J. Symbolic Computation (1995) **11**, 1–000

Parallel Gröbner Bases computattions by elementary methods [†]

J.C. FAUGÈRE

e-mail: jcf@posso.ibp.fr

LITP, Institut Blaise Pascal

Case 168, 4, place Jussieu

F–75252 Paris Cedex 05

(Received 8 June 1995)

The main purpose of this paper is to present parallelized versions of Buchberger and FGLM algorithms which are the most powerful tool to compute Gröbner bases. In the case of integer coefficients, the basic idea is to use a probabilistic DAG of dependencies generated by an auxiliary fast modular computation in order to avoid vanishing syzygies in first time and to exploit the parallelism without disturbing the sequential strategy in a second time; the result of this computation is only probabilistic but we show that proving it can be done quickly. A parallelized version of Buchberger algorithm is also presented in the case of modular computation. We achieved about 31 times speedup with 8 processors for a large robotic problem (integers), ($\bigwedge^2$). For a big modulo p problem – cyclic 7 – we report a 5 (resp. 6.2) rate of acceleration for Buchberger algorithm (resp. FGLM algorithm) on a Alliant with 8 processors. The details of an efficient C++ implementation are given together with results of tests performed with significant examples.

1. Introduction

One of the main tools for solving algebraic systems is the computation of Gröbner bases (also called standard bases); we refer to Buchberger65, Buchberger70, Buchberger79, Buchberger85, Davenport et al.93 and Becker et al.93 for basic facts on this notion. In the zero dimensional case (finite number of solutions) an efficient way to achieve this is Faugere et al.94b, Gianni et al.94, Lazard92, Faugere94 a four steps algorithm (Lazard93b): first find a Gröbner base for a total degree ordering, change of ordering (by using FGLM for instance), obtain a list of triangular sets, solve numerically using the previous result. Although this class of problems is special, the majority of polynomial systems arising in practice have this property. Another usual subclass of polynomial problems is constituted by polynomials whose coefficients are in the ring **Z**; but very

[†] Supported by EEC project PoSSo (Polynomial Systems Solving)

often there is a huge gap between computing over the integers and computing over integers modulo a prime p; trace methods Traverso98 are a first attempt to reduce this gap, a second one is presented in this paper. The main purpose of this paper is to present parallelized versions of the first and second steps. The methods exposed here are elementaries in the sense that we do not try to modify deeply the sequential computation: on the contrary we try to mimic closely as possible the best known sequential strategy. In spite of that relative simplicity the methods are very efficient.

Section 3 is devoted to the parallelization of the first step, referred to as the "Buchberger Algorithm"; it is the more random step in the sense that it is difficult, if not impossible, to predict a priori the computational time and the growth of the coefficients. It is the main section of the whole paper and it is divised in four parts.

Since, as sketched, before the shape of computation is totally different when computing with integers or with modulo p integers we need a special algorithm for the latter case and it is the object of subsection 3.5; for typical big problems Cyclic 7 (resp. T6) we reduce the total CPU time by a factor of $5 = \frac{1080\ sec}{216\ sec}$ (resp. 6.2)

In the case of integer coefficients, we use an auxiliary modular to compute the DAG of dependencies; we can use this DAG to avoid the computation of vanishing critical pairs (subsection 3.3) and to parallelize the computation of the ones (subsection 3.4)

computation , more exactly we first show how to obtain a not probabilistic algorithm from a probabilistic one and give some experimental results which prove that, surprisingly, the extra cost for that "check part" is relatively small and can be reduced to almost zero by using a sufficient number of processors. Then we will explain how to deduce from the graph of dependencies generated by an appropriate modular computation a parallelized algorithm that preserve the selection strategies and all heuristics found recently Giovini et al.91. Without parallelization we have a very efficient algorithm (non linear gain) which enable very big computations (realistic examples from robotics are shown Faugere et al.94a); with only 4 processors we can divide the computation time of the first step by a factor 2.5.

we obtain a speedup varying from 5.5 to 31 for well known and realistic examples on a shared multiprocessor environment with 8 processors (Alliant).

The quality of the computer implementation of Gröbner bases algorithms can have a profound effect on their performance. Hence, “paper and pencil” descriptions of Gröbner bases algorithms are not enough; this why we provide for each algorithm presented in this paper an efficient C++ implementation part of the Gb system and a detailed list of experimental results.

All the tools needed for solving algebraic equations are implemented by the author in a C++ (59 000 lines) very efficient software called GB[†]; GB is known as the fastest system at this time. We have a first implementation of the method exposed in this paper: one can use GB over a network of workstations (heterogeneous), or better on a multiprocessors workstation with share memory (for instance Sparc Center). The experimental timings concord with the computational model used to present the algorithm.

In section 2 we describe recent works in area of parallelizing Buchberger algorithm.

[†] GB is freely available (source and binaries) by anonymous ftp: `posso.ibp.fr`

2. Related Works

We list briefly works related to parallelizing the Buchberger algorithm. We recall that our goal is to parallelize the *fastest* Gröbner bases algorithm, thus recent results (such as Giovini et al.91 and Faugere et al.94b) imply to reconsider most of the following papers.

Senechaud89

computes Boolean Gröbner bases: it is a very special case of ideals for which there are specialized efficient algorithms; the same is true for toric ideals and binomial ideals (Pottier94 and Bsturmfels91).

Vidal90

proposed a parallel algorithm on a shared memory multiprocessor; the algorithm does not rely on the sugar strategy; he obtained a speedup of $14 = \frac{1103\ sec}{79\ sec}$ with 12 processors on the the biggest example (katsura 5), but this example can be solved only $2 secs$ with *one* processor !

Grabe et al.94

reports a parallel version of Gröbner bases algorithm with factorization on a distributed memory environment. This works only for "well splitting examples" which are not very common.

Siegl94

implements in ||MAPLE|| of a Gröbner bases algorithm using factorization: toy examples are solved in a very long time.

Sawada et al.94

describes also a Gröbner bases algorithm on a distributed parallel machine with 256 processors. They obtained significant speedup when the number of processors is < 32 but the implementation (and the strategy) seems not very efficient for one processor. We include in our experiment the most significant examples found here.

Attardi et al.94

are the closest of this paper; they present a parallelized form of Buchberger algorithm that is strategy-accurate. But they do not remove superfluous critical pairs before parallelizing; moreover their algorithm is implemented on a network of workstations and because the heavy cost of communications its seems not pratical; no experimental timings are given.

It follows from this listing that the designer of parallel Gröbner bases algorithm must take car of:

The implementation and the choice of strategy must be up to date and efficient at least when the number of processor is *one*.

The methods should be sufficiently general to handle properly a list of well known systems (such as the PoSSo test suite for instance).

The algorithm must work not only on toy examples but also for big difficult and realistic examples.

3. Parallelization of Buchberger Algorithm

3.1. Introduction

3.1.1. Notation

We define now some terms used in this paper but we donc explain the classical theory of Gröbner bases; for the reader unfamiliar with Gröbner bases language we refer to Becker et al.93.

