parallel model
Brent-Kung (1983)	$O(n)$	$O(n)$	Systolic array
Purdy (1983)	$O(n)$	$O(n)$	Systolic array
Kannan et al. (1987)	$O(\frac{n \log \log n}{\log n})$	$O(n^{2+\epsilon})$	CRCW PRAM
Adleman et al. (random, 1988)	$O(\log^2 n)$	$e^{O(\sqrt{n \log n})}$	CRCW PRAM
Chor-Goldreich (1990)	$O(n / \log n)$	$O(n^{1+\epsilon})$	CRCW PRAM
Sorenson (1994)	$O(n / \log n)$	$O(n^{1+\epsilon})$	CRCW PRAM
Sedjelmaci (2001)	$O(n / \log n)$	$O(n^{1+\epsilon})$	CRCW PRAM
Sorenson (random, 2010)	$O(\frac{n \log \log n}{\log n})$	$O(n^{6+\epsilon})$	EREW PRAM

The following example shows how it works (see [17]):

Example 1:

Let $A = (912672, 815430, 721161, 565701, 662592)$. After 4 iterations, we obtain $GCD(A) = 3$ (recall $\alpha = a_I - a_J$).

$$\begin{array}{c}
 \left(\begin{array}{c} 912672 \\ 815430 \\ 721161 \\ 565701 \\ 662592 \\ \hline \alpha = 58569 \\ (I, J) = (2, 4) \end{array} \right) \rightarrow \left(\begin{array}{c} 34137 \\ 54033 \\ 58569 \\ 38580 \\ 18333 \\ \hline 4443 \\ (0, 3) \end{array} \right) \rightarrow \left(\begin{array}{c} 4443 \\ 717 \\ 810 \\ 3036 \\ 561 \\ \hline 93 \\ (1, 2) \end{array} \right) \\
 \\
 \rightarrow \left(\begin{array}{c} 72 \\ 93 \\ 66 \\ 60 \\ 3 \\ \hline 3 \\ (4, -) \end{array} \right) \rightarrow \left(\begin{array}{c} 0 \\ 0 \\ 0 \\ 0 \\ 3 \\ \hline 3 \\ \text{STOP} \end{array} \right).
 \end{array}$$

3 TWO NEW PARALLEL GCD ALGORITHMS

3.1 The Δ 2-GCD algorithm

In Section 2, only the $O(\log n)$ most significant bits are considered for computing α . However, if we consider the case when the only pair (a_i, a_j) of integers that match with their $O(\log n)$ most significant bits are all equals, i.e.: $a_i = a_j$, then $\alpha = |a_i - a_j| = 0$ (see the illustrative example given in Section 1, question Q1). So the answer of the first question Q1 whether or not the pigeonhole principle always provides a non-zero α is no.

Addressing the second question Q2: In the worst case, we can have $O(n)$ integers that match with their $O(\log n)$ most significant bits. So we must compare all the $O(n^2)$ pairs (a_i, a_j) to know if there exists among them a pair (a_i, a_j) such that $a_i \neq a_j$. This can be done in $O(1)$ parallel time but with no less than $O(n^3)$ processors, which is larger than the expected $O(n^{2+\epsilon})$ processors. So we must ask less. The idea is to use only $O(\sqrt{n})$ integers, so that all the $O(n)$ comparisons can be achieved in $O(1)$ parallel time with $O(n^2)$ processors. On the other hand, in case the pigeonhole principle holds, then we can find a pair (a_i, a_j) of distinct integers that

match roughly with their $1/2 \log n$ most significant bits. So we only introduce a factor 2 in the parallel time and only $O(n^2)$ processors are needed. Roughly speaking, the first new algorithm called Δ 2-GCD uses three successive tests, starting from the easiest one :

- **Test 1:** Is there a small enough $a_i > 0$ so that we can consider it straightforwardly as an α ?
- **Test 2:** Does the pigeonhole algorithm provide an $\alpha > 0$?
- **Test 3:** Use a new transformation R based on continued fractions (see [16]) and test if $R = 0$?

In case the third test fails, i.e.: $R(a_i, a_j) = 0$ for all the pairs of integers (a_i, a_j) , with $i, j \leq \sqrt{n}$, then this means that $\gcd(a_i, a_j) = \gcd(a_j, R(a_i, a_j)) = a_i$ and the pair (a_i, a_j) is replaced by $(0, a_i)$. This new transformation is called **reduce**. So we reduce by half the number of $O(\sqrt{n})$ positive integers considered (the other half of integers are all zeroes). Moreover, it could be iterated at most $O(\sqrt{n})$ times since, at each step, we add $O(\sqrt{n})$ new zeros in the vector A (see the illustrative Example 5). Thus this new transformation **reduce** will guarantee the termination and the parallel performance $O(n/\log n)$ time with $O(n^{2+\epsilon})$ processors (see Sections 4 and 5). We derive a new algorithm called Δ 2-GCD which corrects the Δ -GCD algorithm [17] described in Section 2.

Input: A vector $A = (a_0, a_1, \dots, a_{n-1})$ of n positive integers, $n \geq 4$ and $\max \{a_i\} < 2^n$.

