Parallel Rational Arithmetic for Computer Algebra Systems: Motivating Experiments *†

B. Buchberger, T. Jebelean ‡
RISC-LINZ

—

Research Institute for Symbolic Computation

Johannes Kepler University, A4040 Linz, Austria (Europe)

Tel: +43 (7236) 3231-20 Fax: +43 (7236) 3338-30

e-mail: `buchberg@risc.uni-linz.ac.at`

e-mail: `tjebelea@risc.uni-linz.ac.at`

—

Abstract

We report on a preliminary statistical analysis of algebraic computations that shows the dominating effect of integer/rational number arithmetic in certain time-consuming computations. We observed that, with increasing length of the input coefficients, nearly all the time spent in typically large algebraic computations is actually consumed by integer/rational number operations. For example, in Gröbner bases computation, for 5 digit coefficients, the percentage is already 86%, for 50 digit coefficients it grows to 99.7%. Also, the absolute time increase of 40 times is exclusively due to integer/rational operations.

Therefore we propose to design and implement a dedicated parallel hardware unit for rational number arithmetic for computer algebra systems. Attempts at long integer parallel arithmetic for cryptography have already been made. However, parallel arithmetical units for the purpose of computer algebra will most probably need a complete re-design.

*In the frame of POSSO Project (Polynomial Systems Solving – ESPRIT III Basic Research Action 6846) tasks T1.5 (Bignums), T2.16 (Design of dedicated microprocessors), T2.17.(Parallel symbolic computing) and the Gröbner Bases Project of the Austrian Bundesministerium für Wissenschaft und Forschung.

†A preliminary version was presented at the ACPC (Austrian Center for Parallel Computation) workshop in Weinberg, Austria, April 27 – 28, 1992.

‡T. Jebelean was supported by a fellowship from the Austrian Ministry of Science and Research.

Introduction

The need for *exact computations* is becoming more and more acute both in theoretical Mathematics and Computer Science and in industrial applications of these fields. A large collection of advanced computer algebra algorithms has already been created, which practically covers all the necessities (see e.g. [Buchberger, Collins, Loos 83]). Most of these algorithms are implemented in computer algebra systems available on all popular computers. However, the day-to-day use of these algorithms for actual problems occurring in scientific research and design has a major draw-back: the computing time spent in solving almost any non-trivial problem is so long, that in most cases the use of computer algebra systems is practically impossible. Therefore, any significant speed-up of computer algebra systems will result in a significant expansion of the area of tractable problems.

The *purpose of our research* is to reduce the time spent in large algebraic computations significantly by implementing a special purpose parallel hardware unit for integer/rational arithmetic.

The opportunity of carrying out this project relies on whether a significant speed-up of the computer algebra systems could be achieved by only speeding up the integer/rational arithmetic component of the system. Obviously this depends on the proportion of time spent within this component. Also, by investigating the growth of the total computation time when the length of the numbers involved increases, one may obtain a good judgement about the possible speed up.

The experiments were done using `saclib` (the C-implementation by Hoon Hong (RISC 1991) of the well known computer algebra system SAC-2 [Collins 80]) on the Digital DEC-station 5200 (MIPS RISC architecture).

Analyzing Gröbner bases computation and related algorithms we observed 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 on the account of integer/rational operations.

The experiments used various different techniques for measuring the computation time and for tracing the running algorithms:

- system clock;
- repeated call of the sub-algorithm measured;
- book-keeping of input arguments for separate evaluation;
- run-time tracing of routine-calling, time spent and input arguments.

The results were checked for correctness by comparing the output of different methods.

The measurements were done at different algorithmic levels of the computer algebra system:

- Long integer multiplication (straightforward digit by digit) takes from 0.66 milliseconds (100 decimal digits) to 55 milliseconds (1000 digits). For this range no speed up is possible by using algorithms with better theoretic complexity.

- Important integral polynomial algorithms as *factorization* (IPFAC) and *remainder sequence* (IPSPRS) heavily rely on long integer multiplication (IPROD) and division (IQR). The combined time-weight of IPROD and IQR increases from 25% to 85% within IPFAC and from 33% to 90% within IPSPRS when increasing the input coefficient length from 3 to 150 decimal digits. Other algorithms (polynomial GCD, polynomial resultant), which are implemented using modular arithmetic, spent only less than 10% of the time in long integer arithmetic.
- Some high level algorithms, as Gröbner bases, spent most of the computation time within integer/rational arithmetic. For instance, when increasing the coefficient length from 1 to 10 decimal digits, the proportion of rational operations grows from 62% to 98%, and the total computation time from 0.8 to 21 seconds. Also, by tracing in detail the behavior of the algorithm for various inputs, we observed that the dramatic increase of the total computation time is exclusively due to the increase of the input coefficients length.