A polynomial f is an ordered list of monomials; the order might be any admissible ordering Robbiano85 but the most useful ones are: the lexicographical (LEX) ordering and the degree–reverse–lexicographical (DRL). DRL is generally the one for which the computation of a Gröbner has the best theoretical and practical complexity. On the other side a DRL Gröbner base does not give solutions by itself. In practice the coefficient of polynomials are of two kind:

integers (that is to say big integers with no size limitation)
integers modulo a small prime p (represented by a word in a computer memory)

We call the reduction of a polynomial p_1 by a polynomial p_2 an *elementary operation*, the result is given by:

$$m_1 \times p_1 - m_2 \times p_2 \qquad m_i \ monomials$$

The S-polynomial of polynomials p_1 and p_2 is the reduction of $m_1 \times p_1$ by p_2 for some monomial m_1 and thus is an elementary operation. Reduction of a polynomial by a list of polynomials is a sequence of elementary operations. By "Buchberger Algorithm" we mean any algorithm that compute a Gröbner base of a list of polynomials by a series of elementary operations; from this restricted point of view the classical sugar algoritm Giovini et al.91 and the new algorithm Lazard93a are "Buchberger Algorithm" even if they are very different. Hence the methods describe here apply to all kind of Buchberger's algorithms and variants including different strategies.

We denote by

$$p \mapsto normalForm(p, G)$$

the reduction of p modulo I where p is a polynomial, G is a Gröbner base and I the ideal generated by G; we recall that

$$normalForm(p, G) = 0$$

iff p is member of I. $normalForm(p, G)$ is also a sequence of elementary operations.

By a *modular computation* associated to a system of polynomials S, we mean the computation of a Gröbner base of the input system S'_p where S'_p is the image of S by the morphism (see Traverso98, Lazard93b):

$$coef \times X^\alpha \mapsto (coef \ mod \ p) \times X^\alpha$$

3.1.2. Overview of the section

Computing efficiently a Gröbner base depends strongly on:

Avoid unnecessary critical pairs. Even with Buchberger's first ans second criteria most of the critical pairs considered during a computation will reduce to zero (pairs reducing to zero may represent 95% of the total time in the integer case and 80% for modulo p coefficients); in Moller et al.92 algorithm is given which show how this can be avoided by computing syzygies: this algorithm can detect more superfluous critical pairs than any other method, but unfortunately the time spend of computing syzygies is a bottleneck and the algorithm is of no practical interest.

The selection strategy: it is well known that selection strategies in Gröbner bases algorithms are a crucial point Giovini et al.91: one of the most important effect of a good strategy (sugar strategy is probably the best) is to reduce the size of coefficients, and since 99% of the time is spent in arithmetical operations over the integers $(+, \times)$ it is very important to not perturb the strategy. Even for modular computation and at least for big computations it seems better to strictly follow the sugar strategy.

This section of the paper is devoted to prove that:

for integers coefficients and non homogenenous systems, there is a very simple method to avoid superfluous critical pairs. We claim that the computation of Gröbner basis can be significantly speeded up by a factor range from 4.2 to 7.4 for small problems and from 14 to ∞ for large problems.

for integers coefficients and more processors we can reduce once more time the CPU time given by the last method by a factor varying from 3.9 to 6.5 with 8 processors and shared memory.

for modulo p coefficients on a shared multiprocessor we obtain speedup between 5 and 6.4 for 8 processors.

In each case the structure of our presentation is as follows. We give the algorithm, details on the implementation and the practical results and speedup on *significant* and well known problems are listed.

There is a non linear gain in efficiency due to this method. Our method use an auxiliary modular computation as in Gröbner trace algorithm of Carlo Traverso Traverso98 but it differs from it in two main points: first we show that it could easily be converted into a exact method by doing an extra Gröbner computation, the time spent in verification is small for very big problems and can be divide out by the number of available processors (we could even use a network of workstations without share memory). Secondly we explain how to find a way to parallelize the main part of the computation from the DAG of dependencies found by a modular computation; the method is well adapt for workstations with a small number of processors (4 seems to be quite enough) with share memory and was experimented on Sparc Center 2000 and Alliant Fx2800.

3.2. Verify that a list of polynomials is a Gröbner base

In the rest of the paper we denote by step X a specific part of an algorithm and by τ_X the corresponding CPU time of that part. We recall that coefficients of the polynomials are integers.

Let us suppose that we already have a Gröbner bases implementation and a function

```
Input:   $F$ list of polynomials.
         $I$ the generated ideal.
Output: $G$ the Gröbner base of $I$.
{ The result of this algorithm is probabilist, but CPU time is. }
"Guess" a list $G_1$ of polynomials of $I$.
$G_2 \leftarrow \{\text{normalForm}(x, G_1) \mid x \in F\}$
{ We call it step G (Guess). }
Delete all zeros from $G_2$.
if $G_2 = \emptyset$ then
  { Check that $G_1$ is a Gröbner base: }
  $G_3 \leftarrow$ Gröbner $G_1$
else
  $G_3 \leftarrow$ Gröbner $G_1 \cup G_2$
{ We call this last Gröbner call step C (Check). }
$G \leftarrow G_3$
return $G$
```

Figure 1. Heuristic Gröbner Computation

normalForm, a Gröbner completion algorithm Lazard93b fitted in Figure 3.2 can easily be writen.

The algorithm works fine if we can guess a G_1 such that we have with a good probability the conditions:

$G_2 = \emptyset$, that is to say G_1 is a generator of I.
$G = G_1$ it means that G_1 is already a Gröbner base.

If we don't execute step C, then we have only a probabilistic algorithm. We can achieve step G by doing simultaneously a modular computation – G$_M$ – and the same computation over integers – G$_I$ – but in G$_I$ we don't try to compute syzygies reducing to zero in G$_M$. With a good probability (see Traverso98), the previous conditions3.2 hold. Lazard93b says: "The last two verifications may be almost as difficult as computing the Gröbner base by not modular method". The surprising empirical result shown here, is that $\tau_C \ll \tau_G$ for very big computation and $\tau_C \approx \tau_G$ in other cases. Compute G_2 is an easy and fast task if we have a *normalForm* function; it is obvious that G_2 could be parallelized. Consequently step C is equivalent to verify that a list of polynomials reduce to zero wrt to a *fixed* list of polynomials, hence check that a specific polynomial reduce to zero can be done *independently* of the others, thus we can divide τ_C by n, if we have a set of n processors (not necessary with share memory).

3.3. Avoid superfluous critical pairs

3.3.1. Algorithm and implementation

The algorithm is plotted in Figure 3.3.1. The implementation in C++ follow strictly this scheme: it can be run on a network of two distant workstations. We use Unix sockets to communicates data between the master and slave process; actually in the real implementation we send the exponant of the leading monomial found in the modular computation; if we detect a failure it is better to choose a new prime and to rerun the whole computation; this test is to be used only to screen out obvious errors. One has not to allow for communication costs: the amount of data to transfer is very small in front the total CPU time of each process. In Gb, two kind of fast servers for prime $p < 2^{16}$ and for $p < 2^{31}$ were implemented. In practice a such $p = 59999$ has never failed.

```
Input:   $F$ list of polynomials.
         $I$ the generated ideal.
Output:  $G$ the probabilistic Gröbner base of $I$.
critpairs ← form the list of critical pairs
child ← fork("remote workstation")
if child process then
  choose a small random prime $p$ and set the ground ring to $Z/pZ$
  $F \leftarrow F$ with coefficients modulo $p$
$G \leftarrow F$ while critical $\neq$ 0 do
  if child process then
    $h$ = normalForm(first(critpairs),$G$)
    if $h = 0$ then
      send to(parent process,"ZERO")
    else
      send to(parent process,"NON ZERO")
  else
    if get from(child process) = "NON ZERO" then
      $h \leftarrow$ normalForm(first(critpairs),$G$)
  $G \leftarrow$ Update ($h$,$G$)
  critpairs ← Update ($h$, rest(critpairs))
return $G$
```

Figure 2. Probabilistic Gröbner Computation avoiding vanishing pairs

3.3.2. Experimental Results

A list of examples

In the rest of this paper we will evaluate our parallel Gröbner bases programs on the benchmarks listed below. We take the list of Sawada et al.94 except toy examples (namely katsura n and cyclic n for $n < 6$).

