Two fast parallel GCD algorithms of many integers

SEDJELMACI Sidi Mohamed
LIPN Laboratory, CNRS UMR 7030
University of Paris-Nord.
Av. J.-B. Clément, 93430, Villetaneuse, France
sms@lipn.univ-paris13.fr
9 October 2015

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.

Index Terms—GCD of many integers; Parallel algorithms; Parallel Complexity of GCD; Complexity analysis;

I. INTRODUCTION

The computation of the GCD of two integers is not known to be in the NC class, nor is it 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 the author [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]).

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 obtained the same parallel time complexity as for computing the GCD of two integers, with probability \( > 1/2 \). 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 conserve 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.

A first version of a parallel algorithm with this parallel performance was recently presented as a poster in [17]. The main idea was to use under some conditions the pigeonhole principle on the \( n \) input integers of the vector \( A = (a_0, \ldots, a_{n-1}) \).

There exists two integers \( (a_i, a_j) \) that match with their \( O(\log n) \) most significant bits. Their difference \( \alpha = |a_i-a_j| \) is in fact \( O(\log n) \) bits less w.r.t. the larger size of input integers \( a_i \)'s. Then we divide all the other components of \( A \) by \( \alpha \) to obtain a new vector \( A' \) where all its components are \( O(\log n) \) bits less. The algorithm just iterates this process.

However, there are some drawbacks to this first version since 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 with their 3 most significant bits, namely \( (255,255) \) and \( (65,65) \). Unfortunately, in both cases \( \alpha = 0 \). This is important since we must divide by \( \alpha \), so how to do it if no such an \( \alpha > 0 \) 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 in a first algorithm by adding a new transformation to the early version 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 \( O(n) \) bits. The general case of \( m \) integers of \( O(n) \) bits is similar. It is addressed in Section V. We recall the \( \Delta \)-GCD algorithm presented in [17] in Section II. A modified version of \( \Delta \)-GCD algorithm as well as a new parallel GCD algorithm are presented in Section III. Section IV is devoted to the correctness of our new algorithms. Section V 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 VI. We conclude with some remarks in Section VII.

II. THE \( \Delta \)-GCD ALGORITHM

A. Notations

Throughout the paper, \( A \) is a vector of \( n \) integers \( A = (a_0, a_1, \cdots, a_{n-1}) \), with \( a_i \geq 0 \), \( n \geq 4 \) (unless in Section V-B where \( A \) may have \( m \) integers of \( O(n) \) bits). We use in Section III an integer parameter \( k \) satisfying \( k = O(\log n) \).

We note \( \gcd(A) = \gcd(a_0, a_1, \cdots, a_{n-1}) \) and \( t_i \) is the integer formed by the \( \log(n) \) most significant bits of \( a_i \), i.e.: \( t_i = a_i \div \log n \). We use \( \gcd(0,0) = 0 \).

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).

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 \( O(n) \) bit integers can be achieved in \( O(1) \) parallel time with \( O(n^{1+\epsilon}) \) processors (see [18] for more details).

B. Basic results

The \( \Delta \)-GCD algorithm is based on the following results which are variant forms of the pigeonhole principle:

**Lemma 1:** Let \( A = \{ a_1, a_2, \cdots, a_n \} \) be a set of \( n \) distinct positive integers, such that \( n \geq 2 \) and \( a_n/n < a_1 < a_2 < \cdots < a_n \). Then

\[
\exists i \in \{ 1, 2, \cdots, n-1 \} \text{ s.t. } a_{i+1} - a_i < \frac{a_n}{n}.
\]

**Proof:**

By contradiction. If we assume that \( \forall i, a_{i+1} - a_i \geq \frac{a_n}{n} \), then

\[
a_n - a_1 = (a_n - a_{n-1}) + \cdots + (a_2 - a_1) \geq (n-1) \frac{a_n}{n} = a_n - \frac{a_n}{n}.
\]

So \( a_1 \leq \frac{a_n}{n} \), a contradiction with the assumption \( a_1 > \frac{a_n}{n} \), hence the result.

