ICOT Technical Report: TR-713

TR-713

The Concurrent Constraint Language GDCC and Its Parallel Constraint Solver

by
D. Hawley

Nobember, 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

The Concurrent Constraint Language GDCC and Its Parallel Constraint Solver

David Hawley
Fourth Laboratory,
Institute for New Generation Computer Technology (ICOT)
4-28, Mita 1-chome, Minato-ku, Tokyo 108, Japan

Abstract

We describe the current state of development of the concurrent constraint language GDCC (*Guarded Definite Clauses with Constraints*), is a member of the cc (*Concurrent Constraint*) family of languages which supports multiple solvers and recursive queries in a committed-choice framework. GDCC models computation as the interaction of multiple cooperating agents through the exchange of information via querying and asserting the information into a central repository. Concretely, this paradigm is embedded in a guarded (conditional) reduction system, where the guards contain the queries and assertions. Control is achieved by requiring that the queries in a guard are true (entailed), and that the assertions are consistent (satisfiable), with respect to the current state of the repository.

GDCC is an experimental cc language, which supports a user-specified set of sorts and constraint symbols in a committed-choice framework, and is intended to be a research tool for investigating issues of constraint-solving in concurrent programming languages, such as problem decomposition, use of multiple solvers and hybrid techniques, ordering of constraints, management of semi-decidable solution methods, debugging techniques, etc. We introduce the language, its implementation in KL1[NaI89a], and a parallel constraint solver for rational polynomials based on a parallel implementation of the Buchberger Algorithm. The Buchberger Algorithm is a basic technology for symbolic algebra, and several attempts at its parallelization have appeared in the recent literature, with some good results for shared memory machines. The algorithm we present is designed for the distributed-memory Multi-PSI, but nevertheless shows consistently good performance and speedups for a number of standard benchmarks from the literature.

1 Introduction

Constraints, that is formulas describing conditions on objects in some domain, is an interesting and important programming paradigm that has a voluminous literature. In the last five years, the integration of constraint programming with Prolog has received a considerable amount of attention from both the viewpoint of applications and theory[LaM87, DiH88, Wal89, BeP89] based on the theoretical foundation of Jaffar and Lassez[JaL87]. As for extending this work from the sequential to the concurrent frame, there is little published work, among which is a report of some preliminary experiments in integrating constraints into the PEPSys parallel logic system[Hen89]), and a proposal, the *Concurrent Constraint* programming languages, for integrating constraint programming with concurrent logic programming languages[Sar89]. The cc programming language paradigm models computation as the interaction of multiple cooperating agents through the exchange of information via querying and asserting the information into a central repository. Concretely, this paradigm is embedded in a guarded (conditional) reduction system, where the guards contain the queries and assertions. Control is achieved by requiring that the queries in a guard are true (entailed),

and that the assertions are consistent (satisfiable), with respect to the current state of the repository.

This paper introduces Guarded Definite Clauses with Constraints (GDCC), an experimental cc language, which supports a user-specified set of sorts and constraint symbols in a committed-choice framework, and is intended to be a research tool for investigating issues of constraint-solving in concurrent programming languages, such as problem decomposition, use of multiple solvers and hybrid techniques, ordering of constraints, management of semi-decidable solution methods, debugging techniques, etc. It is implemented in KL1 and is currently running on the Multi-Psi parallel logic machine at ICOT.

The paper is organized as follows. We first informally review some of the vocabulary of the constraint logic and concurrent constraint schemes, and then we introduce the GDCC language. We then introduce the rational polynomial constraint system, discuss its suitability in the cc framework, and then briefly present a concurrent constraint solver for the system.

2 Constraints in Logic Programming

Constraint Logic Programming

Constraint logic programming (CLP), proposed by Jaffar and Lassez [JaL86], is an extension of logic programming in which unification is replaced by the solving of equations over some theory, and which can be then further generalized to allow non-equational relations as constraints on variable values. Similarly to Prolog, a CLP program comprises predicates and clauses, where clauses have the (abstract) syntax

Head :- Constraints, Goals.

and is executed depth-first, left-to-right, with constraints taking the place of unification. Operationally, (head) unification serves a dual purpose in logic programming, first to bind actual (goal) and formal (clause) parameters, and secondly to prune the search. Constraints take both these roles. Two major suitability requirements were given for a constraint solver to be used in a constraint logic language:

satisfaction-complete - since pruning of the proof tree is based on the constraints, the constraint solver must be able to determine the satisfiability or unsatisfiability of any constraint with respect to the accumulated constraint set.

incrementality - since constraints are added dynamically during execution, the constraint solver must be able to add new constraints to the accumulated constraint set efficiently, and

Jaffar and Lassez also observe that although the mode of invocation does not affect the correctness of constraint evaluation, it does have a large effect on the efficiency of evaluation. For example, in order of increasing cost, the query `?- X=2, Y=1, X+Y=3` can be treated as a test, and `?- X+Y=3, X=1` can be handled by constraint propagation, while `?- X+Y=3, X-Y=1` must be handled as a system of simultaneous equations, by some method such as gaussian elimination. Likewise, the global ordering of constraint evaluation has a large effect on efficiency.