Output: $\gcd(a_0, a_1, \dots, a_{n-1})$.

```

If  $A = (a_0, a_0, \dots, a_0)$  then Return  $a_0$ ;
/*  $A$  is a constant vector */
 $(\alpha, I) := (a_0, 0)$  ;  $p := n$  ;  $N := \lfloor \sqrt{n} \rfloor$  ;
While  $(\alpha > 1)$  Do
  For  $(i = 0)$  to  $(n - 1)$  ParDo
    If  $(0 < a_i \leq 2^n/p)$  then
       $\{(\alpha, I) := (a_i, i)$  ;  $S := 1$   $\}$  ;
    else  $S := 0$  ; /* No small  $a_i$  */
  Endfor
If  $(S = 0)$  then  $(\alpha, I) := \text{pigeonhole}(A, N)$  ;
If  $(I = -1)$  then
  /* The pigeonhole algorithm fails */
   $R := 0$  ;
  For  $(i, j = 0)$  to  $(N - 1)$  ParDo
     $x_{ij} := R_{ILE}(a_i, a_j)$  ;

```

```

If ( $x_{ij} > 0$ ) then
  {  $(\alpha, I) := (x_{ij}, i)$ ;  $R := 1$ ;  $a_I := x_{ij}$  }
  /* We can divide all the  $a_i$ 's by  $\alpha = x_{ij}$  */
Endif
Endfor
If ( $R = 0$ ) /*  $\forall i, j, R_{ILE}(a_i, a_j) = 0$  */
  then  $A := \text{reduce}(A, N)$ ;
Endif
Endif
If ( $I \geq 0$ ) then  $A := \text{remainder}(A, \alpha, I)$ ;
  /* We divide all the  $a_i$ 's but  $a_I$  by  $\alpha$  */
If ( $\exists a_k \neq 0$  s.t.:  $\forall i \neq k \Rightarrow a_i = 0$ ) then
  Return  $a_k$ ;
   $p := np$ ; /*  $p$  is  $O(\log n)$  bits larger */
Endwhile
Return  $\alpha$ .

```

THE $\Delta 2$ -GCD ALGORITHM ALGORITHM.

Remarks: The variables S, I and R are linked with **Test 1**, **Test 2** and **Test 3**. Their meanings are:

- $S = 1$ if there exists a small a_i , i.e.: $0 < a_i \leq 2^n/p$ and $S = 0$ otherwise.
- $I \geq 0$ if there exists $\alpha > 0$, s.t.: $0 < \alpha \leq 2^n/p$ and $I = -1$ otherwise.
- $R = 1$ if there exists i, j , s.t.: $R_{ILE}(a_i, a_j) > 0$ and $R = 0$ otherwise.

The functions $\text{remainder}(A, \alpha, I)$, $\text{pigeonhole}(A, N)$, R_{ILE} , Par-Ext-ILE and $\text{reduce}(A, N)$ are described below.

The remainder procedure just divides all the components of A by α and consider their remainders. It proceeds as follows:

```

Input:  $A = (a_0, \dots, a_{n-1})$ , with  $n \geq 4$ ,  $0 \leq I \leq n-1$ ,
  and  $\alpha > 0$ .
Output:  $A' = (a'_0, \dots, a'_{n-1})$ , s.t.:  $a'_i = a_i \bmod \alpha$ 
  for all  $i \neq I$  and  $a'_I = a_I = \alpha$ .

   $a_I = \alpha$ ;
  For ( $i = 0$ ) to  $(n-1)$  ParDo
    If ( $i \neq I$ ) then  $a_i := a_i \bmod \alpha$ ;
  Endfor
Return  $A$ .

```

THE REMAINDER ALGORITHM.

The pigeonhole algorithm is based on Corollary 1 with the first $O(\sqrt{n})$ integers of A , namely $(a_0, a_1, \dots, a_{N-1})$, with $N = \lfloor \sqrt{n} \rfloor$. The algorithm returns a pair (α, I) such that $\alpha = a_I - a_J > 0$ is small enough or, in the case there is no such pair, it returns $(\alpha, I) = (a_0, -1)$. It is described below:

```

Input:  $A = (a_0, \dots, a_{n-1})$ ,  $N = \lfloor \sqrt{n} \rfloor$ ,  $n \geq 4$ .
  /* Actually only the subset  $B = (a_0, \dots, a_{N-1})$ 
  is considered. */
Output:  $(\alpha, I)$ , s.t.:  $0 < \alpha = |a_i - a_j| \leq \max\{B\}/N$ ;
  where  $B = (a_0, \dots, a_{N-1})$ .
   $I = i$ , if such pair  $(a_i, a_j)$  exists

```

with $0 \leq i, j \leq N-1$ and
 $(\alpha, I) = (a_0, -1)$ otherwise.

```

For ( $i, j = 0$ ) to  $(N-1)$  ParDo
   $t_i := O(\log N)$  most significant bits of  $a_i$  for
  each  $0 \leq i \leq N-1$ ;
  /*  $t_i = a_i \text{ div } 2^{s - \lfloor \log N \rfloor}$  */
  /* where  $s = \lfloor \log \max\{B\} \rfloor + 1$  */
  If ( $t_i = t_j$  and  $a_i \neq a_j$ ) then
     $(\alpha, I) := (|a_i - a_j|, i)$ ;
  else  $(\alpha, I) := (a_0, -1)$ ;
  /* The pigeonhole test fails:  $I = -1$  */
Endfor
Return  $(\alpha, I)$ .

```

THE PIGEONHOLE ALGORITHM.

Example 2: (pigeonhole)

We consider the example given in the Introduction, Q1, where $A = (255, 255, 65, 65, 193, 161, 129, 97, 65)$, and $n = 9$, we choose $N = 4$ (instead of $N = 3$ just to illustrate what happens). So $B = (255, 255, 65, 65)$ and two pairs of integers match with their 3 most significant bits, namely $(255, 255)$ and $(65, 65)$. Unfortunately, in both cases $a_i = a_j$, so the pigeonhole algorithm returns $(\alpha, I) = (255, -1)$.