A straightforward consequence is the following:

**Corollary 1:** Let \( A = \{ a_1, a_2, \cdots, a_n \} \) be a set of \( n \) distinct positive integers, with \( n \geq 2 \), then \( (1 \leq k, i \neq j \leq n) \)

\[
\min \{ a_k \mid a_i - a_j > 0 \} \leq \frac{\max \{ a_i \}}{n}.
\]

We recall below the \( \Delta \)-GCD algorithm presented in [17]:

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

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

If \( A = (a_0, a_1, \cdots, a_0) \) then Return \( a_0 \);

\[ \text{// } A \text{ is a constant vector} \]

\[ \alpha := a_0; I := 0; p := n; \]

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

\[ \text{For } (i = 0) \text{ to } (n-1) \text{ ParDo} \]
\[ \text{If } (0 < a_i \leq 2^n/p) \text{ then } \{ \alpha := a_i; I := i; \}
\]

Endfor

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

\[ \text{// } \text{Compute in parallel } I, J \text{ and } \alpha */\]
\[ \alpha := \min \{ \{ a_i - a_j > 0 \} = a_I - a_J; a_I := \alpha; \]

Endif

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

\[ p := np; \text{// } p \text{ is } O(\log n) \text{ bits larger */} \]

Endwhile

Return \( \alpha \).

**The \( \Delta \)-GCD Algorithm.**

**Remarks:**

1) If the result \( \alpha \) 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].

3) The property \( (\exists i, j, \text{s.t. } a_i \neq a_j) \) is always valid for any non constant input vector \( A \).

The following example shows how it works:

**Example 1:**

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

\[
\begin{pmatrix}
912672 & 34137 & 4443 \\
815430 & 54033 & 717 \\
721161 & 58569 & 810 \\
565701 & 38580 & 3036 \\
662592 & 18333 & 561 \\
\alpha & 58569 & 93 \\
(I, J) = (2, 4) & (0, 3) & (1, 2)
\end{pmatrix}
\]
III. TWO NEW PARALLEL GCD ALGORITHMS

A. The \( \Delta 2 \)-GCD algorithm

In [17] we have considered the \( O(\log n) \) most significant bits for computing \( \alpha \). 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 \( \alpha \) 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 \( \Delta 2 \)-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 \( \alpha \) ?
- **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 \textit{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 the transformation \textit{reduce} will guarantee the termination and the parallel performance \( O(n/\log n) \) time with \( O(n^{2+\epsilon}) \) processors (see Sections IV and V). We derive a new algorithm called \( \Delta 2 \)-GCD which corrects the \( \Delta \)-GCD algorithm [17] described in Section II.

### Table I

<table>
<thead>
<tr>
<th>Authors</th>
<th>Time</th>
<th>Nb. of proc.</th>
<th>Parallel model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brent-Kung (1983)</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
<td>Systolic array</td>
</tr>
<tr>
<td>Purdy (1983)</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
<td>Systolic array</td>
</tr>
<tr>
<td>Kannan et al. (1987)</td>
<td>( O(2 \log \log n) )</td>
<td>( O(n^{2+\epsilon}) )</td>
<td>CRCW PRAM</td>
</tr>
<tr>
<td>Adleman et al. (random, 1988)</td>
<td>( O(\log^2 n) )</td>
<td>( e^{O(\sqrt{n \log n})} )</td>
<td>CRCW PRAM</td>
</tr>
<tr>
<td>Chor-Goldreich (1990)</td>
<td>( O(n/\log n) )</td>
<td>( O(n^{2+\epsilon}) )</td>
<td>CRCW PRAM</td>
</tr>
<tr>
<td>Sorensen (1994)</td>
<td>( O(n/\log n) )</td>
<td>( O(n^{2+\epsilon}) )</td>
<td>CRCW PRAM</td>
</tr>
<tr>
<td>Sedeljmaci (2001)</td>
<td>( O(n/\log n) )</td>
<td>( O(n^{2+\epsilon}) )</td>
<td>CRCW PRAM</td>
</tr>
<tr>
<td>Sorensen (random, 2010)</td>
<td>( O(2 \log \log n / \log n) )</td>
<td>( O(n^{2+\epsilon}) )</td>
<td>EREW PRAM</td>
</tr>
</tbody>
</table>

### Input:

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

### Output:

- \( \gcd(a_0, a_1, \cdots, a_{n-1}) \).

If \( A = (a_0, a_0, \cdots, a_0) \) then Return \( a_0 \);

/* \( A \) is a constant vector */

(\( \alpha, I \) := (0, 0) ; \( p := n \) ; \( N := \sqrt{n} \); 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 \(A := \text{divide}(A, \alpha, I)\);
/* We divide all the \(a_i\)’s but \(a_I\) by \(\alpha\) */
If \((\exists a_k \neq 0 \text{ s.t.: } \forall i \neq k \Rightarrow a_i = 0)\) 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 \(\text{divide}(A, \alpha, I), \text{pigeonhole}(A, N), R_{ILE}, \text{Par-Ext-ILE}\) and \(\text{reduce}(A, N)\) are described below.

