J. Symbolic Computation (1996) **21**, 411–425

Strategy-Accurate Parallel Buchberger Algorithms[†]

GIUSEPPE ATTARDI[‡] AND CARLO TRAVERSO[§]

[‡]*Dipartimento di Informatica, Università degli Studi di Pisa*
Corso Italia 40, I-56125 Pisa, Italia
[§]*Dipartimento di Matematica, Università degli Studi di Pisa*
via F. Buonarroti 2, I-56123 Pisa, Italia

(Received 1 June 1995)

We describe two parallel versions of the Buchberger algorithm for computing Gröbner bases, one for the general case and one for homogeneous ideals, which exploit coarse grain parallelism. For the general case, to avoid the growth in number and complexity of the polynomials to reduce, the algorithm adheres strictly to the same strategies as the best sequential implementation. A suitable communication procotol has been designed to ensure proper synchronization of the various processes and to limit their idle time. We provide a detailed analysis the maximum potential degree of parallelism that is achievable with such architecture. The analysis corresponds to the results of our experimental implementation and also explains similar results obtained by other authors.

1. Introduction

The Buchberger algorithm for the computation of Gröbner bases is one of the fundamental algorithms for polynomial system solving. From the perspective of computational complexity the algorithm is intractable, but in practice it can solve a considerable number of interesting problems and there are good indications that problems arising from real situations are far from the worst case of the algorithm.

Increases in computer performance 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 those due to the hardware.

Parallel variations of the Buchberger algorithm have been attempted in different contexts, but usually with marginal success; most likely because 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 concentrated on the speed-up of concurrency but exhibit disappointing performance in absolute speed, inferior to an efficient sequential implementation.

In this paper we describe two parallel formulations of Buchberger algorithm, one for

[†] This research has been funded in part by the ESPRIT Basic Research Action, project no. 6846, PoSSo (Polynomial System Solving).

Table 1. Number of pairs with different strategies.

Benchmark	*Lex*	*RLex*	*Benchmark*	*Lex*	*RLex*
`cyclic4`	24	12	`pavelle4`	30	34
`cyclic5`	464	157	`pavelle5`	124	83
`katsura4`	76	39	`robbiano`	43	44
`lazard`	145	53	`rose`	44	44
`morgenstern`	53	57	`trinks`	58	32

the general case and one for homogeneous ideals. The common characteristic of these formulations is that the algorithm strictly simulates a sequential implementation. This is an unusual approach for designing a parallel algorithm, but it turns out to be appropriate for the Buchberger algorithm: it is well known that the efficiency of a computation with Buchberger algorithm depends on the application of proper strategies, which determine the order of polynomial reductions, and even an occasional deviation from the strategy may lead to a dramatic coefficient or combinatorial growth. Adhering to a strategy which imposes an ordering seems to restrict parallelism; this antinomy is solved through a "process manager", which subdivides the whole computation in many independent tasks, and has the responsibility of "pasting" the different parallel activities into the proper sequence, sometimes even discarding intermediate results not complying with the strategy. The same type of concern has been raised in Bündgen *et al.* (1994) for the Knuth–Bendix completion procedure.

In order to validate our approach, we performed both a theoretical analysis and some experiments with an implementation of the algorithm. The theoretical analysis considers an ideal computational model with an infinite number of processors and no communication costs. From simulated traces of executions on such a model we show that there is a finite limit to the potential speed-up achievable with any parallel implementation of the Buchberger algorithm. This limit is moreover fairly small and therefore this constitutes a somewhat negative result on the possibility of obtaining significant improvements from parallelism, at least from coarse grain parallelism.

Apparently this result is in contrast to previous related work (Vidal, 1990; Chakrabarti and Yelick, 1994), which claim almost linear or superlinear speed-up. A more careful look at those results provides an explanation.

The results by Vidal (1990) and Sawada *et al.* (1994) show that the speed-up stops farly soon with a small number of processors (12 to 16) for all the test cases considered.

The results by Chakrabarti and Yelick (1994) seem influenced by a non-optimal choice of term ordering. Table 1 for example presents a comparison between the number of pairs generated with the ordering used in their algorithm (total degree ordering with ties resolved by lexicographic order) and the one used in our implementation (degree ordering plus reverse lexicographic order).

Consider, for instance, the `lazard` benchmark which is discussed in Chakrabarti and Yelick (1994) as exhibiting more than linear speed-up. Clearly a large number of irrelevant pairs is produced in their implementation and therefore a parallel algorithm appears to perform better with more processors since it has good chances of sidestepping irrelevant computations.

An algorithm exploiting better strategies will show less striking improvements with the increase in the number of processors.

Table 2. Comparison of parallel and sequential implementations.

algorithm	Chakrabarti–Yelick	Sawada *et al.*	PoSSo
machine	CM-5	PIM/m	SparcStation5
processors	20	16	1
`cyclic5`	10	4.5	0.84
`cyclic6`		102	23.52
`katsura4`	4.1	1.8	0.28
`katsura5`		9.5	3.55
`pavelle5`	15		0.39

That margins for improvements are smaller when better strategies are employed is confirmed if we compare the effective runtimes of the Chakrabarti–Yelick implementation on a 20 processor CM-5[†], with the implementation by Sawada *et al.* (1994) on a PIM/m with 256 nodes, and with our own sequential implementation in C++ (Attardi and Traverso, 1995) on a single SparcStation5, as shown in Table 2 (times are in seconds).

