Integer and Rational Arithmetic on MasPar

Tudor Jebelean*

Research Institute for Symbolic Computation
A-4232 Hagenberg, Austria (Europe)
tudor@risc.uni-linz.ac.at

Abstract. The speed of integer and rational arithmetic increases significantly by systolic implementation on a SIMD architecture. For multiplication of integers one obtains linear speed-up (up to 29 times), using a serial–parallel scheme. A two-dimensional algorithm for multiplication of polynomials gives half-linear speed-up (up to 383 times). We also implement multiprecision rational arithmetic using known systolic algorithms for addition and multiplication, as well as recent algorithms for exact division and GCD computation. All algorithms work in "least-significant digits first" pipelined manner, hence they can be well aggregated together. The practical experiments show that the timings depend linearly on the input length, demonstrating the effectiveness of the systolic paradigm for multiple precision arithmetic.

1 Introduction

Systolic parallelization of multiprecision arithmetic in the "most-significant digits first" (MSF) pipelined manner was considered by [13] and other authors (see [12], chapter 3), using the signed-digit redundant representation.

Our approach is different: we use "least-significant digits first" (LSF) algorithms, because this allows pipelined aggregation of the various operations. Also, these algorithms use standard representation of multiprecision integers in an arbitrary radix (typically a power of 2), which makes them suitable for implementation on multiprocessor architectures.

SIMD parallelization of computer algebra algorithms did not receive much attention in the literature. [15] reports a 45 times speed–up of Gröbner Basis computations by parallelizing multiprecision algorithms on the SIMD like Convex vector processor, but most of the speed-up is due to some improvements in list processing operations and to the use of 64-bit arithmetic. Univariate polynomial multiplication by parallelizing both the level of coefficient operations and binary-digit operations was considered by [1] on the ICL DAP computer (SIMD architecture).

The MasPar computer – shortly presented in Section 2 – is particularly suitable for the implementation of systolic algorithms. Of paramount importance for

* Supported by the Austrian *Fonds zur Förderung der wissenschaftlichen Forschung*, project P10002 MAT

our application are: the fast communication between adjacent processors, and the high efficiency of global broadcasting.

In Sections 3 and 4 we describe the implementation of long integer and of univariate integral polynomial multiplication. We use multiprecision variants of serial / parallel multipliers which can be easily derived from the "school method". Apparently these algorithms were the first to be considered for hardware multiplication – see [2]. In both algorithms, one of the operands is present in the array at the beginning of computation, while the elements of the other one are broadcasted to the parallel processors, one at each step. The first algorithm pipelines the result out via the first processor, while the second algorithm leaves the result in the array. The second algorithm is suitable for embedding into polynomial multiplication scheme, yielding an algorithm with two–level systolic parallelism, which maps naturally onto the two–dimensional architecture of MasPar.

For multiplication of multiprecision integers we obtain almost linear speed-up over the classical sequential algorithm (29 times for 30 digit integers, efficiency 95%). The two-dimensional algorithm for multiplication of univariate integral polynomials gives almost half-linear speed-up (383 times for polynomials of degree 29 with multiprecision coefficients of 15 digits, efficiency 43%).

In Sections 5 – 9 we present the systolic implementation of a rational operation which is widely used in typical algebraic computations – e. g. in Gröbner Bases [4]. Besides multiplication, one also uses addition, division, and GCD computation. Theoretically, **addition** in standard representation cannot be improved by systolic parallelization, but practically it runs in constant time, because the carry chain seldom exceeds two digits. Since **division** is with null remainder, one can use the exact division algorithm recently introduced in [7] - some systolic parallelizations of exact division are described in [8] - we choose the one which is most suitable in the present context. The most complicated operation is the **computation of the GCD**, which is implemented using the systolic parallelization [9] of the recently developed algorithm from [6, 14].

The most important conclusion of the practical experiments is that one obtains linear timings - that is, running time depends linearly on the length of the input numbers. This demonstrates the effectiveness of the systolic paradigm for the implementation of long integer and long rational arithmetic.

2 The MasPar Computer

We present here only those features which are relevant for our approach.

MasPar is a SIMD distributed memory machine, with 1024 Processing Elements (PE's), arranged in a 32 by 32 mesh (torus). The PE's are driven by a sequential Array Control Unit (ACU). The device can be programmed in the language C, with some special extensions for handling MasPar parallelism.