The R_{ILE} and Par-Ext-ILE algorithms are described in [15, 16]. ILE stands for Improved Lehmer Euclid and Par-Ext-ILE stands for a parallelization of an extended version of ILE . We just have added some special cases to the original version of R_{ILE} : $R_{ILE}(0, v) = R_{ILE}(u, 0) = 0$ and $R_{ILE}(0, 0) = 0$. Roughly speaking, $R_{ILE}(u, v)$ computes the continued fractions of order $O(n)$ for the rational v/u .

We note by n and p , respectively the number of significant bits of u and v . We use the parameter $k = 2^m$ with $m = O(\log n)$. We recall below the reductions R_{ILE} and Par-Ext-ILE .

```

Input:  $u \geq v \geq 0$ ,  $k = 2^m$ ;  $m = O(\log n)$ , s.t.:
   $n - p + 1 < m$  and  $2p > 2m + n + 2$ .
Output:  $R_{ILE}(u, v) = |su + tv| < 2v/k$ , with  $1 \leq |s| \leq k$ .

```

If ($u = 0$ or $v = 0$) **then Return** $R_{ILE} = 0$;

Step 1:

```

   $n :=$  The number of significant bits of  $u$ ;
   $p :=$  The number of significant bits of  $v$ ;
   $\lambda := 2m + n - p + 2$ ;
   $u_1 := \lfloor u/2^{p-\lambda} \rfloor$ ;  $v_1 := \lfloor v/2^{p-\lambda} \rfloor$ ;

```

Step 2: For ($i = 0$) **to** k **ParDo**

```

   $q_i := \lfloor iu_1/v_1 \rfloor$ ;  $r_i := iu_1 - q_iv_1$ ;
  If ( $r_i < v_1/k$ ) then  $(s, t) := (i, -q_i)$ ;
  If ( $v_1 - r_i < v_1/k$ ) then  $(s, t) := (-i, q_i + 1)$ ;
End ParDo

```

Step 3: Compute in parallel $R_{ILE} = |su + tv|$;

Return R_{ILE} .

THE PARALLEL VERSION OF R_{ILE} ALGORITHM.

Example 3: (R_{ILE} , see [15]) Let $u = 1,759,291$ and $v = 1,349,639$. Their binary representations are respectively:

$$\begin{aligned} 11010110 \ 1100000111011 &= 1,759,291 \\ 10100100 \ 1100000000111 &= 1,349,639 \end{aligned}$$

We have $n = p = 21$ (the number of significant bits of u and v). If we take $m = 3$, we obtain $k = 8$, $\lambda = 2m + 2 = 8$, $u_1 = 214$ and $v_1 = 164$ (the bits representing u_1 and v_1 are in bold). The extended Euclidean algorithm (EEA) with u_1 and v_1 yields the first successive integers q , r , t and s ($r = su + tv$).

q	r	s	t
	214	1	0
	164	0	1
1	50	1	-1
3	14	-3	4
3	8	10	-13

In our example, we obtain $s = -3$, $t = 4$, $r = 14 < v_1/k = 164/8 = 20.50$ and $R_{ILE} = |-3u + 4v| = 120,683 < 2v/k$.

It is proved in [15], that the computation of R_{ILE} can be achieved in $O(1)$ parallel time with $O(n^{1+\epsilon})$ processors in CRCW PRAM model, thanks to pre-computed look-up tables. These table look-up perform all the arithmetic operations of two $O(\log n)$ bits integers in $O(1)$ parallel time with $O(n^{1+\epsilon})$ processors.

Par-Ext-ILE stands for Parallel Extended ILE. It is similar to R_{ILE} since it returns R_{ILE} and the associated Bézout matrix M with the same parallel performance in CRCW PRAM model [16]. The matrix M satisfies $\det(M) = \pm 1$, $M \times (u, v)^T = (R, R_{ILE})^T$ with $0 \leq R_{ILE} < 2v/k$, $1 \leq R \leq v$ and $\gcd(u, v) = \gcd(R, R_{ILE})$ (see [16], for more details).

Example 4: (**Par-Ext-ILE**)

The previous example 3 yields **Par-Ext-ILE**(u, v) = (M, R) , with $R = R_{ILE}(u, v) = 120,683$ and $M = \begin{pmatrix} -3 & 4 \\ 10 & -13 \end{pmatrix}$.

Unlike the pigeonhole principle, the transformation **reduce** will guarantee the termination and the parallel performance of the $\Delta 2$ -GCD algorithm (see Section 4 and 5). In fact, it could be iterated at most $O(\sqrt{n})$ times since, at each step, we add $O(\sqrt{n})$ new zeros in the vector A . The **reduce** procedure is the following:

Input: $A = (a_0, \dots, a_{N-1}, \dots, a_{n-1})$; $n \geq 4$, s.t.:
 $R_{ILE}(a_{2i}, a_{2i+1}) = 0$ for all $0 \leq i \leq N-1$
with $N = \lfloor \sqrt{n} \rfloor$.