1 Timing Techniques

This section presents the techniques used for timing the computations.

Unix system provides several functions for time tracing of programs: `prof`, `gprof`, `profil`, `monitor`. However, they are not well suited for our purpose:

- The results of these measurements are only "statistically" correct, since the timings are done by inspecting the program counter at regular time intervals (1/60 seconds).
- As in any other computer algebra system, `saclib`'s basic operations are list manipulations, which are called by all the other (e.g. integer arithmetic) algorithms. The use of `prof` will just indicate that most of the time is spent within these list operations. Even the use of `gprof`, which tries to give for each routine the total time spent including the subroutines, is not useful, because `gprof` computes these times by using the static routine calling graph of the program. Or, in `saclib`, all the routines share the same list operations, hence this approach does not give a significant result.
- It is not possible using these tools to hide certain routines which are not of interest for us.

Of course, one could still use these tools by making significant changes in the source code. We preferred, however, to implement our own tools, which requires the same programming effort but gives us more control and flexibility.

1.1 System clock

For all timings we used the `saclib` routine ACLOCK, which gives the CPU time exclusive garbage collection time. This time is obtained via the Unix call `times`, which unfortunately has an accuracy of 17 milliseconds only. This is too coarse for our needs: for instance, multiplication of integers up to 500 decimal digits takes less than 17 milliseconds. Even getting access to the native clock-tick (4 milliseconds) of the DEC-station 5200, which is used by `profil`, does not significantly increase the resolution (240 decimal digits integers are multiplied in 4 milliseconds).

Therefore, one cannot use a straightforward approach in order to measure the time taken by various sub-algorithms of a running program, hence we had to use various other timing schemes.

In one case (run-time tracing of routine calling) we used the ACLOCK time, as approximative as it is, for getting the time spent in each routine. This is because, on one hand, a more exact timing in this case would be too difficult to implement, with respect to our main research theme. On the other hand, the results obtained are accurate enough for our purpose. This tracing was done for a high level algorithm (Gröbner bases computation), which calls many times (up to 1000) the basic arithmetic routines, and we are interested mainly in the sum of all the times for each routine, rather than the individual time of each call. One can use statistical arguments to prove that in such a situation the sum of the coarse measured times is a good approximation of the real sum. We also checked the correctness of the results by repeated runs and by sample cross-checking with other timing methods.

1.2 Repeated call

In order to find the computation time needed for long integer multiplication of various length integers, we just repeated the computation until the total time was significantly higher then the clock resolution.

Also, in order to find the time took by a specific integer operation, say long integer division, within a higher level algorithm, say polynomial factorization, we modified the source code of the integer division routine, by inserting a supplementary call of integer division with the same arguments. Hence, the obtained program calls executes two times each division. The difference between the running time of this program and the running time of the initial version gives the time spent within division.

1.3 Book-keeping

Perhaps the most accurate measurement method is to keep a trace of the arguments of each call, and then use this trace to compute separately the time spent with each call. However, this process is quite tedious and time consuming. We made use of it only for sample cross checking of the results obtained by other methods.

1.4 Run-time tracing

This method was used for tracing Gröbner bases computation. We developed a set of routines for tracing and timing, and we inserted calls to these routines at the beginning and at the end of each function of the algorithm which was subject to tracing. The timing provided by the tracing routines is corrected such that it does not include the time spent within the tracing routines themselves. This timing is only "statistically" correct, as we mentioned at a previous subsection. However, the method allows to spot the time consuming parts of the algorithm quickly and to trace the interesting routines, because it provides flexible ways of turning on/off the tracing, both at compile time and at run time. Also, it allows the tracing of the arguments of each function call, in order to find out which characteristics of the arguments have a higher impact on the computing time.

```
BETA-  Milliseconds
digits  Classic/Karatsuba

  2:  0.049  0.055 o
  4:  0.135  0.143 o
  6:  0.265  0.275 o
  8:  0.440  0.447 o
 10:  0.657  0.663  o
 12:  0.916  0.933  o
 14:  1.217  1.225  o
 16:  1.567  2.455   o
 18:  1.967  2.856   *+
 20:  2.389  3.333   *+
 22:  2.867  3.833     *+
 24:  3.400  4.333     *+
 26:  3.943  4.917      *+
 28:  4.555  5.500       *+
 30:  5.250  6.335       *+
 32:  5.917  9.362        *  +
 34:  6.695 10.082         *  +
 36:  7.443 10.887         *   +
 38:  8.280 11.663          *   +
 40:  9.167 12.557           *    +
 42: 10.000 13.387            *   +
 44: 10.943 14.220             *   +
 46: 12.250 15.165              *   +
 48: 13.085 16.085               *   +
 50: 14.335 17.085                *   +
 52: 15.335 18.085                 *   +
 54: 16.420 18.920                  *   +
 56: 17.750 20.080                     * +
 58: 19.085 21.085                      * +
 60: 20.420 23.920                       *   +
 62: 21.585 28.500                         *       +
 64: 23.170 31.670                          *         +
 66: 24.670 33.170                            *        +
 68: 25.830 34.510                             *         +
 70: 27.670 35.840                                *       +
 72: 29.340 37.330                                 *       +
 74: 30.830 38.500                                   *       +
 76: 32.500 39.830                                     *      +
 78: 33.830 41.000                                      *       +
 80: 35.670 42.830                                        *       +
 82: 39.000 43.670                                           *     +
 84: 39.670 45.330                                            *     +
 86: 41.000 46.680                                             *      +
 88: 43.320 48.320                                                *    +
 90: 44.660 49.680                                                  *    +
 92: 47.660 51.320                                                     *  +
 94: 49.000 52.680                                                      *   +
 96: 51.000 54.000                                                        *  +
 98: 53.340 55.340                                                          *  +
100: 55.320 57.340                                                             * +
```

