TR-677

A Buchberger Algorithm for Distributed Memory Multi-Processors

by
David Hawley

August, 1991

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03)3456-3191 ~ 5
Telex ICOT J32964

Institute for New Generation Computer Technology

A Buchberger Algorithm for Distributed Memory Multi-Processors

David J. Hawley
Institute for New Generation Computer Technology
1-4-28 Mita, Minato-ku, Tokyo 108 Japan.
hawley%icot.jp@relay.cs.net

Abstract

Gröbner Bases are a mathematical tool that has received considerable attention in recent years. Since the Buchberger Algorithm for computing these objects is expensive in both space and time, several attempts at parallelization have been made, with good results for shared-memory multi-processors, but not for distributed memory machines. We present an algorithm that delivers substantial speedups on distributed memory multi-processors, and an incremental version of the algorithm which is suitable for use as the solver in a constraint logic language.

1 Introduction

Recently, there have been several attempts made to parallelize the Buchberger algorithm, with generally disappointing results[6, 7], except for shared-memory machines[10, 3]. Parallelization has been tackled at two levels: a coarse grain parallel rewriting of the S-polynomials and/or testing for subsumption and critical pairs, and a fine-grain rewriting of single S-polynomials. The feasibility of the latter seems restricted to shared-memory architectures. An interesting concurrent logic programming (data-flow) approach implemented on Transputers was reported by Siegl [8], with good speedups on the small examples shown, but absolute performance was only fair.

We are interested in using Gröbner Bases as the core of constraint solvers for concurrent constraint programming languages.[1] In this application, the input set of polynomials is not given at the start of the computation, but is generated concurrently by some other process, possibly depending on the intermediate sets of basis polynomials. In this paper, we give distributed algorithms suitable for two abstractions of this application: the *static* case in which the complete set of input polynomials is available at the start of the Gröbner Base calculation, and the *dynamic* case, in which the input polynomials are sent at arbitrary intervals from some processor(s).

We begin by reviewing some terminology. The standard definitions are augmented by some useful definitions from the term-rewriting systems literature. Let there be a certain ordering among monomials and let a system of polynomial equations be given. An equation can be considered a rewrite rule which rewrites the greatest monomial in the equation to the polynomial consisting of the remaining monomials. For example, if the ordering is lexicographic, a polynomial equation, $Z - X + B = A$, can be considered as a rewrite rule, $Z \rightarrow X - B + A$. Two rewrite rules $L_1 \rightarrow R_1$ and $L_2 \rightarrow R_2$, of which L_1 and L_2 are not mutually prime, are termed a *critical pair*, since the least common multiple of their left-hand sides can be rewritten in two different ways. The S-polynomial of such a pair is

$$\text{S-poly}(L_1, L_2) = R_1 \frac{\text{lcm}(L_1, L_2)}{L_2} - R_2 \frac{\text{lcm}(L_1, L_2)}{L_1}$$

[1] One such language, GDCC[4], is under development by the constraint logic programming group at ICOT.

A rule $L_1 \rightarrow R_1$ is said to subsume rule $L_2 \rightarrow R_2$ if L_2 is a multiple of L_1. Finally, Gröbner Bases can be characterized by the property that for all pairs (P,Q) in the given set of equations, S-poly(P,Q) rewrites to zero.

There are two main sources of polynomial-level parallelism in the Buchberger Algorithm, the parallel reduction of a set of polynomials, and the parallel checking for subsumption and critical pairs of a new rule against the other rules. Since the latter is inexpensive, we must concentrate on parallelizing the coarse-grained reduction component for shared-memory architectures. However, since the convergence rate of the Buchberger Algorithm is very sensitive to the order in which polynomials are converted into rules, an implementation must be careful to select "small" polynomials early for inclusion in the developing basis. The key idea underlying the algorithms in this paper is that of sorting a distributed set of polynomials, and we will use the "asynchronous enumeration sort" [1, pp. 178–181] as our point of departure.

