Proceedings of PASCO'94

A strategy-accurate parallel Buchberger algorithm*

Giuseppe Attardi
Dipartimento di Informatica
Corso Italia 2 - Pisa
attardi@di.unipi.it

Carlo Traverso
Dipartimento di Matematica
Via Buonarroti 2 - Pisa
traverso@dm.unipi.it

Abstract: We describe a parallelized form of Buchberger algorithm that is strategy-accurate, i.e. that exactly simulates the sequential algorithm. The algorithm has been experimentally implemented on a a network of workstations, and notwithstanding some problems the experimentation gives evidence that the approach is viable. Plans for further work are discussed.

Keywords: Buchberger algorithm, Gröbner bases, strategies, reductors

1 Introduction

The Buchberger algorithm for the computation of Gröbner bases is one of the fundamental algorithms for polynomial system solving. The algorithm is intractable from the perspective of computational complexity, but in practice it can solve a considerable number of interesting problems (that are of course not the worst-case ones).

Increases in the performance of computers, both in speed and in size, have made more problems practically solvable; however the most striking progress has been made in the algorithm itself: a better understanding of some key points, and a better tuning of the strategies have produced advances well beyond the advances due to the hardware.

Parallel variations for Buchberger algorithm have been attempted in different contexts, but usually with little success; in our opinion the main drawback is that a naive use of parallelism destroys the organizing effect of a good strategy, and leads the computation toward the theoretical worst-case complexity, making it unfeasible. Most parallel implementations that we know of only work for toy examples, which are already solved very quickly by any efficient sequential implementation.

In this paper we describe a parallel formulation of Buchberger algorithm. The main characteristic of this formulation is that the algorithm strictly simulates a sequential implementation. This may seem a crazy approach for a parallel algorithm, but this turns out to be crucial for the Buchberger algorithm: it is widely known that the success of a computation with Buchberger algorithm strictly depends on a rigid application of correct strategies, and even an occasional deviation from the strategy may lead to a dramatic coefficient or combinatorial growth. Strictly adhering to a strategy seems not to allow parallelization; this antinomy is solved through a "process manager", that subdivides the whole algorithm in many simple tasks, and has the responsibility of "pasting" the different parallel processes into the proper sequence, even discarding some intermediate results apparently useful, but not complying with the strategy. The same type of concern is addressed in [7] for the Knuth-Bendix completion procedure.

This is possible only since experimentations on the "simplification strategy" (see below) seem to give support to the "older first, never delete" strategy. The same strategy was chosen in many other implementations, since it is the straightforward one.

The algorithm has been implemented on a network of independent processors. The algo-

*Research performed with the contribution of CEC under contract ESPRIT BRA 6846 POSSO, Polynomial System Solving

rithm however can be implemented also in other architectures, e.g. shared-memory architectures.

Even though using a network involves duplication of significant amount of information on many workstations, this architecture has the advantage of being very common, hence the algorithm could be more widely applied. Shared memory processors might be better, (Cfr. [21]) but they pose different problems, and are less frequently available. The transmission overhead is sensible, but may be reduced through a more clever binary encoding of the data. The experiments show that the load is evenly distributed, and bottlenecks do not appear in the large examples for which a parallel implementation makes sense.

The implementation is based on ECoLisp, an *embeddable* Common Lisp, which can be linked with C based applications [1]. The algorithm implementation derives from AlP*I*, the system for experimenting with the Buchberger algorithm developed in Pisa, [20]. The research has been conducted with partial support of the Commission of the European Communities, contract ESPRIT BRA 6846 POSSO (Polynomial System Solving).

The prototype implementation uses a slightly different form of the communication protocol described in this paper, which requires more messages, but has the same overall structure. The speedup obtained has been uneven, but usually weak; we expect that a careful implementation of the algorithm as described in this paper could give better results.

We discuss at the end of the paper further parallelization possibilities of Buchberger algorithm.