Sawada *et al.* (1994) report results for up to 256 processors, but they consistently get worse beyond 16 processors.

Since results from benchmarking can be quite misleading, the theoretical analysis that we present later provides some useful insights.

Besides the analysis, we report our experiments with a prototype implementation which confirm the results of the analysis.

The algorithm has been implemented on a network of independent processors for reasons of availability but it would perform quite better on a shared-memory architecture where communication costs are lower.

2. A Sketch of Buchberger Algorithm

We recall the basics of Buchberger algorithm, in order to establish the notation. A more complete description may be found in any of the standard references e.g. Becker and Weispfenning (1993), Buchberger (1985), Cox *et al.* (1991), Mishra (1993), Pauer and Pfeifhofer (1988), and in particular Gebauer and Möller (1988) for the most efficient version.

Let k be a field, and assume that we have a *term-ordering* in a polynomial ring $k[X] = k[x_1, \dots, x_n]$; a *power-product* in $k[X]$ is a product of indeterminates, a *monomial* is a product of a non-zero 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 with respect to the term-ordering. The *leading power-product of a pair* of polynomials $\langle f, g\rangle$ is the least common multiple of $Lpp(f)$ and $Lpp(g)$. The S-polynomial of polynomials f and g, is defined as $Spoly(f, g) = (Lc(g)/\gcd(Lpp(f), Lpp(g))f - (Lc(f)/\gcd(Lpp(f), Lpp(g))g$. We say that f *reduces to* f' *by* f_i *at* τ, and write $f \xrightarrow{\tau}_{f_i} f'$ 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 $\tau_i = Lpp(f_i)$ and $\tau_i\mu = \tau$, then $f' = a_i\rho - a\mu\rho_i$. If $\tau = Lpp(f)$ the reduction is called Lpp-reduction. $f \downarrow_R$ is the irreducible form of polynomial f w.r.t. polynomial set R.

Let $(f_1, \dots, f_m) = I \subseteq k[X] = k[x_1, \dots, x_n]$ be a finitely generated ideal.

[†] We show the comparison on the three benchmarks that have been reported in Chakrabarti and Yelick (1994) with 20 nodes, but the difference appears in all of them.

The computation of a Gröbner basis of I is done in two phases:

(1) add elements to $R = \{f_1, \ldots, f_m\}$, obtaining a redundant Gröbner basis;
(2) discard redundant elements and interreduce R, obtaining the (uniquely determined) reduced Gröbner basis of I.

Phase (1) is the most expensive part of the algorithm, and consists of an initialization and a main loop. The main loop is governed by a set of *critical pairs*, called the *pair queue*, which consists of pairs of elements of R, called *reducers*.

Each iteration of the loop selects one pair $\langle f_i, f_j \rangle$ from the *pair queue*, computes its S-polynomial $Spoly(f_i, f_j) = f$, reduces f to h, then, if h is not zero, adds h to R and updates the *pair queue*, whereby some pairs are deleted, and some pairs involving the new element h are added. When the *pair queue* becomes empty, the loop ends and R contains a redundant Gröbner basis.

Phase (2) consists in sorting R with increasing leading power-product, discarding the elements whose leading power-product is multiple of some other one, then reducing each element using only the preceding ones.

The reduction of an S-polynomial f has two phases: the *Lpp*-reduction and the total reduction, which does not affect correctness, but usually improves performance in practice. Both phases consist in rewriting f with elements of the basis.

A rewriting step consists in finding an $f_i \in R$ such that $f \xrightarrow{\tau}_{f_i} f'$ and replacing f with f'.

In the *Lpp*-reduction, the power-product τ is usually the highest rewritable power-product, while in the total reduction any rewritable power-product is considered. Hence, whenever $Lpp(f)$ is rewritable, it is rewritten during the *Lpp*-reduction and it remains unchanged during the total reduction. Other rewriting schemes may be applied: the algorithm will remain correct, but in general the number of rewritings necessary to obtain a non-rewritable polynomial will increase.

When τ can be rewritten with more than one f_i, a *simplification strategy* is applied to select one of them. Several simplification strategies are possible, but experimentally the best one seems to be the trivial strategy (Giovini *et al.*, 1991): select the polynomial which was inserted earlier into the basis. This is because earlier polynomials tend to be less complicated than newer ones, which undergo many more reductions. The experiments confirm that this strategy leads to lower coefficient growth.

After simplification, if h is non-zero, it is added 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) and the *pair queue* is updated using the following function *UpdateQueue*(f, Q, R):

(1) delete from the queue Q each pair $\langle f_i, f_j \rangle$ such that both $Lpp(f_i)$ and $Lpp(f_j)$ strictly divide $Lpp(f)$ [this is the B-criterion of Gebauer and Möller (1988)];
(2) consider the set P of all pairs $\langle f_i, f \rangle$, $f_i \in R$; discard from P all elements $\langle f_i, f \rangle$ such that $Lpp(f_i)$ and $Lpp(f)$ are coprime, as well as any other pair having leading power-product equal or multiple to the leading power-product of $\langle f_i, f \rangle$ (T-criterion);
(3) sort P 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);
(4) merge P with the *pair queue*, using again the *selection strategy*;
(5) return the list of pairs removed from Q.