Output: $A' = (a'_0, \dots, a'_{N-1}, a'_N, \dots, 0, \dots, 0)$
with $\gcd(A') = \gcd(A)$. /* The last $n - N$
components of A' are all zero. */

Step 1: /* Use parallel R_{ILE} transformation */

For ($i = 0$) **to** $(N - 1)$ **ParDo**
 $(M, R_{2i+1}) := \text{Par-Ext-ILE}(a_{2i}, a_{2i+1})$;
 $(R_{2i}, R_{2i+1})^T := M(a_{2i}, a_{2i+1})^T$;
/* $R_{2i+1} = R_{ILE}(a_{2i}, a_{2i+1}) = 0$ */
 $(a_i, a_{i+N}) := (0, R_{2i})$;
Endfor

/* $A = (0, \dots, 0, R_0, \dots, R_{2N-2}, \dots, a_{n-N-1})$ */
Step 2: /* Left-shift N times A */

For ($i = 0$) **to** $(n - 1 - N)$ **ParDo** $a_i := a_{i+N}$;
For ($i = n - N$) **to** $(n - 1)$ **ParDo** $a_i := 0$;

Return A .

/* $A = (R_0, \dots, R_{2N-2}, a_{2N}, \dots, 0, \dots, 0)$ */

THE REDUCE ALGORITHM.

Example 5: (**reduce**) Let $n = 10$ and $N = \lfloor \sqrt{n} \rfloor = 3$.

Let $A = (350, 150, 260, 390, 330, 550, 343, 411, 503, 739)$, with $\max\{A\} < 2^n = 1024$. We only consider the first $6 = 2N$ integers of A , i.e.: $(350, 150, 260, 390, 330, 550)$. We obtain for

$(a_0, a_1) = (350, 150)$, the Bézout matrix $M = \begin{pmatrix} 1 & -2 \\ -3 & 7 \end{pmatrix}$

and $M \times (350, 150) = (R_0, R_1) = (50, 0)$. Similarly $(R_2, R_3) =$

$(130, 0)$, $(R_4, R_5) = (110, 0)$ and we have in turn:

Step 1 yields $A = (0, 0, 0, 50, 130, 110, 343, 411, 503, 739)$.

Step 2 yields $A = (50, 130, 110, 343, 411, 503, 739, 0, 0, 0)$.

So **reduce**($A, 3$) gives rise to 3 zeroes in A .

3.2 The BA-GCD algorithm

The second new algorithm called BA-GCD is based on the **Par-Ext-ILE** transformation [16]. BA stands for Best Approximation. It is similar but simpler than $\Delta 2$ -GCD since it does not use the **pigeonhole** algorithm.

Input: A vector $A = (a_0, a_1, \dots, a_{n-1})$ of n positive integers, $n \geq 4$ and $\max\{a_i\} < 2^n$.

Output: $\gcd(a_0, a_1, \dots, a_{n-1})$.

If $A = (a_0, a_0, \dots, a_0)$ **then Return** a_0 ;

/* A is a constant vector */

$(\alpha, I) := (a_0, 0)$; $p := n$; $N := \lfloor \sqrt{n} \rfloor$;

While $(\alpha > 1)$ **Do**

For ($i = 0$) **to** $(n - 1)$ **ParDo**

If $(0 < a_i \leq 2^n/p)$ **then** $(\alpha, I) := (a_i, i)$;

else $I := -1$;

Endfor

If $(I = -1)$ **then** /* No small a_i */

$R := 0$;

For $(i, j = 0)$ **to** $(N - 1)$ **ParDo**

$x_{ij} := R_{ILE}(a_i, a_j)$;

If $(x_{ij} > 0)$ **then**

{ $(\alpha, I) := (x_{ij}, i)$; $R := 1$; $a_I := x_{ij}$ } ;

/* We can divide all the a_i 's by x_{ij} */

Endif

Endfor

If $(R = 0)$ **then** $A := \text{reduce}(A, N)$;

/* $R = 0$ means $\forall i, j, R_{ILE}(a_i, a_j) = 0$ */

Endif

If $(I \geq 0)$ **then** $A := \text{remainder}(A, \alpha, I)$;

/* We divide all the a_i 's but a_I by $\alpha > 0$ */

If $(\exists a_k \neq 0$ s.t.: $\forall i \neq k \Rightarrow a_i = 0)$ **then**

Return a_k ;

$p := np$;

Endwhile

Return α .

THE BA-GCD ALGORITHM.

4 CORRECTNESS

We prove in the following that all the transformations used in algorithm *BA* or $\Delta 2$ preserve the GCD.

Lemma 4.1: Let n, I be two integers, $n \geq 2$ and $0 \leq I \leq n-1$. Let $A = (a_0, a_1, \dots, a_{n-1})^T$ and $V = (v_0, v_1, \dots, v_{n-1})^T$ be two integral vectors defined by $v_I = a_I$ and $\forall i \neq I$, $v_i = a_i - q_i a_I$ for some integers q_i . Let M be the associated matrix defined by $V = M A$, then $\det(M) = \pm 1$.

Proof: By induction on the size n of the matrix M . \square

Lemma 4.2: Let $n \geq 2$ be an integer. Let $A = (a_0, \dots, a_{n-1})^T$ and $V = (v_0, v_1, \dots, v_{n-1})^T$ be two integral vectors. Let M be a square $n \times n$ matrix with integral entries, such that $V = M A$. If M is unimodular, i.e.: $\det(M) = \pm 1$, then $\gcd(a_0, \dots, a_{n-1}) = \gcd(v_0, \dots, v_{n-1})$.