Data: The ACU and each PE have their own internal memory for data. In C language, one has to use `plural` to declare the variables which are allocated on the parallel PE's. A `plural` variable will have one instance on each PE, possibly containing different values. The variables which are not `plural` are called *singular* and are allocated on ACU.

Program flow: The operations involving only *singular* variables are executed sequentially on the ACU. The operations involving `plural` variables are executed in parallel on the PE's. All PE's execute the same instructions synchronously. However, at certain moments some of the PE's may be "masked" (by conditional instructions), and then they execute nothing.

Data communication between ACU and PE's: The ACU accesses all the PE's in parallel. Hence, data can be broadcasted to all the PE's in one step. For instance, if `a` is *singular* and `b` is `plural`, then the assignment "`b = a;`" will send the value of `a` to all active PE's. The reverse operation is not possible, but one can use "`a = globalor(b);`", which performs a bitwise logical OR on all `b`'s in the active PE's. Also, a plural variable (say `b`) on a particular PE can be accessed using `proc[i].b` (linear addressing, $0 \leq \mathtt{i} \leq 1023$) or `proc[y][x].b` (2D addressing, $0 \leq \mathtt{x}, \mathtt{y} \leq 31$). The `proc` construct may be used in either left or right hand side of assignments, thus yielding the means to store/load values to/from particular PE's.

Data communication between PE's: Data may be moved between adjacent PE's by using the `xnet` construct. For instance, if `b` and `c` are plural variables, then "`xnetW[1].b = c;`" means "store the value of `c` into `b` of the left neighbor". This is also executed in parallel on all active PE's.

3 Long Integer Multiplication

The inputs A, B and the output $C = A*B$ are multiprecision integers represented as lists of positive digits in radix β:

$$A = a_0 + a_1 * \beta + \ldots + a_{n-1} * \beta^{n-1}, \quad B = b_0 + b_1 * \beta + \ldots + b_{m-1} * \beta^{m-1}.$$

$$C = c_0 + c_1 * \beta + \ldots + c_{n+m-1} * \beta^{n+m-1}.$$

The classical algorithm for multiplication consists of a double loop whose innermost instruction is:

$$(carry, c_{i+j}) \leftarrow c_{i+j} + b_j * a_i + carry.$$

This algorithm is inherently sequential, because each step uses the *carry* produced by the previous step. We change this by using a list $Y = (y_0, y_1, \ldots, y_{n+m-1})$ to hold the carries produced at each step. This list has as many elements as C has. The computation then proceeds in two stages:

- Stage 1: The additions and the multiplications are performed and the carries are produced. Since the outermost loop (over A) is performed sequentially, the carries produced in one step may be used in the following step.
- Stage 2: The carries are absorbed into C, by adding each y_k to c_{k+1}. These additions may produce new carries, which are again stored in Y list, and the absorption stage is repeated until all the carries become zero.

This scheme allows the parallelization of the inner loop (over B), leading to the systolic algorithm shown in Fig. 1. The local variables on each processor are denoted by $\overline{B} = (\overline{b}_0, \overline{b}_1, \ldots, \overline{b}_m)$, $\overline{C} = (\overline{c}_0, \overline{c}_1, \ldots, \overline{c}_m)$, $\overline{Y} = (\overline{y}_0, \overline{y}_1, \ldots, \overline{y}_m)$. The vector A is not stored in the processors. Rather, at each iteration of the main loop, one element of A is send to all the processors for the computation in line { 6}. During Stage 1, the m processors act as a window which moves along the vector C, one element at a time. In other words, C is piped through the string of m processors. During Stage 2, the window is fixed on the last m elements of C.

In practice we use an $m + 1^{th}$ processor whose $\overline{b}_m, \overline{c}_m$, and $\overline{y}_m$ are zero all the time. This boundary processor does not participate in the computation, but its presence avoids boundary tests.