2 A sketch of Buchberger algorithm

We give a short description of (our interpretation of) Buchberger algorithm, just to fix the notation. The algorithm is otherwise considered to be known. Refer to any of the standard references (e.g. [3], [5], [10], [14], [16], and in particular [11] for the most efficient version) for a more complete description.

Let k be a field, and assume that we have a term-ordering in a polynomial ring $k[X] = k[x_1, \ldots, x_n]$; a power-product in $k[X]$ is a product of indeterminates, a monomial is a product of a constant $\in k$ and a power-product. The leading power-product $Lpp(f)$, the leading monomial $Lm(f)$ and the leading coefficient $Lc(f)$ of a polynomial f are defined in rapport with the term-ordering. Given two polynomials f, g, we define $Sp(f,g) = (Lc(g)/\gcd(Lpp(f), Lpp(g))f - (Lc(f)/\gcd(Lpp(f), Lpp(g))g$.

Assume that we have a finitely generated ideal $(f_1, \ldots, f_m) = I \subseteq k[X] = k[x_1, \ldots, x_n]$, and that we want to compute a Gröbner basis of I.

This is done in two phases:

a) incrementally add elements to $B = \{f_1, \ldots, f_m\}$ (obtaining a redundant Gröbner basis)

b) discard redundant elements and interreduce B, obtaining the (uniquely determined) reduced Gröbner basis of I.

The first phase is the main part of the algorithm, and consists of an initialization phase and of a main loop. The main loop is governed by a set of *critical pairs*, i.e. a set of pairs of elements of B, called the *pair queue*, and the loop ends when this queue is empty. The *leading power-product of a pair* is the least common multiple of the leading power-products of their elements.

Each iteration of the loop selects one pair (f_i, f_j), computes its S-polynomial $Sp(f_i, f_j) = f$, simplifies it (this is another loop, that will be discussed in a following paragraph), then, if f has been reduced to a polynomial $\bar{f}$ different from 0, add $\bar{f}$ to G, adjust the pair queue (in this phase some pairs are deleted, and some pairs involving the new element $\bar{f}$ are added). This ends the main loop. When the queue is empty, B is a redundant Gröbner basis.

The final reduction consists in sorting B with increasing leading power product, discarding the elements whose leading power-product is multiple of some other one, then simplifying each element using only the preceding ones.

The inner loop (the simplification loop) has two phases: the Lpp-reduction and the total reduction (the second phase is optional, but experimentally convenient). Both consist in rewriting one monomial of the polynomial f with an element f_i of the basis in the following way: if $f = a\tau + \rho$ and $f_i = a_i\tau_i + \rho_i$, with a, $a_i \in k$ constants, τ and τ_i power-products such that τ_i is the leading power-product of f_i and $\tau_i\mu = \tau$, then we replace f with $a_i\rho - a\mu\rho_i$. The power-product τ usually is the highest rewritable power-product, hence as long as the leading power-product of f is rewritable, one has to rewrite it, (this is the Lpp-reduction), and in the second phase the leading power-product of f remains unchanged. However any other rewriting scheme can be applied: the algorithm remains correct, but in general more rewritings are necessary to obtain a non-rewritable polynomial.

When a monomial can be rewritten, it may be possible to rewrite it in different ways (i.e. with different f_i); several strategies have been tested, but experimentally the best strategy seems to be the trivial one: pick the first one, i.e. add new elements to the end of the basis, and when looking for possible simplifications scan the basis from the first element on. The experiments confirm that this strategy leads to lower coefficient growth.

At the end of the simplification loop, if f is not zero, three actions are performed:

a) f is appended to the basis (optionally after making it monic — or primitive, if k is a quotient field of a factorial ring A and we want to use A-arithmetic).

b) a pair (f_i, f_j) is deleted from the queue if the leading power-products of (f_j, f) and of (f_j, f) both strictly divide the leading power-product of (f_i, f_j) (this is the B-criterion of [11]).

c) consider the set S of all pairs of elements (f_i, f); discard from S all elements (f_i, f) such that the leading power-products of f_i and of f are coprime, as well as every other pair having leading power-product equal or multiple to the leading power-product of (f_i, f) (T-criterion).

d) sort S in a convenient order (*selection strategy)*, and discard every pair whose leading power-product is equal or multiple of a preceding leading power-product (M-criterion).

e) merge S with the pair queue, using again the selection strategy.