The initialization phase builds the initial basis R and the initial *pair queue* Q by iteratively adding to R $f_i \downarrow_R$, and invoking *UpdateQueue*.

Note that to perform *UpdateQueue*, just the leading power-product of f needs to be known. Hence, in the perspective of parallelization, the *pair queue* could be adjusted as soon as the *Lpp*-reduction of f is completed, without waiting to perform the total reduction.

In summary, the algorithm is as follows:

```
$R := \emptyset$
$Q := \emptyset$
**for each** $f_i \in \{f_1, \ldots, f_m\}$
    $UpdateQueue(f_i \downarrow_R, Q, R)$
    $R := R \cup \{f_i \downarrow_R\}$
**while** $Q \neq \emptyset$
    $\langle f, g\rangle := First(Q)$
    $Q := Q \backslash \{\langle f, g\rangle\}$
    $h := LppReduce(Spoly(f, g), R)$
    $h := h \downarrow_R$
    **if** $h \neq 0$ **then**
        $UpdateQueue(h, Q, R)$
        $R := R \cup \{h\}$

**procedure** $LppReduce(f, R)$
  **for each** $f_i \in R$
    **let** $\tau =$ the highest power-product of $f$ s.t. $f \xrightarrow{\tau}_{f_i} f'$ **then**
      $f := f'$
**return** $f$
```

There are two degrees of indeterminacy in the Buchberger algorithm: the *selection strategy* for pairs and the *simplification strategy* for reducers. We assume that both strategies are induced by an ordering, i.e. we sort the set of pairs (or of reducers) and choose the first one.

It is well known that a correct choice of the strategies is critical for the performance of the algorithm. A wrong choice, even a single apparently harmless modification, may lead to an incredible growth of coefficient values and of number of pairs to process. See Giovini *et al.* (1991) for a description of a good strategy and a striking example.

The Buchberger algorithm offers opportunity for parallelization because of the exponential number of reducers produced by the algorithm. Coarse-grain parallelism can be exploited to perform as many reductions as possible in parallel.

A simple-minded parallelization is the following: assign different pairs to different processors, and add to the basis new elements as soon as they are produced by any of the processors. In this way however, reduced polynomials are produced in an order which deviates from any proper strategy and all benefits of concurrency are lost.

On the other hand, a strategy accurate implementation where a processor must wait for all previous reductions to be completed, may waste a lot of time since the computing time of different pairs is uneven. Apparently, up to now most parallel versions of the algorithm have fallen into these two categories.

A notable exception is the solution in Sawada *et al.* (1994), where several nodes proceed independently performing reductions: each node works on a disjoint subset of the simplificands. A manager node selects the minimum among the reduced polynomials computed by the nodes, adds it to the basis and sends it to all nodes for use in further reductions. Such algorithm has been implemented on a distributed memory parallel machine and the results reported are consistent with the analysis on the potential speed-up that we present in Section 4.

Our parallel algorithm tries to avoid both difficulties; the strategy is followed strictly, and a processor is only idle when all currently feasible reductions have been completed: since complex problems generate hundreds of simplificands which outnumber the processors.

When the input is homogeneous the influence of the strategy is considerably lower, provided that the selection strategy proceeds in increasing degree. An algorithm for this case is presented in Section 7.

3. Parallelizing the Main Loop

We describe a parallel Buchberger algorithm exploiting coarse grain parallelism on multiple independent processors. One of the processors (the *process manager*) performs a special task while the others (the *simplifiers*) all perform the same task.

We concentrate on the part of the algorithm that computes a redundant, non-reduced Gröbner basis.

The communications are always between one processor and the process manager, in a star-like topology. Each processor has a local copy of the current basis: experience has shown that the size of the basis is small with respect to the total amount of data produced during a computation.

We describe separately the process manager and the simplifiers. In the descriptions we will use the following operations which are typical in the Buchberger algorithm:

FindReducer(m, R)**:** find a reducer that can reduce monomial m (i.e. whose leading monomial divides m). Returns *NIL* if none exists.
Reduce(m, p, r)**:** return p' such that $p \overset{m}{\rightarrow}_r p'$.

3.1. THE SIMPLIFIERS

Each simplifier maintains two lists of polynomials and related status variables:

R: the list of reducers;
S: the list of simplificands;
cs: the current simplificand from S;
cm: the current monomial within cs.

Each reducer can be identified by its position in the list, since it will never change; each simplificand, which initially is the S-polynomial of a critical pair $\langle r_i, r_j \rangle$, can be identified by the pair $\langle i, j \rangle$.

We assume that communication occurs over a reliable stream protocol like the one provided by sockets over TCP, without discussing the specific format of messages. More sophisticated libraries like MPI (Message Passing Interface Forum, 1994) might be used to achieve portability to a wide class of distributed memory architectures.

A simplifier executes a loop, listening at an input channel to requests from the manager. According to the request received an action is performed, while if no request is received one reduction step is performed.

