A New Modular Division Algorithm and Applications

M.S. Sedjelmaci and C. Lavault

LIPN CNRS UMR 7030,
Université Paris-Nord
Av. J.-B. Clment, 93430 Villetaneuse, France.
E-mail: sms@lipn.univ-paris13.fr

Abstract

The present paper proposes a new parallel algorithm for the modular division $u/v \bmod \beta^s$, where u, v, β and s are positive integers ($\beta \geq 2$). The algorithm combines the classical add-and-shift multiplication scheme with a new propagation carry technique. This "Pen and Paper Inverse" (*PPI*) algorithm, is better suited for systolic parallelization in a "least-significant digit first" pipelined manner. Although it is equivalent to Jebelean's modular division algorithm [5] in terms of performance (time complexity, work, efficiency), the linear parallelization of the *PPI* algorithm improves on the latter when the input size is large. The parallelized versions of the *PPI* algorithm leads to various applications, such as the exact division and the digit modulus operation (dmod) of two long integers. It is also applied to the determination of the periods of rational numbers as well as their p-adic expansion in any radix $\beta \geq 2$.

1 Introduction

The modular division of two positive integers u and v modulo a radix $\beta \geq 2$ to the power s is defined as $(u/v) \bmod \beta^s$. It is used in many topics in theoretical computer science, such as p-adic computation, cryptography, Computer Algebra systems, etc. When s is small, say less than a machine word (i.e., 8, 16 or 32 bits), the modular division is completed by several algorithms in $O(s^2)$ time complexity. Such is the case for the *Extended Euclidean algorithm* (*EEA*) in [7]. However, the *EEA* is not efficient for large s, because it entails long quotient-remainder divisions of two long positive integers at each step.

In [5], Jebelean proposes a more efficient algorithm: *Modiv*. It is better suited for systolic parallelization in a *LSF* (*"least-significant digit first"*) pipelined manner. Actually, experiments performed in [5] using SACLIB Computer Algebra system indicate more than 20 times speed-up in computing the inverse modulo a power of $\beta = 2^{29}$. The speed-up was measured in comparison with the *EEA* algorithm improved by Lehmer [8] and Collins [3].

Jebelean also presents *Ediv*, a new algorithm for exact division of two long integers which improves on the classical quotient-remainder algorithm described in [7], when it is known in advance that the remainder is zero. Although it is presented as an application to *Ediv*, *Modiv* may rather be considered as the true main algorithm. As a matter of fact, *Ediv* is obtained easily by running *Modiv* with the specific parameter $s = \ell_\beta(u) - \ell_\beta(v) + 1$ (see Subsection 2.1 for the notation $\ell_\beta(u)$). Hence, we rather focus on modular division algorithms throughout the paper.

In Section 2, we compare and discuss sequential and parallel versions of *Modiv* and of the *"Pen and Paper Inverse"* (*PPI*) algorithm for modular division. Section 3 is devoted to the parallelization of the *PPI* algorithm, where a linear parallel *PPI* (*ParPPI*) algorithm using a new carry propagation technique is presented. Section 4, gives applications of the *ParPPI* algorithm to the exact division and the "digit modulus" (dmod) operations on two long integers, as well as to the determination of the periods of rational numbers and their p-adic expansion in any given radix $\beta \geq 2$. Section 5 concludes with some remarks.

2 Modular Division Algorithms

Let $x \equiv (u/v) \pmod{\beta^s}$. *Modiv* follows the process below. After shifting the operands u and v until they become relatively prime to β, x_0 (the least-significant digit of x) is found from u_0 and v_0 (the least-significant digits of u and v) in the form $x_0 \equiv u_0 v_0^{-1} \pmod{\beta}$.

Next, once x_0 is obtained, the next coefficient x_1 is performed by applying again the same process to $(u - x_0 v)/\beta$ and v.

2.1 Notation and Example

Throughout the paper we assume that any integer is expressed in radix $\beta \geq 2$, where β and v are coprime. As in [13], $\ell_\beta(u)$ denotes the number of digits needed to represent u in radix β and $\gcd(a, b)$ denotes the greatest commun divisor of integers a and b.

The algorithm designed in the paper is based on a simple technique: the classical "pen and paper" multiplication, which also works in *LSF* (*"least-significant digit first"*) manner.

Table 1: The *PPI* computation of $(37,229/1,543) \bmod 2^7$.