The divide procedure just divides all the components of \(A\) by \(\alpha\) and consider their remainders. It proceeds as follows:

Input: \(A = (a_0, \cdots, a_{n-1})\), with \(n \geq 4\), \(0 \leq I \leq n - 1\), and \(\alpha > 0\).
Output: \(A' = (a'_0, \cdots, a'_n)\), s.t.: \(a'_i = a_i \mod \alpha\) for all \(i \neq I\) and \(a'_I = a_I\).
For \((i = 0)\) to \((n - 1)\) ParDo
If \((i \neq I)\) then \(a_i := a_i \mod \alpha\);
Endfor
Return \(A\).

THE DIVIDE ALGORITHM.

The pigeonhole algorithm is based on Corollary 1 with the first \(O(\sqrt{n})\) integers of \(A\), namely \((a_0, a_1, \cdots, a_{N-1})\), with \(N = \lfloor \sqrt{n} \rfloor\). The algorithm returns a pair \((\alpha, I)\) 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: \(A = (a_0, \cdots, a_{n-1})\), \(N = \lfloor \sqrt{n} \rfloor\), \(n \geq 4\).
/* Actually only the subset \(B = (a_0, \cdots, a_{N-1})\) is considered. */
Output: \((\alpha, I)\), s.t.: \(0 < \alpha = |a_i - a_j| \leq \max \{B\}/N\); where \(B = (a_0, \cdots, a_{N-1})\).
\(I = i\), 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 \(0 \leq i \leq N - 1\);
/* \(t_i = a_i \div 2^{\log N}/\) */
/* where \(s = [\log \max \{B\}] + 1 \) */
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 consider \(N = 4\) (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 \(\text{Par-Ext-ILE}\) algorithms are described in [16], p.530 and p.532 respectively. ILE stands for Improved Lehmer Euclid and \(\text{Par-Ext-ILE}\) stands for a parallelization of an extended version of ILE. We just have added some special cases 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 \(\text{Par-Ext-ILE}\).

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 \(v\);
\(\lambda := 2m + n - p + 2;\)
\(u_i := \lfloor u/2^{\lambda} \rfloor;\) \(v_i := \lfloor u/2^{\lambda} \rfloor;\)
Step 2: For \((i = 0)\) to \(k\) ParDo
\(q_i := \lfloor u u_i/v_i \rfloor;\) \(r_i := u u_i - q_i v_i;\)
If \((r_i < v_i/k)\) then \((s,t) := (i, -q_i);\)
If \((v_i - r_i < v_i/k)\) 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.
If many processors are in write concurrency in Step 2 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.

**Example 3:** ($R_{ILE}$)

Let $u = 1,759,291$ and $v = 1,349,639$. Their binary representations are respectively:

$$1101011010100001110111 = 1,759,291$$
$$10100100101000000111 = 1,349,639$$

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 $q$, $r$, $s$ and $t$ ($r = su + tv$).

<table>
<thead>
<tr>
<th>$q$</th>
<th>$r$</th>
<th>$s$</th>
<th>$t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>214</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>164</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>50</td>
<td>1</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>14</td>
<td>-3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>10</td>
<td>-13</td>
</tr>
</tbody>
</table>

In our example, we obtain $s = -3$, $t = 4$, $r = 14 < v_1/k = 164/8 = 20.50$ and

$$R_{ILE} = | -3u + 4v | = 120,683 < 2v/k.$$

In [16], p. 530, we prove 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], pp. 532-533 for more details).

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

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

Unlike the pigeonhole principle, the transformation reduce will guarantee the termination and the parallel performance of the $\Delta 2$-GCD algorithm (see Section IV and V). 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 vector $A = (a_0, a_1, \ldots, a_{n-1})$ of $n$ positive integers, $n \geq 4$ and $\max \{a_i\} < 2^n$.

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

If $A = (a_0, a_0, \ldots, a_0)$ 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

**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.:

$$(a_0, a_1, \ldots, a_5) = (350, 150, 260, 390, 330, 550).$$

We obtain for $(a_0, a_1) = (350, 150)$, the Bézout matrix $M = \left( \begin{array}{cc} 1 & -2 \\ -3 & 7 \end{array} \right)$ 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 $\text{reduce}(A, 3)$ gives rise to 3 zeroes in $A$.

**B. The BA-GCD algorithm**

The second new algorithm called BA-GCD is based on the Par-Ext-ILE transformation ([16], p. 530 and p. 532). 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 $A = (a_0, a_1, \ldots, a_{n-1})$ of $n$ positive integers, $n \geq 4$ and max $\{a_i\} < 2^n$.