Katsura-6 (Katsura86): (7 variables and 7 polynomials 64 solutions)
Katsura-7 (Katsura86): (8 variables and 8 polynomials 128 solutions)
Cyclic-6 (Bjork85, Backelin89, Davenport97): (6 variables and 6 polynomials 156 solutions)
Cyclic-7 (Bjork85, Backelin89, Backelin et al.91): (7 variables and 7 polynomials 924 solutions)
T-6 (Backelin et al.91): (7 variables and 6 polynomials dimension 1)

In Faugere et al.94a we used the techniques exposed in subsection 3.3 to solve big problems arising in a robotic problem: a *parallel manipulator* is a body (*platform*), the spatial position of which is commanded by fixing the distance of six points of the platform to six fixed points of the space (the *base*). When one wants to compute the position of the platform from its geometry and the lengths of the linear actuators, one is led to a system of algebraic equations which has several solutions.

We summary a special configuration of the parallel manipulator by symbols such as $(\bigwedge\!\!\!\bigvee^2)$ (`Spat2N` in text form), $(\bigwedge^2|^2)$, $(\bigwedge\bigvee|^2)$, ... These examples will be add to the PoSSo Test suites; they are useful for our purpose because they are:

realistic in the sense that they come from the physical world.
very big, they give rise to an enormous coefficient growth and are good tests of how efficient the integer arithmetic is.
unreachable with classical Gröbner bases algorithms (at least the biggest).
they do not have too much geometrical properties but have some. In some sense they are othogonal to the cyclic-n problem which have a lot of symmetries.

By testing our ideas on different kind of examples, we will validate our theoretical models.

Gb and other Computer Algebra systems

Even if it is not an easy task to compare system, one can say that Gb is among the fastest system for computing Gröbner bases. Because of the big numbers of computer algebra systems which implement a Gröbner bases function, it would be very difficult (and very long) to test a comprehensive list of examples. Thus, we restrict ourselves to *one* well known system, cyclic n (see Backelin89 and Backelin et al.91); for each system we report on figure 3.3.2 (see page 8) its aptitude to compute the Gröbner base for the *non homogenenous integer* Cyclic n problem in a less than one day. The fact that it is a non linear scale is best explained by referring to Table 3.3.2.

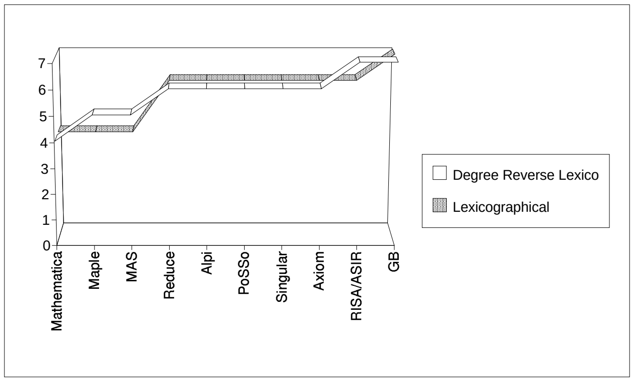

Figure 3. Comparing Gb with other systems

n	5 6	6	7
$\frac{time\ to\ solve\ Cyclic\ n}{time\ to\ solve\ Cyclic\ n-1}$	10.7	10.3	4427.6

Tab 3.3.2: Relative time for solving cyclic n

This list include the following systems: Reduce Version 4.3 (Fitch85, Hearn87), Axiom Version 2.0 (Jenks et al.87, Jenks et al.92), Alpi Version 1.95 (Alpi, Traverso et al.89), MAS Version 0.7, Maple Version V.2 (Char et al.91), Singular version 0.9.0e (Grassman et al.94 and Grassman et al.95) but we do not include a system such Macaulay (Stillman et al.89) which works only with modular p computation. We mention also that the Bergman (Backelin et al.92) is also very fast. The main drawback of this table is that it do not report that a program such as the PoSSo library is faster than Reduce. Nevertheless we can conclude from this table that Gb is one of the fastest system.

Timings on Sparc 10

We give a first table of computing time (Sparc Station 10): `groebner` is a classical Gröbner bases, `tgroebner` is an implementation in GB of the previous method (two processors for G_I and G_M thus τ_G is a parallel process time; one processor for C).

	`groe-bner`	`tgroe-bner`	τ_G	τ_C	saving of time
T6 (Homog)	4'3"	4'48"	48"	4'	**0.85**
Homog Cyclic 6	16"3	7"3	2"	5"3	**2.2**
Cyclic 6	14"	3"3	1"7	5"6	**4.2**
Katsura 6	5'14"	1'1"	16"	45"	**5.2**
$(\bigwedge\!\!\!\bigvee^2)$	16'55"	2'18"	47"	1'31"	**7.4**
Katsura 7	2h15'39"	9'5"	5'51"	3'14"	**15**
$(\bigwedge^2 \|^2)$	2h24'24"	10'20"	7'5"	3'15"	**14**
$(\bigwedge \bigvee \|^2)$	18h48'	55'5"	47'19"	7'46"	**20.7**
$(\|^6)$	$>$ 4 days	1h16'48"	58'22"	18'26"	$>$ **75**
Cyclic 7	∞	4h3'31"	3h52'2"	11'29"	∞

Tab 3.3.2: Using two processors Sparc 10 (50Mhz)

Dire que Cyclic 7 est calcule avec l'algo de Daniel
Le cas de T6 est etrange homogene ?

3.4. Parallelization of non vanishing critical pairs

Our aim in this section is to compute G_I on a multiprocessors workstation using the DAG of dependencies generated by G_M. We first need to fix notations:

3.4.1. Definitions

We want to compute a Gröbner base for the DRL ordering (in fact ordering doesn't matter but experimental tests have been made only for the DRL ordering because it is the most important case in practice), of the input system $F = (f_1, \ldots, f_r)$. A Buchberger algorithm can be view as a series of polynomial computations f_i $(i > r)$, (we compute all the polynomials one after the other) where f_i is the S-polynomial of f_{a_i} and f_{b_i} reduced by a set of previously computed polynomials (i.e. whose indexes are $< i$), namely $f_{u_{i,j}}$ for $j = 1, \ldots, v_i$ $(u_{i,j} < i)$. In other words, a_i, b_i and $u_{i,j}$ are indexes of all the polynomials which occur in the computation of f_i and $v_i + 2$ is the number of such polynomials; we denote by S_i the *list* of indexes $\{a_i, b_i, u_{i,j}\ \forall j \in \{1, \ldots, v_i\}\}$. Elements of S_i are not necessarily unique. The final Gröbner (not reduced) base is a sub series of the whole polynomials series (f_i): $(f_{i_1}, \ldots, f_{i_k})$. (One can suppose that $f_i \neq 0$ for all i as explained before). We obtain the DAG of dependencies (see Fig 3.4.1, page 10) of vertices $(1, \ldots, i_k)$: if $i > j$ we join j and i by an edge if $a_i = j$, or $b_i = j$ or $u_{i,l} = j$ for some $l \leq v_i$.[†]

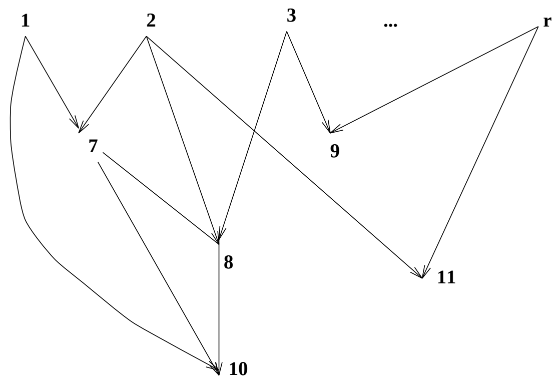

Figure 4. DAG of dependencies.