The same actions are performed to initialize the procedure, extracting f from the initial basis. Optionally, in this phase, each element extracted can be simplified using the preceding elements.

Remark that to perform the points b) to e) above, one only has to know the leading power-product of f, hence (already considering parallelization) one can adjust the pair queue as soon as the *Lpp*-reduction of f is completed, without waiting for the total reduction to be completed.

There are two points in which there is a big freedom in the Buchberger algorithm: the selection strategy and the simplification strategy (the choice of a simplification between the different simplifications possible for a single monomial). We assume that both strategies are orderings, meaning that the relative order of two pairs (or reducers) does not depend from the others, hence the changes are always additions or deletions.

It is a common experience that a correct choice of the strategies is critical for the complexity of the algorithm. A wrong choice, maybe a single seemingly harmless modification, may lead to incredible growth of the coefficients and of the number of pairs to process. See [13] for a description of a good strategy and a striking example.

A "wrong" parallelization is the following: give different pairs to different processors, and add to the basis new elements as soon as they appear as a result of one of the processors. On the other hand, a strategy accurate implementation where a processor must wait for the previous pairs to be completed, may waste a lot of time: the computing time of different pairs is uneven. Apparently, up to now all parallel versions of the algorithm have fallen into these two categories; our remark might explain some problems of other implementations.

On the other way, when the input is homogeneous the influence of the strategy is considerably smaller, provided that the selection strategy proceeds in increasing degree. It might be interesting to test a strategy-inaccurate version in these cases.

Our parallel algorithm tries to avoid both difficulties; the strategy is strictly followed, and a processor is rarely idle: if a processor is waiting, it can ask for more work to do.

3 Parallelizing the outer loop

We describe a parallelized Buchberger algorithm, with $n+1$ processors. One of the processors (the *process manager*) performs a special task and the other processors (the *simplifiers*) all perform the same task.

We completely describe an algorithm that computes a redundant, non reduced Gröbner basis.

The communications (input and output) are always between one processor and the central manager (in a star-like topology). Every processor has a complete copy of the current basis. This can be seen as a useless duplication of information. However, the experience with other implementations shows that the information contained in the basis is frequently small with respect to the total amount of the computation. A sequential computation may take several hours, and the final output is usually relatively short. A basis element in a simplifier is called a *reducer*.

We describe separately the two types of processes that run in the different processors. In this descriptions we assume that some operations involved in the usual Buchberger algorithm are known. These include:

FindReducer: given a monomial, find a reducer that can reduce it (i.e. whose leading monomial divides the leading monomial) or prove that none exists

Divide: given a polynomial, one of its monomials, and a reducer of the monomial, simplify the polynomial with the reducer

AddPolynomial: given a Buchberger algorithm in execution, add a polynomial to the basis, updating correspondingly the reducers and the critical pairs

For these functions a polynomial is considered to have a *current monomial*, i.e. a monomial on which we are currently acting; the current monomial can be empty (meaning that we are at the end of the polynomial); the result of **Divide** is again a polynomial with a current monomial, that is the first monomial of the tail that has been modified.

3.1 The simplifiers

A simplifier executes an infinite loop, listening at an input channel. According to the message received an action is performed, and if no message is received a default action is taken.

Each simplifier maintains two data sets, composed of polynomials:

- the list of reducers;
- the list of simplificands, of which one is the current simplificand, and has a current monomial.

Each simplificand can be identified by a critical pair of reducers, that is the critical pair that has given origin to it; each reducer can be identified by its position in the list (that will never change).

