Systolic Multiplication on MasPar *

Tudor Jebelean
RISC-LINZ

—

Research Institute for Symbolic Computation
Johannes Kepler University, A–4040 Linz, Austria (Europe)
Tel: +43 (7236) 3231-49, -20 Fax: +43 (7236) 3231-30
e-mail: `tjebelea@risc.uni-linz.ac.at`

—

Abstract

We investigate the possibility of speeding-up multiplication by systolic implementation on MasPar (SIMD architecture) computer. For multiplication of multiprecision integers we obtain almost linear speed-up over the classical sequential algorithm (29 times for 30 digit integers, efficiency 95%), using a systolic parallelization scheme for a linear array of processors. By embedding this algorithm in a systolic two-dimensional scheme for multiplication of univariate integral polynomials, we obtain almost half-linear speed-up for this operation (383 times speed-up for polynomials of degree 29 with multiprecision coefficients of 15 digits, efficiency 43%). These experimental results show the usefulness of the systolic paradigm for increasing the efficiency of algebraic computations.

Also, this research constitutes a relevant example of fruitful cooperation between two ACPC member organizations situated at different locations, by remote access to computing equipment, exchange of documentation, and useful discussions.

Introduction

The speed of multiprecision arithmetic has a decisive influence on the speed of computer algebra systems. By performing a series of measurements upon typical algebraic algorithms [Buchberger, Jebelean 92], we found out that most of the computation time is spent in integer/rational operations, and this proportion increases when the length of the input coefficients increases. Also, the absolute total time increases dramatically, exclusively on behalf of integer/rational operations. For example, in Gröbner Bases computation [Buchberger 85], when increasing the coefficients length from 1 to 10 decimal digits, the proportion of rational operations grows from 62% to 98%, and the total computation time grows by a factor of 25.

This paper is part of a of a research aimed at speeding up computer algebra systems by systolic parallelization of long integer and long rational arithmetic. Systolic parallelization

*Acknowledgements:
Austrian Forschungsförderungsfonds, project S5302-PHY (Parallel Symbolic Computation);
Austrian Ministry for Science and Research, project 613.523/3-27a/89 (Gröbner Bases) and doctoral scholarship;
POSSO project(Polynomial Systems Solving – ESPRIT III BRA 6846);

of rational arithmetic in the "most-significant digits first" (MSF) pipelined manner was considered by [Trivedi, Ercegovac 77], using the signed bit representation of rationals as continued fractions.

Our approach is different: we use "least-significant digits first" (LSF) algorithms, because this allows pipelined aggregation of various multiprecision operations (e.g. addition/subtraction units, on-line multiplication [Atrubin 65], exact division [Jebelean 92a], [Jebelean 92b], LSF greatest common divisor [Brent, Kung 83]). Also, our 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. We demonstrate in this paper the possibility of embedding this kind of systolic algorithms into higher level algebraic algorithms (e.g. multiprecision multiplication into polynomial multiplication).

Systolic multiplication was considered by many authors (see [Jebelean 92c] for a survey), one of the first being [Atrubin 65]. [Weeks 88] obtained 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. In particular, univariate polynomial multiplication by parallelizing both the level of coefficient operations and binary-digit operations was considered by [Beadsworth 81] on the ICL DAP computer (SIMD architecture).

The algorithms we use are similar to the systolic polynomial multiplication scheme of [Kung 81], obtained by a natural parallelization of the inner loop of the classical algorithm. 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.

The experimental results show the effectiveness of the systolic paradigm for increasing the efficiency of algebraic computations. For multiplication of multiprecision integers we obtain almost linear speed-up over the classical sequential algorithm (29 times for 30 digit integers, efficiency 95%), using a systolic parallelization scheme for a linear array of processors. 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%).

Last but not least, this research was done in the frame of a joint research project of the ACPC (Austrian Center for Parallel Computation), and constitutes a relevant example of cooperation between two member organizations situated at different locations. The author (from RISC–Linz) ran the experiments remotely on the MasPar computer of the Technical University of Vienna. This activity was facilitated by the organization of the ACPC network, which allows researchers from the member organizations to use all the computing facilities of the other members, and by the constant exchange of documentation and informations between the ACPC members. Also, the author benefited from useful discussions and help from the other MasPar users, occasioned by the regular ACPC workshops and meetings. This shows that organization of cooperation frames between academic groups with similar interests (in our case: parallel computing) and starting joint projects involving several such groups is likely to add new potential to the research activity.