Proof: Let $d = \gcd(a_0, \dots, a_{n-1})$ and $\delta = \gcd(v_0, \dots, v_{n-1})$. Since each v_i is a linear combination of the a_j 's, then $d \mid v_i$ for all index i , so $d \mid \delta$. The matrix M^{-1} exists and has integral entries because $\det(M) = \pm 1$, so $A = M^{-1} V$. Similarly, each a_i is a linear combination of the v_i 's, so $\delta \mid d$ and $\delta = d$.

Proposition 4.1: (Correctness) Let A_k be the vector obtained at the end of the k -th while loop iteration (in both $\Delta 2$ -GCD and BA-GCD) and let $\gcd(A) = \gcd(a_0, \dots, a_{n-1})$. Then $\forall k \geq 1$, $\gcd(A_k) = \gcd(A)$.

Proof: We consider two cases $\alpha = a_I$ or $\alpha = |a_I - a_J| > 0$.

Case 1: Let $\alpha = a_I$ and the transformation $a'_i = a_i \bmod a_I$ for $i \neq I$ and $a'_I = a_I$. Then, by Lemma 4.1 and Lemma 4.2, we have $\gcd(a'_0, a'_1, \dots, a'_{n-1}) = \gcd(a_0, a_1, \dots, a_{n-1})$. In fact, if $i \neq I$, then $a'_i = a_i \bmod a_I$ and the matrix associated to this transformation has determinant ± 1 .

Case 2: Let $\alpha = |a_I - a_J|$ for some indices I, J with $0 \leq I < J < n$ and let T be the transformation defined by $a'_i = a_i \bmod \alpha$ for $i \neq I$ and $a'_I = \alpha$. We split this transformation T in two parts $T = T_2 \circ T_1$. T_1 is the elementary transformation: $a'_I = \alpha = |a_I - a_J|$ and $a'_i = a_i$, for $i \neq I$. The determinant of the matrix associated to T_1 is obviously ± 1 . The transformation T_2 is similar to that of Case 1, since α becomes one of the component of the vector $A = (a_i)$.

5 COMPLEXITY ANALYSIS

Since BA and $\Delta 2$ algorithms use the same reductions except the **pigeonhole** in BA, it is enough to only analyse the $\Delta 2$ -GCD algorithm. Pre-computed look-up tables perform all the arithmetic operations of two $O(\log n)$ bits integers in $O(1)$ parallel time with $O(n^{1+\epsilon})$ processors, for any $\epsilon > 0$ (see [15, 18] for more details).

5.1 The case of n integers :

The following Lemma shows why the total number of iteration in the while loop in $\Delta 2$ and BA is $O(n/\log n)$ and sum of the bit length of all the quotients involved in both algorithms $\Delta 2$ and BA is $O(n)$.

Lemma 5.1: Let S be total number of iteration of the while loop in $\Delta 2$ and BA, and let k_i be the maximum bit length of the quotient $q_j = \lfloor a_j/\alpha \rfloor$ at the iteration i , with $0 \leq j \leq n-1$, $0 \leq i \leq S$. Then

$$(i) \quad S = O(n/\log n) \quad , \quad (ii) \quad \sum_{i=0}^S k_i = O(n).$$

Proof: We observe that there are three kind of reductions in both algorithms $\Delta 2$ and BA:

- (1) Dividing by α with a small $a_I = \alpha < \max\{A\}/n$.
- (2) using a pigeonhole technique $a_I = \alpha = |a_I - a_j|$, where a_I is not small enough and $\alpha < \max\{A\}/n$.
- (3) using R_{ILE} reduction where $\alpha = x_{ij} = R_{ILE} \neq 0$.

Proof of (i): First we consider the cases (1) and (2). For each iteration i , $0 \leq i \leq S$, $A^{(i)} = (a_0^{(i)}, \dots, a_{n-1}^{(i)})$ denotes the current vector A . We define $\alpha_i = \min\{a_j^{(i)}, |a_p^{(i)} - a_q^{(i)}| > 0\}$, and $\beta_i = \max\{a_j^{(i)}\}$, where $0 \leq j, p, q \leq n-1$.

At iteration $i+1$, all (but one) the integers $a_j^{(i)}$ are divided by α_i , with $j \neq I$, so that $a_j^{(i+1)} = a_j^{(i)} \bmod \alpha_i$ and $a_I^{(i+1)} = \alpha_i$. Now $a_I^{(i+1)}$ becomes the largest integer at iteration $i+1$, i.e.: $\beta_{i+1} = a_I^{(i+1)} = \alpha_i$ and we have $\alpha_i < \beta_i/n$. So $\alpha_{i+1} < \beta_{i+1}/n = \alpha_i/n$ and $\alpha_i < \alpha_0/n^i$. For $i = S$, we obtain $1 \leq \alpha_S < \alpha_0/n^S \leq 2^n/n^S$, $n^S < 2^n$ and $S \log n < n$.

In case Lehmer like R_{ILE} reduction with $(a_I^{(i)}, a_{I+1}^{(i)})$, then $\alpha_i = R_{ILE}(a_I^{(i)}, a_{I+1}^{(i)}) < (2/k) \max\{a_I^{(i)}, a_{I+1}^{(i)}\} < (2/k)\beta_i$. So at iteration $i+1$, we have $a_j^{(i+1)} = a_j^{(i)} \bmod \alpha_i$, for all $j \neq I$ and $\beta_{i+1} = a_I^{(i+1)} = a_I^{(i)} = \alpha_i < (2/k)\beta_i$, hence $\beta_{i+1} < (2/k)\beta_i$ and $\beta_i < (2/k)^i \beta_0$, where k is such that $\log k = \theta(\log n)$. For $i = S$, we obtain $1 \leq \beta_S < (2/k)^S \beta_0 < (2/k)^S 2^n$. Then $(k/2)^S < 2^n$ and $S < n/\log(k/2)$. Since $\log k = \theta(\log n)$ then we obtain $S = O(n/\log n)$.