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

If $A = (a_0, a_0, \ldots, a_0)$ 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 \(a_i\) */
\[ R := 0; \]
\[ \text{For } (i, j = 0) \text{ to } (N - 1) \text{ ParDo} \]
\[ x_{ij} := R_{\text{LFE}}(a_i, a_j); \]
\[ \text{If } (x_{ij} > 0) \text{ then} \]
\[ \{ (\alpha, I) := (x_{ij}, i); R := 1; a_{ji} := x_{ij} \}; \]
/* We can divide all the \(a_i\)’s by \(x_{ij}\) */
Endif
Endfor
\[ \text{If } (R = 0) \text{ then } A := \text{reduce}(A, N); \]
/* \(R = 0\) means \(\forall i, j, R_{\text{LFE}}(a_i, a_j) = 0\) */
Endif
\[ \text{If } (I \geq 0) \text{ then } A := \text{divide}(A, \alpha, I); \]
/* We divide all the \(a_i\)’s but \(a_I\) by \(\alpha > 0\) */
\[ \text{If } (3a_k \neq 0 \text{ s.t. } \forall i \neq k \Rightarrow a_i = 0) \] then
\[ a_k := np; \]
\[ \] Endwhile
Return \(\alpha\).

**THE BA-GCD ALGORITHM.**

**IV. 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 \(A = (a_0, a_1, \ldots, a_{n-1})^t\) and \(V = (v_0, v_1, \ldots, v_{n-1})^t\) be two integer vectors defined by \(v_I = a_I\) and \(\forall i \neq I, v_i = a_i - a_I a_{ji}\) for some integers \(q_I\). Let \(M\) be the associated matrix defined by \(V = M A\), then \(\det(M) = 1\).

**Proof:** By induction on the size \(n\) of the matrix \(M\).

**Lemma 4.2:** Let \(n \geq 2\) be an integer and let \(A = (a_0, a_1, \ldots, a_{n-1})^t\) and \(V = (v_0, v_1, \ldots, v_{n-1})^t\) be two integer vectors. Let \(M\) be a square \(n \times n\) matrix with integral entries, such that \(V = M A\). If \(M\) is unimodular, i.e., \(\det(M) = 1\), then \(\gcd(a_0, \ldots, a_{n-1}) = \gcd(v_0, \ldots, v_{n-1})\).

**Proof:**
Let \(d = \gcd(a_0, \ldots, a_{n-1})\) and \(\delta = \gcd(v_0, \ldots, v_{n-1})\). Since each \(v_i\) is a linear combination of the \(a_j\)’s, then \(d | v_i\) for all index \(i\), so \(d | \delta\). 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 \(d | d\) 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 \(\gcd(A) = \gcd(a_0, \ldots, a_{n-1})\). 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 \(a'_I = a_i \mod a_I\) for \(i \neq I\) and \(a'_I = a_I\). Then, by Lemma 4.1 and Lemma 4.2, we have \(\gcd(a'_0, a'_1, \ldots, a'_{n-1}) = \gcd(a_0, a_1, \ldots, a_{n-1})\). In fact, if \(i \neq I\), then \(a'_i = a_i \mod a_I\) and the matrix associated to this transformation has determinant \(\pm 1\).

**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 \(a'_i = a_i \mod \alpha\) 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 \(\pm 1\).

The transformation \(T_2\) is similar to the transformation described in Case 1, since \(\alpha\) becomes one of the component of the vector \(A = (a_i)\).

**V. COMPLEXITY ANALYSIS**
Since \(\Delta 2\)-GCD and BA algorithms are similar, 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 [18], [16] for more details).

**A. The case of \(n\) integers:**

**Proposition 5.1:** (Complexity analysis of divide)
The computation of all the quotients \([a_i/a_I]\) 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 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. Therefore the algorithm terminates after \(O(n/\log n)\) iterations of the while loop.