{ 0} $C \leftarrow$ `IntSysMul.1`(A, B) [Systolic integer multiplication, version 1]
{ 1} $\quad \overline{B}, \overline{C}, \overline{Y} \leftarrow (0, \ldots, 0)$ [$m + 1$ positions]
{ 2} $\quad$ **for** $j = 0, 1, \ldots, m-1$ **do** [load B sequentially]
{ 3} $\qquad \overline{b}_j \leftarrow b_j$
{ 4} $\quad$ **for** $i = 0, 1, \ldots, n-1$ **do** {Stage 1: add and multiply]
{ 5} $\qquad$ **for** $j = 0, 1, \ldots, m-1$ **in parallel do**
{ 6} $\qquad\quad (\overline{y}_j, \overline{c}_j) \leftarrow \overline{c}_j + \overline{b}_j * a_i + \overline{y}_j$ [compute]
{ 7} $\qquad\quad c_i \leftarrow \overline{c}_0$ [extract next digit of C]
{ 8} $\qquad$ **for** $j = 0, 1, \ldots, m-1$ **in parallel do**
{ 9} $\qquad\quad \overline{c}_j \leftarrow \overline{c}_{j+1}$ [shift $\overline{C}$ left]
{10} $\quad$ **while** `globalor`$(\overline{Y})$ **do** [Stage 2: absorb carries]
{11} $\qquad$ **for** $j = 0, 1, \ldots, m-1$ **in parallel do**
{12] $\qquad\quad (\overline{y}_{j+1}, \overline{c}_j) \leftarrow \overline{c}_j + \overline{y}_j$
{13} $\qquad \overline{y}_0 \leftarrow 0$
{14} $\quad$ **for** $j = 0, 1, \ldots, m-1$ **do** [extract rest of C sequentially]
{15} $\qquad c_{n+j} \leftarrow \overline{c}_j$

Fig. 1. Systolic multiprecision multiplication, version 1.

The parallel loops {5}, {8}, {11} and the initialization {1} require constant time. The other loops are {2}: n steps, {4}: n steps, {10}: at most m steps, and {14}: m steps. Hence $T_{\text{systolic}} = O(n + m)$. For balanced–length operands: $T_{\text{systolic}} = O(n)$.

We implemented the algorithm `IntSysMul.1` on MasPar MP1, using only the first row of 32 processors, and the classic algorithm, using only one processor. Fig. 2 shows the timings in milliseconds for the two algorithms (systolic timings are scaled by 10). The speed–up is linear w.r.t. input length and ranges between 4 (at 5 digits) and 29 (at 30 digits). The efficiency ranges between 81% and 96%.

Second version: If C is to be used in subsequent computations (as it is the case in polynomial multiplication), then it is useful to leave it in the array,

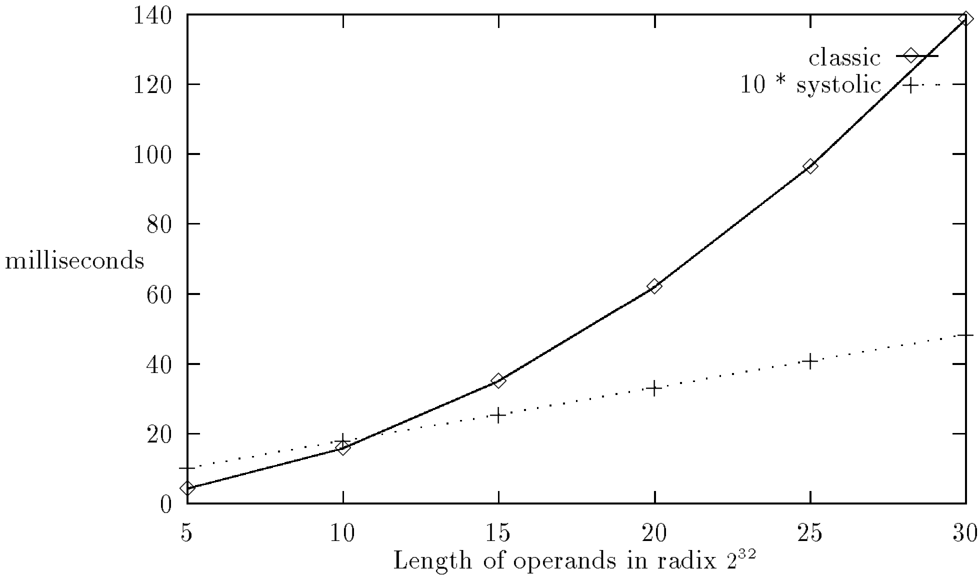

Fig. 2. Comparative timings for multiprecision multiplication.

instead of pipelining/extracting it. Using $n+m+1$ processors, C can be stored in $\overline{C}$, and then $\overline{B}$ and $\overline{Y}$ must be shifted rightward one position at each step. However, most of the time only m of the $n+m+1$ processors do useful work. This results in a lower efficiency of parallelism (1/2 for balanced-length operands).