The list of possible requests and their parameters is the following, with the corresponding actions:

INIT *init-data*
: Set R and S to $\{\}$, *cs* to *NIL*; the parameter *init-data* conveys information about the current ring (number of variables, coefficients, ordering, ...).

POLY *polynomial*
: Append *polynomial* to R.

PAIR *pair*
: Let p be *Spoly*(*pair*) and insert it in S, in the proper position according to the ordering for pairs specified by the *simplification strategy.*

START
: Set *cs* to *First*(S) and *cm* to *Lpp*(*cs*). Perform one reduction step.

FINISH *pair*
: Suspend simplification, let p be the simplificand corresponding to *pair* in S if already present, otherwise let p be *Spoly*(*pair*). Compute the polynomial $h = p\downarrow_R$. Send message **DONE** *pair h.* Set *cs* to *NIL.*

DISCARD *pair*
: Delete the polynomial corresponding to *pair* from S. In case the polynomial being discarded is *cs*, set *cs* to *NIL.*

SUSPEND
: Suspend simplification, set *cs* to NIL and send back message **INPUT** to the manager, signaling to be ready to receive next polynomial.

END
: Terminate the activity of the simplifier.

If no message is received and *cs* is not *NIL*, then one reduction step is performed, which consists in the following:

```
(1) $r := FindReducer(cm, R)$
(2) If $r \neq NIL$ then
    (2.1) $cs := Reduce(cm, cs, r)$
    (2.2) If $cs = 0$ then send a **ZERO** *pair* message, where *pair* is the pair which originated cs, and delete *cs* from $S$
(3) *advance*
(4) If $cs = NIL$, send an **IDLE** command.
```

The *advance* step can be performed in two ways, one of which is completely strategy-accurate, but does not fully exploit parallelism, the other is partially strategy-accurate, and can be used only in some cases. We will refer to the first variant as *totally accurate*, to the second as *partly inaccurate.*

In the first case *advance* sets *cs* to the next simplificand and *cm* to *Lpp*(*cs*), in the second case it sets *cm* to the next monomial of *cs*, and if no such monomial exists, sets *cs* to the next simplificand and *cm* to $Lpp(cs)$. In both cases the next simplificand of the last simplificand is *NIL*.

Note that when *cs* is *NIL*, after sending an **IDLE** message, the simplifier remains idle waiting for requests by the manager from the network.

It is possible that an **IDLE** message is sent while a request from the manager is already underway. The protocol has been designed to take into account such asynchrony of communications, as discussed later.

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 performs the usual initializations of Buchberger algorithm, sends an **INIT** command to each simplifier, then for each polynomial *pol* in the basis sends a message **POLY** *pol* to each simplifier. Then it partitions the set of pairs in as many subsets as the simplifiers and distributes each subset to a different simplifier by means of messages **PAIR** *pair*. It them sends a **FINISH** request with the first pair to the simplifier to which it is assigned, and a **START** to each of the remaining simplifiers. Then the manager waits for messages form the simplifiers.

For each critical pair in Q, the manager maintains the address of the simplifier to which it has been assigned, if any. It also maintains information on whether each simplifier is idle or not.

The manager reacts to messages as follows:

DONE *pair pol*
: Remove *pair* from Q; transmit **POLY** *pol* to all the simplifiers. Update the queue with $removed := UpdateQueue(pol, Q, R)$. For each pair $p \in removed$ already assigned to a simplifier, transmit **DISCARD** p to that simplifier. Add *pol* to the basis R. If Q is not empty and q_1 is the first pair, then transmit request **FINISH** q_1 to the simplifier to which the q_1 assigned, or to the first idle simplifier in case the first pair is yet unassigned. Partition the set of unassigned pairs in Q among the remaining simplifiers and distribute each subset to them by transmitting **PAIR** messages.

IDLE
: Send to the simplifier any pending messages.

ZERO *pair*
: Remove *pair* from Q.

When Q becomes empty, the manager sends an **END** request to each simplifier and the algorithm terminates, with the current basis (present in all the nodes) as a redundant Gröbner basis.

Notice that we use two different primitives for communication: *send* and *transmit. Send* is an immediate form of communication which can be used when it is ensured by the protocol that the receiver is listening. *Transmit* is a buffered primitive which involves an exchange to synchronize with the receiver. In fact the receiver might be busy performing a complex reduction and may not read immediately the message: if the sender tries to send a long message (e.g. a long polynomial), the operation on the socket would block and the manager would wait.

To overcome these problems, we implement buffered transmission through the operation *transmit*(*message*, *dest*), which involves an exchange of messages. The *message* is recorded in a queue for *dest*, and if queue was empty the request **SUSPEND** is sent to *dest*. When the message **INPUT** is received back from *dest*, then all messages present in the queue for *dest* are sent to *dest*.

Note 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, but a simplifier must accept requests for discarding a non-existing pair.

4. Simulation

We have organized a simulation of the parallel algorithm from traces of a sequential implementation.

We assume that: 1. an unlimited number of processors is available, so each pair can be assigned to a different processor; 2. reduction steps take the same amount of time, called a tick; 3. communication takes no time.

These assumptions are not met in practice, but they allow us to compute a theoretical maximum speed-up for the algorithm.