We do not discuss here the format of data and their semantics. The list of possible messages, and the corresponding arguments is the following, with the corresponding actions:

INIT `init-data` This message sets reducers and simplificands to the empty list; the `init-data` set the current ring (number of variables, coefficients, ordering,...).

POLY `pol n` Append the polynomial `pol` to the reducer set; reset the simplificand list to the leading monomial of the first element. The integer n is the position of the polynomial in the list, and is sent to synchronize the processes in case that the medium does not ensure that the message are received in the same order in which they have been sent. If the position number n is incorrect, this means that a previous **POLY** command has not been received; in that case, wait until it arrives.

START `pair` Insert the S-polynomial of the critical pair `pair` in the list of reducers, in the correct order (the order is defined by the simplification strategy on the pairs associated to the polynomials). If the new polynomial precedes the current polynomial, set the current polynomial to it.

FINISH `pair n` Suspend the current work, extract the polynomial corresponding to `pair` from the simplificand list, completely reduce it, send a message **DONE** `pair pol` where `pol` is the result of the reduction if `pol` is not 0, or a **ZERO** `pair` otherwise. The number n is a check of the length of the polynomial basis; if it is incorrect, this means that some **POLY** message has still to be received; in that case, wait until all the polynomials are available.

DISCARD `pair` delete the polynomial corresponding to `pair` from the simplificand list. In the case that the current monomial is in the polynomial that is discarded, set the current monomial to the polynomial that immediately follows. Remark that when a **DISCARD** command is received, the corresponding polynomial is not assured to exist.

DELAY suspend current work, get ready to receive next polynomial and send back message **INPUT** to the manager.

END terminate the activity of the simplifier.

If no message is received, then a standard action is performed, We give two different forms of the standard action, one of these is completely strategy-accurate, but may cause a bottleneck, the other is partially strategy-accurate, and can be used only in some cases. The difference is in the `advance` function: in the first case it causes the current monomial to be the leading monomial in the next polynomial, in the second case it causes the current monomial to be the next monomial of the current polynomial, and if the current polynomial is at end, to be the leading monomial of the next polynomial.

We will refer to the first variant as *totally accurate*, to the second as *partly inaccurate*.

The standard action consists in the following:

- If the current monomial can be simplified by one of the reducers, perform the simplification, otherwise `advance`;
- If the current polynomial is zero, then send a **ZERO** `pair` message, delete the zero polynomial from the simplificands, and `advance`;
- If we are at the end of the simplificand list, send an **IDLE** command.

When an **IDLE** message is sent, no further action is taken until a message is received. This avoids the accumulation of too many **IDLE** commands, hence the need from the process manager to check them.

It is of course possible that an **IDLE** message is sent while a message with more work is already arriving. In that case more work to do might arrive when the simplifier is no longer idle. The description of **START** takes care of this possibility.

3.2 The process manager

The task of the process manager consists in maintaining the queue of pairs, distributing tasks to the simplifiers, and dispatching messages.

The process manager starts performing the usual initializations of Buchberger algorithm, sends an **INIT** command to every simplifier, then for every polynomial `pol` in the basis sends a message **POLY** `pol` to every simplifier. Then it waits for messages form the simplifiers (which will issue immediately **IDLE** messages).

For every critical pair that has to be computed, the manager maintains the address of the simplifier to which it has been assigned, if any.

The actions of the manager are the following:

- When an **IDLE** message is received from a simplifier, the first unassigned pair is sent to the simplifier with a **START** message; if no pair is available, the message is discarded (this cannot cause deadlocks: either all the critical pairs have already been assigned, or eventually a new polynomial will arrive, that will restart all the simplifiers, and will cause an **IDLE** message to be resent, if no work exists)
- When a **DONE** `pair pol` is received, `pair` is removed from the pair queue; `pol` is added to the basis, a **POLY** `pol n` (where n is the position number of the polynomial in the basis) is broadcast to all the simplifiers. If an already assigned pair gets removed from the queue, a **DISCARD** `pair` is sent to the relevant simplifier, before discarding the pair. Then a **FINISH** message is sent to the simplifier to which the first of the remaining pairs is assigned (in case that the first pair is unassigned, wait until a processor is idle, and send to it a **FINISH** message with the first pair).
- When a **ZERO** `pair` message is received, the corresponding pair is deleted. When all the pairs are deleted, the algorithm terminates, and the current basis (present in all the nodes) is a redundant Gröbner basis.