4 Polynomial Multiplication

The inputs $\mathcal{A}, \mathcal{B}$ and the output $\mathcal{C} = \mathcal{A} * \mathcal{B}$ are integral univariate polynomials represented as lists of multiprecision integers:

$$\mathcal{A} = A_0 + A_1 * x + \ldots + A_{N-1} * x^{N-1}, \quad \mathcal{B} = B_0 + B_1 * x + \ldots + B_{M-1} * x^{M-1}.$$
$$\mathcal{C} = C_0 + C_1 * x + \ldots + C_{N+M-2} * x^{N+M-2}.$$

The innermost loop of the classical algorithm performs the operation $C+A*B$ over long integers. In this case there is no problem in parallelizing the inner loop of the algorithm, using M computing units. As in the case of `IntSysMul.1` (fig. 1), $\mathcal{C}$ is piped through the string of these M computing units. However, each of these computing units must be able to compute $C + B * A$ on long integers. This is exactly what is done by the second version of the integer systolic algorithm, if the initialization of C is removed. Therefore, one can use a row of processors for each of the above M computing units. Overall, one needs a matrix of $(M+1)*(n+m+2)$ processors, where n, m are the maximum lengths of the coefficients of $\mathcal{A}, \mathcal{B}$.

The local variables of each processor are denoted by $\overline{\mathcal{B}} = (\overline{b}_{J,j})$, $\overline{\mathcal{B}}' = (\overline{b'}_{J,j})$, $\overline{\mathcal{C}} = (\overline{c}_{J,j})$, $\overline{\mathcal{Y}} = (\overline{y}_{J,j})$. $\overline{\mathcal{B}}'$ contains the coefficients of $\mathcal{B}$, shifted rightward as required by the integer algorithm, and gets from $\overline{\mathcal{B}}$ the non–shifted values at the beginning of each main cycle. The $(M+1)^{th}$ row and the $(n+m+2)^{th}$ column of processors ensure the boundary conditions and do not participate in the computation. The coefficients of $\mathcal{A}$ are not stored in the parallel processors. Rather, they are send to all the processors, one digit at each step, for the computation in line { 9} (see Fig.3).

{ 0} $\mathcal{C} \leftarrow$ `PolySysMul`$(\mathcal{A}, \mathcal{B})$ [Systolic polynomial multiplication]
{ 1} $\quad\overline{\mathcal{C}}, \overline{\mathcal{B}}, \overline{\mathcal{Y}} \leftarrow (0, \ldots, 0)$ [in parallel]
{ 2} $\quad$ `for` $(J, j) = (0,0), (0,1), \ldots, (M-1, m-1)$ `do` [load $\mathcal{B}$ sequentially]
{ 3} $\qquad \overline{b}_{J,j} \leftarrow b_{J,j}$
{ 4} $\quad$ `for` $I = 0, 1, \ldots, N-1$ `do` [scan coefficients of $\mathcal{A}$]
{ 5} $\qquad$ `for` $J = 0, 1, \ldots, M-1$ `in parallel do` $[C_{I+J} \leftarrow C_{I+J} + B_J * A_I]$
{ 6} $\qquad\quad \overline{b}'_{J,j} \leftarrow \overline{b}_{J,j}$ [restore $\overline{\mathcal{B}}$]
{ 7} $\qquad\quad$ `for` $i = 0, 1, \ldots, n-1$ `do` [Stage 1: add and multiply]
{ 8} $\qquad\qquad$ `for` $j = 0, 1, \ldots, n+m$ `in parallel do`
{ 9} $\qquad\qquad\quad (\overline{y}_{J,j+1}, \overline{c}_{J,j}) \leftarrow \overline{c}_{J,j} + \overline{b'}_{J,j} * a_{I,i} + \overline{y}_{J,j}$ [compute, shift $\overline{\mathcal{Y}}$]
{10} $\qquad\qquad\quad \overline{b'}_{J,j+1} \leftarrow \overline{b'}_{J,j}$ [shift $\overline{\mathcal{B}}$]
{11} $\qquad\quad$ `while globalor(`$\overline{\mathcal{Y}}$`) do` [Stage 2: absorb carries]
{12} $\qquad\qquad$ `for` $j = 0, 1, \ldots, n+m$ `in parallel do`
{13} $\qquad\qquad\quad (\overline{y}_{J,j+1}, \overline{c}_{J,j}) \leftarrow \overline{c}_{J,j} + \overline{y}_{J,j}$
{14} $\qquad$ `for` $j = 0, 1, \ldots, n+m$ `do` [extract C_I sequentially]
{15} $\qquad\quad c_{I,j} \leftarrow \overline{c}_{0,j}$
{16} $\qquad$ `for` $(J, j) = (0,0), (0,1), \ldots, (M-1, m-1)$ `in parallel do`
{17} $\qquad\quad \overline{c}_{J,j} \leftarrow \overline{c}_{J+1,j}$ [shift $\overline{\mathcal{C}}$ upwards]
{16} $\quad$ `for` $J = 0, 1, \ldots, M-1$ `do` [extract rest of $\mathcal{C}$ sequentially]
{17} $\qquad$ `for` $j = 0, 1, \ldots, n+m$ `do` [extract C_{N+J} sequentially]
{18} $\qquad\quad c_{N+J,j} \leftarrow \overline{c}_{J,j}$

