A NETWORK IMPLEMENTATION OF BUCHBERGER ALGORITHM

GIUSEPPE ATTARDI — CARLO TRAVERSO

Dipartimento di Informatica – Dipartimento di Matematica
Università di Pisa

INTRODUCTION

We report on a parallel implementation of the Buchberger algorithm on a network of workstations. The algorithm differs from other existing parallel implementations in two important respects: the algorithm will very rarely have idle processors; and the running of the algorithm is strictly sequential.

This last assertion may seem crazy for a parallel algorithm, but is very important 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 responsability of "pasting" the different parallel processes in a sequential algorithm, even discarding some results that appear useless because they do not comply with the strategy.

This is possible only since the experimentations on the "simplification strategy" (see below) seem to give support to the "older first, never delete" strategy. There are theoretical motivations that support this strategy (these motivations were suggested by D. Lazard), but up to now they are conjectural. The same strategy was chosen in many other implementations, since it is the straightforward one.

The implementation was made on a network of workstations connected in ETHERNET. We think that this is a good choice, even if this choice forces to duplicate a large amount of information on many workstations. This architecture is very frequent in many research centers, hence our implementation can be easily applied. Shared memory processors might be better, (Cfr. [Vi]) but they pose different problems, and are less frequently available.

The implementation is based on AKCL, extended with network primitives from DELPHI Common-LISP, [AD]. One processor has to manage the communications with the other processors, hence many input-output channels. The other processors communicate only with this one, and they check the input stream periodically, to determine whether the manager has instructions for them.

Research with the contribution of "Ministero dell'Università e della Ricerca Scientifica e Tecnologica" and Consiglio Nazionale delle Ricerche 30/1/1991, 1177.

The implementation derives from AlPI, the system for experimenting the Buchberger algorithm developped in Pisa, and will be in a near future integrated in it, ([TD], [ADT]).

A sketch of Buchberger algorithm

We give a short description of (our interpretation of) Buchberger algorithm, just to fix notations. Refer to any of the standard introductions (e. g. [Bu], [PP], [Ro]) for a more complete description.

Let k be a fied, and assume that we have a term-ordering in a polynomial ring $k[X] = k[x_1, \ldots, x_n]$; a term in $k[X]$ is a product of indeterminates, a monomial a product of a constant $\in k$ and a term. The leading term $Lt(f)$, 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 $Sp(f,g) = (Lc(g)/\gcd(Lt(f), Lt(g))f - (Lc(f)/\gcd(Lt(f), Lt(g))g$.

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 term of a pair* is the least common multiple of the leading terms of their elements, and is often used in the algorithm, so in practice it is pre-computed and stored with the pair.

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 term, discarding the elements whose leading term 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 Lt-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 terms such that τ_i is the leading term of f_i and $\tau_i\mu = \tau$, then we replace f with $a_i\rho - a\mu\rho_i$. The term τ usually is the highest rewritable term, hence as long as the leading term of f is rewritable, one has to rewrite it, (this is the Lt-reduction), and in the second phase the leading term 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, and there are hints that one can prove connections between this strategy and algorithms based on linear algebra with controllable complexity.

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 terms of (f_j, f) and of (f_j, f) both strictly divide the leading term of (f_i, f_j)

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 terms of f_i and of f are coprime, as well as every other pair having leading term equal or multiple to the leading term of (f_i, f).

d) sort S in a convenient order (*selection strategy)*, and discard every pair whose leading term is equal or multiple of a preceding leading term

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

The same actions are performed to intialize the procedure, taking f from the initial basis. Optionally, in this phase, before taking an element, one can simplify it using the preceding elements.

Remark that to perform the points b) to e) above, one only has to know the leading term of f, hence (already considering parallelization) one can adjust the pair queue as soon as the Lt-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 simplifiers) 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.

A "wrong" parallelization is the following: give to different processors different pairs, and add to the basis new elements as soon as they appear as a result of one of the processors. On the other side, if a processor has to wait for the previous pairs to be completed, it may happen that a lot of time is lost: the computing time of different pairs is uneven. Apparently, up to now all parallelizations have been of these kinds; our remark might explain some problems of other implementations.

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

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 *pair reducers*) all perform the same tasks.

We completely describe an algorithm that computes a redundant, non reduced Gröbner basis. At the end we give a short description of a parallel implementation of the final total reduction.

The communications (input and output) are always from one processor to the central manager. Every processor (excluding the pair manager, that has only the leading terms) 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 small with respect to the total amount of the computation. A sequential computation may take several hours, and the final output, including the formatting of the redundant basis, only a couple of minutes.)

The pair reducers. We describe the state of one of the pair reducers in the course of the algorithm, and the actions to perform.