Broadcasting a polynomial to the simplifiers involves a sequence of messages: **DELAY** from manager to simplifier, **INPUT** back from simplifier to manager and finally **POLY** from manager to simplifier. This protocol is needed to cope with buffering of communications as explained in the section on the implementation.

Remark that it is possible that a **ZERO** and a **DISCARD** message are issued at both ends at the same time; the protocol described ensures that no inconsistency may result, ensuring that deleting a non-existent item is not an error.

3.3 Correctness and robustness of the algorithm

We consider the algorithm in the totally accurate variant; the partially inaccurate variant will be discussed at the end.

We prove that the algorithm is correct, that the polynomials obtained are the same of the sequential algorithm, and that the algorithm does not depend on the order in which the messages are received. The algorithm may hang only if some message is lost.

We perform a reduction only in two cases:

- the monomial is the leading monomial of the polynomial, or
- all the pairs that precede the current pair in the strategy have already been completely simplified, and the full list of polynomials is available.

In both cases, the simplification performed is exactly the one that would have been performed by the sequential algorithm: a monomial is simplified by the first available reducer in both cases, and if no reducer is available at some moment for a leading monomial, then we pass to the tail simplification only if we are sure that the head will never be simplifiable.

For the partly-inaccurate variant this remark is false; this variant can be adopted only if one can prove that the result of the simplifications will be unaffected by this change. This is for example true for the "normal selection strategy", but it is not true for other strategies.

The totally accurate algorithm unfortunately has an intrinsic limitation, due to the fact that at most one pair reducer is performing non-*lt* reductions at one time. In most of the examples computed the non-*lt* reduction phase accounts for less than one tenth of the whole algorithm, hence one can expect from the parallel algorithm a maximum speed-up of a factor 10, independently of the number of processors. This will result in some cases in having all pairs distributed to the simplifiers, with their *Lpp*-reduction completed with the current basis, waiting for the first total reduction to be completed.

It is hence very important to prove conditions in which the partly-inaccurate algorithm can be safely used.

3.4 Number of the messages

Here we count the total number of the messages that are exchanged in the algorithm; we will see that this number is small. Let n be the number of simplifiers, m' is the number of elements in the input basis, m the number of the polynomials in the (redundant) basis computed, p the number of pairs that are reduced in the algorithm.

INIT is sent n times;

POLY is sent nm times;

DELAY is sent nm times at most;

INPUT is sent nm times at most;

START is sent p times at most;

FINISH is sent p times at most;

DISCARD is sent p times at most — experimentally, this message is very infrequent;

END is sent n times;

IDLE is sent at most $nm + p + n$ times (every **IDLE** is followed either by a **START** or by a **POLY** to resume the work, or by an **END** to finish);

DONE is sent $m - m'$ times at most;

ZERO is sent $p - m + m'$ times at most (this corresponds to those pairs for which a **DONE** is not sent) and a **DISCARD** is not received).

Hence the total number of messages is at most $4nm + 5p + 2n$. Remark that every message is short except **POLY** and **DONE**, that have a polynomial as argument; **INIT** has an argument that encodes the arithmetic setting, and its length corresponds to the complication of the setting; all other command arguments are composed of a pair at most.

Remark that in shared memory setting the **DONE** and **POLY** messages do not require that the polynomial is passed as an argument, but only his address has to be passed. This remark can also be used simulating the shared memory architecture using a file to simulate the shared memory. This allows to have only short messages, but of course accessing a remote file through the network is rather liable to increase the network traffic than to reduce it.