Fig. 3. Systolic polynomial multiplication.

Time complexity: We do not count the parallel loops {5}, {8}, {12} and {16}. The loop {2} is performed $M * m$ times, the loop {4} (N times) has several inner loops: {7} n times, {11} at most m times, and {14} $n + m$ times, hence $N * (n+m)$ is dominating. Finally, the loop {16} is repeated $M * (n+m)$ times. All in all: $T_{\text{systolic}} = O((N + M) * (n + m))$. For balanced–length operands: $T_{\text{systolic}} = O(N * n)$.

We implemented the algorithm `PolySysMul` on MasPar MP1, using the matrix of 32 by 32 processors, and the classical algorithm, using only one processor. The experiments used polynomials with 5 to 30 coefficients, whose size ranges between 5 and 15 words. The timings of the classical algorithm show the characteristic parabola, and grow up to 32 seconds. Figure 4 shows the speed–up – the maximum is 383. The efficiency ranges between 38% and 43%, approaching the 50% theoretical limit.

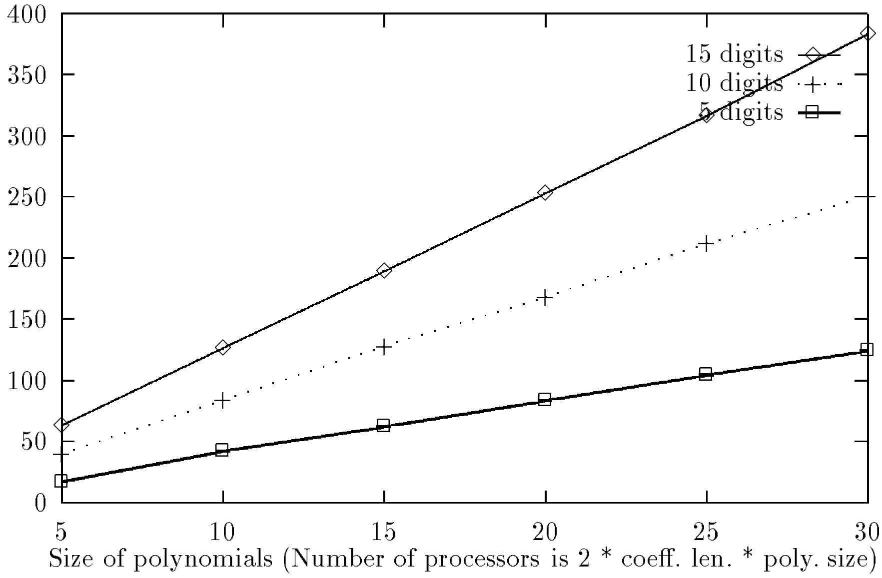

Fig. 4. Speed-up of systolic polynomial multiplication.

5 Rational Arithmetic

The operation which we implement is *rational reduction*, that is, given rational numbers $\frac{A}{B}$, $\frac{C}{D}$, $\frac{X}{Y}$, find $\frac{E}{F} = \frac{A}{B} - \frac{X}{Y} * \frac{C}{D}$, where E/F is normalized. This operation is heavily used, for instance, in Gröbner bases computation [4], for inter-reduction of polynomials. All the basic operations with long integers are involved in this reduction: $E' = A * Y * D - B * X * C$, $F' = B * Y * D$, $G = GCD(E', F')$, $E = E'/G, F = F'/G$, where the last two divisions are *exact divisions*. Note that we do not use here Henrici's approach [5], which gives poor results in the context of systolic parallelization.