We would like to extend the constraint logic paradigm to a concurrent paradigm, both to take advantage of potential parallelism in the logic component and constraint solver component, and to address problems of controlling the parallelism with particular reference to the order of constraint evaluation. The next section reviews a framework for starting to deal with these issues.

Concurrent Constraint Programming Languages

Concurrent Constraint programming languages [Sar89], are a generalization to concurrency of the CLP languages. We will present a brief summary of the basic concepts of cc. The logical interpretation of CLP programs is replaced by the notion of cooperating agents, which

communicate via queries and assertions into a (consistent) global database of constraints called the *store*. Saraswat remarks that the CLP scheme is too weak, since it lacks control features suitable for concurrent languages. Accordingly, in cc constraints occurring in program text are classified by whether they are querying or asserting information, into Ask and Tell constraints respectively. [1] Asking and Telling are required to be stable operations.

The following definitions are adapted freely from [SaA89, Sar89, Mah87]. We define the following sets: S is a finite set of *sorts*, including the distinguished sort HERBRAND, F a set of *function symbols*, C a set of *constraint symbols*, P a set of *predicate symbols*, and V a set of *variables*. A sort is assigned to each variable and function symbol. A finite sequence of sorts, called a *signature*, is assigned to each function, predicate and constraint symbol. We write $v : s$ if variable v has sort s, $f : s_1 s_2 \ldots s_n \rightarrow s$ if functor f has signature $s_1 s_2 \ldots s_n$ and sort s, and $p : s_1 s_2 \ldots s_n$ if predicate or constraint symbols p has signature $s_1 s_2 \ldots s_n$. We require that terms are well-sorted, according to the standard inductive definitions. An *atomic constraint* is a well-sorted term of the form $c(t_1, t_2, \ldots, t_n)$ where c is a constraint symbol, and a *constraint* is a set of atomic constraints. Let Σ be the many-sorted vocabulary $F \cup C \cup P$. A *constraint system* is a tuple $(\Sigma, \triangle, V, C)$, where $\triangle$ is a class of Σ structures.

We define the following meta-variables: c ranges over constraints, g,h range over atoms, q ranges over clauses, and p ranges over predicates.

We now define the four relations *answers*, *accepts*, *rejects*, and *suspends*. The constraint c *answers* c_l if

$$\triangle \models (\forall x_g)(c \Rightarrow \exists x_l . c_l)$$

c *accepts* c_l if

$$\triangle \models (\exists)(c \wedge c_l)$$

and c *rejects* c_l if

$$\triangle \models (\forall x_g)(c \Rightarrow \neg(\exists x_l . c_l))$$

where x_g are the variables in c, and x_l are the variables in c_l but not in c. Note that the property *answers* is strictly stronger than *accepts*, and that *accepts* and *rejects* are complementary. We say that c *suspends* c_l, if c accepts, but does not answer c_l.

A cc language program is comprised in the usual way of clauses. A clause is defined as a tuple (head,guard,tell,body), where "head" is a term with unique variables as arguments, "guard" is a tuple $< c_a, c_t >$, c_a, c_t and "tell" are a constraints, and "body" is a set of terms. The constraint c_a is said to be *ask-moded*, while c_t and "tell" are said to be *tell-moded*. We abuse notation somewhat in the following definitions. Constraint s *confirms* guard $< c_a, c_t >$ if

$$s \textit{ answers } c_a \wedge s \textit{ accepts } c_t$$

constraint s *suspends* $< c_a, c_t >$ if

$$s \textit{ accepts } c_a \wedge s \textit{ accepts } c_t$$

and s *rejects* q if

$$s \textit{ rejects } c_a \vee s \textit{ rejects } c_t$$

Informally, a clause $(h, <\text{a,t}>, c, b)$ is a candidate for goal g in the presence of store s if s. $g{=}h$ *confirms* $<a,t>$ and s. $g{=}h$ *accepts* a. A goal g *commits* to candidate clause $(h, <\text{a,t}>, c, b)$, by updating the store s with $t \cup c$, and replacing g by b. A goal fails if $\forall q.q = (h, a, t, b) \wedge s \wedge g = h \Rightarrow s$ *rejects* $<\text{a,t}>$. Deciding *confirms* for multiple clauses and commitment for multiple goals can be done in parallel.

The next section introduces the GDCC language, which is an instance of the cc scheme.

[1] In a language such as Flat Guarded Horn Clauses[Ued86] querying and Ask-constraints correspond to guard unification and guards, while asserting and Tell-constraints correspond roughly to output unification and body unifications respectively.

3 GDCC

GDCC is an experimental cc language, designed and implemented in the stream/process model natural to committed-choice programming. It supports multiple plug-in constraint solvers with a standard stream-based interface, so that users can add new domains and solvers, and delays the binding of domain to solver to the start of each query, so that users can easily specify variant solvers. We feel that the added flexibility is appropriate at this stage. It should be noted that a solver for a given domain may only support a subset of the constraint symbols and modes. The GDCC language includes most of KL1 as a subset; KL1 builtin predicates and unification can be regarded as a distinguished domain HERBRAND (after Saraswat).