Table 1: Multiplication time for classic and Karatsuba's algorithm

2 Results

2.1 Long integer multiplication

Because long integer multiplication is one of the most costly operations, we first measured the time spent by this algorithm and the possibilities of speed up. The present algorithm uses the straightforward digit by digit multiplication. Plotting the multiplication time against operands length we obtained the expected parabola (0.7 milliseconds for 100 decimal digits, 14,3 ms for 500 dd, 55.3 ms for 1000 dd). We also measured the time taken by Karatsuba's algorithm [Karatsuba, Ofman 62], finding that it tends to become lower than the initial algorithm only from 1000 decimal digits upwards. Table 1 shows the comparative timings of the two algorithms. A "BETA-digit" is a one digit number in the radix used by `saclib` (2^{29}), and roughly corresponds to 9 decimal digits.

As for the FFT algorithm [Schönhage, Strassen 71], the theoretical time-complexity is favorable only for $n > 200$. In our case (`saclib` uses 2^{29} as radix) this means about 1800 decimal digits, which is outside our range of interest.

Also, we found out that the time needed for accessing the operands and building the result in the memory takes 1/10 (for 100 decimal digits) to 1/100 (for 1000 decimal digits) of multiplication time. This may be seen as a higher bound of possible speed up using an external hardware device for multiplication only. The detailed results are listed in Table 2.

```
BETA-  Milliseconds  Fraction
digits  access/total

  2:  0.012  0.049   4  *
  4:  0.022  0.135   6   *
  6:  0.031  0.265   8     *
  8:  0.040  0.440  11       *
 10:  0.050  0.657  13        *
 12:  0.058  0.916  16          *
 14:  0.068  1.217  18            *
 16:  0.078  1.567  20             *
 18:  0.087  1.967  23               *
 20:  0.096  2.389  25                *
 22:  0.106  2.867  27                  *
 24:  0.114  3.400  30                    *
 26:  0.125  3.943  32                     *
 28:  0.133  4.555  34                      *
 30:  0.143  5.250  37                        *
 32:  0.151  5.917  39                          *
 34:  0.161  6.695  42                            *
 36:  0.172  7.443  43                            *
 38:  0.180  8.280  46                              *
 40:  0.190  9.167  48                                *
 42:  0.200 10.000  50                                 *
 44:  0.207 10.943  53                                   *
 46:  0.219 12.250  56                                     *
 48:  0.227 13.085  58                                      *
 50:  0.235 14.335  61                                        *
 52:  0.244 15.335  63                                          *
 54:  0.252 16.420  65                                           *
 56:  0.262 17.750  68                                             *
 58:  0.275 19.085  69                                              *
 60:  0.265 20.420  77                                                   *
 62:  0.290 21.585  75                                                  *
 64:  0.302 23.170  77                                                   *
 66:  0.307 24.670  80                                                     *
 68:  0.320 25.830  81                                                      *
 70:  0.327 27.670  85                                                        *
 72:  0.340 29.340  86                                                         *
 74:  0.347 30.830  89                                                           *
 76:  0.357 32.500  91                                                            *
 78:  0.367 33.830  92                                                             *
 80:  0.373 35.670  96                                                                *
 82:  0.383 39.000 102                                                                    *
 84:  0.397 39.670 100                                                                  *
 86:  0.400 41.000 102                                                                    *
```

```
 88:  0.411 43.320 105                                                            *
 90:  0.422 44.660 106                                                            *
 92:  0.433 47.660 110                                                              *
 94:  0.439 49.000 112                                                               *
 96:  0.450 51.000 113                                                                *
 98:  0.467 53.340 114                                                                 *
100:  0.467 55.320 119                                                                   *
```