Proof of (ii): The maximum bit length of the quotient at the iteration i is $k_i = 1 + \lfloor \log \lfloor \beta_i/\alpha_i \rfloor \rfloor \leq 1 + \log(\beta_i/\alpha_i)$.

For all the three cases we have $\beta_{i+1} = \alpha_i$. So for all $i \geq 1$, we have $\beta_i/\alpha_i = \alpha_{i-1}/\alpha_i$ and $k_i \leq 1 + \log(\alpha_{i-1}/\alpha_i)$. Hence $\sum_{i=0}^S k_i \leq S + \log(\beta_0/\alpha_0) + \sum_{i=1}^S \log(\alpha_{i-1}/\alpha_i) < S + \log \beta_0 = O(n/\log n) + O(n) = O(n)$. \square

Proposition 5.1: (Complexity analysis of remainder)

The computation of all the quotients $\lfloor a_i/a_I \rfloor$ during the whole $\Delta 2$ -GCD algorithm costs $O(n/\log n)$ time with $O(n^{2+\epsilon})$ processors in CRCW PRAM model, for any $\epsilon > 0$.

Proof: We consider the worst case (maximum of divisions) and recall that the algorithm terminates after $O(n/\log n)$ iterations of the while loop (Lemma 5.1).

Let t_i be the time cost at iteration i , $1 \leq i \leq S$, with $S = O(n/\log n)$. So the total parallel time is $t(n) = \sum_{i=1}^S t_i$. Arithmetic operations with $O(\log n)$ bits can be done in $O(1)$ parallel time with table look-up [15, 18].

Let k_i be the maximum bit length of all the quotients $q_j = \lfloor \frac{a_j}{\alpha} \rfloor$ at iteration i , with $\sum_{i=1}^S k_i \leq n$. Then $t_i = O(\min\{k_i/\log n, \log n\})$. In fact, if $k_i \leq \log n$ then $t_i = O(1)$ parallel time with table look-up. If $k_i > \log^2 n$, then a division between two n bits integers costs $t_i = O(\log n)$.

Otherwise, if $\log n < k_i < \log^2 n$, then each quotient q_i has roughly $k_i / \log n$ digits of $O(\log n)$ bits, so $t_i = O(\frac{k_i}{\log n})$. The total number of processors is $n \times O(n^{1+\epsilon}) = O(n^{2+\epsilon})$ and the parallel time is, up to a constant (recall that $\sum_{i=1}^S k_i \leq n$)

$$t(n) = \sum_{i=1}^S \min\left\{\frac{k_i}{\log n}, \log n\right\} = \sum_{k_i < \log n} 1 + \sum_{\log n < k_i < \log^2 n} \frac{k_i}{\log n} + \sum_{k_i > \log^2 n} \log n = A + B + C.$$

We have $A \leq \sum_{i=1}^S 1 = S = O(\frac{n}{\log n})$ and

$$B \leq \sum_{k_i < \log^2 n} \frac{k_i}{\log n} = \frac{1}{\log n} \sum_{i=1}^S k_i \leq \frac{n}{\log n}.$$

$C = \sum_{k_i > \log^2 n} \log n = \log n \sum_{k_i > \log^2 n} 1$. Let P be the number of all the k_i 's satisfying $k_i > \log^2 n$. Then $P \log^2 n \leq k_1 + k_2 + \dots + k_P < n$, so $P < \frac{n}{\log^2 n}$, hence $C \leq \frac{n}{\log n}$ and $t(n) = O(\frac{n}{\log n})$. \square

Theorem 5.1: The $\Delta 2$ -GCD algorithm computes in parallel the GCD of n integers of $O(n)$ bits in length, in $O(n/\log n)$ time using $O(n^{2+\epsilon})$ processors in CRCW PRAM model, for any $\epsilon > 0$.

Proof: The algorithm is correct thanks to Proposition 4.1. Testing if there are any small a_i , i.e.: $0 < a_i \leq 2^n/p$ can be done easily in $O(1)$ parallel time with $O(n \log n)$ processors.

- Cost of the **pigeonhole** algorithm: We consider the $O(\sqrt{n})$ first terms a_i of A . There are only $O(n)$ pairs to compare and each comparison costs $O(1)$ parallel time with $O(n)$ processors, so the total of all the comparisons $t_i = t_j$ and $a_i \neq a_j$ cost $O(1)$ parallel time with $O(n^2)$ processors.

- Cost of the **reduce** algorithm: Thanks to look-up tables, the computation of all $R_{ILE}(a_i, a_j)$ and **Par-Ext-ILE** cost $O(1)$ parallel time with $O(n^{1+\epsilon})$ processors (see [16] for more details). The cost of left-shift N times the vector A is similar. There are $O(N^2) = O(n)$ pairs (a_{2i}, a_{2i+1}) considered. On the other hand **reduce** can be called at most $O(\sqrt{n})$ times since, at each step, we add $O(\sqrt{n})$ new zeros in the vector A . So the total cost of the **reduce** algorithm is $O(\sqrt{n})$ parallel time with $O(n^{2+\epsilon})$ processors. Note that the case where **reduce** is called $O(\sqrt{n})$ times is obvious. It will remain $O(1)$ non zero integers and their GCD can be achieved in $O(n/\log n)$ time with $O(n^{1+\epsilon})$ processors.