4 Heuristic considerations

In this section we explain the experimental findings that make plausible that this form of the Buchberger algorithm works fine. These findings come from experimentation with the sequential Buchberger algorithm implemented in the AlP*I* program ([20]).

The problems with any parallel algorithms are: transmission overhead, useless duplication of computing, deadlocks and bottlenecks. This may happen even in case of an ideal network. Implementations have moreover problems arising from the network software, like the amount of buffering of messages: we will not address these problems, even if any successful implementation has to take care of them.

Transmissions

The basis has to be transmitted in full to every processor. One form of basis transmission occurs in any case: the output. We can measure the output transmission time, and although it is often quite large, the computing time is usually much larger.

Useless computations

When adding a polynomial, one can delete pairs from the queue. Experimentally, this does not happen often, and the pairs deleted are often not the first in queue. Hence useless computations should not be frequent.

Idle processors

This may happen for two reasons: either the central manager is busy (usually, to add one element to the basis), or no pairs are available.

In AlP*I* one has the possibility to compute separately the reduction time and the pair managing time. There is a big difference between ideals generated by binomials (difference of monic monomials) and other ideals: for the first type the pair managing time often exceeds one half of the total computing time, for the others usually the pair managing time is no more than 2% of the total computing time. Hence binomial ideals are unsuited for our implementation. In a section at the end we discuss shortly other ideas for these ideals. For other polynomials, waiting for the central manager to complete managing the queue is usually not a problem unless the number of processors is very high, and in this case one can easily divide the task of the central manager in two different processes, the queue adjustment and the task distribution, in such a way that this problem too can be solved.

Moreover, the low influence of managing pairs in the total computing time heavily depends on a correct implementation: marginal changes in the algorithm may cause the pair managing time to raise to a considerable fraction of the total computing time.

Experimentally, for the long-running computations (and we are interested to them), the queue has more often hundreds of elements than scores, and the queue is almost empty only sometimes in the very first and very last phase. Hence, the lack of work to do is quite unusual.

5 Implementation and improvements

5.1 The experimental implementation

The algorithm has been experimentally implemented in Common Lisp, using ECoLisp. The protocol described here addresses some of the drawbacks of an earlier implementation.

The major problem has been synchronizing the processors for communication. In fact, the network software provides buffering of messages but only with a limited size. If the buffer fills up, portions of messages gets lost. Since the activities of the manager and the simplifiers are asynchronous, it can happen for instance that a number of quite long polynomials are generated while one simplifier is busy performing a complex reduction. If the manager broadcasts each new polynomial to all simplifiers as soon as it is generated, it can happen that many such polynomials queue up in the input channel of the busy simplifier and fill up its buffer.

The new protocol overcomes this problem, by having the manager to send a new polynomial only when a simplifier is ready to receive it. Before sending the polynomial, the manager sends a very short message **DELAY** to notify the simplifier that it should wait for a new polynomial. The simplifier will respond with either an **IDLE**, **ZERO**, **DONE**, or **INPUT** message, and goes into receive mode. After receiving one of these messages, the manager can safely send the new polynomial.

The protocol described here further improves on the prototype implementation by trying to reduce the number of messages exchanged between the manager and the simplifiers. The simplifiers in fact can decide autonomously which pair to reduce, after completing one reduction, in lack of further instructions from the manager. In the prototype implementation each simplifier would have to ask the manager for a new pair to reduce.

The present implementation works fine when there are few nodes and the polynomial are not too long, but in that case the transmission overhead is quite large, and the speed-up is modest.

Some deeper analysis of the experiments is required to understand what limits the speed-up. Note that the length of the polynomials that have to be transmitted is quite unusual in parallel computing, and on the other hand a parallel implementation is interesting especially when the polynomials are large hence the computation of a single simplification takes a long time (seconds or even minutes are quite usual in some interesting examples).