2 The Algorithm

We begin by considering the "asynchronous enumeration sort" algorithm, which is suitable for distributed memory machines. In the sort algorithm, each processor has a complete set of the input items, and a copy of the *ownership* function which is a many-to-one function from items to processors. Each processor independently compares the items it owns to all the other items in order to determine the items' ranks in the sorted sequence. The method for outputting the items in sorted sequence chosen, because of its applicability to the Buchberger algorithm, is that each processor listens to the output of all the other processors, and outputs its equations when the count reaches the respective ranks.

The sorting algorithm is adapted as follows. Each processor contains a complete set of basis polynomials (called *rules*) and non-basis polynomials, and a load-distribution function which logically partitions the polynomials by specifying which processor "owns" what polynomials. The position in the output (rule) sequence of each polynomial is calculated by its owning processor based on an associated key (for example, the leading power product) which is identical in every processor, and does not change during reduction. Each polynomial is output when it becomes the smallest one remaining. The critical-pairs and subsumptions are calculated independently by each processor, so that the processors' sets of polynomials stay synchronized. As a background task, each processor rewrites the polynomials it owns, starting with those lowest in the sorted order. Termination of the algorithm is detected independently by each engine, when the input equation stream is closed, and there are no non-basis polynomials remaining.

The dynamic problem requires more complex control, in order to prevent the arrival of input polynomials at different times at each processor from causing processors to have inconsistent views about the set of non-basis polynomials and possibly about the output (rule) sequence. Figure 1 shows the algorithm for the dynamic case. This version requires additional information about the basis and non-basis sets of each engine to be made known, eventually, to every other engine.

A serious drawback to the algorithm is that it cannot take advantage of "magic polynomials". That is, since the key which determines the output position of a polynomial is fixed before reduction begins, the key is only a rough approximation of the actual preferability of a polynomial after reduction. A possible refinement is to resort the set of polynomials within each processor inside the same "output slots" owned by that processor.

Since the result for the static algorithm is straightforward, and a special case of the result for the dynamic algorithm, we will only prove correctness for the dynamic version. We would like to show that the processors have the same view of the output (rule) sequence.

Lemma 2.1 *For every $t \geq 0$, exactly one processor outputs to Channel[t].*
Proof by induction on t. Assume $t = 0$. Since B_i is updated exactly when t is incremented, we have $B_i = \mathcal{B}_i = 0$ and $P_i = K_i$. We call a processor i synchronized *if $\mathcal{S}_i = K_i$; only synchronized processors can output (line 12). By definition, $\mathcal{S}_i \subseteq \bigcap_j K_j$, and so for all synchronized processors*

```
comment
    S = stream of polynomials.
    S_i = subset of S that engine i knows has been received by every engine.
    B_i = subset of B_i that engine i knows has been received by every engine.
    Code to maintain S_i and B_i is omitted.

(1)   do i=1, N
(2)       spawn engine(i,S,Channel) on processor i

(3)   engine(I,S,Channel)
(4)       S_i := P_i := K_i := ∅
(5)       [B_i := B_i] := ∅; t := 0
(6)       do forever
(7)           choose
(8)               guard     receive X from S
(9)                         B_i = B_i
(10)              do        K_i := K_i ∪ {X}, P_i := P_i ∪ {X}
(11)              guard     (p := min(P_i)) is irreducible w.r.t. B_i
(12)                        ω(p) = i, [S_i = K_i]
(13)              do        output p to Channel[t++]
(14)                        P_i := P_i ∪ {spoly(p,q) | q ∈ B_i} − {p}
(15)                        B_i := B_i ∪ {p}
(16)              guard     receive p from Channel[t++]
(17)              do        P_i := P_i ∪ {spoly(p,q) | q ∈ B_i} − {p}
(18)                        B_i := B_i ∪ {p}
(19)              guard     (L := {q | q ∈ P_i, ω(q) = i, p is reducible by B_i}) ≠ ∅
(20)              do        Rewrite L by B_i
(21)              guard     P_i = ∅, [S is closed]
(22)              do        output B_i to Channel[t]
(23)                        stop
(24)          endchoose
(25)      enddo
```

[a]The **choose** (**guard** Cond **do** Action)* **endchoose** construct specifies a non-deterministic guarded choice. Execution will suspend until at least one of the conditions obtains, and then the action corresponding to one of the guards whose condition obtains will be executed; the testing of guard conditions has no observable effect until an associated action is chosen.