$\cdots\ \cdots\ \cdots\ 1 \ \leftarrow x$ unknown $\times\ 0\,0\,0\,0\,1\,1\,1 \ \leftarrow v \bmod 2^7$ $\cdots\ \cdots\ \cdots\ \cdots 1$ $\cdots\ \ \cdots\ \ \cdots$ $\cdots\ \ \cdots\ \ \cdots$ $=1\,1\,0\,1\,1\,0\,1 \ \leftarrow u \bmod 2^7$	$1\,1\,0\,1\,0\,1\,1 \ \leftarrow x = 107$ $\times\ 0\,0\,0\,0\,1\,1\,1 \ \leftarrow v \bmod 2^7$ $1\,1\,0\,1\,0\,1\,1$ $1\,0\,1\,0\,1\,1$ $0\,1\,0\,1\,1$ $=1\,1\,0\,1\,1\,0\,1 \ \leftarrow u \bmod 2^7$
Before the *PPI* algorithm	After the *PPI* algorithm

Roughly speaking, the algorithmic scheme consists in guessing and filling the missed digits of x from right to left, so that the puzzle adapts the classical multiplication, and the process stops when the s-th digit of x is found. This is the reason why the algorithm is called the *"Pen and Paper Inverse"* algorithm, or *PPI* for short.

In Table 1 below, a simple example describes this inverse multiplication scheme, whith $u = 37,229$, $v = 1,543$, $\beta = 2$ and $s = 7$.

3 The Algorithms *Modiv* and *PPI*

Let us recall the sequential and the parallel versions of the algorithm *Modiv* proposed in [5].

3.1 The sequential algorithm *SeqModiv*

Input: $u,\ v > 0$, with $\gcd(v, \beta) = 1$, and s a positive integer.
Output: $(u/v) \bmod \beta^s$.

$v_0 := v \bmod \beta$; $a := v_0^{-1} \bmod$ /* initialization */
for $k := 0$ **to** $s-1$ **do**
  $x_k := au \bmod \beta$;
  $u := (u - x_k v \bmod \beta^{s-k})/\beta$
endfor
return $x = (x_{s-1}, \ldots, x_0)$

The Sequential Algorithm SeqModiv.

The time complexity of *SeqModiv* is $O(s^2)$. In order to compare the parallel version of *Modiv* with the parallelization of the *PPI* algorithm, we first recall *ParModiv*, the parallel version f *Modiv*. In *ParModiv*, a carry save technique is used to handle the carry propagation (see [6, 9]).

For every integer x such that $|x| < \beta^2$, the notation $(a, b) := x$ is defined as

$$(a, b) \overset{def}{=} (x \text{ div } \beta, x \bmod \beta), \text{ with } 0 \le b < \beta,$$

where div is the "integral division": $x \text{ div } \beta \overset{def}{=} \lfloor x/\beta \rfloor$.

3.2 The Parallel Algorithm *ParModiv*

Input: $u,\ v > 0$, with $\gcd(v, \beta) = 1$, and s a positive integer.
Output: $(u/v) \bmod \beta^s$.

for $i := 1$ **to** s **pardo** $y_i := 0$ **endfor** /* initialization */
  $a := v_0^{-1} \bmod \beta$
for $k := 0$ **to** $s - 2$ **do**
  $x_k := au_k \bmod \beta$;
  $y_{k+1} := y_{k+1} - (x_k v_0) \text{ div}\beta$ /* update of the first carry */
    for $i := 1$ **to** $s - k - 1$ **pardo**
      $(y_{k+i+1}, u_{k+i}) := u_{k+i} - x_k v_i + y_{k+i}$
    endfor
endfor
  $x_{s-1} := au_{s-1} \bmod \beta$
return $x = (x_{s-1}, \ldots, x_1, x_0)$

The Parallel Algorithm ParModiv.

3.3 The PPI Algorithm

Assume u and β are coprime. Otherwise, $u = \beta^q u$' for some positive integers q and u', such that u' and β are coprime. Then, $x_i = 0$ for $i = 0, 1, \ldots, (q - 1)$. Since $(u/v) \bmod \beta^s = \beta^q((u'/v) \bmod \beta^{s-q})$, the algorithm variables reduces to the parameters u' and $s' = s - q$.

In the algorithm the s least-significant digits of u only are needed; and thus we may use $u \bmod \beta^s$ instead of u. Set

$$u \bmod \beta^s = \sum_{i=0}^{s-1} u_i \beta^i \quad \text{and} \quad v \bmod \beta^s = \sum_{j=0}^{s-1} v_j \beta^j,$$

and if $\ell_\beta(u) < s$, let $u_i = 0$ for all $i \ge \ell_\beta(u)$.