We note $N_{\text{all}} = i_k$ the total number of non null polynomials and $N_{\text{base}} = k$ the size of the Gröbner base.

3.4.2. Independent nodes

First of all we can detect in the DAG of dependencies some characteristic pattern (see Fig 3.4.2, page 11) which can help us in parallelizing: for instance if a node i which does not depend of the previous node $i - 1$, we can compute i and $i - 1$ *at the same time.*

Of course the question is then: what proportion of such patterns one may expect in a typical computation ? We have developed along with GB a small program call "*flow*" which analyzes the output of a modular Gröbner computation. We note:

$$N_{\text{once}} = Card\{k \mid \exists j, Card(S_i \cap \{k\}) = \delta_{i,j} \quad \forall i\}$$

$$N_{\text{ind}} = N_{\text{independent}} = Card\{i \mid S_i \cap \{i-1\} = \emptyset\}$$

In other words, N_{once} (resp. $N_{\text{independent}}$) is the number of patterns similar to the left (resp. the right) part figure 3.4.2.

[†] In fact in GB we compute only nodes which have a descendant in $\{i_1, \ldots, i_k\}$ even if in practice this number is *very small.*

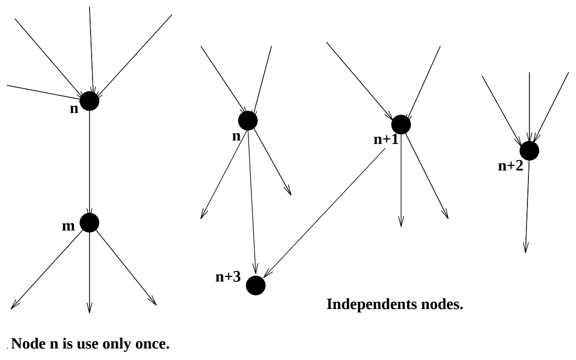

Figure 5. Characteristic pattern

	N_{all}	N_{once}	N_{ind}	$\frac{N_{\text{ind}}}{N_{\text{all}}}$
C_6	98	10	11	11.2%
$(\|^6)$	109	9	26	23.8%
$(\bigwedge^2\|^2)$	82	10	32	39%
$(\bigwedge\bigvee\|^2)$	105	11	22	20.9%

Tab 3.4.2: Number of patterns

It is to be noticed that this "rate of independencies" is not negligible, but represent only 10/25% of the overall computation. That is to say that a Buchberger algorithm is a very *sequential* algorithm.

3.4.3. Almost independent nodes.

Let us extract a typical list of dependencies from the $(\bigwedge^2|^2)$ example:

$$\begin{array}{l} \vdots \\ a_{29} = 19,\ b_{29} = 22,\ v_{29} = 8, \\ S_{29} = [19, 22, 20, 18, 22, 15, 18, 19, 20, 28] \\ \vdots \end{array}$$

It means that the knowledge of 28 (f_{28}) is strictly necessary for computing node 29 (f_{29}), which require to perform $v_{29}+1 = 9$ elementaries operations; but at the other side, the first 8 steps can be done as soon as we have computed f_i, $i < 22$, in other words we can compute $\frac{8}{9} = 88\%$ of f_29 as soon as we know f_22. We introduce now the following measure:

$$p_i = \begin{cases} 0 & \text{if } a_i = i-1 \text{ or } b_i = i-1 \\ 1 & \text{if } maxS_i < i-1 \\ \frac{j_0}{v_i+1} & \text{else, with} \\ & \quad j_0 = min\{j \leq v_i \mid u_{i,j} = i-1\} \end{cases}$$

p_i tell us which percentage of f_i we can compute without knowing f_{i-1}. It is now easy to find the number of such elements f_i: we can compute: $N_{\text{almost}} = Card\{i \mid p_i > 6\%\}$. *flow* give both p_i and N_{almost}:

	C_6	$(\vert^6)$	$(\bigwedge^2 \vert^2)$	$(\bigwedge\bigvee \vert^2)$
N_{almost}	60	52	52	60
$\frac{N_{\text{almost}}}{N_{\text{all}}}$	61%	48%	63%	57%

Tab 3.4.3: Quasi independent nodes

These numbers explain why one can expect good result of the previous outlined method, however they are not sufficient to estimate the computational time.

3.4.4. A computational model

To estimate the computational time, p_i is not sufficient because

we have fixed arbitrarily the limit of 66%.
even if we know that $p_i > 66\%$ we don't know when f_i can be computed (in the previous example, we have seen that f_{29} is almost computed since f_{22} is known).
we have to test the dependency of node i not only with $i-1$, but also with $i-2$, $i-3$, ...

flow will give a good estimate if we can continue to restrict ourselves to analyze the DAG of dependencies, that is to say if we can suppose that each element of $\cup_i S_i$ has the same weight in the global computational time; so we are making the following simplifying hypothesis:

H$_1$: We have N_{all} processors (a typical number is 100 for N_{all}) at our disposal. Of course it is not a realistic estimation; it is introduce here to simplify the algorithm presentation: processor number i will have the task to compute f_i. In practical cases only a few processors are needed as it will be seen later.

H$_2$: Time for sending messages is negligible: this is only true if we a share memory architecture in which case it is sufficient to send index of polynomials. (sending message in that case is done by mean of share memory, so it is very fast).

H$_3$: We suppose that all elementaries operations over polynomials can be performed in a constant time t_0: this hypothesis is globally false because polynomials are growing during the computation (both the number of monomials and the size of coefficients); nevertheless we can estimate as a first approximate that f_i and f_{i+j}, for j small, have almost the same size; thus the assertion **H**$_3$ is *locally* well verified.

As a result, we can write that the sequential computational time of G_I is of the same order than $T_{seq} = t_0(N_{\text{all}} + \sum_i v_i)$. We associate – at initialization time – to each processor i, a stack P_i of constituted of all elements of S_i (a_i is at the top of the stack). On can say that f_i is known when the corresponding stack P_i is empty; of course, at the beginning, stacks P_i for which $i < r$ are empty. Now we can write very easily our main algorithm (see Fig 3.4.4, page 13 and Fig 3.4.4, page 13).

```
while $\exists i_0$ such that $P_{i_0} \neq \emptyset$ do
  for $i \leftarrow 1$ to $N_{\text{all}}$ do
    {first($P_i$) return the first element at the top
     of the stack $P_i$ without popping. }
    if $P_i \neq \emptyset$ and $P_{\text{first}(P_i)} = \emptyset$ then
      $j \leftarrow$ pop ($P_i$)
      $f_i \leftarrow$ Reduction of $f_i$ by $f_j$
```

Figure 6. Gröbner pipeline computation

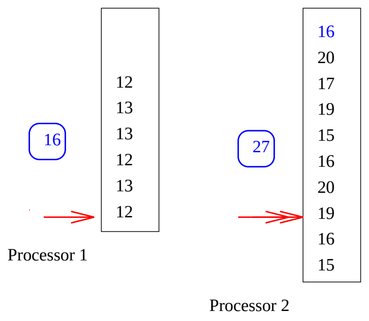

Figure 7. Each processor has its own stack.