[0] $C \leftarrow$ `IntClasMul`(A, B) [Classical integer multiplication]
[1] $\quad C \leftarrow (0, \ldots, 0)$ [$m + n$ positions]
[2] $\quad$ **for** $i = 0, 1, \ldots, n - 1$ **do**
[3] $\qquad carry \leftarrow 0$
[4] $\qquad$ **for** $j = 0, 1, \ldots, m - 1$ **do**
[5] $\qquad\quad (carry, c_{i+j}) \leftarrow c_{i+j} + b_j * a_i + carry$
[6] $\qquad$ **if** $carry \neq 0$
[7] $\qquad\quad$ **then** $c_{i+m} \leftarrow c_{i+m} + carry$

Figure 1: Classical multiprecision multiplication

1 Sequential classical multiplication

1.1 Integers

The input to the multiplication algorithm consists of two multiprecision integers A, B represented as lists of positive digits in radix $\beta = 2^{29}$, least-significant digits first. The output is the multiprecision integer $C = A * B$.

Suppose:

$$A = a_0 + a_1 * \beta + \ldots + a_{n-1} * \beta^{n-1},$$
$$B = b_0 + b_1 * \beta + \ldots + b_{m-1} * \beta^{m-1},$$
$$C = c_0 + c_1 * \beta + \ldots + c_{n+m-1} * \beta^{n+m-1}.$$

Then

$$C = (a_0 + a_1 * \beta + \ldots + a_{n-1} * \beta^{n-1}) * B,$$

that is:

$$C = (a_0 * B) + (a_1 * B) * \beta + \ldots + (a_{n-1} * B) * \beta^{n-1}.$$

Hence, one can initialize $C = (0, \ldots, 0)$, and then, in n steps $(0 \leq i \leq n - 1)$, one can compute the lists $a_i * B$ and add them to C, starting at the position of c_i. At each individual step a *carry* is produced, which must be added to the next digit of C.

The description of the algorithm is presented in Fig.1.

Note that the addition in line [7] is not really necessary here. Indeed, at the end of the i^{th} step, C contains the product $(a_0 + a_1 * \beta + \ldots + a_{i-1} * \beta^{i-1}) * B$, which is the product of an i digits integer by an m digits integer. This product has at most $i + m$ nonzero digits, hence c_{i+m} is zero. However, this addition will be necessary when we embed this algorithm into polynomial multiplication, for computing $C + A * B$.

The theoretical time complexity of this algorithm depends on the innermost loop (line [5]), which is performed $n * m$ times, hence

$$T_{\text{classic}} = O(n * m).$$

For balanced–length operands

$$T_{\text{classic}} = O(n^2).$$

1.2 Polynomials

The input to the multiplication algorithm consists of two integral univariate polynomials $\mathcal{A}, \mathcal{B}$, represented as lists of multiprecision integers, least-degree coefficients first. The output is the polynomial $\mathcal{C} = \mathcal{A} * \mathcal{B}$.

[0] $\mathcal{C} \leftarrow$ `PolyClasMul`$(\mathcal{A}, \mathcal{B})$ [Classical polynomial multiplication]
[1] $\quad \mathcal{C} \leftarrow (0, \ldots, 0)$ [$M + N - 1$ positions]
[2] $\quad$ **for** $I = 0, 1, \ldots, N - 1$ **do**
[3] $\quad\quad$ **for** $J = 0, 1, \ldots, M - 1$ **do**
[4] $\quad\quad\quad C_{I+J} \leftarrow C_{I+J} + B_J * A_I$

Figure 2: Classical multiplication of polynomials

Let us denote

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

Then

$$\mathcal{C} = (A_0 + A_1 * x + \ldots + A_{N-1} * x^{N-1}) * \mathcal{B},$$

that is

$$\mathcal{C} = (A_0 * \mathcal{B}) + (A_1 * \mathcal{B}) * x + \ldots + (A_{N-1} * \mathcal{B}) * x^{N-1}.$$

Similarly to the case of integers, one obtains the algorithm presented in Fig.2.

The innermost loop of this algorithm performs the operation $C + A * B$ over long integers, which is exactly what `IntClasMul` does, when the initialization in line [1] is removed.