Table 2: Fraction of time taken by accessing operands/result

2.2 Algorithms on integral polynomials

Typical subalgorithms used by most computer algebra applications were also subject to experiments investigating the proportion of time spent in integer arithmetic. The results show:

- Linear complexity algorithms (as summation, difference, negation) consume an insignificant time of the computation (up to 5%).
- Multiplication (IPROD) and division (IQR) consume a significant part of the computation in some basic algorithms as IPFAC (integral polynomial factorization) and IPSPRS (integral polynomial "subresultant" polynomial remainder sequence). The proportion of time spent in IPROD and IQR increases dramatically when the length of the coefficients grows (see Fig.1 and Fig.2).
- Integer arithmetic consumes an insignificant time (up to 10%) within the algorithms which rely on modular arithmetic, as IPRES (integral polynomial resultant) and IPGCD (integral polynomial GCD). Actually, these algorithms use modular arithmetic because the integer arithmetic routines are so time consuming. It may be that by using a high speed unit for integer multiplication/division, the old straightforward versions of these algorithms will have faster implementations then the modular ones.

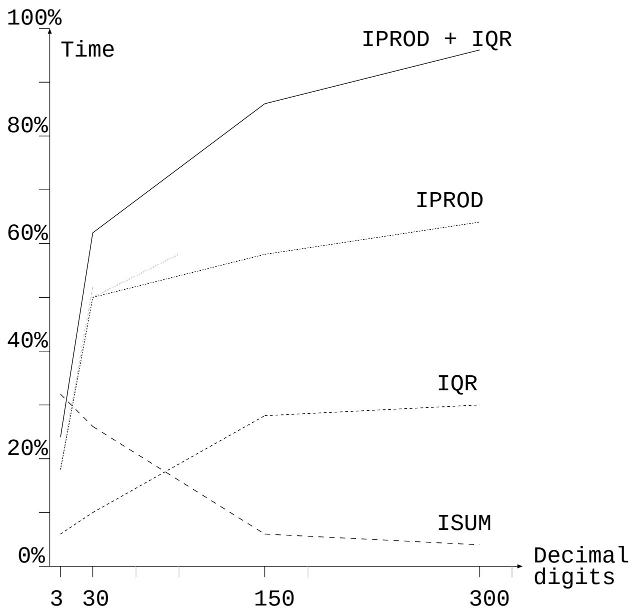

Figure 1: Integer arithmetic within IPFAC

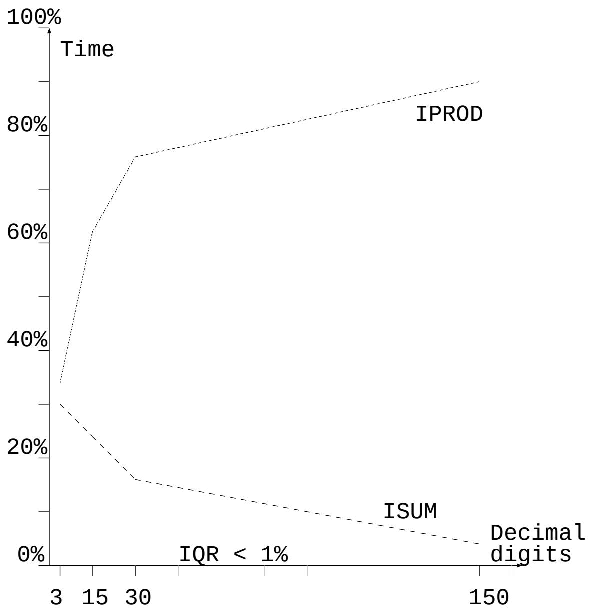

Figure 2: Integer arithmetic within IPSPRS

The following tables present the details of the measurements. The columns have the following meaning:

CoeffLength/Dec : input coefficients length in decimal digits (approximative);

CoeffLength/Bin : input coefficients length in binary digits (exact);

IPROD : time percentage spent in long integer multiplication;

IQR : time percentage spent in long integer division (quotient and remainder);

ISUM : time percentage spent in long integer summation;

Rest : time percentage spent in operations other than integer arithmetic;

P+QR : sum of IPROD and IQR;

Var/# : number of variables of the input polynomials;

Degs : polynomial degree in each variable;

Dense : density of input polynomials;

Raw data : absolute timings;

Repeat : number of repeated computations, with different inputs of same characteristics.

The absolute timings were obtained by the "double-call" method. That is, the timings in the column **Raw data/total** are of the original `saclib` program, and the timings in the column **Raw data/IPROD**, for instance, are obtained by replacing the original IPROD routine with a double call to it, then timing the obtained program on the same inputs and subtracting the original time.