The *For* loop in the algorithm can be done in parallel, hence $T_{//}$ – the number of iteration multiplied by t_0 – is an estimation of the computational time. *flow* can simulate this algorithm (only the DAG of dependencies is needed):

	C_6	$(\|^6)$	$(\bigwedge^2 \|^2)$	$(\bigwedge \bigvee \|^2)$
T_{seq}	2165	7499	2848	5004
$T_{//}$	630	2824	932	1553
$\frac{T_{seq}}{T_{//}}$	3.4	2.7	3.1	3.2
N_{procs}[†]	16	29	14	28
$N_{\approx\ \text{procs}}$	3.44	2.66	3.06	3.22

Tab 3.4.4: Theoretical parallel time

Consequently we can estimate to 3 the gain factor of the method to be compare with the results of previous section (τ_G in 3.3.2). It remains to see if we can restrict the number of actual processors in order to validate $\mathbf{H}_1$; *flow* compute also (last two lines of the tabular) the maximal number (resp. the average number) of processors needed for a particular computation. When one read $N_{\approx\ \text{procs}} = N_{\text{average processors}} = 4$ in the tabular, it means that most of the time we can compute f_r, f_{r+1}, ..., f_{r+3} at the same time, thus computing polynomials of roughly the same size (see $\mathbf{H}_2$. All of these results show that we can validate our hypothesis (see page 13) and that a number of 4 processors seems enough in most cases.

3.4.5. Implementation in GB.

In GB we have made two kind of implementations (C++): one for a distributed network of heterogeneous workstations and a true parallel implementation for multiprocessors computers (Sparc Center[†], Alliant[‡]) with share memory. The first one was only a debugging program because of the prohibitive cost of communications. Nevertheless we adopt in both cases the syntax of sending messages and objects, which is semantically rich and enable us to have the same code to describe the algorithms. The natural architecture client/server of GB (see Fig 3.4.5, page 15) was very helpful to implement the method.

The central module called GB is an interpreter which sends among other things the initial system of polynomials to the master server. Ideally each servers should be associate with a physical processor; it is planed in a future version of GB to suppress the master server. The master don't compute things by itself but it has to:

Use the DAG of dependencies to dispatch computation between servers (in the same way of *flow*, see algorithm 3.4.4, page 13).

Execute and read data from the Modular Server. (Very often this server is running on a remote workstation)

See below for definitions of N_{procs} and $N_{\approx\ \text{procs}}$.

[†] Thanks to J. Hollman in Sweden.

[‡] Thanks to J. Marchand École Polytechnique

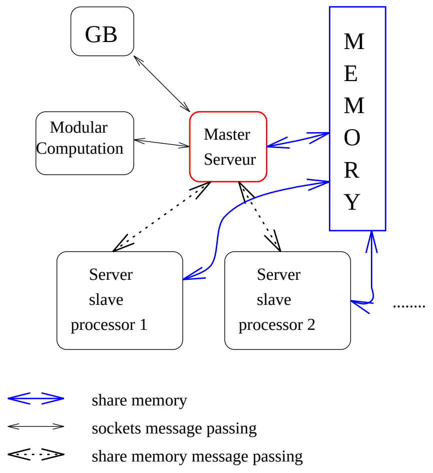

Figure 8. Architecture of GB.

Run all other processes.
Schedule computations between servers.
Synchronize all the slave servers.
Send messages and polynomials: in the distributed implementation it is necessary to send only a minimum of data over the network; thus the master server has for each slave server a list of all the polynomials which are already in its memory. In the case of share memory there is no need to send polynomials, all the messages consist of indexes which are copy in a part of the memory share among all processes devoted for sending messages.
Share memory is write protected by a semaphore[†] in order to prevent concurrent access of the same part of memory.
Collect polynomials whose computation is finished and all available processors. When the master server "receive" a polynomial it checks that the degree of the leading monomial is the equal to the exponent found by the modular computation program.

At the opposite side, task of a slave server is

compute one f_i at a time and wait while the corresponding stack is not ready (that is to say when the top of the stack is not known yet); when the stack is empty the server send f_i to its master; wait for receiving a new index j and begin the computation of f_j. In our first implementation we follow the description of algorithm 3.4.4 but it would be better to begin the computation of a new f_j instead of waiting.
each server has its own memory zone under the control of a local Memory Management – MM – (describe in Faugere94). It works as follow: when one enter in a function the system store a mark at the beginning of the free memory; when a function return the programmer indicates to the MM (eventually) which data are "stable" or are supposed to be stable; in such a case the MM release all memory

[†] On Alliant machines this mechanism was not available.

above the previous mark, then compacts and slides the data at the beginning of free memory. Such a Memory Management system is well adapted to our problem: when the slave server end a computation it release all its local memory (the previous mark is at the beginning of memory) and copy the corresponding polynomial in share memory (a mark shared by all processors);

3.4.6. Timings on Alliant FX2800

We give now the results of our first implementation using 1,2 or 3 effective processors (remember that master server use a processor). Therefore we are not exactly in the ideal case of 4 *active* processors. All the timings are obtain by making the difference between the date at the end and date at the beginning of a computation; thus it is exactly the time the user spent in front of the workstation to have the Gröbner base. All the timings are round to the nearest second.

	1 processor	2 processors	3 processors
$(\bigwedge\!\!\bigvee^2)$	220"	136"	110"
Cyclic 6	7"	5"	3"

Tab 3.4.6: Alliant FX2800 8 processors

Here is the table for $(\bigwedge\!\!\bigvee^2)$ example; *Sequential* is the classic `groebner` function; *Sequential trace* is a sequential implementation of trace method including verifications; *Parallel* is the parallel implementation using 3 processors.

	Sequential	Sequential trace	Parallel
$(\bigwedge\!\!\bigvee^2)$	3635"	434"	179"
C_6	47.8"	11"	4"

Tab 3.4.6: Sequential/Parallel Alliant (3 procs)

For this examples, the parallel implementation is 2.5 faster, in other we use 80% of each processors.

The main messages of this sbusection are summarized in Fig. 3.4.6 where the speedup of our method is illustrated in the case of 1.5, 4 or 8 processors and five examples:

The reminder of this section is concerned with studying the parallelization of Buchberger algorithm in a very different case: modular computations.