The innermost loop (line [4]) is performed $N * M$ times, and it requires $O(n * m)$ steps, where n, m are the maximum lengths of the coefficients of $\mathcal{A}, \mathcal{B}$. Hence

$$T_{\text{classic}} = O(N * M * n * m).$$

For balanced–length operands

$$T_{\text{classic}} = O(N^2 * n^2).$$

2 Systolic algorithms

2.1 The MasPar computer

In designing the parallel algorithm, we make use of the specific architecture of MasPar. We present here only those features which are relevant for our approach. More details can be found in [MasPar 90, 92].

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.

```
int a;
plural int b, c;
...
a = a*5;
if(b==0)
   c = c/2;
```

Figure 3: A sample MasPar program

- 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", and then they execute nothing. The "masking" is done by using C language conditional constructs. For instance, if the program is like in Fig.3, then `a` is a *singular* variable, and `b, c` are `plural` variables. Hence, "`a = a*5;`" is executed on ACU, and then "`c = c/2;`" is executed on those PE's on which `b` is zero (they are called *active* PE's). The other PE's are "masked" and execute nothing at this step.
- 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 more delicate, because `b` might have different values on different processors. One may 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[x][y].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. A similar result can be obtained with "`b = xnetE[1].c;`". Any of the 8 directions can be used (horizontally, vertically, diagonally). At the PE array borders the data wraps around in a torus like fashion.

2.2 Integers

Let us modify the algorithm `IntClasMul` such that the inner loop can be executed in parallel. In the present form of the algorithm, the inner loop is inherently sequential, because each step uses the *carry* produced by the previous step. Therefore, we shall devise a different scheme for *carry* propagation. We use 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

```
C ← IntParMul(A, B) [Parallel integer multiplication]
    C, Y ← (0, ..., 0) [m + n positions]
    for i = 0, 1, ..., n − 1 do [Stage 1: add and multiply]
        for j = 0, 1, ..., m − 1 in parallel do
            (y_{i+j+1}, c_{i+j}) ← c_{i+j} + b_j * a_i + y_{i+j}
    while(there are nonzero carries) do [Stage 2: absorb carries]
        for j = n, n + 1, ..., n + m − 1 in parallel do
            (y_{j+1}, c_j) ← c_j + y_j
        \overline{y}_n ← 0
```

Figure 4: Parallel integer multiplication

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

Figure 5: Systolic multiprecision multiplication, version 1.

is repeated until all the carries become zero.

The parallel algorithm is presented in Fig.4

One can see that m processors are necessary to implement this algorithm. 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.

The first version of the systolic algorithm (Fig.5) is a slight modification of this parallel algorithm. In order to make SIMD implementation easier, we emphasize the local variables on each processor. These variables 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]. We also introduce 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.

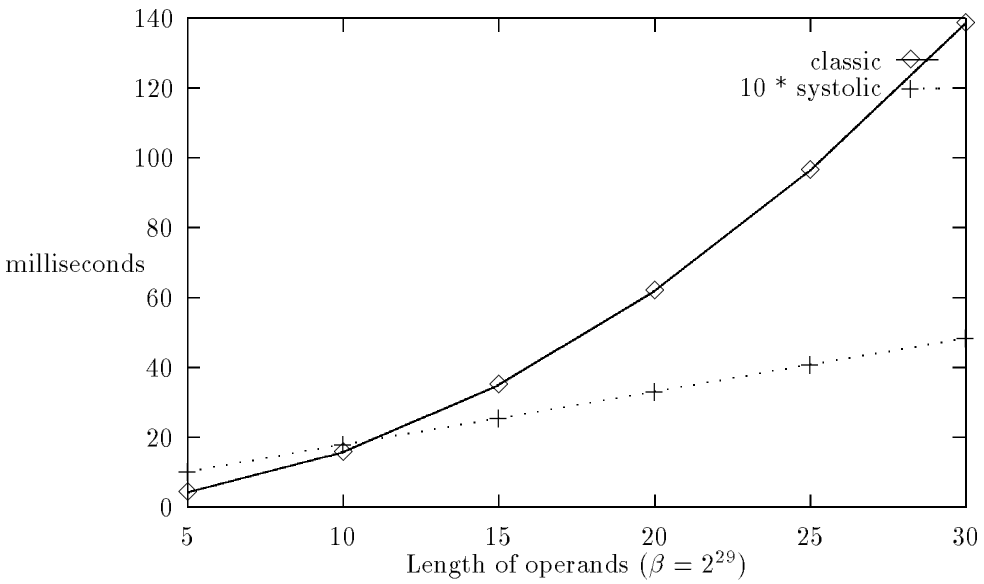

Figure 6: Comparative timings for multiprecision multiplication.