Every pair reducer has a copy of the current basis, a *working list* of critical pairs (sorted with respect to the selection strategy), each with a partial simplification of the corresponding Sp, a flag (stating if the processor is the first that has to answer), and a current pair being reduced, an input channel and an output channel.

a) If the flag is TRUE, then the current pair is to the first pair in the list; in this case, the processor does not need to check the input channel, and proceeds to perform the simplification of the first pair. As soon as the Lt-simplification is completed, the processor sends the Lt to the output channel, and proceeds with the rest of the reduction; as soon as the reduction is completed, it sends the reduced polynomial to the output channel, resets the flag to FALSE, adds the polynomial to the basis, deletes the first pair (the one that has been completed), selects the first pair as current.

b) If the flag is FALSE, one has to check the input channel; one can receive four types of messages:

- b1) a new element of the basis: this is added to the basis, the first element of the working list becomes current.
- b2) a message to reset the flag: this is set to TRUE, the first element of the working list becomes current.
- b3) a new pair to process: it is inserted in the working list (in the correct selection strategy order) and, [if the new pair precedes the current one in this ordering], the new pair becomes current.
- b4) an interrupt message, stating that one of the pairs is useless: such pair is deleted, and if it was the current one, the next pair becomes current.

c) If the flag is FALSE and the input channel is empty, then try to perform one Lt-reduction[1] to the polynomial corresponding to the current pair. If no reduction

[1] This point has a variant, consisting in trying here to perform a reduction whatsoever: an Lt-reduction if possible, otherwise a reduction of the largest possible monomial. This point will be discussed later.

can be performed, then the next pair becomes current: if no next pair exists, send a message to the output channel, and wait until a message arrives in the input channel. If the polynomial reduces to 0, then send a message to the output channel, delete the pair, and move on to the next pair (as above, send a message if the next pair does not exist).

The process manager. Its tasks consist in keeping the queue of pairs, distributing the tasks to the pair reducers, and dispatching messages. The process manager maintains the pair queue, which consists of elements which are records of four fields: (i, j, τ_{ij}, ϕ), where i, j are the indices of the elements of the pair, τ_{ij} is the least common multiple of the leading terms of f_i and f_j (this is often used to sort the elements of the queue) and ϕ is either NIL or the address of the processor to which the pair is currently assigned for simplification.

The process manager is usually idle, waiting for messages from the pair reducers.

a) If a pair reducer asks for more pairs to simplify, it is sent the first unassigned pair, if one available exists; otherwise, the manager records the processor as idle.

b) If a pair reducer has reduced a pair to zero, delete the pair from the queue. If this was the first element of the queue, perform d) below.

c) If a pair reducer sends a new addition to the basis, then send the new element to all other pair reducers, erase the first element of the queue (this first element was necessarily the one that has generated the new element: the other pairs are not authorized to add an element to the basis). Then perform d).

d) Take the first element out of the queue, and if it is assigned to a processor, send to this processor a request to set the flag to TRUE. Otherwise, if nobody is reducing the first pair (this may happen, if all the processors are busy reducing other pairs given to them before the appearance of the pair that is first in queue) wait until a processor asks for a new pair: this will eventually happen.

e) if a pair reducer sends back the leading term of the forthcoming addition to the basis, then a series of actions has to be taken:

e1) check all existing elements of the queue, erase those that have become useless (sending an interrupt to all pair reducers that are currently reducing them)

e2) prepare all new pairs that have to be added to the queue, sorted in the right strategy order

e3) merge the new pairs in the existing pair queue.

Remark that point e2) might be long to perform, and may be given to another processor, maybe an idle pair reducer: such processor already has almost all the information needed. (In our first implementation this was not done, and the process manager does not listen to other messages until point e) is complete.) However, from our experience with the sequential version, the time spent in adjusting the pair queue is a small fraction of the total computing time in all the complicated examples, with the relevant exception of ideals generated by binomials, where the time spent in adjusting the queue may exceed one half of the total execution time. In these ideals our parallel implementation is not supposed to work reasonably; anyway, for these ideals, that are important in finding positive integer solutions to linear systems (see [O], [P], [C]) special implementations using shared memory processors or vectorial processors are being planned.

During the execution of e2) other processors may ask for more pairs: one can

give them elements from the old queue. It is improbable (at least for sufficiently complex examples) that the pair reducer that has given the leading term of the future addition to the basis will be ready to send the new element before the completion of the whole of e), so we do not consider this eventuality (if it happens, somebody will wait).

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 ([TD]).

The problems with any parallel implementation are: transmission overhead, useless duplication of computing, and idle processors, either for bad synchronization or for bottlenecks.