Table 3 presents the results concerning the time spent by integer arithmetic within IPFAC (integral polynomial factorization). Two polynomials with the indicated characteristics were randomly generated, then multiplied, and then the time took by factorization of the product was measured.

CoeffLength		IPROD	IQR	ISUM	Rest	P+QR	Var	Degs	Dense
Dec	Bin	%	%	%	%	%	#		
3	10	18.5	6.3	32.6	43.0	24.9	3	2,2,2	1/2
30	100	51.5	10.4	25.9	12.1	61.9	3	2,2,2	1/2
150	500	57.3	28.6	6.9	7.3	85.8	3	2,2,2	1/7
300	1000	64.9	30.1	4.8	0.2	95.0	3	2,2,2	1/10

Raw data (milliseconds)

IPROD	IQR	ISUM	rest	P+QR	total	repeat
47,416	16,252	82,396	110,133	63,688	256,197	*5
272,636	55,323	137,004	64,052	327,959	529,015	*3
100,834	50,300	12,200	12,781	151,134	176,115	*2
279,381	129,546	20,712	714	408,927	430,353	*1

Table 3: Integer arithmetic within IPFAC

Table 4 presents the results concerning the time spent by integer arithmetic within IPSPRS (integral polynomial "subresultant" polynomial remainder sequence). Two polynomials P1, P2 with the indicated characteristics were randomly generated, then the time took by IPSPRS(P1,P2) was measured.

Table 5 presents the results concerning the time spent by integer arithmetic within IPRES (integral polynomial resultant). Two polynomials P1, P2 with the indicated characteristics were randomly generated, then the time took by IPRES(P1,P2) was measured.

Length Dec	Length Bin	IPROD %	IQR %	ISUM %	Rest %	P+QR %	Var #	Degs	Dense
3	10	33.2	--	29.6	37.3	33.2	3	3,3,3	1
15	50	62.8	--	23.1	14.1	62.8	3	3,3,3	1
30	100	75.1	--	16.0	8.9	75.1	3	3,3,3	1
150	500	90.1	--	3.0	6.3	90.1	3	3,3,3	1

Raw data (milliseconds)

IPROD	IQR	ISUM	rest	P+QR	total	repeat
10,918	--	9,734	12,282	10,918	32,934	*10
82,796	--	30,482	18,524	82,796	131,802	*10
127,269	--	27,084	15,011	127,269	169,364	*5
525,178	--	17,650	36,525	525,178	579,353	*1

Table 4: Integer arithmetic within IPSPRS

CoeffLength Dec	CoeffLength Bin	IPROD %	IQR %	ISUM %	Rest %	P+QR %	Var #	Degs	Dense
3	10	0.4	1.3	0.0	98.2	1.7	3	4,4,4	1/2
30	100	0.8	4.2	1.4	93.7	4.9	3	3,3,3	1/2
150	500	1.4	1.7	5.9	91.0	3.0	3	3,3,3	1/2
300	1000	2.6	0.4	8.8	88.6	2.5	3	2,2,2	1/2

Raw data (milliseconds)

IPROD	IQR	ISUM	rest	P+QR	total	repeat
1,034	2,951	155	231,358	3,985	235,498	*5
2,254	11,749	3,950	265,063	14,003	283,016	*5
5,113	6,229	22,362	342,631	11,342	376,335	*1
5,983	1,051	24,502	245,781	7,034	277,317	*3

Table 5: Integer arithmetic within IPRES

2.3 Gröbner bases computation

High level algorithms which are of use in typical applications are the most interesting for us. Such an algorithm is Gröbner bases computation [Buchberger 65, Buchberger 85], which is used in solving systems of polynomial equations, quantifier elimination, and many others. We performed experiments using the implementation by B. Buchberger and W. Windsteiger [Buchberger 90, Windsteiger 90] on top of `saclib`. The Gröbner bases algorithm was applied upon two dense polynomials in two variables, with the highest power products x^3y^2 and x^2y^3 (total degree lexicographic ordering, $y < x$).

When increasing the input coefficients length from 1 to 50 decimal digits, the computation time increases from 2 to 339 seconds, and the weight of rational arithmetic increases from 62% to 99.7%, as displayed in Fig.3 and Fig.4. Table 6 and Table 7 show the detailed timings (milliseconds).

The timings are obtained by calling ACLOCK (CPU time minus garbage collection) at the beginning and at the end of each traced routine. Since these timings are only "statistically" correct, we cross-checked them using the "double-call" method.

We also traced the different computations and compared the traces, observing that exactly the same steps are performed in each case, and the computation time can be split in:

- computation time independent on the coefficients length (operations with exponents, decisions of how the algorithm proceeds), which remains practically unchanged in all the examples (0.3 to 1.1 seconds);
- computation time dependent of the coefficients length (difference and product of coefficients), which increases very much as the length of the coefficients grows (from 0.5 to 331.1 seconds).