Let us find the time complexity of this algorithm. 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:

$$T_{\text{systolic}} = O(n).$$

In fact, for large β, the **while** loop [10] usually requires only 2 steps. Indeed, after the first step, the carries can be at most 1. The probability that more than 1 step is necessary to absorb them is less than the probability that at least one of the elements of C is $\beta - 1$, which probability is lower than m/β.

We implemented the algorithm **IntSysMul.1** on MasPar MP1, using only the first row of 32 processors. In order to evaluate the speed–up, we also implemented the algorithm **IntClasMul**, using only one processor.

Fig.6 shows the timings in milliseconds for the two algorithms (systolic timings are scaled by 10). The speed–up (systolic time over classical time), and the efficiency (speed–up over number of processors) are shown in Fig.7 and Fog.8 . The efficiency was computed taking into account only those processors which actually participate in the computation (that is m).

If C is to be used in subsequent computations, then it is useful to leave it in the array, instead of pipelineing/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. This second version of the systolic algorithm is presented in Fig.9.

The theoretical complexity is the same as before. However, in steps [5]–[7] 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). On the other hand, having C steady in the processors allows simple embedding of this algorithm into systolic algorithms for operation with polynomials.

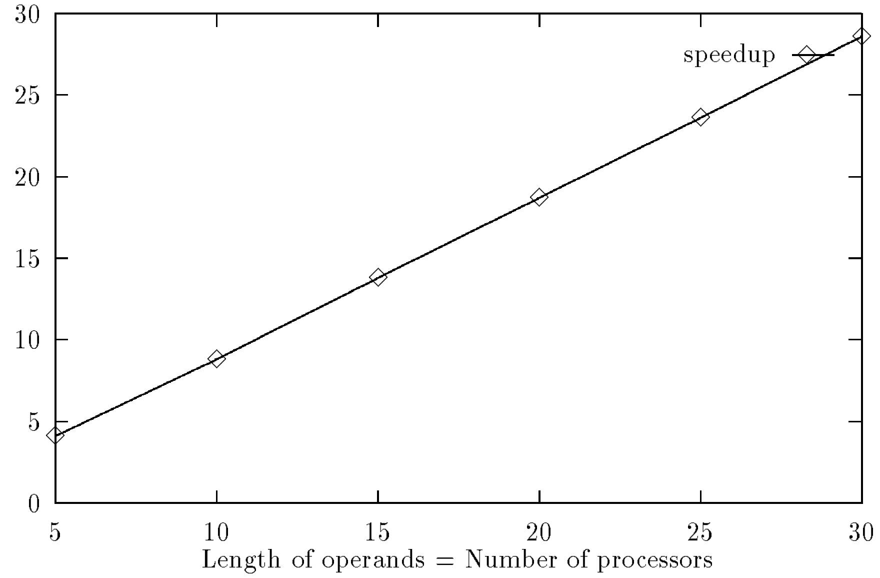

Figure 7: Speed–up of systolic multiprecision multiplication.

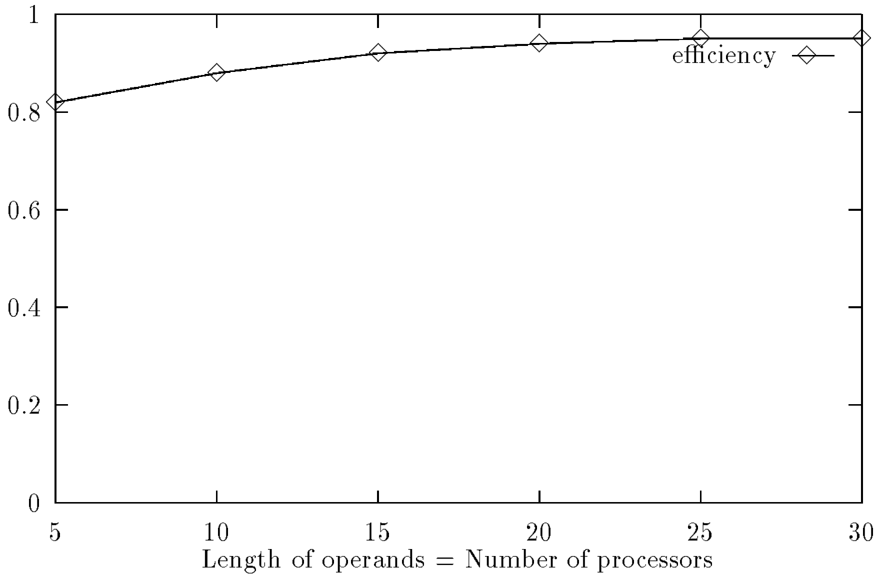

Figure 8: Efficiency of systolic multiprecision multiplication.