[b]The algorithm for the static problem is obtained by changing all references to the stream S to the set of input polynomials P, replacing line(4) with "$P_i := P$", and deleting the framed code.

Figure 1: Algorithm for Dynamic Problem

$K_i = K = \bigcap_j K_j$. Therefore there is a unique minimum $p \in P_i$. Let $m = \omega(p)$. If processor m is synchronized, then it outputs p as soon as p has been fully rewritten, otherwise it waits until synchronization (which will eventually occur, if S is finite). In either case, t is incremented. After output, K_m will not change until $\mathcal{B}_m = B_m$ (line 9), which also freezes the value of K. We are then guaranteed that no (other) engine can output until receiving p, and incrementing t.
Assume $t = t_1 > 0$. Now $P_i - K_i$ are the identical sets of critical pairs from the first t_1 rules. We argue similarly to the base case to obtain the required result.

Corollary 2.1 *Each processor receives the same sequence of rules.*

Theorem 2.1 *For all $p, q \in S$, S-poly(p,q) rewrites to zero.*
The proof follows easily from the above corollary.

3 Implementation and Results

The dynamic algorithm was implemented on the Multi-PSI, a distributed-memory multi-processor designed as a development platform for operating systems and applications based on concurrent logic programming concepts. The user-level language, KL1[9, 5], is a data-flow language that executes at 128 K reductions/second on a single Multi-PSI node.

The central data structure in the implementation is a sorted list of items of work, comprising input polynomials, critical pairs, and requests to simplify rules. Priorities correspond to the key associated with each polynomial. In the current implementation for rules and input polynomials we use the largest power product as the key, and for S-polynomials we use the largest power product after canceling the largest power product of each of the two parent polynomials. The complete execution of one piece of work is broken down into stages; for example, a critical pair is first converted to a S-polynomial, rewritten, and finally normalized. Based on this breakdown, we pipeline the execution of the entire list, giving us maximum overlap between communication and local computation. Although this implementation only deletes critical pairs arising from subsumed rules, a full implementation of Buchberger's criteria for filtering useless critical pairs should also be possible.

The implementation of the $\mathcal{S}$ and $\mathcal{B}$ variables in the dynamic algorithm is based on *ACK* (acknowledgment) messages. However, the additional latency introduced applies only to the acceptance of new input polynomials, and the number of $\mathcal{B}$ related ACK messages can be decreased by updating the $\mathcal{B}$ variables less frequently. Information about processor load is piggybacked onto the ACK messages, in order to construct the ω load-distribution function dynamically (being careful to build it identically on each processor).

Finally, the calculation of the coefficients of non-basis polynomials is improved by delaying until a rule to rewrite the associated power product has been found. At that point, the coefficient expression is evaluated using divide-and-conquer, and compared to zero. This strategy results in several fold speed improvements in some examples.

Example	1 PE	2 PE	4 PE	6 PE	8 PE	12 PE	16 PE
Runge-Kutta 1	2.335	1.841	1.493	1.704	1.334	1.514	1.751
Katsura 3	8.854	5.008	3.748	3.359	3.369	3.297	3.463
Little Trinks	49.710	23.592	22.096	14.897	12.351	14.469	15.682
Big Trinks	188.641	94.428	84.328	67.630	46.512	45.070	45.106
Katsura 4	2000.056	1520.289	725.004	377.047	301.324	243.900	209.529

Table 1: Absolute Performance of Dynamic Algorithm (sec)

The benchmarks presented here are from Boege et. al.[2], with Katsura 4 modified to use total degree reverse lexicographic ordering, as do all the others. Except for Katsura 4, the speedup curve (Figure 2) eventually becomes flat, reflecting the limits of polynomial-level parallelism in these examples. The absolute performance of the algorithm is only fair. However, reimplementing the polynomial and rational arithmetic in a standard von Neumann language should bring about a 1–2 order of magnitude performance improvement in the bulk of the computation (measured at over 90%), without affecting the parallelism. Although reimplementation would change the ratio between computation time and communication time/ latency, we conjecture a significant improvement in overall performance.