This proves that the computation duration increase of 170 times is exclusively due to the rational/integer arithmetic. By speeding up rational arithmetic, let's say, 20 times, a 20 times speed up of the Gröbner bases computation would be achieved.

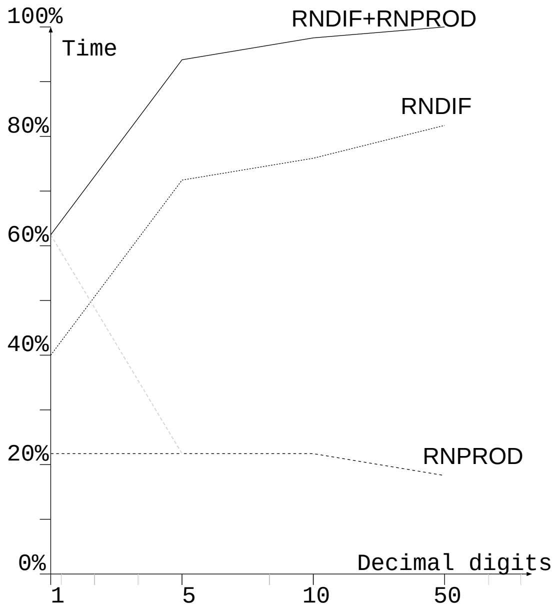

Figure 3: Rational arithmetic within Gröbner Bases

Length dec dig	RNPROD %	RNDIF %	Rest %	RNPROD+DIF %
1	22.1	40.3	37.6	62.4
5	21.6	72.8	5.6	94.4
10	22.5	75.3	2.2	97.8
50	17.5	82.2	0.3	99.7

Raw data (milliseconds)

RNPROD	RNDIF	Rest	RNPROD+DIF	total
184	336	313	520	833
1,638	5,536	426	7,174	7,600
4,800	16,063	466	20,863	21,329
58,062	273,096	1,142	331,158	332,300

Table 6: Rational arithmetic within Gröbner Bases

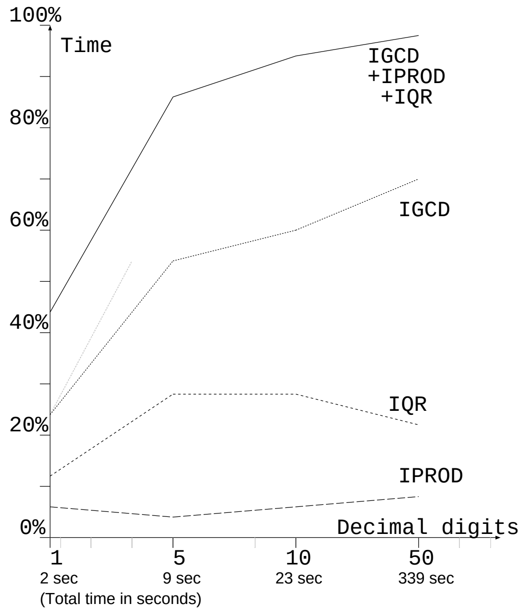

Figure 4: Integer arithmetic within Gröbner Bases

1 dec digs		5 dec digs		10 dec digs		50 dec digs		
total:	2040	total:	9160	total:	23150	total:	339082	
51%	1058	11%	1053	5%	1263	0%	1225	rest
3%	62	0%	84	0%	163	0%	518	INEG
1%	36	0%	59	0%	116	0%	534	ISUM
6%	134	4%	447	5%	1358	7%	23982	IPROD
12%	253	28%	2625	27%	6345	21%	74414	IQR
24%	495	53%	4890	60%	13903	70%	238407	IGCD
43%	882	86%	7962	93%	21606	99%	336803	IPROD+IQR+IGCD

Table 7: Integer arithmetic within Gröbner Bases

Table 8 shows the comparative analysis of the traces for 1 digit and 5 digit coefficients.

A trace contains the following information for each traced routine:

- the name of the routine at the beginning and at the end of the call;
- the "total" time spent within the routine, including the called routines;
- the "local" time spent within the routine, excluding the traced subroutines;
- the arguments of the routine;
- trace of *if* statements reached within the routine;
- trace of *while* statements reached within the routine, including the trace of each loop.

Choosing of the traced routines was done as follows:

- The routines whose behavior does not depend on the coefficients were not traced "inside" (called subroutines, *if* statements, *while* statements). Examples of such routines are:
  - `is-greater-mon`: comparison between two monomials with respect to the total-degree lexicographic order;
  - `product-monic-mon-dpl`: product of a power product and a polynomial ("shifting");
  - `criteria`: decides whether the S-polynomial of two polynomials should be computed.
- The routines whose behavior depends on the coefficients were also traced "inside", down to the level of rational arithmetic operations. Examples of such routines are:
  - `GB-reduce-all`: the main routine of GB algorithm;
  - `complete-normal-form-poly`: reduces a polynomial with respect to the existing base;
  - `difference-dpl`: difference of two polynomials.