[0] $C \leftarrow$ `IntSysMul.2`(A, B) [Systolic integer multiplication, version 2]
[1] $\quad \overline{B}, \overline{C}, \overline{Y} \leftarrow (0, \ldots, 0)$ [$n + m + 1$ 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** $j = 0, 1, \ldots, n + m - 1$ **in parallel do**
[6] $\qquad\quad (\overline{y}_{j+1}, \overline{c}_j) \leftarrow \overline{c}_j + \overline{b}_j * a_i + \overline{y}_j$ [compute, shift $\overline{Y}$]
[7] $\qquad\quad \overline{b}_{j+1} \leftarrow \overline{b}_j$ [shift $\overline{B}$]
[8] $\quad$ **while globalor**$(\overline{Y})$ **do** [Stage 2: absorb carries]
[9] $\qquad$ **for** $j = 0, 1, \ldots, n + m - 1$ **in parallel do**
[10] $\qquad\quad (\overline{y}_{j+1}, \overline{c}_j) \leftarrow \overline{c}_j + \overline{y}_j$
[11] $\quad$ **for** $j = 0, 1, \ldots, n + m - 1$ **do** [extract C sequentially]
[12] $\qquad c_j \leftarrow \overline{c}_j$

Figure 9: Systolic multiprecision multiplication, version 2.

2.3 Polynomials

In this case there is no problem in parallelizing the inner loop of the algorithm `PolyClasMul`, using M computing units. As in the case of `IntSysMul.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 algorithm `IntSysMul.2`, if the initialization of C is removed from line [1] (Fig.9). Therefore, we shall use a row of processors for each of the above M computing units. Overall, we need 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 right as required by the algorithm `IntSysMul.2`, 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. Their local variables are zero all the time. 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.10).

Let us compute the 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. In order to evaluate the speed–up, we also implemented the algorithm `IntClasMul`, using only one processor.

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

Figure 10: Systolic polynomial multiplication.

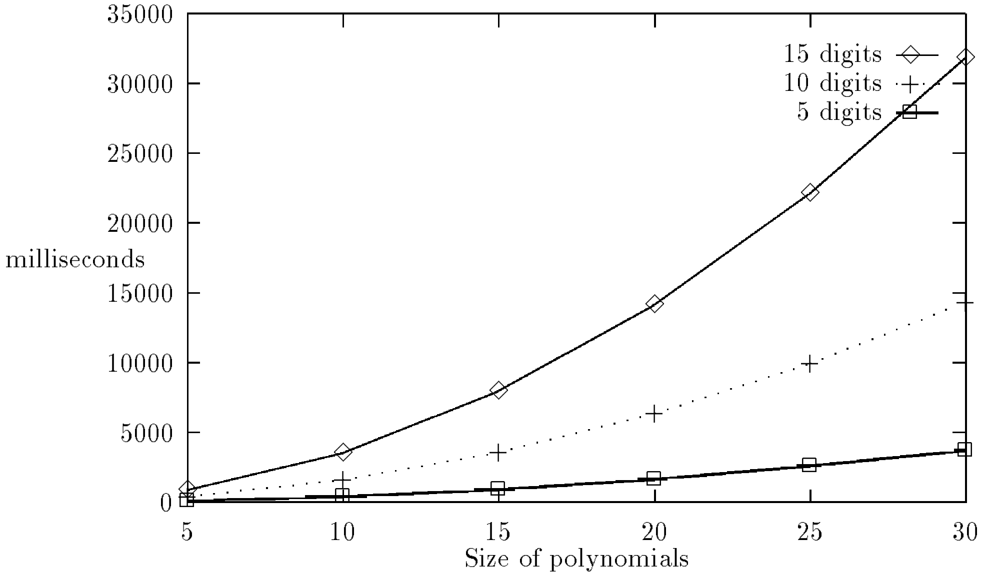

Figure 11: Timings of classical polynomial multiplication.