We compute, for every processor and every tick, whether the processor is idle, or is performing an *Lpp* reduction, or a total reduction. If the reduction involves a basis element that is not yet available, the processor is considered idle[†]. A computed basis element is made available in one extra tick to the other processors, and only after all the previous elements have been made available.

We consider:

1. the parallel run time, i.e. the number of ticks necessary for all the processor to stop (the *depth*);
2. the sequential run time (in which each processor starts only when the former ones have finished), i.e. the number of reduction steps necessary (the *length*);
3. the maximum number of processors simultaneously active (the *width*).

The quotient of length and depth is the *theoretical speed-up*, and the width is an estimate of the number of processors necessary to achieve this speed-up.

Figure 1 provides a graphical display of the state of each processor throughout the execution of one typical example. Table 3 reports depth, width and speed-up for this and other standard benchmarks [from Boege *et al.* (1986) and Faugère *et al.* (1993)].

In the figures, a dotted line (−−−) is an initial idle state, a thick line (▬) is an *Lpp* reduction, a medium line (▬) is a total reduction, a thin line (——) is an intermediate idle state.

Figure 1 shows that three pairs are generated initially and assigned to three distinct processors. After the first processor completes the total reduction, the 4th element of the basis is produced, which allows the second processor to proceed and generate the 5th element, and so on.

[†] In the algorithm presented earlier a processor should be considered idle also if a total reduction is needed and a preceding pair is still unfinished; but the algorithm can be modified allowing earlier total reduction. The implementation of this variant is however more difficult, since it may be necessary to back up a (strategically) wrong reduction to use a previously unavailable element.

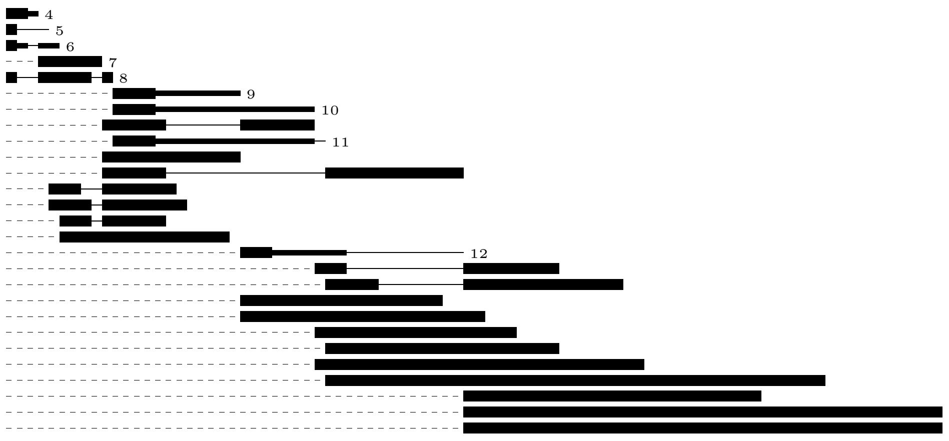

Figure 1. Pavelle4.

Table 3. Simulation summary.

Benchmark	*Depth*	*Width*	*Speed-up*	*Benchmark*	*Depth*	*Width*	*Speed-up*
`cyclic4`	18	4	2.11	`ceva`	81	17	7.42
`cyclic5`	197	29	7.84	`valla`	676	159	41.64
`cyclic6`	818	91	16.10	`fateman`	344	14	4.02
`cyclic7` (-last)	9399	136	11.08	`pavelle4`	88	10	5.50
`katsura4`	119	15	5.03	`caprasse`	131	25	6.67
`katsura5`	405	32	7.49	`gerdt`	401	61	13.68
`liu`	80	11	5.49	`cassou`	147	31	9.53
`butcher`	463	50	9.70				

5. Heuristic Considerations

In this section we explain the experimental findings that support our choices in the design of the parallel Buchberger algorithm. These findings come from experimentation with the sequential Buchberger algorithm implemented in the AlP*I* program (Traverso and Donati, 1989).

USELESS COMPUTATIONS

When adding a polynomial, some pair may get deleted from the queue. Experimentally, this does not happen often, and the pairs deleted are often not the first in queue. Hence useless computations should be infrequent.

IDLE PROCESSORS

This may happen only when a simplifier has completed all its partial reductions and no new pairs are available.

Experimentally, for the long-running computations in which we are interested, the queue has often hundreds of elements, enough to keep all the simplifiers continuously busy. The queue is short only briefly in the very first and very last phase.

In an earlier implementation of the algorithm, we had arranged that the manager would send pairs to simplifiers on demand, whenever they had completed all their previously assigned reductions. This turned out to slow down the computation, since the latency in communication over the network meant that simplifiers would be often waiting for receiving new pairs. For instance the benchmark `valla` would take 77.83 seconds instead of 24.75 seconds with the current implementation on 29 simplifiers.

Balance between manager and simplifiers

Experimenting with the algorithm we have compared the reduction time and the pair managing time. We noticed a significant difference between ideals generated by binomials (difference of monic monomials) and other ideals: for the former kind the pair managing time often exceeds one half of the total computing time, while for other ideals the pair managing time is usually less than 2% of the total computation time. Hence for binomial ideals one should consider parallelizing the process manager task as well.