The input/output is intermingled with the computations as follows: the operands are loaded into the array during multiplication, exact division outputs the result during computation.

6 Multiplication and Addition

Each **multiplication** is performed according to the second variant of the systolic algorithm presented in Section 3. This scheme requires one operand to be present in the array, while the other is loaded during multiplication. Therefore, first C and D are sequentially loaded into the array (this is the only I/O operation which does not overlap with actual computation). Subsequently, $Y * D$ and $X * C$ are computed, then $A * Y * D$, $B * Y * D$ and $B * X * C$.

Addition (subtraction) is performed using complement representation. The operands are represented by filling-up the array with additional words (*sign-words*) which equal zero for a positive operand and $2^{32}-1$ for a negative operand. The actual addition is performed in digit-parallel fashion, using the ripple-carry scheme. This scheme has a linear-time worst-case complexity, but, however, the probability of the worst-case situation is extremely low when using high-radix digits. In fact, in our experiments we never encountered a situation when the carry propagation needed more than 2 steps.

The only situation when the carry could systematically ripple along many words is the case of subtraction when the result is positive. In order to limit the number of steps in this case, we use a *mask*, which indicates the significant words of the result.

The experimentally measured running time is in fact constant and takes less than 0.1% of the entire rational reduction operation.

7 Greatest Common Divisor

GCD computation is the most complicated and also the most time-consuming operation. Also, parallelization of the classical Euclidean algorithm – or Lehmer improved scheme [10] – is difficult, because of the carry propagation. An algorithm in which the decisions are taken using the *least-significant* digits of the operands is the *binary* algorithm of [11], which was adapted for systolic computations by [3] – the so called PlusMinus algorithm. These algorithms, however, work at *binary* level, hence they are less suitable for implementation on multiprocessor machines working at *word* level.

Therefore, we parallelize here the *generalized binary* algorithm from [6], which works least-significant digits first, and also at word level. This algorithm needs some further adaptations in order to be suitable for systolic parallelization. Namely, the problem is that the generalized binary algorithm finds an *approximation* G' of the true $G = GCD(A, B)$, which in the sequential version is corrected by computing $G = GCD(A, B, G') = GCD(A \bmod G', B \bmod G', G')$. These computations are difficult to parallelize systolically, hence we want to avoid them. One way would be to replace the division with remainder by exact division, whose result is also suitable for finding the true GCD, and then continue the computation using the systolic PlusMinus algorithm or an improved version for high-radix computation. We do not use this approach here, but an exact version of the generalized binary algorithm as described in [9]. The only cause for this is the simplicity of the implementation.

$G \leftarrow$ `IntSysGCD`(A, B) $[C \leftarrow GCD(A, B)]$
  $(A, B) \leftarrow$ `ShiftTwo`(A, B) [shift common zeroes]
  `while` $B \neq 0$ [main loop]
    $A \leftarrow$ `ShiftOne`(A) [shift A]
    $B \leftarrow$ `ShiftOne`(B) [shift B]
    $(x, y, x', y', s) \leftarrow$ `Cofactors`(a_0, b_0) [compute cofactors]
    $A' \leftarrow$ `LinComb`(x, A, s, y, B) [first linear combination]
    $B' \leftarrow$ `LinComb`$(x', A, 1 - s, y', B)$ [second linear combination]
    `if` $A' \neq 0$ [replace]
      `then` $(A, B) \leftarrow (A', B')$
      `else` $(A, B) \leftarrow (B', A')$
  [end of main loop: B is 0, A is the GCD]
  $G \leftarrow$ `ComplIfNeg`(C) [complement G if negative]

Fig. 5. Systolic multiprecision GCD computation.

The outline of the algorithm is presented in Fig. 5. The routine `ShiftOne`(X) shifts the least-significant bits out of the nonzero X. The routine `ShiftTwo`(X, Y) shifts out the *common* least-significant bits from its arguments (of which at least one must be nonzero). Both routines operate in (almost) constant time, because the probability that many least-significant *words* are null is very small.