Figures 11, 12, 13 and 14 show the timings, the speed–up, and the efficiency. The efficiency was computed taking into account only those processors which actually participate in the computation (that is $M * (n + m)$).

Conclusions

The experimental curves match well the theoretical expectations:

- quadratic time for classic algorithm;
- linear time for systolic algorithm;
- linear speed–up;
- constant efficiency.

The high efficiency obtained for long integer multiplication demonstrates the effectiveness of the systolic parallelization approach. Even in the case of polynomial multiplication, the efficiency approaches the 50% theoretical limit, due to the design of **`IntSysMul.2`**. We believe this efficiency is also quite good in this case, considering the high number of processors used and the inherent limitations of the SIMD model.

When considering the problem from a strictly practical point of view, one should interpret this results with caution. Namely, measuring the speed–up against the running time of the classic algorithm on one PE gives insight about the efficiency of the algorithm, but not about the practical speed–up which can be obtained for a real-life application. We timed the classic algorithm on a fast sequential machine (DECstation 5200), using a good optimizing compiler (GNU C), and for practically the same code we obtained a speed–up of 80 to 100 over the PE runs. This means that only a fraction of the speed–up obtained by running on the MasPar are still valid from the pure practical point of view.

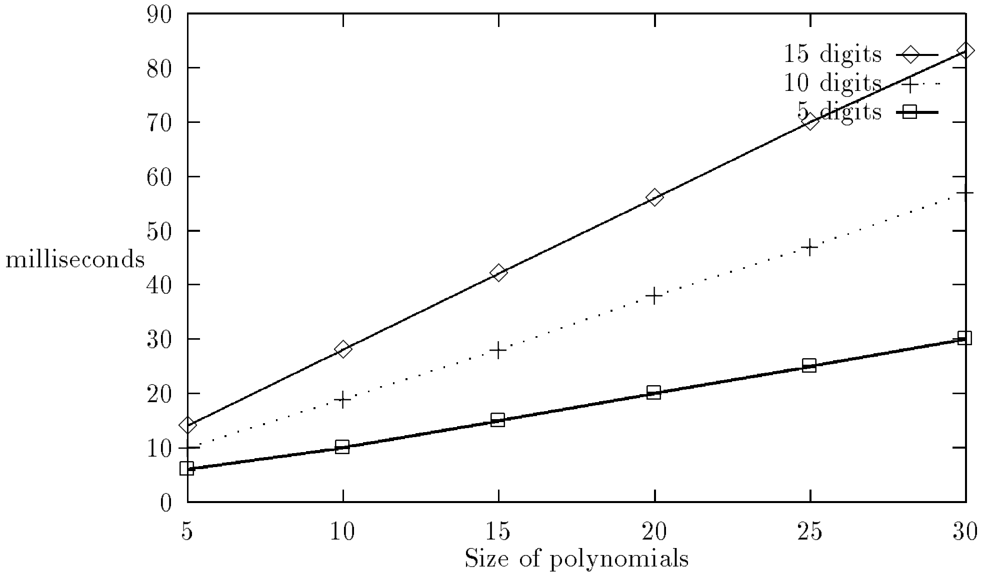

Figure 12: Timings of systolic polynomial multiplication.

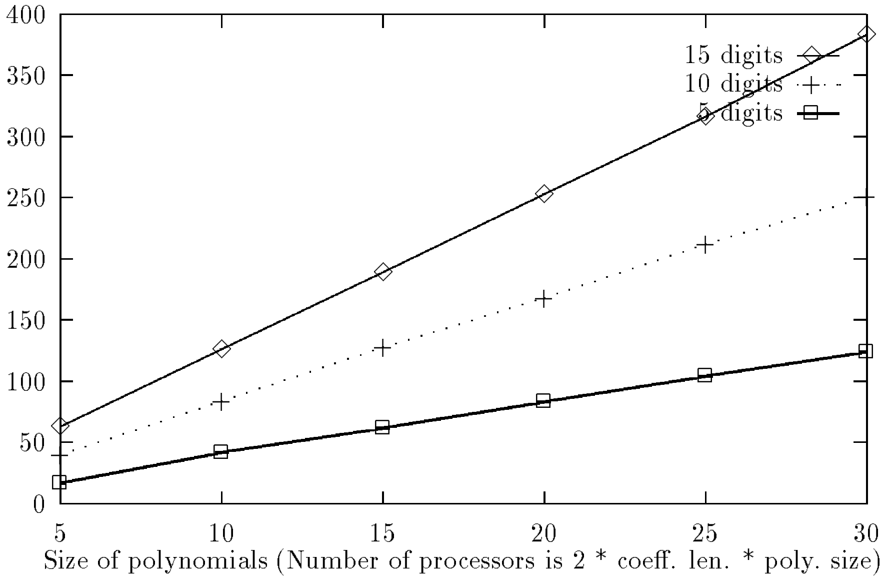

Figure 13: Speedup of systolic polynomial multiplication.