Keeping low the influence of managing pairs in the total computing time also 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.

6. Implementation

The algorithm has been implemented in ECoLisp, an *embeddable* Common Lisp, which can be linked with C based applications (Attardi, 1995), in our case with the socket library for communication. The core of the implementation derives from AlP*I* (Traverso and Donati, 1989).

Table 4 shows the results of the experiments with up to 29 simplifier nodes. For each benchmark we present the computation time (Comp.), the time for transmission of polynomials (Comm.) and the difference between the two (Diff.). We also report the total number of messages (# msg) and the number of messages exchanging polynomials (P. msg).

The relative timings should be considered, since the Lisp implementation is inherently slower than a C++ implementation (Attardi and traverso, 1995) by a factor of 2 to 6 in the sequential version of these benchmarks.

It is apparent that the cost of transmission of polynomials over the network becomes dominant as the number of processors increases. Nonetheless if we factor out this cost we can notice a speed-up which for some benchmarks (`katsura5`, `ceva`, `cyclic6`) approaches the predicted theoretical limit.

7. A Degree-accurate Algorithm for Homogeneous Ideals

When an ideal is homogeneous, it is known that the strategies have much less influence, provided that the pairs are chosen in increasing degree. The reason is that higher degree elements in the Gröbner basis cannot reduce lower degree ones, and that the degree of a polynomial does not change during the reduction. Hence when all the critical pairs of degree lower than n have been processed, we have all the elements of the Gröbner basis of

Table 4. Execution times.

Benchmark		2 simp.	4 simp.	8 simp.	16 simp.	29 simp.
`katsura5`	Comp.	9.45	6.95	5.35	5.98	8.85
	Comm.	1.31	1.88	2.66	4.45	7.11
	Diff.	8.14	5.07	2.69	1.53	1.14
	# msg	225	332	485	739	1097
	P. msg	21	63	147	315	588
`ceva`	Comp.	90.65	47.21	38.20	38.95	59.58
	Comm.	4.36	7.08	10.97	29.02	52.81
	Diff.	86.29	40.13	27.23	9.93	6.77
	# msg	408	580	823	1172	1724
	P. msg	27	81	189	405	756
`valla`	Comp.	39.60	26.06	20.30	25.98	24.75
	Comm.	4.95	6.09	6.05	10.6	13.20
	Diff.	34.65	19.97	14.25	15.38	11.50
	# msg	1847	2360	3182	4376	5988
	P. msg	78	234	546	1170	2028
`cyclic6`	Comp.	81.66	79.66	31.88	41.16	50.33
	Comm.	6.60	9.55	15.06	30.93	40.36
	Diff.	75.06	70.11	16.82	10.23	9.97
	# msg	1177	1687	2449	3731	5761
	P. msg	93	279	651	1365	2418

degree lower than n, and if the elements obtained are interreduced, they are the elements of the reduced Gröbner basis of degree up to n.

This allows to describe a completely different form of Buchberger algorithm, that is easily parallelizable.

We divide the algorithm in blocks, one per degree, and each block is divided in three parts. We name these three parts $\mathcal{A}_n$, $\mathcal{B}_n$, $\mathcal{C}_n$. In a sequential implementation, we proceed degree-wise, and for each degree n we perform first $\mathcal{A}_n$, then $\mathcal{B}_n$, then C_n. In a parallel implementation, some form of parallelism between blocks is possible, and the blocks themselves are highly parallelizable.

The Gröbner basis G under construction, is divided degree-wise; G_n is the set of elements of degree n of G, $G_{<n}$ is the set of elements of degree $< n$. The set P of critical pairs too is divided degree-wise, and P_n is the set of pairs of degree n (the degree of a pair is the degree of the lcm of the leading monomials), and $P_{>n}$ is the set of pairs of degree $> n$.

We allow in P also *singleton pairs*, i.e. polynomials; the S-polynomial of a singleton pair is the polynomial itself. This gives more uniformity to the algorithm.

Now we describe $\mathcal{A}_n$, $\mathcal{B}_n$, $\mathcal{C}_n$ as partly independent processes; each one is intrinsically parallelizable.

- $\mathcal{A}_n$ takes the elements of P_n, computes their S-polynomial, and reduces them with the elements of $\mathcal{B}_{<n}$. The result is sent to $\mathcal{B}_n$.
- $\mathcal{B}_n$ interreduces the elements obtained from $\mathcal{A}_n$, producing elements of G_n.
- $\mathcal{C}_n$ gets the input from $\mathcal{B}_n$ and outputs elements of $P_{>n}$.

In particular,

- $\mathcal{A}_n$ requires $G_{<n}$ and P_n; it can start as soon as there is an element of P_n, it can output as soon as $\mathcal{B}_{n-1}$ is complete, and stops as soon as $\mathcal{C}_{n-1}$ is complete and P_n is empty.
- $\mathcal{B}_n$ gets the input from $\mathcal{A}_n$, and produces G_n as output; it can output only when $\mathcal{A}_n$ is complete.
- can be completed only after the completion of $\mathcal{B}_n$.