Transmissions. The basis has to be transmitted in full to every processor. One form of basis transmission occurs in AlP*I*: the output. At the end of the pair reduction, the redundant basis computed is formatted and printed. This may take a couple of minutes for a basis that was computed in a couple of hours. Moreover, the formatting for the screen in AlP*I* is complicated: a polynomial is transformed in a man-readable string, then the string is divided in pieces that, if possible, do not exceed the screen width, then the string pieces are output to the screen, that still has to manage scrolling. Communications of machine-readable polynomials are of course more rapid.

Moreover, in AlP*I* it is possible to store the current basis on disk, and every polynomial needed has to be re-read every time (with random disk access) and re-parsed. Even with this extreme communication problem the overhead seldom exceeds 20%, and can be often be compensated by reduced garbage collection.

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 the other polynomials, waiting for the centralmanager to complete the queue managing is usually no 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 adjusting and the task distribution, in such a way that this problem too can be solved.

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 usually no problem. There is anyway a possible bottleneck.

An intrinsic limit of the main variant. The possibility of parallelization of the algorithm proposed has an intrinsic limitation, due to the fact that at most one pair reducer is performing non-lt reductions at the same time. In most of the examples computed the non-lt reduction phase accounts for about 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 reducers, with their Lt-reduction completed with the current basis, waiting for the first total reduction to be complete

The variant suggested in the footnote above does not have this limitation, but has a drawback: the fragments of the algorithm performed by the pair simplifiers no longer coincide with the corresponding fragments of the sequential algorithm. We try to explain why the difference in the variant is small, and only amounts in duplicating some simplifications.

The simplification procedure steps consist in rewriting a monomial with a polynomial whose leading term is smaller than the monomial. The choice of the simplification strategy "older first, never delete" that we use makes sure that the same monomial at different points of the algorithm is always rewritten with the same polynomial.

Assume that we are in the sequential algorithm, and that at some moment we forget once the existence of the last simplifier; then instead of the Lt-reduction with the last simplifier, we cannot perform another Lt-reduction (otherwise the choice of the Lt-reduction would have been this last one), and we perform the rewriting of (say) the second monomial.

It may happen that the Lt-reduction would have erased or modified this same monomial, so the same Lt-reduction performed later re-introduces the same monomial, with another coefficient; hence the inner reduction performed is a waste of time: we have to perform again the same rewriting with another coefficient. These considerations may be generalized, and we see that at the end the result of the simplification is always the same, but the work is larger.

One can also remark that the phenomenon described above is only possible if either the forgot polynomial is not totally reduced, or the Lt of the polynomial is a proper multiple of the Lt of the reducer; the first possibility never happens, and the second one is experimentally not frequent, hence the variant might be a very good idea. To settle the issue one has however to experiment largely with many processors, and this will take time (and money) to be done.

We did not worry to compute statistics for the assertions in this section: they are just feelings justified by the practice. It is planned to compute statistics for transmission time, idle time and useless computation time for the parallel algorithm.

Experimental results

A first experimental implementation was made and tested, using AKCL (Austin-Kyoto Common LISP) modified incorporating some features of DELPHI COMMON-LISP on the workstations of the Department of Mathematics (we thank the col-

legues of the Department that had to stand our pair reducers creeping in at inconvenient times).

We have tested the first implementation on a network with 7 workstations (3 SUN SPARC-1, 8, 8 and 16 MB; 1 SUN 3-60, 16MB; 3 IBM PC-RT, 16MB). The configuration is sufficiently heterogeneous to be significant, but also to make it difficult to give a correct interpretation of the timings: we give them in raw form.

The examples tested are the same already considered in [TD] or [GT], with the same name (excepted the examples due to Bjørk, wrongly credited to Arnborg). The term-ordering is the deree-reverse lexicographic.

We summarize the results in the following table, that gives the execution times in seconds for the computation of each example sequentially on the diffenent type of workstations, in average (harmonic mean on the 7 execution times), and in parallel on 2 (one manager and one reducer, both SPARC-1), 3 (SPARC-1) and 7 workstations. It was impossible to perform the experiments in ideal conditions (no other processes active on the workstations), so the results are not really significant, but anyway they are encouraging.

We did not arrive to finish our tests in time, also due to system maintenance on some of the machines at the last moment. The table, or an equivalent one, will be filled in next week-end. Up to now, the algorithm has run on up to 4 workstation, but with great unevenness of timings, also due to the varying load. We hope to be more convincing soon.