The comparative analysis is done by a program which:

- Checks whether in the two traces the same routines are called in the same order, and the same *if* and *while* statements occur, with the same number of loops.
- Compares whether the arguments of each call are identical.
- Counts the number of calls of each routine and sums the total and local timings.
- Outputs the results in a table like the Table 8.

# Calls SameArgs	DiffArgs	Time in trace1 total	local	Time in trace2 total	local	Routine name
0	1	1538	0	15610	0	GB_reduce_all
0	1	0	0	0	0	normalized_ps
0	7	0	0	249	0	normalized_dpl
0	5	0	0	0	0	RNINV
0	5	0	0	249	0	product_coef_dpl
0	1073	202	202	3739	3739	RNPROD
1	0	0	0	0	0	labeled_ps
0	4	34	0	100	0	all_reduced
6	0	0	0	0	0	merge
2	0	17	0	49	0	normal_form_lp
21	0	0	0	0	0	unlabeled_ps
20	0	1488	0	15210	0	complete_normal_form_poly
0	382	1488	0	15210	0	partial_normal_form_poly
7	0	0	0	0	0	split
0	5	0	0	0	0	thinned_pairs
2	0	0	0	0	0	insert_lps
465	0	0	0	0	0	is_multiple_mon
2974	0	117	117	116	116	is_greater_mon
61	0	0	0	0	0	quotient_monic_mon
0	53	285	0	3607	0	product_mon_dpl
0	968	252	50	3590	84	product_mon
968	0	0	0	16	16	product_el
0	57	1102	0	11637	0	difference_dpl
978	0	820	34	11269	50	difference_mon
0	978	786	33	11219	33	RNDIF
0	1050	50	50	16	16	RNNEG
0	978	703	703	11170	11170	RNSUM
0	28	0	0	17	0	negative_dpl
71	0	0	0	0	0	CONC
0	4	0	0	0	0	updated_pairs
14	0	17	0	51	0	head_reduced_lp
0	14	17	0	51	0	head_reduced_poly
1	0	17	0	51	0	completely_reduced_lps
10	0	0	0	0	0	criteria
0	4	50	0	151	0	S_polynomial_poly
4	0	0	0	0	0	lcm_monic_mon
8	0	34	34	0	0	product_monic_mon_dpl
72	0	0	0	0	0	negative_mon

Table 8: Comparative trace analysis (Gröbner Bases for 1 vs 5 decimal digits)

3 Conclusions

Most probably one can find better techniques for more exact timings, and one can also find many other important computer algebra algorithms which could be relevant to this research. However, we believe that the results we obtained up to now already offer a sufficiently good motivation. That is, they prove the usefulness of designing and implementing a parallel hardware unit for integer/rational arithmetic, to be used by computer algebra systems.

The memory (lists) accessing time sets an inconvenient bound on the speed up for any arithmetic operation we might consider to be performed by an additional hardware unit. For instance, in long integer multiplication, 1/10 (for 100 digits) to 1/100 (for 1000 digits) of the computation time is spent for loading the operands and storing the result. Therefore, such an use is not worthwhile for linear time operations like integer summation or negation, and is not very efficient for other integer operations like integer multiplication or division. Rather, one might expect a very high speed up by implementing rational arithmetic, because then several integer operations can be performed, without having to access the main memory for the operands of each operation, but only for the initial/final data. Since cellular algorithms are well suited to pipelining, the intermediary results of the computation inside the special unit can be pipelined between the different modules. Also, one may implement cellular algorithms for specific operations needed in rational arithmetic, rather then implementing modules for the basic integer arithmetic only. For instance, the cellular multiplier described in [Katona 82] is also capable of computing $a * b + c * d$ in the same number of steps as necessary for computing $a * b$.

In [Jebelean 91], [Buchberger, Jebelean 91] we investigate the possibility of using cellular/systolic techniques for designing long integer arithmetic parallel hardware. The existence of several systolic algorithms for multiplication and division, ([Wallace 64], [Atrubin 65], [Dadda 65], [Habibi, Wintz 70], [Gex 71], [Brent, Kung 81], [Kung 81], [Katona 82], [Danielsson 84]), as well as the pipelining capabilities [Kung 82] and the versatility of cellular automata (changing small details in the cellular program can switch its functionality from one integer operation to another – see [Gex 71], [Kamal, Singh, Agrawal 75], [Preparata, Vuillemin 90]), enable us to believe that cellular/systolic techniques can be successfully applied to the design of an efficient parallel hardware unit for rational arithmetic.