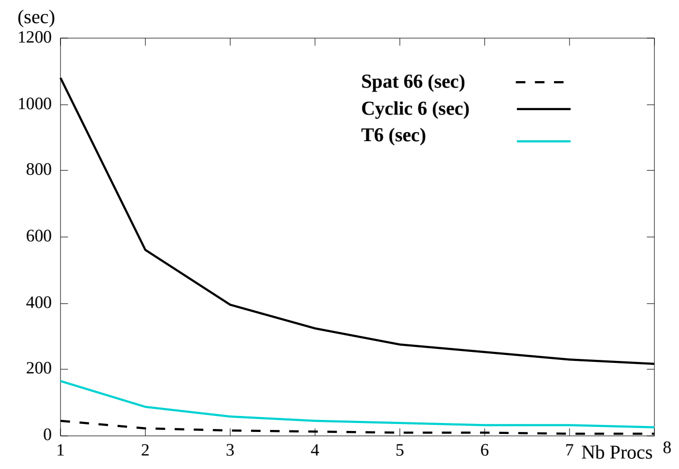

Figure 9. Time in secs/Nb of Processors Modulo p (Alliant 8 procs)

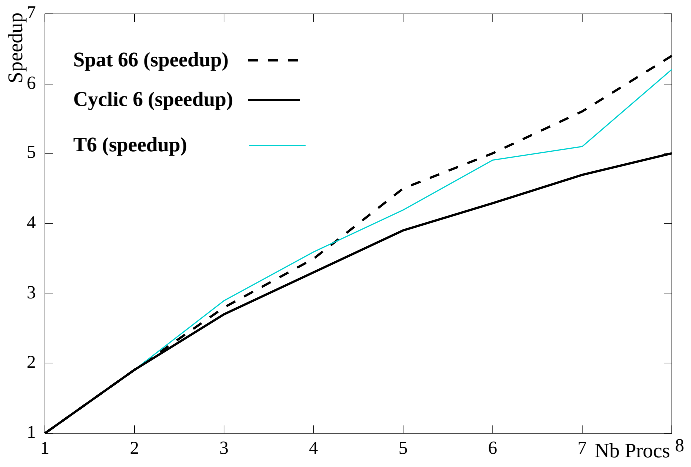

Figure 10. Speedup/Nb of Processors Modulo p (Alliant 8 procs)

3.5. A parallel Buchberger algorithm for modulo p coefficients

The point of view in this paragraph is necessarily different because:

arithmetic are no more the most important part of the computation.
we do not know the graph of dependencies

On the other side, the fact that most of time is spent in computing vanishing critical pairs is always true: $95\% \approx \frac{3366\ sec}{785 sec}$ (resp. $92\% \approx \frac{162\ sec}{20\ sec}$) of the time for cyclic 8 (resp. cyclic 7) on an Alpha workstation.

4. Parallelization of FGLM Algorithm

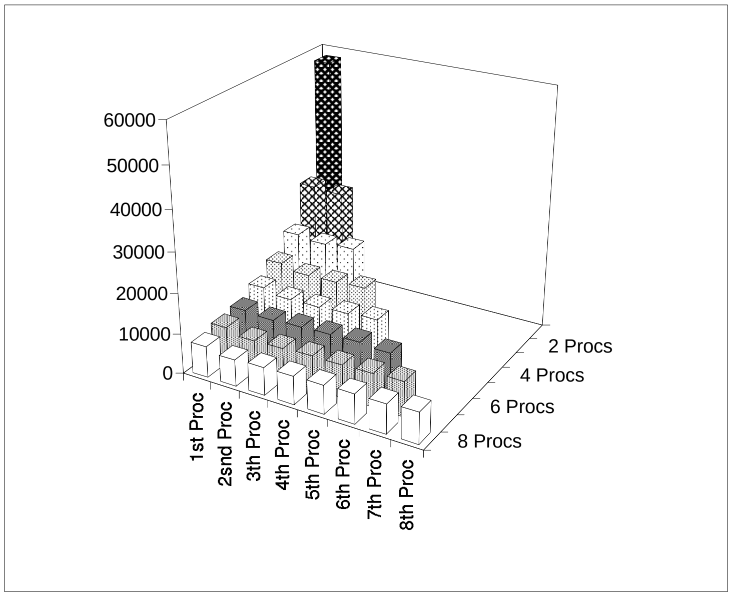

Figure 11. Activity by proc on Alliant ($(|^6)$ Modulo p).

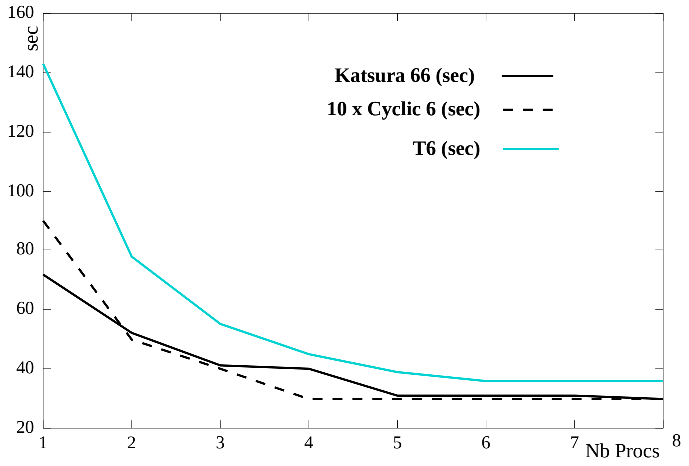

Figure 12. Time in secs/Nb of Processors Comput. Part BigInt (Alliant 8 procs)

5. Conclusion

We have shown in this paper that:

The cost for verifying that a list of polynomials is a Gröbner base is small and can be easily parallelized.

By using 3-4 processors we can divide the computational of a trace method by 2-3 using the DAG of dependencies.

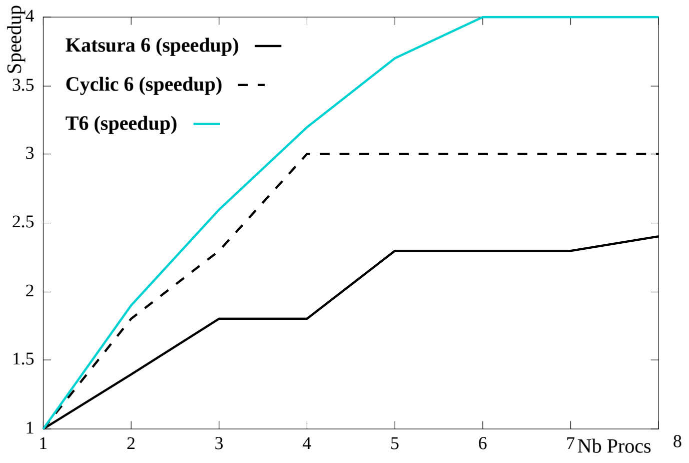

Figure 13. Speedup/Nb of Processors Comput. Part BigInt (Alliant 8 procs)

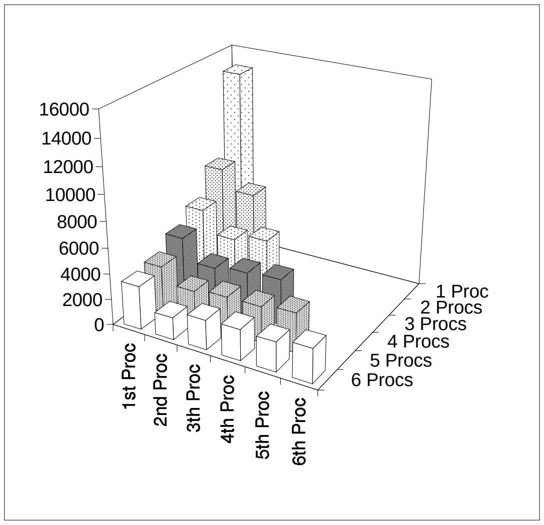

Figure 14. Activity by proc on Alliant (T6 Big Int).