5.2 Using the POSSO library

The choice of LISP as an implementation language is good for the flexibility, but may impose an unreasonable overhead for a version to be used routinely.

The POSSO project is developing a solver in library form, using the C++ language. We will use the library to prepare an efficient version of the simplifiers; the central control probably may remain in LISP. Moreover the efficient binary protocol POSSO-XDR will be used for the distributed version.

POSSO-XDR [17] is a binary data encoding protocol extending the XDR protocol, specialized for the transmission of data concerning polynomials. It has been designed with special care in encoding and decoding efficiency. For very large data it can be combined with a compression algorithm.

5.3 Improvements and revisions

We list in this section some possible improvements and revisions of the algorithm.

It may happen that at some stage one of the processors begins many computations, and this may cause a bottleneck at a later stage. One obvious possibility is to determine a maximum number of pairs to give to a single processor (this may be dynamically adjusted, looking at the length of the queue). A danger is to be stuck when every processor has many pairs, and none has the first in queue; this may happen, but is easy to avoid.

Another possibility is to give to every processor a minimum number of pair to process: this in order to minimize the time in which a processor is idle (between the **IDLE** message and the receiving of another pair). Also, if a pair exists that precedes the first pair given to a pair reducer, send this pair, to do first the work with higher priority.

A considerable improvement can be the possibility to reclaim from one processor a partially completed computation; this may be useful if one processor is idle, and another has a long list to complete, or even if one processor appears to slow down the others. This can very well happen when using a large set of workstations for which this task has to be performed in the spare time. If at some moment the workstation is used for another time-consuming job, we want to stop using it, but we want to use the work done up to that moment.

All these improvements require that the central manager controls the algorithm in a more clever way; but they risk that the central manager becomes the bottleneck.

A real improvement may be an implementation on a shared memory multiprocessor machine; the details of a port to this architecture are more or less obvious.

5.4 Parallelizing the central manager

The central manager can work asynchronously: when it updates the pair queue, it is possible to continue to handle the requests of the simplifiers.

In the case that the load of the central control results too heavy, it is possible to split it in different processors; the updating is divided into updating the old queue (B-criterion), preparing the new queue (M-criterion) and merging the two. The B-criterion is immediately parallelizable (it consists in separate tests); the preparation of the new queue can be made through the preparation of separate queues, and a merging with cancellations. The possibilities of parallelization of merging are well known.

A further parallelization, or rather a different way of dispatching messages, is possible when the physical network is a wide area network. In each local component we may have a local manager, that is seen as a manager by the local simplifiers, and as a simplifier by the central manager.

5.5 Homogeneous ideals

Homogeneous ideals are less sensible to strategies, at least until the pairs are processed in increasing degree. Slightly different algorithms can be used; in particular, one can consider separately the reduction by elements of degree lower than the current degree, and by elements of the current degree. This latter is a form of linear algebra.

A completely different approach is hence reasonable: proceed degreewise, without regard of the strategies, subdividing the two parts as sketched above; the smaller-degree reducers need to have the old elements, and polynomial arithmetic capabilities, and the current-degree reducers instead only need to accumulate the new elements and interreduce them with linear algebra.

As soon as a pair is reduced by the first type of reducers, it can be sent to the second type, that automatically provide a reduced basis. The pairs of the current degree can be even subdivided evenly to the different processors, up to equalizing the load when one becomes idle.

This architecture is well fitted with the Hilbert-driven algorithm of [12], since the linear reducers can send a stop to the pair reducers when the end of the degree is reached. This makes the two algorithms radically different.

Of course at the end of the degree a big management has to be made, to redistribute the work to the different processors.

5.6 Binomial ideals

As already remarked, binomial ideals, i.e. ideals generated by power-product differences are especially useful for some applications, and have special properties (see [15], [9], [18]). In particular, every polynomial appearing in the Buchberger algorithm is difference of two power-products. In most applications, whenever the two power-products of a binomial polynomial have a common factor, we are authorized to divide by it, hence the two power-products have disjoint variables. This allows to represent a binomial as vector with integer coefficients.