	SPARC1	SUN-3	PC-RT	Average	2	3	7
Katsura4	13″15	25″73			16″53		
Katsura5	2′55″	7′13″			1′57″		
Bjørk5	15″43	1′03″			20″		
Bjørk6							
Cohn 2	4′55″	21′22″					
Cohn 3							
Valla							
Butcher	42″31	3′03″			59″	25″	

We are planning further experiments with more workstations, including experiments with workstations in different locations connected in INTERNET. These experiments, as well as more details on the implementation, will be reported in [ADT]

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 partly done 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 in a situation of this type: we are using a large set of workstations for which this task is a task to be performed in spare time (for example, in UNIX with a high "nice" setting). 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: they risk that the central manager becomes the bottleneck.

Another modification might be the following: a pair reducer does not have automatically the full basis, but only the leading terms; as soon as an element of the basis is needed for a simplification, the element is asked to the process manager, and stored. Less used elements might be discarded, if the space becomes tight.

This does not seem to be a true improvement, since usually almost any new pair will need almost every element to be simplified. Moreover, the communication problem increases. This could however be an useful idea for parallel computers with a large shared memory and a small cache memory for every processor.

Binomial ideals.

As remarked above, binomial ideals, i. e. ideals generated by term differences have special properties. In particular, every polynomial appearing in the Buchberger algorithm is difference of terms. This allows to use special data structures: for example, we may represent polynomials as pairs of vectors, instead of lists of pairs (vector . coefficient), that is the usual representation for polynomials in Gröbner basis computations. The fact that polynomials use a fixed amount of memory makes it easy to use shared memory processors (the space reserved for the reduction of a single polynomial is fixed and small). There is no risk of growth for coefficients or for the length of polynomials, so even the strategies have to be reconsidered. Operations on polynomials are easily vectorized: the key operations are term product, divisibility test, quotient, term-ordering comparison, and they correspond to simple vector operations.

The reduced need of garbage collections, and the fact that no long integers are needed (unless exponents grow too much; this too can happen!) show that LISP is maybe not necessary any more: C and FORTRAN might be considered instead.

Using the special properties of binomial ideals should improve considerably the performance of the algorithm; one might expect that in that case most of the time would be spent in the pair queue management, hence different parallelizing ideas should be found. Moreover, vectorial operations could be used in this management, and vectorization seems to be easier and more useful than parallelization.

A preliminary analysis of a vectorial implementation of Buchberger algorithm for binomial ideals has been made, and a preliminary implementation might follow

in a short time; the relevance of the problems that can be solved with Gröbner basis computation of binomial ideals (e. g. diophantine equations and integer programming, [Po], [Co]), makes this issue very interesting for applications.

References

[AD] Attardi, G., Diomedi, S., *Multithread Common Lisp*, ESPRIT MADS TR87/1, DELPHI, 1985.

[ADT] Attardi, G., Donati, L., Traverso, C., A|*PI User Manual and Implementation Report*, (in preparation), Dip. Mat. Pisa. Available by anonymous FTP on `gauss.dm.unipi.it`.

[B1] Buchberger, B., *An Algorithm for Finding a Basis for the Residue Class Ring of a Zero-Dimensional Polynomial Ideal*, Aequationes Mathematicae **4** (1970), 374–383, (Ph.D. Thesis, Math. Inst. Univ. Innsbruck, 1965)..

[B2] Buchberger, B., *Gröbner bases: an algorithmic method in polynomial ideal theory*, Recent trends in multidimensional systems theory, N. K. Bose, ed., D. Reidel Publ. Comp., 1985, pp. 184–232.

[B3] Buchberger, B., *The parallelization of Critical pair completion procedures on the L-machine*, Proc. of the Japanese Symposium on functional programming, 1987, pp. 54–61.

[Co] Conti, P., *Basi di Gröbner e sistemi lineari diofantei*, (preprint), 1990.

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

[Po] Pottier, L., *Minimal solutions of linear diophantine systems : bounds and algorithms*, 1990 (submitted).

[Ro] Robbiano, L., *Gröbner bases: a foundation for commutative algebra*, Computer and Mathematics, Springer Verlag, 1989.

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

[TD] Traverso, C., Donati, L., *Experimenting the Gröbner basis algorithm with the* A|*PI system*, ISSAC 89, A. C. M., 1989.

[Vi] Vidal, J. P., *The computation of Gröbner bases on a shared memory multiprocessor*, Design and implementation of symbolic computation systems, Lecture Notes in Computer Science **429** , Springer Verlag, Berlin–Heidelberg–New York, 1990, pp. 81–90.

Giuseppe Attardi
Dipartimento di Informatica
Corso Italia
I-56100 PISA

E-mail: attardi@dipisa.di.unipi.it

Carlo Traverso
Dipartimento di Matematica
Via F. Buonarroti 2
I-56100 PISA

E-mail: traverso@dm.unipi.it