Cellular techniques have already been successfully applied for designing special purpose hardware for image processing (e.g. [Zsoter, Legendi, Balazs 90]). Also, designing circuits for very long integer multiplication has been done in relation to cryptography (e.g. [Lippitsch, Posch 91]). However, according to our previous investigations [Buchberger, Jebelean 91], the existing cellular hardware is not suitable for the purpose of algebraic computations. Rather, an efficient implementation could eventually be obtained be a specific design only. It seems that no such attempt has not yet been done (see e.g. [Della Dora, Fitch 88]).

Modular arithmetic may also be an interesting candidate for systolic implementation (an attempt is presented in [Davenport, Robert 85]).

References

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

[Buchberger 65] B. Buchberger — *An Algorithm for Finding a Basis for the Residue Class Ring of a Zero-Dimensional Ideal (German)*, Ph.D. Thesis, Univ. of Innsbruck, 1965.

[Buchberger 85] B. Buchberger — *Gröbner Bases: An Algorithmic Method in Polynomial Ideal Theory*, in N. K. Bose (ed.) — *Multidimensional Systems Theory*, D. Reidel Publishing Company, 1985.

[Buchberger 90] B. Buchberger — *An Implementation of Gröbner Bases in Mathematica*, RISC Technical Report 1990.

[Buchberger, Collins, Loos 83] B. Buchberger, G. E. Collins, R. Loos — *Computer Algebra: Symbolic and Algebraic Computations*, Springer Verlag, 1983.

[Buchberger, Jebelean 91] B. Buchberger, T. Jebelean — *Systolic Algorithms in Computer Algebra: State of the Project*, RISC Technical Report 1991.

[Collins 80] G. E. Collins — *ALDES and SAC-2 Now Available*, SIGSAM Bull. 14/2, 19 (1980).

[Dadda 65] L. Dadda — *Some schemes for parallel multipliers* — Alta Frequenza 34(1965): 349–356.

[Danielsson 84] P. E. Danielsson — *Serial/parallel convolvers* — IEEE C-33(1984):652–667.

[Davenport, Robert 85] J. Davenport, Y. Robert — *VLSI and computer algebra: the G.C.D. example* — in J. Demongeot, E. Goles, M. Tchuente (eds) *Dynamical systems and cellular automata*, Academic Press, 1985.

[Della Dora, Fitch 88] J. Della Dora, J. Fitch (eds.) – *Computer Algebra and Parallelism* – Academic Press, 1988

[Gex 71] A. Gex — *Multiplier - divider cellular array* — Electronic Letters 7(1971):442–444.

[Habibi, Wintz 70] A. Habibi, P. A. Wintz — *Fast multipliers* — IEEE C-19(1970): 153–157.

[Jebelean 91] T. Jebelean — *Long Integer Multiplication by Cellular Automata: An Annotated Bibliography*, RISC Technical Report 1991.

[Kamal, Singh, Agrawal 75] A. K. Kamal, H. Singh, D. P. Agrawal — *A generalized pipelined array* — IEEE C-23(1975):533–536.

[Karatsuba, Ofman 62] A. Karatsuba, Yu Ofman — *Multiplication of multidigit numbers on automata* — Sov. Phys. Dokl. 7(1962): 595–596

[Katona 82] E. Katona — *Binary addition and multiplication in cellular space* — Acta Cybernetica 5(1982): 457–464.

[**Knuth 69**] D. E. Knuth — *The Art of Computer Programming. Vol. 2: Seminumerical Algorithms. Chapt. 4: Arithmetic*, Addison-Wesley 1969.

[**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.

[**Kung 82**] H. T. Kung — *Why systolic architectures?* — Computer (IEEE) 15(1982): 37–46

[**Lippitsch, Posch 91** P. Lippitsch, K. C. Posch, R. Posch — *Multiplication as Parallel as Possible*, ACPC Conference 1991.

[**Preparata, Vuillemin 90**] F. P. Preparata, J. E. Vuillemin — *Practical cellular dividers* — IEEE C-39(1990): 605-614.

[**Schönhage, Strassen 71**] A. Schönhage, V. Strassen — *Schnelle Multiplikazion grosser Zahlen* — Computing 7(1971): 281-292.

[**Wallace 64**] C. S. Wallace — *A suggestion for a fast multiplier* — IEEE C-13(1964): 14–17

[**Windsteiger 90**] W. Windsteiger — *An Approach to Object-Oriented Programming in C*, RISC Technical Report 1990.

[**Zsoter, Legendi, Balazs 90**] A. Zsoter, T. Legendi, G. Balázs — *Design and implementation of M1 cellprocessor* — Submitted to CONPAR '90.