On the above assumptions, the algorithm described in Table 1 expresses as

Input: $u,\ v > 0$, with $\gcd(u,\beta) = \gcd(v,\beta) = 1$, and s a positive integer.
Output: $(u/v) \bmod \beta^s$.

$a := v_0^{-1} \bmod \beta$; $x_0 := (au_0) \bmod \beta$; /* initialization */
$c_1 := (x_0 v_0) \text{ div } \beta$
for $k := 1$ **to** $s-1$ **do** /* loop */
  $L_k := \sum_{j=1}^{k} v_j x_{k-j} + c_k$;
  $x_k := a(u_k - L_k) \bmod \beta$; $c_{k+1} := (L_k + x_k v_0) \text{ div } \beta$
endfor
return $x = (x_{s-1}, \ldots, x_0)$

The Sequential PPI Algorithm.

Remarks:

• In the case when $\beta = 2$, the algorithm turns out to be simpler.

• Notice that the loop is very similar to a triangular linear system of the form

$$A\,X \;=\; B,$$

with solution $X = {}^t(x_{s-1}, \ldots, x_1)$, where $B = {}^t(b_{s-1}, \ldots, b_1)$ is a given vector and $A = (a_{i,j})$ $(1 \le i,\ j \le s-1)$, is defined as

$$a_{i,j} \overset{def}{=} \begin{cases} v_1 & \text{if } i = j, \\ v_{i-j+1} & \text{if } i > j, \\ 0 & \text{if } i < j. \end{cases}$$

The parallelization process described in Section 4 applies to the above triangular system.

• In place of s, the constant $m = \ell_\beta(v \bmod \beta^s)$ can be used for updating L_k, because it is the current number of digits of v needed in the computation (see Fig. 1). The constant m allows faster computations: $m \le s$ and, when $m < s$, all the useless computations corresponding to $v_m, \ldots, v_{s-1}$ are eliminated. Hence, L_k is updated as follows:

$$L_k := \sum_{1 \le j \le \mu} v_j x_{k-j} + c_k, \qquad \text{where } \ \mu = \min(k-1, m-1).$$

• For any given k, the carry c_{k+1} satifies the relation $L_k + x_k v_0 = u_k + \beta c_{k+1}$.

Moreover, the worst-case time complexity of the above algorithm occurs for all pairs (u,v) such that $(u+v) \equiv 0 \pmod{\beta^s}$, with output $x = (u/v) \bmod \beta^s = \beta^s - 1$.

In that case, the largest value of L_k is $\beta k - 1$. Therefore, for all k,

$$L_k \le \beta k - 1 \le \beta s - \beta - 1.$$

4 Linear Parallelized *PPI* Algorithms

In the *PPI* algorithm, the output x is obtained step by step, least-significant digit first. The digits a, u_k and the least-significant digit of L_k, namely $L_k \bmod \beta$, are only needed to compute x_k. However, $L_k \bmod \beta$ is obtained after computing all of the two-digits products $v_j \times x_{k-j}$ and their sum.

This variant significantly increases the running time of the algorithm and prevents any efficient systolic implementation. In order to overcome the difficulty, we make use of the followings two facts:

1. Every two-bits product $v_j \times x_{k-j}$ can be computed and added to L_k as soon as x_{k-j} is found.

2. The goal of the parallelization is to break the computation of the sums $\sum_{1\le j\le k-1} v_j x_{k-j}$. As far as the carry propagation is concerned for updating the L_i's (for $i = k+1, \ldots, s-1$), the carry-save technique used in *ParModiv* can be applied successfully to the parallelization of the *PPI* algorithm *ParPPI*.

4.1 A Linear Parallel PPI Algorithm ParPPI

Input: $u,\ v > 0$, such that $\gcd(u, \beta) = \gcd(v, \beta) = 1$, and s a positive integer.
Output: $(u/v) \bmod \beta^s$.

```
for i := 0 to s pardo (y_i, L_i) := 0 endfor ;          /* initialization */
a := v_0^{-1} mod β
   for k := 0 to s − 2 do                              /* main loop */
      x_k := a(u_k − L_k − y_k) mod β
         for i := 0 to s − k − 1 pardo
            (y_{k+i+1}, L_{k+i}) := L_{k+i} + x_k v_i + y_{k+i}
         endfor
   endfor
      x_{s−1} := a(u_{s−1} − L_{s−1} − y_{s−1}) mod β
return x = (x_{s−1}, . . . , x_1, x_0)
```

ParPPI: a Linear Parallel PPI Algorithm (Version 1).