In practice, most of the time spent in Buchberger algorithm for binomial ideals (in a sequential implementation) is the pair management, and not the polynomial reduction. Hence the target for a speed-up is given by the power-product operations. An implementation giving one variable to each processor seems hence recommended.

An implementation of this type has been attempted on the SIMD machine APE, [8], with good results.

References

[1] Attardi, G. , *The Embeddable Common Lisp*, Lisp User and Vendor Conference 1994, Berkeley, August 1994.

[2] C. Battista, S. Cabasino et al., *The APE-100 Computer: the Architecture*, International Journal of High Speed Computing, 1993

[3] Becker, V. Weispfenning Gröbner bases, Springer Verlag, GTM, 1993

[4] Buchberger, B. *An Algorithm for Finding a Basis for the Residue Class Ring of a Zero-Dimensional Polynomial Ideal*, Aequationes Mathematicae vol. 4, 1970, pp. 374–383, (Ph.D. Thesis, Math. Inst. Univ. Innsbruck, 1965).

[5] Buchberger, B. *Gröbner bases: an algorithmic method in polynomial ideal theory*, in: Recent trends in multidimensional systems theory, 1985, N. K. Bose, ed. pp. 184–232, D. Reidel Publ. Comp.

[6] Buchberger, B. *The parallelization of Critical pair completion procedures on the L-machine*, pp. 54–61, in: Proc. of the Japanese Symposium on functional programming, 1987.

[7] R. Bündgen, M. Göbel, W. Küchlin, *A fine-grained parallel completion procedure*, in: ISSAC-94 Proceedings, Oxford 1994.

[8] A. Ciampa, M. Silvestri *The vectorial approach for binomial ideals: an overview*, Technical report (in preparation)

[9] P. Conti, C. Traverso, *Buchberger algorithm and integer programming*, AAECC-9, Springer, LNCS, 1991

[10] D. Cox, J. Little, D. O'Shea, Ideals, varieties, and algorithms, UTM, Springer Verlag, 1992

[11] R. Gebauer, H.M. Möller, *On an installation of Buchberger's algorithm*, J. of Symb. Comp. 6, 1988

[12] P. Gianni, T. Mora, L. Robbiano, C. Traverso, *Hilbert functions and Buchberger algorithm*, (submitted) 1994

[13] Giovini, A., Niesi, G. Robbiano, L., Traverso, C., *"One sugar cube, please" OR Selection strategies in, Buchberger algorithm*, ISSAC 91, ACM, 1991

[14] B. Mishra, Algorithmic Algebra, Springer Verlag, Text and monographs in Computer Science, 1993

[15] F. Ollivier, *Canonical Bases: Relations with Standard Bases, Finiteness Conditions and Application to Tame Automorphisms*, Mega-90, proceedings, Birkhauser, Progress in Mathematics, 1991

[16] Pauer, F., Pfeifhofer, M. *The theory of Gröbner bases*, L'Enseignement Math. vol. 34, 1988, pp. 215–232.

[17] POSSO-XDR, POSSO technical report, draft. Requests to `posso@dm.unipi.it`.

[18] L. Pottier, *Minimal solutions of linear diophantine systems: bounds and algorithms*, Proceedings RTA '91, Como, LNCS **488**, Springer Verlag

[19] Senechaud, P. *Implementation of a parallel algorithm to compute a Gröbner basis on boolean polynomials* pp. 159–166 in: Computer Algebra and parallelism, 1989, Academic Press.

[20] Traverso, C., Donati, L. 1989 *Experimenting the Gröbner basis algorithm with the* AlPI *system* in: ISSAC 89, A. C. M.

[21] Vidal, J. P. *The computation of Gröbner bases on a shared memory multiprocessor* pp. 81–90 in: Design and implementation of

symbolic computation systems, 1990, Lecture Notes in Computer Science **429** Springer Verlag, Berlin–Heidelberg–New York