The GDCC system (figure 1) comprises

1. Pre-processor
   Translates a GDCC source program into KL1 code.
2. Shell
   Performs binding of domain to constraint solver, initiates queries and provides rudimentary debugging facilities. The debugging facilities comprise the standard KL1 trace/spy and PARAGRAPH execution-profiling functions, together with solver-level event logging. The shell also provides limited support for incremental querying, in the form of inter-query variable and constraint persistence.
3. Standard Constraint Solvers
   Interacts with a GDCC program to check guards and satisfy body constraints. At present, there exists a constraint solver for rational polynomials.

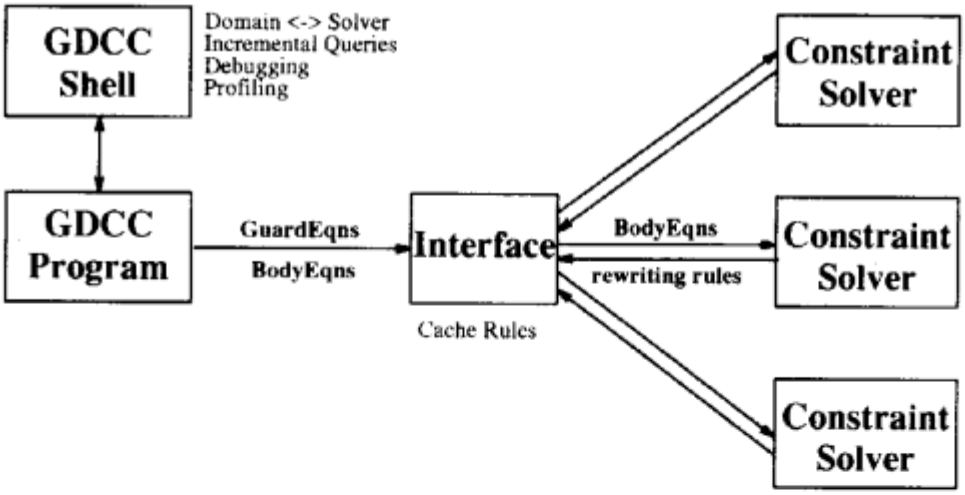

Figure 1: System Construction

Evaluation of GDCC programs

GDCC programs are compiled predicate-by-predicate into KL1, and during execution communicate via stream merge-trees to constraint solvers implemented as processes that encapsulate the entailment and unsatisfiability decision procedures. Abstractly, a constraint solver is an object that provides the following services:

1. Add a (body) constraint to the current store, and fail the whole computation in case of inconsistency.

2. Determining the candidacy of each of a set of clauses with respect to a given goal.
3. Report the canonical form of the final store, as projected over the variables occurring in the query.

This definition focuses on the input-output behaviour, as relating to the mission of constraint solvers: to provide control information to programs, and to report answers to queries. Specifying an interface at this level has the potential of allowing the constraint solver to make mission-oriented decisions on scheduling, speculative versus conservative guard evaluation, etc.

However, in the current implementation we solve constraints one-by-one, following a simple strategy for guard evaluation: For each goal, we evaluate the HERBRAND component of the guard before solving the remaining constraints; we do this in parallel for each clause. This approach is incomplete in the same sense that KL1 is incomplete, in that it is not guaranteed to detect goal failure, since only one of the possible serializations of each guard evaluation is considered.

Constraint solving typically requires manipulating variables in various ways. One basic form of manipulation is the comparison of two (uninstantiated) variables. KL1 provides no mechanism for doing so, and in fact the language's philosophy prohibits such comparisons. Accordingly, we represent non-HERBRAND variables as ground terms. Currently, the programmer is burdened with creating these "variables" via the builtin predicates *alloc/1,2*.

Example

The following example is adapted from [AiS88]. Given an arbitrary quadrilateral, we wish to find the ellipse that passes through the midpoints of the four sides, as illustrated in figure 2. The approach is to calculate the midpoints of the four sides first, and then to calculate the ellipse as a linear transformation of a circle. The transformation matrix is calculated based on the relationship between a unit square an the parallelogram formed by the midpoints. A fragment of a GDCC program (see Appendix A) to solve this problem is:

```
start(P1,P2,P3,P4,X,Y) :- true |
  create_points([A,B,C,D]),
  mid_points([P1,P2,P3,P4,P1],[A,B,C,D]),
  calc_ellipse(A,B,C,D,X,Y).

calc_ellipse(p(X1,Y1),p(X2,Y2),p(X3,Y3),p(X4,Y4),P) :-
        alg:(X1-X2)*(Y3-Y4)=(Y1-Y2)*(X3-X4),
        alg:(X1-X4)*(Y2-Y3)=(Y1-Y4)*(X2-X3) |
  calc_ellipse_okay(P1,P2,P3,P4,P).
```

The entry-point is `start/6` which takes the coordinates of the quadrilateral's vertices, and the two variables on which the ellipse constraint will be imposed. For example, the query `ellipse:start(p(X1,1),p(1,5),p(X2,8),p(-9,9),p(X,Y))` will return constraints on `X` and `Y` such that the point (X,Y) must lie in the ellipse drawn through the midpoints of a quadrilateral whose vertices are (X1,1), (1,5), (X2,8) and (-9,9). The program proceeds by spawning a process to calculate the midpoints of the four sides of the quadrilateral, and a process to calculate the ellipse. The guard in `calc_ellipse/6` *suspends* until, by checking that each of the two pairs of opposite sides have the same slope, it verifies that the midpoints form a parallelogram. Correctness of the method used by `calc_ellipse/6` is ensured by the guard. Additionally, the constraint-solver has a more constrained problem to deal with since the guard guarantees that the midpoint calculations have finished before new constraints are generated from `calc_ellipse/6`. This is typically much more efficient, which shows the value of *control based on information-flow* for concurrent constraint languages.