- Cost of the **remainder** algorithm for the whole $\Delta 2$ -GCD algorithm: By Proposition 5.1, the total complexity of **remainder** algorithm is $O(n/\log n)$ time with $O(n^{2+\epsilon})$ processors. \square

5.2 The case of m integers :

We have similar results for the GCD computation of m integers of $O(n)$ bits with some adjustments.

Proposition 5.2: If we consider the case of m integers, then the computation of all the quotients $\lfloor a_i/a_I \rfloor$ during the whole $\Delta 2$ -GCD algorithm costs $O(n/\log n)$ parallel time with $O(m n^{1+\epsilon})$ processors in CRCW PRAM model, for any $\epsilon > 0$ and m s.t.: $2 \leq m \leq n^{3/2}/\log n$.

Proof: First, if m is a constant w.r.t. n , the result is obvious. Moreover, if $m < \sqrt{n}$, then just add zeros to A to reach $O(\sqrt{n})$ integers, i.e.: set $a_{m+1} = \dots = a_{N-1} = 0$ with

$N = \lfloor \sqrt{n} \rfloor$. This is necessary because **pigeonhole**, R_{ILE} and **reduce** need N integers. So WLOG we assume $m \geq \sqrt{n}$.

The proof is similar to the proof of Proposition 5.1 with some adjustments. We consider the worst case (maximum of divisions) and assume that, at each iteration, there exists $\alpha > 0$ so that the size of all the integers in A are reduced by $O(\log n)$ bits. The only difference is the number of calls of **reduce**. Since **reduce** adds $O(\sqrt{n})$ zeroes in A and A has initially m integers, so the number of calls is at most $O(m/\sqrt{n})$. Thanks to the upper bound of m , the number of calls of **reduce** as well as the number of iterations of the while loop in $\Delta 2$ -GCD are both bounded by $O(n/\log n)$. In fact $m/\sqrt{n} \leq n^{3/2}/(\sqrt{n} \log n) = n/\log n$. If t_i is the time cost at iteration i , with $1 \leq i \leq S$, then $S = O(n/\log n)$. The remainder of the proof is the same as in Proposition 5.1. Consequently we derive the following result.

Theorem 5.2: The $\Delta 2$ -GCD and BA algorithms compute in parallel the GCD of m integers of $O(n)$ bits in length, in $O(n/\log n)$ time using $O(m n^{1+\epsilon})$ processors in CRCW PRAM model, for any $\epsilon > 0$ and m , such that: $2 \leq m \leq n^{3/2}/\log n$.

Proof: Straightforward from Proposition 4.1, Proposition 5.2 and the proof of Theorem 5.1 since the cost of **pigeonhole**, R_{ILE} and **reduce** are quite similar in both cases. \square

5.3 Some comments about BA and $\Delta 2$

As mentioned in the Introduction section, the main drawback of the pigeonhole technique is that it becomes inefficient when there are only a few distinct integers. So we have to combine it with another technique. We have chosen the Lehmer like reduction R_{ILE} . On the other hand the Lehmer like reduction is efficient and a fast parallel integer GCD can be designed without the pigeonhole technique. So, the natural question is why keep both algorithms rather than presenting only the algorithm based on the Lehmer like reduction R_{ILE} ? The reason is that the pigeonhole technique is a new technique. It is suitable for parallel algorithms when there are enough distinct integers, say $O(n^c)$, $c > 0$. It is worthwhile in itself so that it must be studied even with its drawbacks.

6 APPLICATIONS

We suggest below two applications of $\Delta 2$ (or BA) algorithm, namely an extended GCD and an algorithm for solving linear Diophantine equations.

6.1 Extended GCD algorithm

With the same idea, a straightforward Blankinship like algorithm [1] can be designed to compute an extended GCD of n integers. We just add the identity matrix I_n to the first row A , i.e.: consider the $(n+1) \times n$ matrix $\begin{pmatrix} A \\ I_n \end{pmatrix}$. The only difference with Blankinship algorithm is that, at each step, we divide by α instead of the smallest non-zero integer of the first row. The algorithm is the following:

Input: A vector $A = (a_0, a_1, \dots, a_{n-1})$ of n integers of $O(n)$ bits, $n \geq 4$ and $\max\{a_i\} < 2^n$.

Output: n integers x_0, x_1, \dots, x_{n-1} such that

$$\sum_{i=0}^{n-1} x_i a_i = \gcd(A).$$

$\alpha := a_0$; $I := 0$;
 Let $V_i = (a_i, 0, \dots, 1, \dots, 0)^T$ for $i \neq I$ and
 $V_I = (\alpha, 0, \dots, 1, \dots, 0)^T$;

While ($\exists j \neq I$ s.t.: $a_j \neq 0$) **Do**
 Compute I and α (as in Δ 2-GCD algorithm);
If ($I \geq 0$) **then** /* $\alpha > 0$ */
 $V_I := V_I - V_J = (\alpha, \dots, a_i, \dots)^T$;
For ($i = 0$) **to** ($n - 1$) **ParDo**
If ($i \neq I$) **then**
 $q_i := \lfloor a_i / \alpha \rfloor$; $V_i := V_i - q_i V_I$;
Endif
Endfor
Else reduce the columns V_i instead of a_i with
 $0 \leq i \leq N$;
Endwhile
Return $V_I = (\gcd(A), x_0, x_1, \dots, x_{n-1})^T$.