Comparison with *ParModiv*

ParPPI (version 1) and *ParModiv* are linear in terms of *"surface"* (i.e., the maximal number of processors needed in the algorithms) and time complexity. The algorithms are equivalent, but they also slightly differ in the following points:

• Every variable in the *ParPPI* algorithm consists of a single non-negative digit, whereas *ParModiv* uses a signed double digit for the variables y_j.

• In contrast with *ParModiv*, the *ParPPI* algorithm does not change the input u.

• In the *ParPPI* algorithm, all carries y_{k+i} are updated in parallel. By contrast the first carry is updated serially in *ParModiv*.

Although they are not important regarding the design of the algorithms themselves, the above differences cause substantial time improvements when s is large, and these algorithms are intensively used to devise efficient GCD algorithms, for example. (See [6, 11, 13].)

4.2 A New Carry Propagation Technique

We now describe a new carry propagation technique, which propagates the carries alternately. This new technique, called *"alternated carry"*, is illustrated in the second version of *ParPPI*, where the main loop of *ParPPI* (Version 1) designed in Subsection 4.1 can be rewritten as follows:

for $k := 0$ **to** $s - 2$ **do**
    $x_k := a(u_k - L_k) \bmod \beta$
        for $i := 0$ **to** $s - k - 1$ **pardo** $L_{k+i} := L_{k+i} + x_k v_i$ **endfor**
        for $n := 0$ **to** $\lfloor (s - k - 1)/2 \rfloor$ **pardo**
            $L_{k+2n+1} := L_{k+2n+1} + L_{k+2n} \operatorname{div} \beta$; $L_{k+2n} := L_{k+2n} \bmod \beta$
        endfor
endfor

Main Loop of the ParPPI Algorithm (Version 2).

Theorem 4.1 below shows that, whatever the input size s, all the L_i's are bounded. Therefore, the *ParPPI* algorithm (Version 2) is also linear in terms of surface (i.e., the maximal number of processors needed) and time complexity.

Theorem 4.1 *For any* $\beta \geq 2$, $s > 1$, $k \leq s - 1$ *and* $n \leq \lfloor (s - k - 1)/2 \rfloor$,
(i) $L_{k+2n} \leq \beta - 1$.
(ii) $L_{k+2n+1} \leq \beta^2 + \beta - 2$.

Proof:
By induction on k.
Basis: Obviously, $L_{2n} = L_{2n+1} = 0$. So, $L_{2n} \leq \beta - 1$ and $L_{2n+1} \leq \beta^2 + \beta - 2$.

Induction step: for any $i \geq k$, let $L_i(k)$ denote the value of L_i at the end of the k-th iteration. Suppose that inequalities (i) and (ii) hold for a given positive integer k. After the computation of x_{k+1}, and since $x_{k+1},\ v_j \leq \beta - 1$ for any k and j,

$$\begin{aligned} L_{k+1+2n+1}(k) + x_{k+1}v_{2n+1} &\leq \beta^2 - \beta, && \text{for } n \leq \lfloor (s-k-2)/2 \rfloor, (1)\\ L_{k+1+2n}(k) + x_{k+1}v_{2n} &\leq 2\beta^2 - \beta - 1, && \text{for } n \leq \lfloor (s-k-1)/2 \rfloor. (2) \end{aligned}$$

At the end of the $(k+1)$-st iteration Eq. (1) and Eq. (2) yield

$$L_{k+2n+1}(k+1) \leq (2\beta^2 - \beta - 1) \bmod \beta \leq \beta - 1,$$

and

$$L_{k+2n+2}(k+1) \leq \beta^2 - \beta + (2\beta^2 - \beta - 1) \text{ div } \beta = \beta^2 + \beta - 2.$$

Remark: The alternated carry propagation technique uses 3 digits for L_{k+2n+1} and one digit for L_{k+2n}. Thus, only $2s$ digits are needed in the algorithm.

5 Applications

The *ParPPI* algorithm can also be used in several applications. All these algorithms use the same "pen and paper" multiplication technique combined with a carry propagation technique (either Version 1 or Version 2). Since it is new, we rather use the second version in the applications.