The method exposed here does not excluded parallelization of elementaries operations which can be achieved with more processors. Our main goal was to not perturbate the selection strategy (sugar strategy) for integers coefficients which ensures that parallelization will not grow the volume of computations; the second goal was to have an implementation for common workstations with four processors that can be competitive with a very efficient implementation for non trivial problems.

Another important algorithm FGLM Faugere et al.94b – change of ordering of a Gröbner base (step 2 for solving) – is also available in GB both in a sequential and parallel form; it is not describe in this paper (see Faugere94) but works very well. It is also obvious to obtain a parallel algorithm for the last step: find a list of triangular sets. Thus one

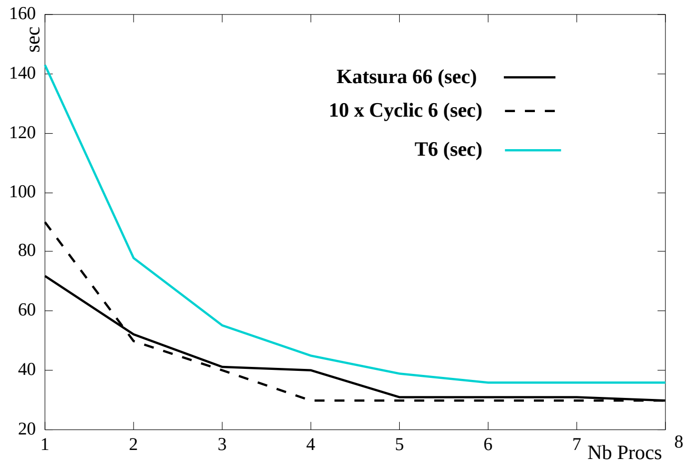

Figure 15. Time in secs/Nb of Processors Comput. + Check BigInt (Alliant 8 procs)

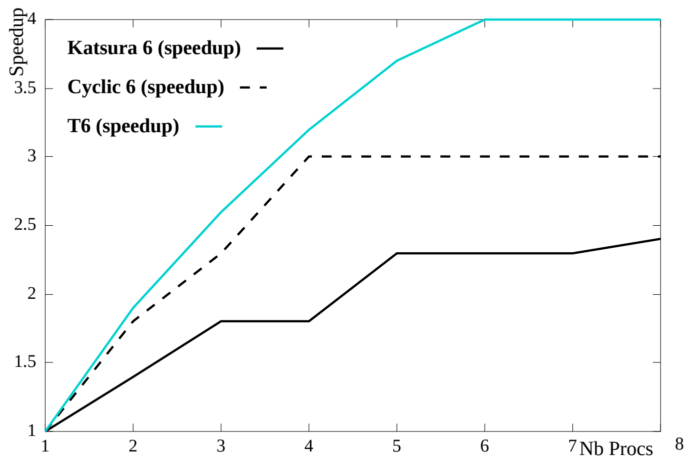

Figure 16. Speedup/Nb of Processors Comput. + Check BigInt (Alliant 8 procs)

can say that a complete parallel implementation for solving algebraic systems is available in GB, of course the first step remain the more difficult part to be parallelized.

Limitation de la // Petit nombre de processeur.

The success of algorithms for Gröbner bases computations, perhaps more than in any other area of Computer Algebra depends on the quality of their computer implementation. Some algorithms presented here which appear not very attractive from a theoretical point of view but are very efficient in practice. For these reasons, we have described C++ programs which implement all the algorithms discussed in this paper.

References

Alpi: a system for experimenting with buchberger algorithm. alpha version in Common Lisp available by anonymous FTP in `gauss.dm.unipi.it` (131.114.6.55).

Attardi (G.) et Traverso (C.). – A strategy-accurate parallel buchberger algorithm. *In: Pasco'94*, éd. par Hong (Hoon). pp. 12–21. – World Scientific.

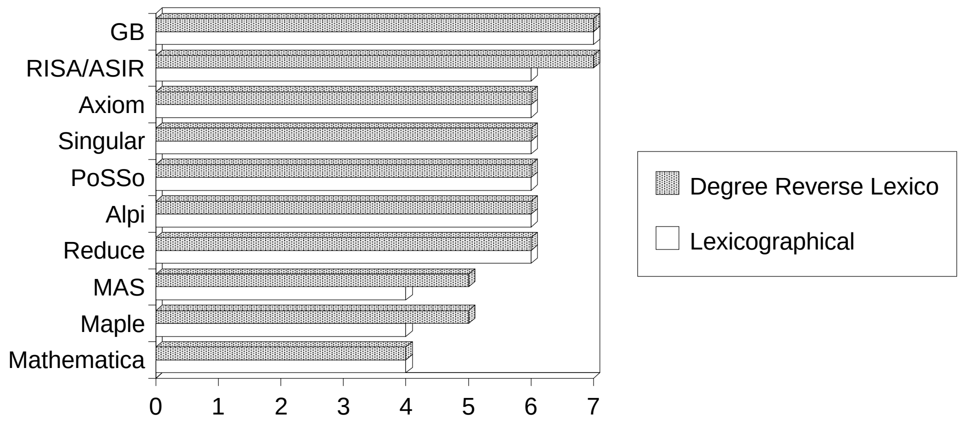

Figure 17. Speedup for BigInt/sugar method (Alliant 8 procs)

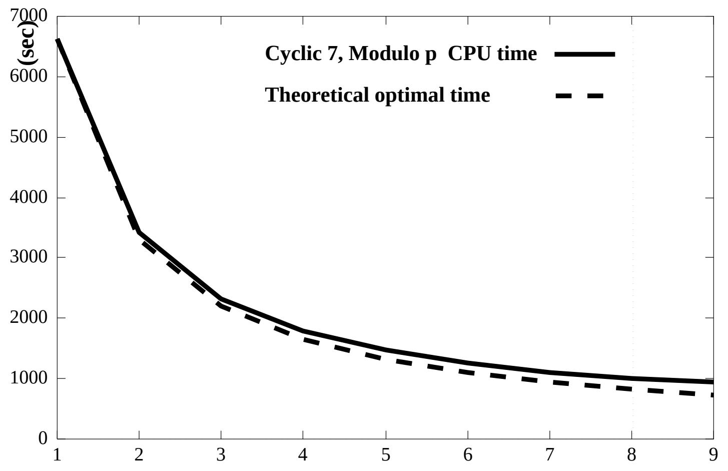

Figure 18. Time in secs/Nb of Processors FGLM Cyclic 7

Backelin (J.) et Fröberg (R.). – How we proved that there are exactly 924 cyclic 7-roots. *In : ISSAC' 91*, éd. par Watt (S. M.). pp. 103–111. – ACM.

Backelin (Jörgen) et Hollman (Joachim). – Calculating gröbner bases fast. *In : DISCO 92*.

Backelin (J.). – *Square multiples n give infinitely many cyclic n–roots.* – Technical Report 8, Reports Matematiska Institutionen, Stockholms Universitet, 1989.

Becker (T.) et Weispfenning (V.). – *Groebner Bases, a Computationnal Approach to Commutative Algebra.* – Springer-Verlag, 1993, *Graduate Texts in Mathematics.*

Björk (G.). – Functions of modulus one on z_p whose fourier transforms have constant modulus. *In : Proceedings of Alfred Haar Memorial Conference, Budapest, Colloquia Mathematica Societatis János Bolyai*, pp. 193–197.