If the Hilbert function of the result is known, or at least an upper bound, we can use the Hilbert-driven algorithm described in Traverso (1996), and there is a fourth series of blocks $\mathcal{H}_n$, that has input from $G_{<n}$ and $\mathcal{B}_n$, and can send an interrupt to $\mathcal{A}_n$. Moreover $\mathcal{A}_n$ and $\mathcal{B}_n$ are slightly modified. To avoid describing two different versions of the algorithm, we describe directly a unique version, containing $\mathcal{H}_n$.

The Hilbert-driven algorithm computes, from the Hilbert function upper bound given as input and $G_{<n}$, the maximum expected number of elements of G_n; when this number is attained, one can discard the other elements of P_n, and all the computations in progress in the current degree can be aborted. If instead for some degree this number is not attained, for higher degree a guess is impossible, and one reverts to the usual algorithm.

The algorithm starts with $B = \emptyset$, and P equal to the input generators as singleton pairs. We describe the algorithm as sequence of $\mathcal{A}_n, \mathcal{H}_n, \mathcal{B}_n, \mathcal{C}_n$. The algorithm stops when at the end of $\mathcal{C}_n$ we have that $P_{>n}$ is empty. We discuss first the definition and parallelization of each step, we will discuss later the possibilities of concurrency for the various steps.

$\mathcal{A}_n$ takes P_n, computes the S-polynomial of each pair, and reduces it with $B_{<n}$. The procedure is easily parallelized since each reduction is independent from the others. The output of each reduction is sent to $\mathcal{H}_n$.

$\mathcal{H}_n$, if active, knows the number h of expected elements of G_n. We do not describe the algorithm, which uses $G_{<n}$ and initial data, see Traverso (1996) for a complete description. $\mathcal{H}_n$ contains a probabilistic checker of linear independence: a test to determine whether a polynomial is linearly independent from previous ones, when whose answer is “YES” then the polynomial is linearly independent, when it is “NO” the polynomial is probably dependent (there are plenty of such algorithms, and of parallel ones too). $\mathcal{H}_n$ sends the polynomial to $\mathcal{B}_n$ if it is independent, and after sending h polynomials, it interrupts $\mathcal{A}_n$, discarding further output from $\mathcal{A}_n$.

If, at the end of A_n, $\mathcal{H}_n$ has not detected h linearly independent polynomials, then the Hilbert-driven algorithm is declared inactive, and the remaining (probably linearly dependent) polynomials have to be discarded with a deterministic algorithm (or accepted if the probabilistic algorithm has been unlucky).

If $\mathcal{H}_n$ is inactive, then it simply forwards the input to $\mathcal{B}_n$.

$\mathcal{B}_n$ gets a set of polynomials (linearly independent if $\mathcal{H}_n$ is active), and reduces them to Gauss–Jordan reduced echelon form. This is linear algebra, for which parallelization is well studied.

$\mathcal{C}_n$ computes, for every element g of G_n, the useful critical pairs of g with the preceding basis elements. This can be done in parallel for the elements of G_n, and moreover there are further parallelization possibilities for each g. But this would require a description of the implementation of the criteria, which is outside the scope of the present paper. Remark that it is not necessary to sort the pairs in a prescribed strategy order, but only to divide them in different degrees; and all pairs of which a g of degree n is an element are of higher degree, (since g is not reducible by the previous elements), hence are in $P_{>n}$.

The correctness of the algorithm is easy to prove, being just a variant of Buchberger algorithm.

From the description of $\mathcal{A}_n, \mathcal{H}_n, \mathcal{B}_n, \mathcal{C}_n$ it is clear that they need not wait for the previous step to be completed to begin their work:

- $\mathcal{A}_n$ can start as soon as P_n is non-empty; an output however can be delivered only after G_n becomes available; hence it is probably too complicated to take care of this possibility. Hence it is better to wait for $\mathcal{B}_{n-1}$, but it is not necessary to wait for $\mathcal{C}_{n-1}$. The end of $\mathcal{C}_{n-1}$ is necessary for the end of $\mathcal{A}_n$.
- $\mathcal{H}_n$ can start as soon as $\mathcal{A}_n$ gives an output, and it is mandatory not to wait that $\mathcal{A}_n$ is ended (otherwise the interrupt to $\mathcal{A}_n$ is useless). In a sequential implementation it is reasonable to mix a step of $\mathcal{A}_n$ and a step of $\mathcal{H}_n$.
- $\mathcal{B}_n$ can begin to work as soon as an element of input arrives. The output can be sent immediately to C_n, that works on the leading power products only, even before the tail reduction is completed. If this is done, it is however necessary that $\mathcal{B}_n$ operates destructively on its data, so that a polynomial sent to $\mathcal{C}_n$ can continue to be modified by $\mathcal{B}_n$, and the critical pairs refer automatically to the polynomials updated by $\mathcal{B}_n$.

This discussion shows that the global parallelization of the algorithm will be fruitful if several critical pairs are present in each degree. This is often the case if the ordering is a DegRevLex, and is much less likely for a Lex ordering.

If the ideal is zero-dimensional, this is a further reason to resort to the FGLM algorithm of Faugère *et al.* (1993), that is linear algebra hence highly parallelizable; in higher dimension the algorithm sketched in Faugère (1994) should be analysed for possible parallelization.