4 Conclusion

This contribution of this work is the parallelization of the Buchberger Algorithm on a distributed memory machine exhibiting substantial speedups and reasonable performance. Reimplementation of the low-level routines in a von Neumann language should substantially improve the latter. The algorithm uses broadcast messages exclusively, and it would be interesting to investigate its performance on a hardware and software platform that supports broadcasting efficiently.

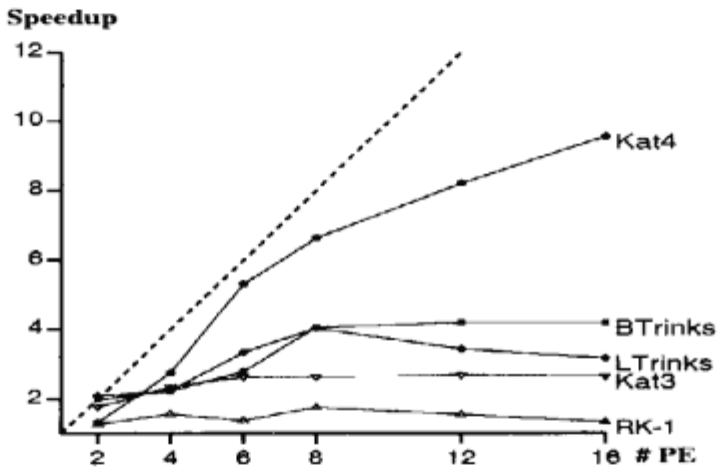

Figure 2: Speedup of Dynamic Algorithm

References

[1] S. G. Akl. *Parallel Sorting Algorithms*. Notes and Reports in Computer Science and Applied Mathematics. Academic Press, 1985.

[2] W. Boege, R. Gebauer, and H. Kredel. Some examples for solving systems of algebraic equations by calculating groebner bases. *J. Symbolic Computation*, 2(1):83–98, 1986.

[3] E. M. Clarke, D. E. Long, S. Michaylov, S. A. Schwab, J. P. Vidal, and S. Kimura. Parallel Symbolic Computation Algorithms. Technical Report CMU-CS-90-182, Computer Science Department, Carnegie Mellon University, Oct. 1990.

[4] D. Hawley and A. Aiba. Guarded Definite Clauses with Constraints - A Preliminary Report. Japan-Italy-Sweden Workshop on Logic Programming and Parallel Processing, Stockholm, Sweden, Aug. 1990.

[5] K. Nakajima, Y. Inamura, N. Ichiyoshi, K. Rokusawa, and T. Chikayama. Distributed implementation of KL1 on the Multi-PSI/V2. In *Proceedings of ICLP'89*, pages 436–451, 1989.

[6] C. G. Ponder. Evaluation of 'Performance Enhancements' in algebraic manipulation systems. In J. D. Dora and J. Fitch, editors, *Computer Algebra and Parallelism*, pages 51–74. Academic Press, 1990.

[7] P. Senechaud. Implementation of a parallel algorithm to compute a Gröbner basis on Boolean polynomials. In J. D. Dora and J. Fitch, editors, *Computer Algebra and Parallelism*, pages 159–166. Academic Press, 1990.

[8] K. Siegl. Gröbner Bases Computation in STRAND: A Case Study for Concurrent Symbolic Computation in Logic Programming Languages. Master's thesis, CAMP-LINZ, Nov. 1990.

[9] K. Ueda and T. Chikayama. Design of the kernel language for the parallel inference machine. *Computer Journal*, December 1990. To appear.

[10] J. P. Vidal. The Computation of Gröbner bases on a shared memory multi-processor. Technical Report CMU-CS-90-163, Computer Science Department, Carnegie Mellon University, Aug. 1990.