5.1 Exact Division Algorithms

Algorithms for exact division compute the exact quotient u/v of two long integers u and v, when it is known in advance that the remainder is zero. The exact division is easily completed by using the *ParPPI* algorithm with the parameter $r = \ell_\beta(u) - \ell_\beta(v) + 1$ in place of s. Then, r is no longer the "input size" but rather expresses the difference between the sizes of inputs u and v [5].

5.2 The dmod Operation

Let $\ell_\beta(u) = s$ and $\ell_\beta(v) = t$, with $s \geq t$. The dmod operation is defined as

$$\text{dmod}_\beta(u,v) \overset{def}{=} |xv - u|/\beta^r, \text{ where } r = s - t + 1 \text{ and } x \equiv (u/v) \pmod{\beta^r}.$$

The algorithm below is similar to the *ParPPI* algorithm. It simply multiplies x_k and v, and simultaneously substracts u from $x_k v$. The subtraction is performed by β-complement. The β-complement of u_k is defined as follows:

$$u'_k \overset{def}{=} \begin{cases} (\beta - u_k) & \text{if } k = r,\\ (\beta - 1 - u_k) & \text{if } k > r. \end{cases}$$

We assume that xv and u are two $(s+1)$ digits numbers; so we set $u_s = 0$.

Input: u, v two positive numbers, $\ell_\beta(u) = s$, $\ell_\beta(v) = t$, with $s \geq t$, and $\gcd(v, \beta) = 1$.
Output: $x \equiv (u/v) \pmod{\beta^r}$ and $\mathrm{dmod}_\beta(u, v) = |xv - u|/\beta^r$, where $r = s - t + 1$.

for $i := 0$ **to** $s+1$ **pardo** $L_i := 0$ **endfor** ; /* initialization */
$a := v_0^{-1} \bmod \beta$; $r := s - t + 1$
for $k := 0$ **to** $r - 1$ **do**
$x_k := a(u_k - L_k) \bmod \beta$
for $i := 0$ **to** $t - 1$ **pardo** $L_{k+i} := L_{k+i} + x_k v_i$ **endfor**
for $n := 0$ **to** $\lfloor (s - k - 1)/2 \rfloor$ **pardo**
$L_{k+2n+1} := L_{k+2n+1} + L_{k+2n} \operatorname{div} \beta$;
$L_{k+2n} := L_{k+2n} \bmod \beta$
endfor
endfor
for $i := 0$ **to** $s - r$ **pardo** $L_{r+i} := L_{r+i} + u'_{r+i}$ **endfor**
for $k := r$ **to** s **do**
for $n := 0$ **to** $\lfloor (s - k)/2 \rfloor$ **pardo**
$L_{k+2n+1} := L_{k+2n+1} + L_{k+2n} \operatorname{div} \beta$;
$L_{k+2n} := L_{k+2n} \bmod \beta$
endfor
endfor
$w := (L_s, \ldots, L_r)$
if $L_{s+1} = 0$ **then** $w := \beta^t - w$
return x, w

The ParPPI dmod *Algorithm.*

Remark: If $L_{s+1} = 0$, $xv - u = w - \beta^t < 0$; and $xv - u$ is easily given by β-complement. Moreover the algorithm provides the sign of $xv - u$. Note also that the algorithm computes the modular division and the digit modulus operations simultaneously.

5.3 A Linear Surface-Time Multiplication

The same algorithm also applies to perform a multiplication which is also linear in terms of surface and time complexity. However, the *ParPPI* multiplication algorithm is not even efficient, since the best sequential multiplication algorithms are $O(s \log^c s)$ (for some constant $c > 0$).

Input: u, v two positive numbers, $\ell_\beta(u) = s$, $\ell_\beta(v) = t$.
Output: $L = uv$.

for $i := 0$ **to** $s + t$ **pardo** $L_i := 0$ **endfor**
for $k := 0$ **to** $t - 1$ **do**
  for $i := 0$ **to** $s - 1$ **pardo** $L_{k+i} := L_{k+i} + v_k u_i$ **endfor**
  for $n := 0$ **to** $\lfloor (s-1)/2 \rfloor$ **pardo**
    $L_{k+2n+1} := L_{k+2n+1} + L_{k+2n}$ div β ;
    $L_{k+2n} := L_{k+2n}$ mod β
  endfor
endfor
for $k := t$ **to** $s + t - 1$ **do**
  for $n := 0$ **to** $\lfloor (s+t-k-1)/2 \rfloor$ **pardo**
    $L_{k+2n+1} := L_{k+2n+1} + L_{k+2n}$ div β ;
    $L_{k+2n} := L_{k+2n}$ mod β
  endfor
endfor
return $L = (L_{s+t-1}, \ldots, L_1, L_0)$

The ParPPI Multiplication Algorithm.