The routine `LinComb`(x, X, s, y, Y) computes the linear combination $x * X \pm y * Y$ (parameter s indicates $+$ or $-$). The routine works for negative operands also, using complement representation. A mask is used in order to indicate the range of correct values.

In order to avoid rippling the carries at each step, X, Y and the result of the linear combination are represented by *two* arrays of values, one array containing the actual digits, and one containing the carries which are not propagated yet. During each linear combination, the carries are propagated only 1 step, after which each carry becomes at most 1 (this decreases the cost of the next multiplication). Note that the least-significant digit of the result (needed for the next reduction step) is always correct.

After the main loop, G is complemented if negative. In fact, G should be also shifted with the same number of binary positions which were shifted out from the inputs at the beginning. However, in the actual implementation we perform `ShiftTwo(...)` *before* calling the GCD routine, thus the normalization is still correctly done. Indeed, the GCD computation is needed for the *normalization* of the rational fraction E'/F'. We shift out of E', F' the common trailing binary zeroes, obtaining $E", F"$. Then the GCD algorithm is used to find $G" = GCD(E", F")$, and then $E = E"/G"$ and $F = F"/G"$ are found by exact division. Note that $G"$ is always odd, which suits well the needs of the exact division algorithm.

The main reduction scheme works only if the operands are multiprecision. If the GCD is single precision, then at some moment both operands A, B might also become single precision. From this moment the [single precision] Euclidean algorithm is used for finding the GCD.

8 Exact Division

The final stage of computation consists in performing the exact divisions by the GCD. As explained in the previous section, the divisor is already odd, hence the exact division algorithm introduced in [7] can be applied without any pre-processing. In [8] several systolic variants of this algorithm are described. We choose for implementation the version which suits well the particular characteristics of this application – see Fig. 6. Namely, the algorithm is simpler because global communication can be used and also the digits of the result are pushed out *during* the computation.

$B \leftarrow$ `IntSysEDIV`(C, A) [$B \leftarrow C/A$]
    $a' \leftarrow$ `ModInv`(a) [find ${a_0}^{-1}$ mod 2^{32}]
    `while` $C \neq 0$ [main loop]
        $b \leftarrow (c_0 * a') \bmod 2^{32}$ [find next digit of the quotient ...]
        $B_{\text{next}} \leftarrow b$ [...and push it out]
        $C \leftarrow$ `LinComb`$(1, C, 1, a', A)$ [$C \leftarrow C - a' * A$]
        `for` $i = 0, 1, \ldots$ `in parallel do` [shift C left]
            $c_i \leftarrow c_{i+1}$
    [end of main loop]

Fig. 6. Systolic multiprecision exact division.

The vector B in this algorithm is external to the processor array – it represents the output of the algorithm. The function `ModInv`(a) is based on the recursion developed in [7], hence we avoid the (expensive) extended Euclidean algorithm. A simplified version of the function `LinComb` from the GCD algorithm is used for performing the operation $C - a' * A$. Again the carries are not propagated at each step, because only the correct value of the least-significant digit of C is needed for continuing the computation.

9 Experimental Results

The algorithms were implemented on a computer MasPar MP-1 having an array of 32 by 32 processors. The programs handle this two-dimensional array as an one-dimensional array of 1024 processors, virtually connecting the rows at their edges.

A straightforward calculation of the time complexity of all the algorithms will be similar to the one for multiplication (see Sect. 3) and will reveal linear complexity. For practical purposes, however, direct timing of the algorithms is even more relevant. We timed the execution for random inputs having length up to 100 32-bit words. That means GCD is computed for operands having (roughly) 300 words, while its output is usually small (single precision). The timings are presented in fig. 7. The times consumed for addition and `ShiftTwo` before GCD computation are 0.70 and 0.35 milliseconds, respectively, and are not shown in the figure.