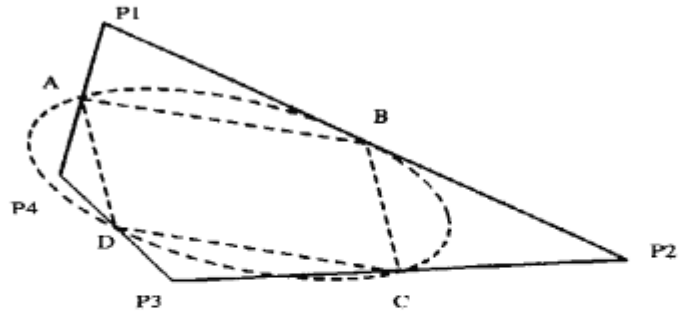

Figure 2: Example: Ellipse through midpoints

4 Solving Rational Polynomial Constraints

In [SaA89, SaS88], Buchberger Algorithm/Gröbner Base constraint solvers for the rational polynomial, boolean and finite-cofinite set domains were shown to fit reasonably well with the Constraint Logic Programming scheme, since they are incremental, and satisfaction-complete. We are interested here to investigate the use of Gröbner Base techniques within the concurrent constraint programming language scheme for the rational polynomial domain, which can be formalized as the constraint system $(\Sigma = F \cup C \cup P, \triangle, V, C)$, where:

$$
\begin{aligned}
S &= \{\mathbf{A}\} \\
F &= \{\times : \mathbf{AA} \rightarrow \mathbf{A}, + : \mathbf{AA} \rightarrow \mathbf{A}\} \\
&\quad \cup \{\text{fraction} :\rightarrow \mathbf{A}\} \\
C &= \{=\} \\
P &= \{\text{string starting with a lowercase letter}\} \\
V &= \{\text{string starting with an uppercase letter}\}
\end{aligned}
$$

with the structure

$$
\begin{aligned}
D(\mathbf{A}) &= \text{the set of all algebraic numbers} \\
D(\times) &= \text{multiplication} \\
D(+) &= \text{addition} \\
D(\text{fraction}) &= \text{the rational number it denotes}
\end{aligned}
$$

and

$$\triangle = \text{axioms of complex numbers}$$

We will start this section with a summary of some well-known results.

In [Buc83], Buchberger introduced the notion of Gröbner Bases and devised an algorithm to compute the Gröbner Base of a given finite set of polynomials. This algorithm has been widely used in the field of computer algebra over the past few years.

Without loss of generality, we can assume that all polynomial equations are in the form of $p = 0$. Let $E = \{p_1 = 0, \ldots, p_n = 0\}$ be a system of polynomial equations, and I the ideal in the ring of all the polynomials generated by $\{p_1, \ldots, p_n\}$. The following close relation between the elements of I and the solutions of E is well known as the Hilbert zero point theorem [Hil90].

Theorem 4.1 *Let p be a polynomial. Every solution of E is also a solution of $p = 0$, if and only if there exists a natural number n such that p^n is an element of I.*

Corollary 4.1 *E has no solution if and only if $1 \in I$.*

Buchberger gave an algorithm to determine whether a polynomial belongs to the ideal. A rough sketch of the algorithm is as follows (see [Buc83] for a precise definition).

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, $Z \rightarrow X - B + A$. 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. A pir of rewrite rules $L_1 \rightarrow R_1$ and $L_2 \rightarrow R_2$, of which L_1 and L_2 are not mutually prime, is 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

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

If further rewriting does not succeed in rewriting the S-polynomial of a critical pair to zero, the pair is said to be *divergent* and the S-polynomial is added to the system of equations. By repeating this procedure, we can eventually obtain a confluent rewriting system. The confluent rewriting system thus obtained is called a *Gröbner Base* of the original system of equations, and can be characterized by all S-polynomials rewriting to zero. The following theorem establishes the relationship between ideals and Gröbner Bases.

Theorem 4.2 *Let R be a Gröbner Base of a system of equations $\{p_1 = 0, \ldots, p_n = 0\}$, and let I be an ideal generated by $\{p_1, \ldots, p_n\}$. A polynomial, p, belongs to I if and only if p is rewritten to 0 by R.*

We could decide entailment based on 4.2, and satisfiability by using the Buchberger Algorithm to incorporate the polynomial to the Gröbner Base as per 4.1, but for our purposes there are some problems with this approach. Firstly, tentatively modifying the Gröbner Base in order to check guard satisfiability is undesirable, particularly if we wish to do so for multiple clauses simultaneously. Secondly, since the relation between the solutions and the ideal described in theorem 4.1 is incomplete, the method of theorem 4.2 is incomplete with respect to deciding entailment. For example, since Gröbner Base of $\{X^2 = 0\}$ is $\{X^2 \rightarrow 0\}$, rewriting using this Gröbner Base cannot show that $X = 0$ is entailed. There are several approaches to solve the entailment problem:

1. Use the Gröbner Base of the radical of the generated ideal, I, i.e. $\{p | p^n \in I\}$. Although it is theoretically possible to compute, there is no efficient implementation.
2. Use the Buchberger Algorithm to add p to the Gröbner Base and then check that the resulting Gröbner Base is equivalent to the original. Unfortunately, this seems as difficult as finding the Gröbner Base of the radical.
3. Use the Buchberger Algorithm to add $p\alpha$ to the Gröbner Base, where α is a new variable. p is in the old ideal iff $1 \in$ the new ideal. This has the unfortunate side-effect of changing the Gröbner Base.
4. Find n such that p^n is rewritten to 0 by the Gröbner Base of the generated ideal. Since n is bounded[CaG88], this is a complete decision procedure. Since the bound is very large, we may prefer the incremental solution of repeatedly raising p to a small positive integer power and rewriting it by the Gröbner Base.

Since we have not run into problems with incompleteness in practice, we have chosen not to implement any of the above strategies.

4.1 Parallel Constraint Solver

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 "distributed enumeration sort" [Akl85] as our point of departure.

We begin by considering the "distributed 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 one-to-one function from items to processors[2]. Each processor independently compares the item it owns to all the other items in order to determine the item's rank 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 equation when the count reaches the item's rank.

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 ω 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 3 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.

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 4.1 *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 $K_i = K = \bigcap_j K_j$. 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 4.2 *Each processor receives the same sequence of rules..*