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 \(q_j = [a_j/a_i]\) at iteration \(i\), with \(\sum_{i=1}^{S} k_i \leq n\). Then
\[ t_i = O(\min \{ \frac{k_i}{\log n}, \log n \}) . \]

In fact, if \(k_i \leq \log n\) then \(t_i = O(1)\) parallel time with table look-up. If \(k_i > \log n\) 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 \(\sum_{i=1}^{S} k_i \leq n\))
\[ t(n) = \sum_{i=1}^{S} \min \{ \frac{k_i}{\log n}, \log n \} \leq \sum_{i=1}^{S} \frac{k_i}{\log n} + \sum_{k_i > \log^2 n} \log n = A + B + C. \]

We have \(A \leq \sum_{i=1}^{S} 1 = S = O(\frac{n}{\log n})\) and
\[ B \leq \sum_{k_i < \log^2 n} \frac{k_i}{\log n} \leq \frac{1}{\log n} \sum_{i=1}^{S} k_i \leq \frac{n}{\log n}, \]

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

**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 $\text{Par-Ext-Ile}$ 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 $O(n^{1+\epsilon})$ 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 divide algorithm for the whole $\Delta 2$-GCD algorithm:** By Proposition 5.1, the total complexity of divide algorithm is $O(n/\log n)$ time with $O(n^{2+\epsilon})$ processors.

**B. 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 $|a_i/a_j|$ 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 $a_{m+1} = \cdots = a_{N-1} = 0$ 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 $O(\sqrt{n})$ zeros 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

$$\frac{m}{\sqrt{n}} \leq n^{3/2}/n \log n = \frac{n}{\log n}.$$ 

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 the proof of 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.

**VI. APPLICATIONS**

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

**A. 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 $A = (a_0, a_1, \cdots, a_{n-1})$.

$$\begin{pmatrix} a_0 & a_1 & \cdots & a_{n-1} \\ 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix}$$

The only difference is that, at each step, we divide by $\alpha$ instead of the smallest non-zero of the first row and just report the same transformations to the columns instead of the $a_i$’s. The algorithm is the following:

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

**Output:** $n$ integers $x_0, x_1, \cdots, x_{n-1}$ such that $\sum_{i=0}^{n-1} x_i a_i = \gcd(A)$.

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

Let $V_I = (a_i, 0, \cdots, 1, \cdots, 0)$ for $i \neq I$ and $V_I = (\alpha, 0, \cdots, 1, \cdots, 0)$;

While ($\exists j \neq I$ s.t.: $a_j \neq 0$) Do

Compute $I$ and $\alpha$ (as in $\Delta 2$-GCD algorithm);

If ($I \geq 0$) then $I \leftarrow I - 1$;

$V_I := V_I - V_I = (\alpha, \cdots)$;

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

If ($i \neq I$) then
\[
q_i := \lfloor a_i / \alpha \rfloor; \quad V_i := V_i - q_i V_f;
\]

**Endfor**

*Else* reduce the columns \(V_i\) instead of \(a_i\) with 
\[0 \leq i \leq N;\]

**Endwhile**

Return \(V_f = (\gcd(A), x_0, x_1, \cdots, x_{n-1})^t\).

**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.:
\[-2a_0 + 6a_1 - 83a_2 + 2a_3 + 84a_4 = 3 = \gcd(A).\]

**B. Solving linear Diophantine equation**

Similarly, we can improve (at least theoretically) the algorithm of Rosser \([14], [11]\) to solve linear Diophantine equation: Given \((n + 1)\) integers \(a_0, a_1, \cdots, a_{n-1}, b\), find \(n\) integers \(x_0, x_1, \cdots, x_{n-1}\), such that 
\[a_0 x_0 + a_1 x_1 + \cdots + a_{n-1} x_{n-1} = b.\]

We also add the identity matrix \(I_{n+1}\) to first row 
\[(a_0, a_1, \cdots, a_{n-1}, -b),\]
\[
\begin{pmatrix}
a_0 & a_1 & \cdots & a_{n-1} & -b \\
1 & 0 & \cdots & 0 & 0 \\
0 & 1 & \cdots & 0 & 0 \\
\vdots & \vdots & \ddots & 1 & 0 \\
0 & 0 & \cdots & 0 & 1
\end{pmatrix}
\]

Then, instead of division by the smallest non-zero integer of the first row, we consider division by \(\alpha\) and just report the same transformations to the columns instead of the \(a_i\)’s.

**VII. 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 \([5], [18], [15]\) 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 mode, 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 \(O(m n^{1+\epsilon})\) 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.

Some remarks are given below:

- Based on the same ideas, one may consider a fast sequential GCD algorithm for \(n\) integers of \(O(n)\) bits in \(O(n^2/\log n)\).
- An improvement can be done by dividing systematically \(\alpha\) by all primes of \(O(\log n)\) bits as described in a recent paper of Sorenson \([19]\).
- Similarly, a Least Significant Bit first approach (LSB) may be considered. However the extended GCD version becomes more difficult.
- An open question: Find a low norm coefficient vector for the extended GCD described in Section VI-A and compare it with other Extended GCD algorithms for many integers like LLL-based GCD for example.

**REFERENCES**