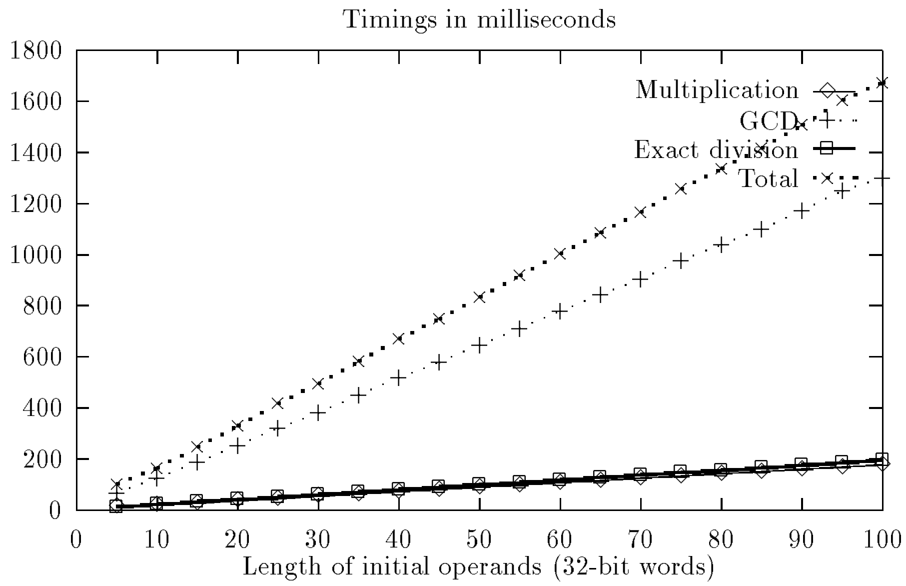

Fig. 7. Timings of the rational reduction and its components.

The most important characteristic of the timing is the *linear dependence* of the lengths of the input. This shows that the systolic model can be effectively used on MasPar architecture for implementing long integer arithmetic.

Further work includes improving the efficiency of the implementation, – especially that of the GCD computation, which takes most of the time – and embedding the rational reduction algorithm in higher-level algebraic computations.

References

1. R. Beardsworth. On the application of array processors to symbol manipulation.

In *SYMSAC'81*, 1981.
2. A. D. Booth. A signed binary multiplication technique. *Q. J. Mech. Appl. Math.*, 4:236–240, 1951.
3. R. P. Brent and H. T. Kung. A systolic algorithm for integer GCD computation. In K. Hwang, editor, *Procs. of the 7th Symp. on Computer Arithmetic*, pages 118–125. IEEE Computer Society, June 1985.
4. B. Buchberger. Gröbner Bases: An Algorithmic Method in Polynomial Ideal Theory. In Bose and Reidel, editors, *Recent trends in Multidimensional Systems*, pages 184–232, Dordrecht-Boston-Lancaster, 1985. D. Reidel Publishing Company.
5. P. Henrici. A subroutine for computations with rational numbers. *Journal of the ACM*, 3:6–9, 1956.
6. T. Jebelean. A generalization of the binary GCD algorithm. In M. Bronstein, editor, *ISSAC'93: International Symposium on Symbolic and Algebraic Computation*, pages 111–116, Kiev, Ukraine, July 1993. ACM Press.
7. T. Jebelean. An algorithm for exact division. *Journal of Symbolic Computation*, 15(2):169–180, February 1993.
8. T. Jebelean. Systolic Algorithms for Exact Division. In *Workshop on Fine Grain and Massive Parallelism*, pages 40–50, Dresden, Germany, April 1993. Published in *Mitteilngen–Gesellschaft für Informatik e. V. Parallel Algorithmen und Rechnerstrukturen*, Nr. 12, July 1993.
9. T. Jebelean. Systolic algorithms for long integer GCD computation. In J. Volkert B. Buchberger, editor, *CONPAR 94 - VAPP VI, Linz, Austria, September*, pages 241–252. Springer Verlag LNCS 854, 1994.
10. D. H. Lehmer. Euclid's algorithm for large numbers. *Am. Math. Mon.*, 45:227–233, 1938.
11. J. Stein. Computational problems associated with Racah algebra. *J. Comp. Phys.*, 1:397–405, 1967.
12. E. E. Swartzlander, editor. *Computer Arithmetic*, volume 2. IEEE Computer Society Press, 1990.
13. K. S. Trivedi and M. D. Ercegovac. On-line algorithms for division and multiplication. *IEEE Trans. on Computers*, C-26(7):681–687, 1977.
14. K. Weber. The accelerated integer GCD algorithm. *ACM Trans. on Math. Software*, 21(1):111–122, March 1995.
15. D. Weeks. Adaptation of SAC-1 algorithms for an SIMD machine. In J. Della Dora and J. Fitch, editors, *Computer Algebra and Parallelism*, pages 167–177. Academic Press, 1989.

This article was processed using the LaTeX macro package with LLNCS style