[2]This idea is easily generalized to a many to one ownership function.

```
     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 3: Algorithm for Dynamic Problem

Theorem 4.3 *For all $p, q \in S$, S-poly(p,q) rewrites to zero.*
The proof follows easily from the above corollary.

4.2 Implementation and Results

The dynamic algorithm was implemented on the Multi-PSI, a distributed-memory multiprocessor designed as a development platform for operating systems and applications based on concurrent logic programming concepts. The user-level language, KL1[UeC90, Nal89b], is a data-flow language that executes at up to 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. A refinement of the algorithm to mitigate the effect of not updating the keys during polynomial reduction is to reorder the subset of polynomials owned by each processor within the "output slots" controlled by that processor, maintaining each subset in increasing order.

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, then rewritten until the leading power-product is irreducible, thirdly fully rewritten, and finally the coefficients are minimized by dividing by their greatest common divisor. Based on this breakdown, we pipeline the execution of the entire list, giving us maximum overlap between communication and local computation. The strategy for executing the pipeline is to give priority to the most advanced element for the engine next to output if its output stage is empty, and otherwise to give priority to the least advanced item; this combination of eager evaluation for the next engine to output a rule and lazy evaluation for all the other engines seems to give the best timings. A nice consequence of have the global set of equations replicated on each processor is that we can use the criteria for detecting useless critical-pairs as described for the sequential implementation of Gebauer and Moller [GeM88]. The criteria must be executed on each processor redundantly.

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).

Table 1: Absolute Performance of Dynamic Algorithm (sec)

Example	1 PE	2 PE	4 PE	6 PE	8 PE	16 PE	SAC
Runge-Kutta 1	1.999	1.702	1.598	1.562	1.689	2.078	.482
LTrinks	14.040	7.351	7.725	8.216	7.944	10.629	.888
BTrinks	52.043	29.691	23.145	27.629	24.645	25.299	138.425
Katsura 3	4.739	2.903	2.330	2.210	2.289	2.594	4.666
Katsura 4	273.666	144.475	101.897	69.708	114.802	92.790	†31.077

† Result is for different ordering that gives better results for this problem.

The benchmarks presented here are from the SAC system as reported in Boege et. al.[BoG86]; with the exception of Katsura 4, all examples use total degree reverse lexicographic ordering. The figures for the SAC system/IBM 3080 are given to show qualitative differences with a standard implementation.

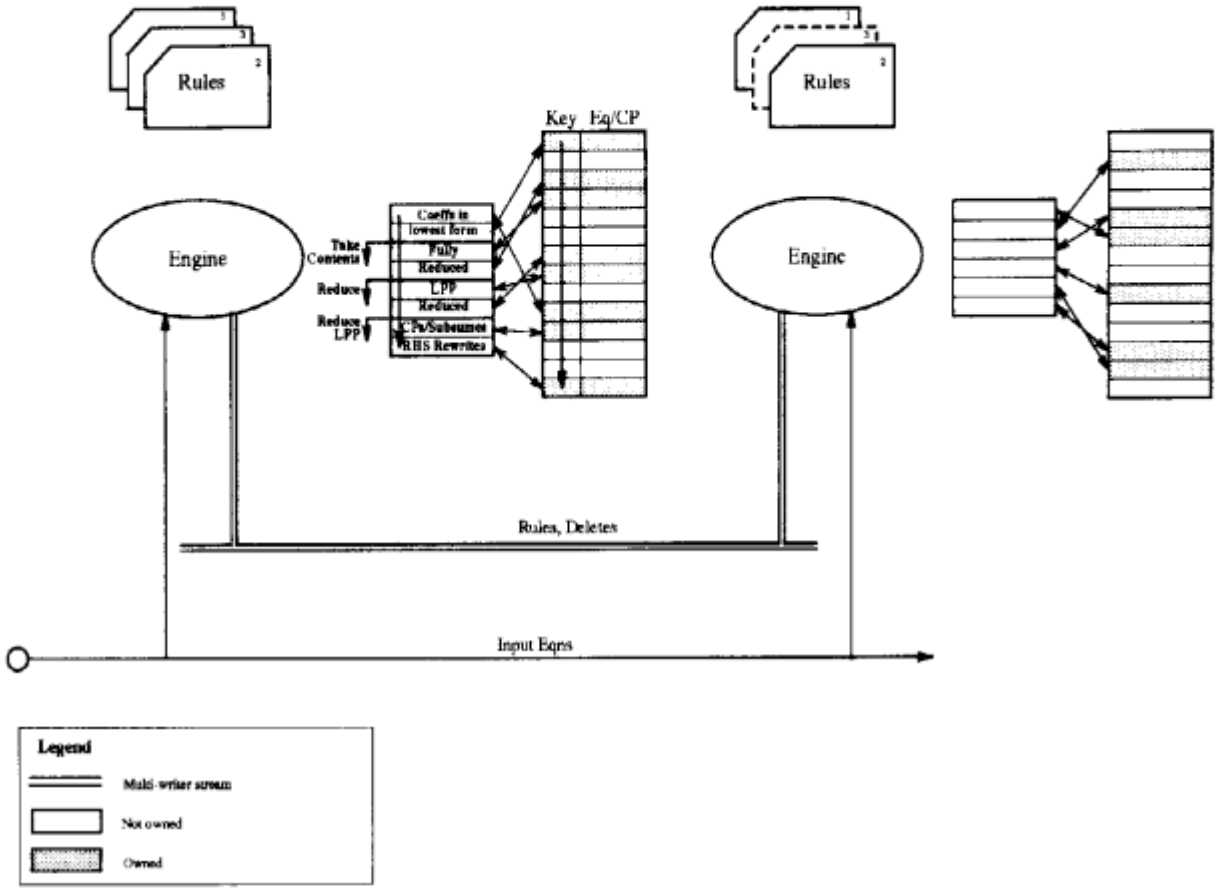

Figure 4: Parallel Buchberger Algorithm

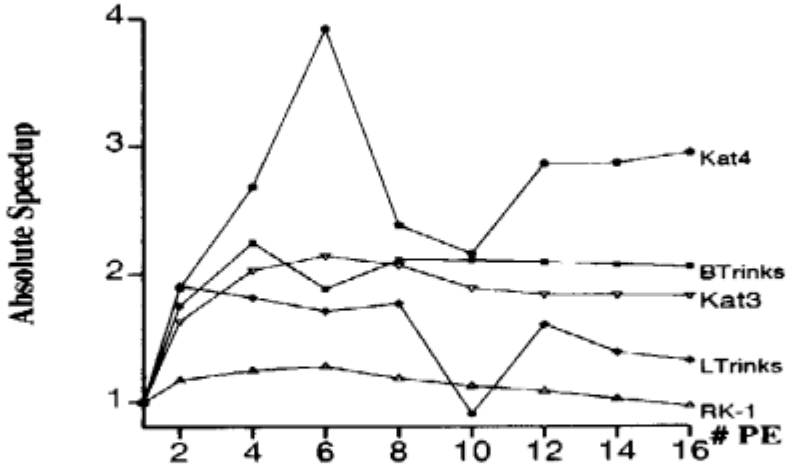

Figure 5: Speedup of Dynamic Algorithm

Except for Katsura 4, the speedup curve (Figure 5) 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 to the extent of passing the performance of SAC and other sequential implementations.

5 Conclusions

We have demonstrated the possibility of developing a concurrent constraint programming language GDCC using the KL1 committed-choice concurrent logic programming language. GDCC is an open system, in the sense that KL1 processes obeying a simple protocol can be used as constraint solvers for user-specified domains. We have demonstrated that Gröbner Base techniques are somewhat applicable to the concurrent context, and presented a constraint solver based on a parallelized Buchberger Algorithm for calculating the Gröbner Base.

The constraint solver exhibits 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.

The performance of the parallel Knuth-Bendix approach to solving Gröbner Bases needs much further study. It well known that the ordering of constraints has a strong effect on the performance of the Buchberger Algorithm, but although GDCC guards can affect the arrival order of constraints occuring in the program text, internally generated constraints (critical pairs) are not subject to this control. Therefore some other means should be found to order the solver's subcomputations, and in this respect research into intelligent scheduling strategies seems worthwhile. Guard-oriented scheduling also opens up the possibility of giving a complete implementation of entailment, and extend the implementation to decide *parametric queries*[Las89]. The latter has many possible applications, including semantic meta-programming.

Finally, the problem of variable management in KL1 may be overcome by global program analysis for some class of GDCC programs, and as program analysis is a very active area of research, we hope to use these results to relieve the programmer of the burden of indicating variable typing and allocation.

References

[AiS88] A. Aiba, K. Sakai, Y. Sato, D. Hawley, and R. Hasegawa. Constraint Logic Programming Language CAL. In *International Conference on Fifth Generation Computer Systems 1988*, pages 263–276, 1988.

[Akl85] Selim. G. Akl. *Parallel Sorting Algorithms*. Notes and Reports in Computer Science and Applied Mathematics. Academic Press, 1985.

[BeP89] H. Beringer and F. Porcher. A Relevant Scheme for Prolog Extensions: CLP (Conceptual Theory). In *6th International Conference on Logic Programming*, pages 131 -148, 1989.

[BoG86] 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.

[Buc83] B. Buchberger. Gröbner bases:An Algorithmic Method in Polynomial Ideal Theory. Technical report, CAMP-LINZ, 1983.

[CaG88] Leandro Caniglia, Andre Galligo, and Joos Heintz. Some new effectivity bounds in computational geometry. In *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes - 6th International Conference*, pages 131–151. Springer-Verlag, 1988. Lecture Notes in Computer Science 357.

[DiH88] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Bertheir. The Constraint Logic Programming Language CHIP. In *Proceedings FGCS-88*, 1988.

[GeM88] R. Gebauer and H. M. Moller. On an installation of buchberger's algorithm. *J. Symbolic Computation*, 6(2 and 3):275–286, 1988.

[Hen89] P. Van Hentenryck. Parallel constraint satisfaction in logic programming: Prelimiary results of chip with pepsys. In *6th International Conference on Logic Programming*, pages 165–180, 1989.

[Hil90] D. Hilbert. Über die Theorie der algebraischen Formen. *Math. Ann.*, 36:473–534, 1890.

[JaL86] Joxan Jaffar, Jean-Louis Lassez, and Michael J. Maher. A Logic Programming Language Scheme. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Functions, Relations, and Equations*. Prentice-Hall, 1986.

[JaL87] J. Jaffar and J-L. Lassez. Constraint Logic Programming. In *Proceedings of the 14th ACM Principles of Programming Languages Conference*, Munich, January 1987.

[LaM87] Catherine Lassez, Ken McAloon, and Roland Yap. Constraint Logic Programming and Options Trading. forthcoming Research Report, IBM T.J. Watson Research Center, 1987.

[Las89] J-L. Lassez. Querying Constraints. forthcoming Research Report, 1989.

[Mah87] Michael J. Maher. Logic Semantics for a Class of Committed-choice Programs. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 858–876, Melbourne, May 1987.

[NaI89a] K. Nakajima, Y. Inamura, K. Rokusawa, N. Ichiyoshi, and T. Chikayama. Distributed implementation of kl1 on the multi-psi/v2. In *6th International Conference on Logic Programming*, pages 436 451, 1989.

[NaI89b] Katsuto Nakajima, Yu Inamura, Nobuyuki Ichiyoshi, Kazuaki Rokusawa, and Takashi Chikayama. Distributed implementation of KL1 on the Multi-PSI/V2. In *Proceedings of ICLP'89*, pages 436–451, 1989.

[SaA89] K. Sakai and A. Aiba. Cal: A theoretical background of constraint logic programming and its applications. *Journal of Symbolic Computation*, 8:589–603, 1989.

[Sar89] V. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Carnegie-Mellon University, Computer Science Department, January 1989.

[SaS88] Y. Sato and K. Sakai. Boolean Gröbner Base, February 1988. LA-Symposium in winter, RIMS, Kyoto University.

[UeC90] Kazunori Ueda and Takashi Chikayama. Design of the kernel language for the parallel inference machine. *Computer Journal*, December 1990. To appear.

[Ued86] K. Ueda. *Guarded Horn Clauses*. PhD thesis, University of Tokyo, Information Engineering, 1986.

[Wal89] C. Walinsky. CLP(Σ^*): Constraint Logic Programming with Regular Sets. In *6th International Conference on Logic Programming*, pages 181 198, 1989.

A Source for Ellipse Example