B.Sturmfels. – Gröbner bases of toric varieties'. *Tohoku Mathematical journal*, vol. 43, 1991, pp. 249–261.

Buchberger (B.). – *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal.* – PhD thesis, Innsbruck, 1965.

Buchberger (B.). – An algorithmical criterion for the solvability of algebraic systems. *Aequationes Mathematicae*, vol. 4 (3), 1970, pp. 374–383. – (German).

Buchberger (B.). – A criterion for detecting unnecessary reductions in the construction of gröbner basis. *In : Proc. EUROSAM 79.* pp. 3–21. – Springer Verlag.

Buchberger (B.). – Gröbner bases : an algorithmic method in polynomial ideal theory. *In : Recent trends in multidimensional system theory*, éd. par Reidel. – Bose, 1985.

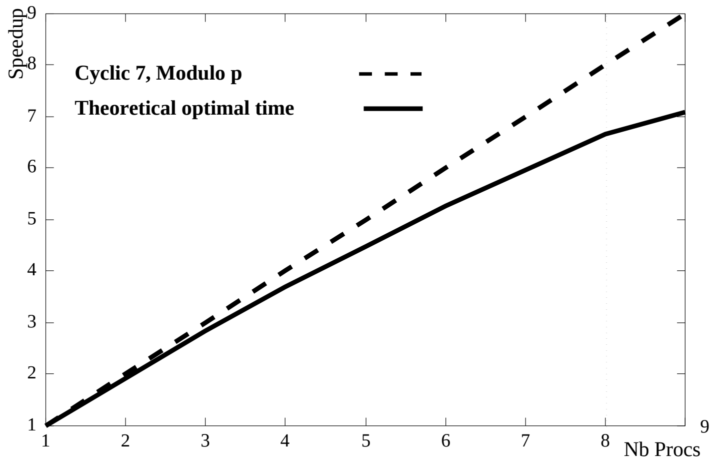

Figure 19. Speedup/Nb of Processors FGLM Cyclic 7

Char (B.), Geddes (K.), Gonnet (G.), Leong (B.), Monagan (M.) et Watt (S.). – *Maple V Library Reference Manual.* – Spinger-Verlag, 1991. Third Printing, 1993.

Davenport (J.), Siret (Y.) et Tournier (E.). – *Calcul Formel.* – Masson, 1993. 2^e édition révisée.

Davenport (J.H.). – *Looking at a set of equations.* – Technical Report 87-06, University of Bath, 1997. Technical report.

Faugère (J.C.) et Lazard (D.). – The combinatorial classes of parallel manipulators. *Mechanism and Machine Theory*, February 1994.

Faugère (Jean-Charles), Gianni (Patrizia), Lazard (Daniel) et Mora (Teo). – Efficient computation of zero–dimensional gröbner bases by change of ordering. *J. Symbolic Computation*, vol. 16, 1994, pp. 329–344.

Faugère (J.C.). – *Résolution des systèmes d'équations algébriques.* – PhD thesis, Université Paris 6, Feb. 1994.

Fitch (J.). – Solving algebraic problems with reduce. *J. Symb. Comp.*, vol. 1, 1985, pp. 211–227.

Gianni (Patrizia), Mora (Teo), Robbiano (Lorenzo) et Traverso (Carlo). – Hilbert functions and Buchberger algorithm. – 1994. (submitted).

Giovini (A.), Mora (T.), Niesi (G.), Robbiano (L.) et Traverso (C.). – One sugar cube, please, or selection strategies in the Buchberger algorithm. *In : Proceedings of the 1991 International Symposium on Symbolic and Algebraic Computation*, éd. par Watt (S. M.). ISSAC. – ACM Press.

Gräbe (H.G.) et Lassner (W.). – A parallel gröbner factorizer. *In : Pasco'94*, éd. par Hong (Hoon). pp. 174–180. – World Scientific.

Grassman (H.), Greuel (G.M.), Martin (B.), Neumann (W.), Pfister (G.), Pohl (W.), Schönemann (H.) et Siebert (T.). – *Standard bases, syzygies and their implementation in SINGULAR.* – Technical Report 251, Universität-Kaiserlautern, March 1994. Preprint.

Grassman (H.), Greuel (G.M.), Martin (B.), Neumann (W.), Pfister (G.), Pohl (W.), Schönemann (H.) et Siebert (T.). – *SINGULAR User Manual.* – Technical report, Universität-Kaiserlautern, 1995.

Hearn (Anthony C.). – *Reduce user's manual, Version 3.3.* – The RAND Corporation, July 1987, report cp 78 edition.

Hollman (J.). – *Theory and Applications of Gröbner Bases.* – PhD thesis, Royal Institure of Technology, Stockholm, Sweden, September 1992.

Jenks (R. D.), Sutor (R. S.) et Watt. (S. M.). – Scratchpad II: An abstract datatype system for mathematical computation. *In : Trends in Computer Algebra, Proc. Internat. Symp.*, éd. par Janßen (R.). pp. 12–37. – Bad Neuenahr, May 1987. Lecture Notes in Computer Science 296.

Jenks (Richard D.) et Sutor (Robert S.). – *Axiom, the Scientific Computation System.* – Springer-Verlag, 1992.

Katsura (K.). – Theory of spin glass by the method of the distribution function of an effective field. *Progress of Theoretical Physics*, no87, 1986, pp. 139–154. – Supplement.

Lazard (D.). – Solving zero-dimensional algebraic systems. *J. Symb. Comp.*, vol. 15, 1992, pp. 117–132.

Lazard (D.). – A new gröbner basis algorithm. – 1993. Not published.

Lazard (D.). – Systems of algebraic equations (algorithms and complexity). *In : Proceedings of Cortona Conference*, éd. par Eisenbud et Robbiano. – Cambridge University Press.

Möller (H.M.), Mora (T.) et Traverso (C.). – Gröbner bases computation using syzygies. – January 1992. preprint.

Pottier (L.). – *Gröbner bases of toric ideals.* – Technical Report 2224, INRIA Sophia Antipolis, april 1994.

Robbiano (L.). – Term orderings on the polynomial ring. *In : Proceedings of Eurocal'85 (Linz).* – Lect. Notes in Comp. Sc., 1985.

Sawada (H.), Terasaki (S.) et Aiba (A.). – Parallel computation of gröbner bases on distributed memory machines. *J. Symb. Comp.*, vol. 18 (3), september 1994, pp. 207–222.

Senechaud (P.). – Implementation of a parallel algorithm to compute a gröbner basis on boolean polynomials. *In : Computer Algebra and Parallelism.* pp. 159–166. – Academic Press.

Siegl (K.). – A parallel factorization tree gröbner basis algorithm. *In : Pasco'94*, éd. par Hong (Hoon). pp. 356–372. – World Scientific.

Stillman (M.) et Bayer (D.). – *Macaulay User Manual*, 1989. available via anonymous ftp on `zariski.harvard.edu`.

Traverso (C.) et Donati (L.). – Experimenting the gröbner basis algorithm with the alpi system. *In : Proceedings of the 1989 International Symposium on Symbolic and Algebraic Computation*, éd. par Press (ACM). ISSAC '89.

Traverso (Carlo). – Gröbner trace algorithms. *In : ISSAC '88, Roma.* pp. 125–138. – Lect. Notes in Comp.Sc.

Vidal (J.P.). – The computation of gröbner bases on a shared memory multiprocessor. *In : Design and implementation of symbolic systems.* pp. 81–90. – Springer Verlag.