5.4 The p-adic Expansion of Rationals

The ordered sequence of the s digits of $x \equiv (u/v) \pmod{\beta^s}$ represents exactly the expected Hensel code $\mathrm{H}(u, v; \beta^s)$. The Hensel code is thus directly given by the *ParPPI* algorithm. Note that the p-adic expansion is obtained step by step, when s tends to infinity.

5.5 Computation of Periods of Rational Numbers

Let u/v be a rational number, with $u < v$, and let v and β be coprime. Let T be the periodic part of the expansion, called the period of u/v, and let t be the length of T in base β. Then, $u/v = T/(\beta^t - 1)$, and thus $vT \equiv (-u) \pmod{\beta^t}$.

Now using the *PPI* algorithm with the parameters $(-u)$, v and t yields $T = (-u/v) \bmod \beta^t$.

Note that T is obtained in a "least-significant digit first" manner, whereas the classical division provides the period in a "most-significant digit first" manner.

6 Conclusion

The new modular division *PPI* algorithms (sequential and parallel variants) enjoy various interesting properties.

It is straightforward to derive many *LSF* algorithms from the *ParPPI* algorithm. Such is the case for the exact division, for the p-adic expansion and the periods of rational numbers, and for the multiplication and the dmod operations as well. All above applications are founded on the one and same identical scheme. This makes it easier to construct interactions between them, and thus provide an homogeneous collection of routines which may be extended later.

The *"alternated carry propagation"* is a new carry propagation technique, which is also fruitful for add-and-shift algorithms.

The parallel algorithms proposed herein follow the same classical multiplication scheme along with a carry propagation technique (either carry save or alternated carry). These algorithms are all linear ($O(s)$) in terms of surface $S(s)$ (i.e., the maximal number of processors needed) and time complexity $T(s)$, where s is the input size. As a consequence, the work (or cost), $W(s) = S(s) \times T(s)$, is $O(s^2)$, and they are neither in $\mathcal{NC}$, nor efficient (recall that the best sequential algorithms solving the same problems are $O(s \log^c(s))$, for some constant $c > 0$). Note also that their parallel speed-up is $O(\log^c s)$, with efficiency $O(\log^c(s)/s)$.

ParModiv and the main *ParPPI* algorithm are equivalent, since *ParModiv* is also surface-time linear (and hence optimal). However, our algorithm may significantly improve when used intensively within several efficient GCD algorithms, for example [6, 8, 11, 13]. A new algorithm is also presented, which performs the modular division as well as the digit modulus of two long positive integers simultaneously.

All the *ParPPI* algorithms described are suitable for systolic implementation in a "least-significant digit first" manner, because all decisions in the procedures are taken by using the lower digits of the operands. Hence they can be well aggregated to other systolic algorithms in the arithmetic of multiprecision rational numbers. *LSF* processing is also used in the most efficient systolic algorithms for multiprecision rational arithmetic. Among them, one may mention long integer multiplication [1] and addition/substraction [7] algorithms, the Brent-Kung systolic GCD algorithm [2] and the algorithms in [4, 5, 6].

The present work continues and complements our previous investigations [10] in improving the algorithm for modular division. The combined effects of these improvements allow several basic routines in Computer Algebra systems to run more efficiently.

References

[1] A.J. Atrubin in *IEEE Trans. on Computers*, C-14 (1965).

[2] R.P. Brent and H.T. Kung in *in VLSI'83*, eds. Anceau and Aas (1983) .

[3] G.E. Collins, *Un. of Wisconsin* (1980).

[4] T. Jebelean, *RISC-Linz Report* (1992).

[5] T. Jebelean in *in Proc. of the International Symposium on Symbolic and Algebraic Computation (ISSAC'93)* (1993).

[6] T. Jebelean in *Journal of Symbolic Computation* (1993).

[7] D.E. Knuth in *The art of computer programming: seminumerical algorithms* (1981).

[8] D.H. Lehmer in *American Math. Monthly* (1938).

[9] J.M. Muller, Masson (1989).

[10] M.S. Sedjelmaci and C. Lavault in *Information Processing Letters* (1997).

[11] J. Sorenson in *J. of Algorithms* (1994).

[12] Earl E. Swartlander Jr, eds. IEEE Computer Society Press (1990).

[13] K. Weber in *J. of symbolic Computation (Special Issue on Parallel Symbolic Computation)* (1996).