```
:- module ellipse.
:- public start/6.
start(P1,P2,P3,P4,P) :- true |
    create_points([A,B,C,D]),
    mid_points([P1,P2,P3,P4,P1],[A,B,C,D]),
    calc_ellipse(A,B,C,D,P).

calc_ellipse(p(X1,Y1),p(X2,Y2),p(X3,Y3),p(X4,Y4),P) :-
        alg:(X1-X2)*(Y3-Y4)=(Y1-Y2)*(X3-X4),
        alg:(X1-X4)*(Y2-Y3)=(Y1-Y4)*(X2-X3) |
    transform_matrix(T),
    calc_transform(T,p(X1,Y1),p(X2,Y2),p(X3,Y3)),
    create_point(Pp),
    transform(Pp,T,P),
    circle_2(Pp).

circle_2(p(X,Y)) :- true |
    alg:X^2 + Y^2 = 2.

transform_matrix(T) :- true |
    alloc(100,A1,B1,C1,A2,B2,C2),
    T=[[A1,B1,C1],[A2,B2,C2]].

calc_transform(T,P1,P2,P3) :- true |
    square_2(LL,UL,UR,LR),
    transform(LL,T,P1),
    transform(UL,T,P2),
    transform(UR,T,P3).

square_2(LL,UL,UR,LR) :- true |
    UL=p(-1,1),  UR=p(1,1),
    LL=p(-1,-1), LR=p(1,-1).

transform(P,[L1,L2],Pp) :- true |
    Pp = p(Xp,Yp),
    transform1(P,L1,Xp),
    transform1(P,L2,Yp).

transform1(p(X,Y),[A,B,C],P) :- true |
    alg:P=A*X+B*Y+C.

mid(p(X1,Y1),p(X2,Y2),p(X3,Y3)) :- true |
    alg:2*X3=X1+X2,
    alg:2*Y3=Y1+Y2.

create_points([]).
create_points([I|Is]) :- true |
    create_point(I),
    create_points(Is).

mid_points([_],[]).
mid_points([A,B|Is],[M|Ms]) :- true |
```

```
        mid(A,B,M),
        mid_points([B|Is],Ms).

create_point(P) :- true | alloc(100,X,Y), P=p(X,Y).