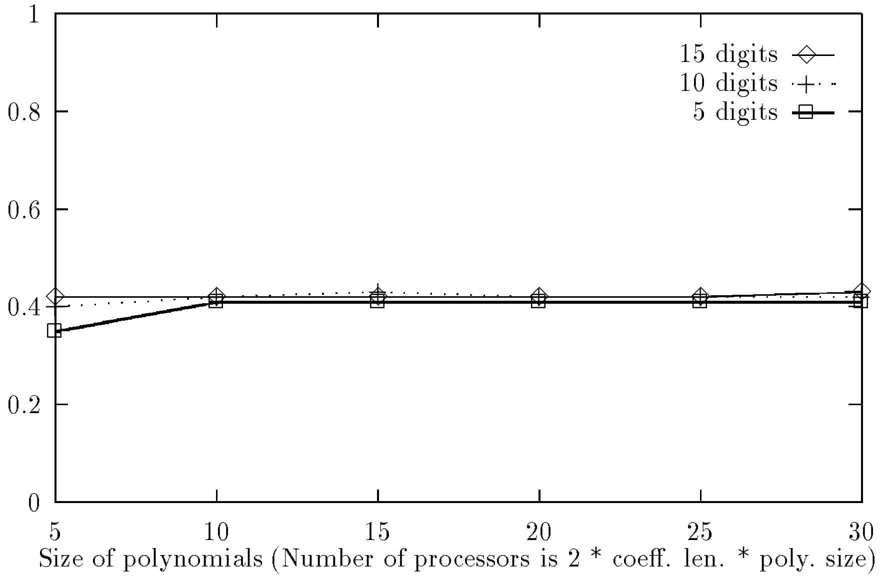

Figure 14: Efficiency of systolic polynomial multiplication.

References

[Atrubin 65] A. J. Atrubin — *A one–dimensional iterative multiplier*, IEEE Trans. C-14 (1965), 394–399.

[Beadsworth 81] R. Beadsworth — *On the application of array processors to symbol manipulation*, Proc. of 1981 ACM symp. on Symbolic and Algebraic Computation, ACM 1981, 126–130.

[Brent, Kung 83] R. P. Brent, H. T. Kung — *Systolic VLSI arrays for linear-time GCD computation*, in V. Anceau, E. J. Aas (eds) – *VLSI'83*, Elsevier (North-Holland), 1983, 145 – 154.

[Buchberger 85] B. Buchberger — *Gröbner Bases: An algorithmic method in polynomial ideal theory*, in N. K. Bose (ed.) – *Multidimensional Systems Theory*, D. Reidel Publishing Co., 1985.

[Buchberger, Jebelean 92] B. Buchberger, T. Jebelean — *Parallel rational arithmetic for Computer Algebra systems: Motivating experiments*, RISC–Linz Report 92-29, May 1992.

[Jebelean 92a] T. Jebelean — *An algorithm for exact division*, RISC–Linz Report 92-35, May 1992; submitted to Journal of Symbolic Computation.

[Jebelean 92b] T. Jebelean — *Systolic algorithms for exact division*, submitted to PARS Workshop, Dresden, April 1993.

[Jebelean 92c] T. Jebelean — *Long integer multiplication by cellular automata: An annotated bibliography*, in progress.

[Kung 81] H. T. Kung — *Use of VLSI in algebraic computation: some suggestions*, Proc. of 1981 ACM symp. on Symbolic and Algebraic Computation, ACM 1981, 218–222.

[MasPar 90] *MP–1 family data–parallel computers*, MasPar Corp., 1990.

[MasPar 92] *MasPar data–parallel programming languages*, MasPar Corp., 1990.

[Weeks 88] D. Weeks — *Adaptation of SAC-1 algorithms for a SIMD machine*, in J. Della Dora, J. Fitch (eds.) – *Computer Algebra and Parallelism*, Academic Press, 1988, 167–177.