8. Conclusions

The Buchberger algorithm seems to lend itself quite naturally to parallelization. Parallelism may arise at fine grain in polynomial arithmetic, or at medium grain in the reduction of polynomials, working on different monomials in parallel. Most attempts at parallelization so far have chosen, however, the coarse-grain parallelism of performing independent reductions in parallel.

We claim that to achieve the most benefit from coarse-grain parallelism the algorithm must exploit good strategies for reductions as those developed for the sequential version of the algorithm. While this may limit the overall amount of parallelism, it avoids the combinatorial growth of reductions which offsets the benefits of parallelism.

To substantiate this claim we designed parallel versions of the Buchberger algorithm which adhere to a reduction strategy. We reported on our experiments with an implementation of one of these algorithms.

A second question which we tried to answer is about the speed-up achievable from coarse-grain parallelism. Our analysis of simulated traces provides estimates on the speed-up and the number of processor necessary to achieve it for various classical benchmarks. The speed-up is significant in some cases but not unlimited.

This confirms and explains our findings and those by other authors, notably Sawada *et al.* (1994), that the speed-up from coarse-grain parallelism in the Buchberger algorithm tops with a relatively small number of processors (16 to 32).

So while strategy accurate parallel algorithms perform better than straightforward aggressive parallel search algorithms, further possibilities for improving the performance of the Buchberger algorithm might come from the exploitation of finer grains of parallelism.

Acknowledgements

The International Computer Science Institute in Berkeley provided access to its network of workstations to carry out our experiments. We are grateful to the anonymous referees for their well founded criticism.

References

Attardi, G. (1995). The embeddable common Lisp. *ACM Lisp Pointers*, **8**(1), 30–41.

Attardi, G., Traverso, C. (1995). The PoSSo Library for Polynomial System Solving. *Proc. AIHENP95*, Singapore, World Scientific Publishing Company.

Becker, T., Weispfenning, V. (1993). *Gröbner bases.* Berlin, Springer-Verlag, GTM 141.

Boege, W., Gebauer, R., Kredel, H. (1986). Some Examples for Solving Systems of Algebraic Equations by Calculating Grobner Bases. *J. Symbolic Computation*, **2**, 83–89.

Buchberger, B. (1965). An Algorithm for Finding a Basis for the Residue Class Ring of a Zero-Dimensional Polynomial Ideal. Ph.D. Thesis, Math. Inst., Univ. of Innsbruck, Austria.

Buchberger, B. (1985). Gröbner bases: an algorithmic method in polynomial ideal theory. *Recent Trends in Multidimensional Systems Theory*, Bose, N.K., (ed.), New York, D. Reidel Publ. Co., pp. 184–232.

Buchberger, B. (1987). The parallelization of critical pair completion procedures on the L-machine. *Proc. Japanese Symposium on functional programming*, 54–61.

Bündgen, R., Göbel, M., Küchlin, W. (1994). A fine-grained parallel completion procedure. *Proc. ISSAC'94*, Oxford.

Chakrabarti, S., Yelick, K. (1994). Distributed data structures and algorithms for Gröebner basis computation. *Lisp and Symbolic Computation*, **7**, 1–27.

Cox, D., Little, J., O'Shea, D. (1991). Ideals, varieties, and algorithms. *UTM*, Berlin, Springer-Verlag.

Faugère, J.C. (1994). Résolution des systèmes d'équations algébriques. Ph.D. thesis, Université Paris 6.

Faugère, J.C., Gianni, P., Lazard, D., Mora, T. (1993). Efficient computation of zero–dimensional Gröbner bases by change of ordering. *J. Symbolic Computation*, **16**(4), 329–344.

Gebauer, R., Möller, H.M. (1988). On an installation of Buchberger's algorithm. *J. Symbolic Computation*, **6**(2 and 3), 275–286.

Giovini, A., Mora, T., Niesi, G., Robbiano, L., Traverso, C. (1991). "One sugar cube, please" OR Selection strategies in Buchberger algorithm. *Proc. ISSAC'91*, ACM, July, 49–54.

Message Passing Interface Forum (1994). MPI: A Message-Passing Interface Standard. *International J. Supercomputing Applications*, **8**(3-4).

Mishra, B. (1993). *Algorithmic Algebra.* Text and monographs in Computer Science, Berlin, Springer-Verlag.

Pauer, F., Pfeifhofer, M. (1988). The theory of Gröbner bases. *L'Enseignement Math.*, **34**, 215–232.

Sawada, H., Terasaki, S., Aiba, A. (1994). Parallel computation of Gröbner Bases on distributed memory machines, *J. Symbolic Computation*, **18**, 3, 207–222.

Senechaud, P. (1989). Implementation of a parallel algorithm to compute a Gröbner basis on boolean polynomials. *Computer Algebra and parallelism*, New York, Academic Press, 159–166.

Traverso, C. (1996). Hilbert functions and Buchberger algorithm. Submitted.

Traverso, C., Donati, L. (1989). Experimenting the Gröbner basis algorithm with the AlP*I* system. *Proc. ISSAC'89*, ACM.

Vidal, J.P. (1990). The computation of Gröbner bases on a shared memory multiprocessor. *Design and implementation of symbolic computation systems*, *Lecture Notes in Computer Science* **429**, Berlin, Springer-Verlag, 81–90.