EXTENDED GCD ALGORITHM.

With the example 1, where

$A = (912672, 815430, 721161, 565701, 662592)$,
 we obtain the coefficient vector $(-2, 6, -83, 2, 84)$, i.e.:
 $-2a_0 + 6a_1 - 83a_2 + 2a_3 + 84a_4 = 3 = \gcd(A)$.

6.2 Solving linear Diophantine equation

We can design a parallel algorithm similar to Rosser's algorithm [11, 14] to solve linear Diophantine equation: Given $(n+1)$ integers $a_0, a_1, \dots, a_{n-1}, b$, find n integers x_0, x_1, \dots, x_{n-1} , such that $a_0 x_0 + a_1 x_1 + \dots + a_{n-1} x_{n-1} = b$. Here we add the identity matrix I_{n+1} to the row $A' = (a_0, \dots, a_{n-1}, -b)$. Then, run the extended GCD algorithm described above.

7 CONCLUSION

This paper confirms an early result of Cooperman et al. [6] showing that the GCD computation of many integers does not cost much more than the GCD computation of two integers with a good probability. It also generalizes the parallel performance of computing the GCD of two integers ([5, 15, 18]) to the case of many integers. In fact we propose two algorithms for computing the GCD of n integers of $O(n)$ bits in $O(n/\log n)$ parallel time with $O(n^{2+\epsilon})$ processors, for any $\epsilon > 0$ in CRCW PRAM model, in the worst case. More generally, the parallel time for computing the GCD of m integers of $O(n)$ bits can be achieved in $O(n/\log n)$ parallel time with $O(mn^{1+\epsilon})$ processors, i.e. the parallel time does not depend on the number m of integers considered satisfying $2 \leq m \leq n^{3/2}/\log n$. We suggest an extended GCD version for many integers as well as an algorithm to solve linear Diophantine equations. To our knowledge, it is the first time that we find deterministic algorithms which compute the GCD of many integers with this parallel performance and polynomial work.

REFERENCES

- [1] W. A. Blankinship, A new version of the Euclidean algorithm, *Amer. Math. Monthly*, 70 (1963), 742-745
- [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] G.H. Bradley, Algorithm and Bound for the Greatest Common Divisor of n Integers, *Communications of the ACM*, 13, 7, 1970, 433-436
- [4] V. Brun, Algorithmes Euclidiens pour trois et quatre nombres, *13th Congr. Math. Scand. Helsinki, S.* (1957) 45-64.
- [5] B. Chor and O. Goldreich, An improved parallel algorithm for integer GCD, *Algorithmica*, 5, 1990, 1-10
- [6] G. Cooperman, S. Feisel, J. von zur Gathen and G. Havas, GCD of many integers, *Computing and Combinatorics, Lect. Notes in Comp. Sciences, Springer-Verlag, Berlin*, 1627 (1999), 310-317
- [7] G. Havas, S. Majewski, Hermite normal form computation for integer matrices, *Technical Report TR0295, the university of Queensland, Brisbane, (1998)*
- [8] G. Havas, S. Majewski, Extended gcd calculation, *Congressus Numerantium*, 111, 1627 (1998), 104-114
- [9] C.S. Iliopoulos, Worst case complexity bounds on algorithms for computing the canonical structure of finite abelian groups and the Hermite and Smith normal forms of an integer matrix, *SIAM J. Computing*, 18 (1989), 658-669
- [10] R. M. Karp, V. Ramachandran, Parallel algorithms for shared-memory machines, *Handbook of Theoretical Computer Science (Vol. A): Algorithms and Complexity*, J. van Leeuwen, Ed., New York, Elsevier (1991), 869-941
- [11] M. Khorramizadeh and N. Mahdavi-Amiri, On solving linear Diophantine systems using generalized Rosser's algorithm, *Bulletin of the Iranian Mathematical Society*, 34, 2, (2008), 1-25
- [12] H. Poincaré, Sur une généralisation des fractions continues, *in C. R. Acad. Sci. Paris Sér. A* 99, (1884) 1014-1016.
- [13] L. Potier, The Euclidean Algorithm in Dimension n , *in Proc. of the International Symposium on Symbolic and Algebraic Computation, ISSAC'96*, (1996) 40-42
- [14] J.B. Rosser, A note on the linear Diophantine equation, *Amer. Math. Monthly*, 48 (1941), 662-666
- [15] S.M. Sedjelmaci, On a Parallel Lehmer-Euclid GCD Algorithm, *in Proc. of the International Symposium on Symbolic and Algebraic Computation (ISSAC'2001)*, 2001, 303-308
- [16] S.M. Sedjelmaci, A Parallel Extended GCD Algorithm, *Journal of Discrete Algorithms*, 6, (2008) 526-538
- [17] S.M. Sedjelmaci, Fast Parallel GCD of many integers, *Poster in the International Symposium on Symbolic and Algebraic Computation (ISSAC'2013), Boston, USA and ACM Communications in Computer Algebra*, Vol. 47, No. 3, Issue 185, (2013), 92-93
- [18] J. Sorenson, Two Fast GCD Algorithms, *J. of Algorithms*, 16, 1994, 110-144
- [19] J. Sorenson, A Randomized Sublinear Time Parallel GCD Algorithm for the EREW PRAM, *Information Processing Letters*, 110, 2010, 198-201
- [20] M.S. Waterman, Multidimensional greatest common divisor and Lehmer algorithm, *BIT*, 17, 1977, 465-478