?- alloc(0,X,Y),alloc(1,A),
   ellipse:start(P1,P2,P3,P4,p(X,Y)),
   P1=p(A,-1),P2=p(-1,1),P3=p(1,1),P4=p(1,-1).
```

B Result of Compiling Ellipse Example

```
 :- module ellipse .
 :- public'$$gdcc$$' / 0 .
'$$gdcc$$' .
 :- public start/ 6 .
start(Var0,Var1,Var2,Var3,Var4,XA):-Var0=Var5,Var1=Var6,Var2=Var7,Var3=Var8,Var4=Var9|
    create_points([Var10,Var11,Var12,Var13],Var14),
    mid_points([Var5,Var6,Var7,Var8,Var5],[Var10,Var11,Var12,Var13],Var15),
    calc_ellipse(Var10,Var11,Var12,Var13,Var9,Var16),
    XA={Var14,Var15,Var16} .
calc_ellipse(Var0,Var1,Var2,Var3,Var4,XA) :- true |
    XA={XA1,{Var5}},
    ( Var6=success,Var7=success,Var8=success
   -> Abort=success,
      transform_matrix(Var9,Var10),
      calc_transform(Var9,p(Var11,Var12),p(Var13,Var14),p(Var15,Var16),Var17),
      create_point(Var18,Var19),
      transform(Var18,Var9,Var20,Var21),
      circle_2(Var18,Var22),
      XA1={Var10,Var17,Var19,Var21,Var22}
    ),
    ( Var0=p(Var23,Var24),
      Var1=p(Var25,Var26),
      Var2=p(Var27,Var28),
      Var3=p(Var29,Var30),
      Var4=Var31
    -> Var5=[suspend([Var11,Var12,Var13,Var14,Var15,Var16,Var32,Var33],Abort,
                     [alg#ask(=,expr(1*(Var11*Var16)+(-1*(Var11*Var33)
                                     +(-1*(Var12*Var15)+(1*(Var12*Var32)
                                     +(-1*(Var13*Var16)+(1*(Var13*Var33)
                                     +(1*(Var14*Var15)+-1*(Var14*Var32))))))))),
                              {Var7,Abort})]),
             suspend([Var11,Var12,Var13,Var14,Var15,Var16,Var32,Var33],Abort,
                     [alg#ask(=,expr(1*(Var11*Var14)+(-1*(Var11*Var16)
                                     +(-1*(Var12*Var13)+(1*(Var12*Var15)
                                     +(1*(Var13*Var33)+(-1*(Var14*Var32)
                                     +(-1*(Var15*Var33)+1*(Var16*Var32))))))))),
                              {Var8,Abort})])],
      Var6=success,
      Var11=Var23,Var12=Var24,Var13=Var25,Var14=Var26,Var15=Var27,Var16=Var28,
      Var32=Var29,Var33=Var30,Var20=Var31
    ; wait(Abort)
    -> Var5 ={}
    ).
circle_2(Var0,XA) :- Var0=p(Var1,Var2) |
    XA=[suspend([Var1,Var2],[alg#tell(=,expr(-2*1+(1*Var1**2+1*Var2**2)))])].
transform_matrix(Var0,XA) :- Var0=Var1  |
    Var1=[[Var2,Var3,Var4],[Var5,Var6,Var7]],
    XA=[new_variable(Var2,100),new_variable(Var3,100),new_variable(Var4,100),
        new_variable(Var5,100),new_variable(Var6,100),new_variable(Var7,100)].
calc_transform(Var0,Var1,Var2,Var3,XA):-Var0=Var4,Var1=Var5,Var2=Var6,Var3=Var7|
    square_2(Var8,Var9,Var10,Var11,Var12),
    transform(Var8,Var4,Var5,Var13),
```

```
    transform(Var9,Var4,Var6,Var14),
    transform(Var10,Var4,Var7,Var15),
    XA={Var12,Var13,Var14,Var15} .
square_2(Var0,Var1,Var2,Var3,XA) :- Var0=Var4,Var1=Var5,Var2=Var6,Var3=Var7  |
    Var5=p(-1,1),
    Var6=p(1,1),
    Var4=p(-1,-1),
    Var7=p(1,-1),
    XA={} .
transform(Var0,Var1,Var2,XA) :- Var0=Var3, Var1=[Var4,Var5], Var2=Var6  |
    transform1(Var3,Var4,Var7,Var8),
    transform1(Var3,Var5,Var9,Var10),
    Var6=p(Var7,Var9),
    XA={Var8,Var10} .
transform1(Var0,Var1,Var2,XA):-Var0=p(Var3,Var4),Var1=[Var5,Var6,Var7],Var2=Var8 |
    XA=[suspend([Var3,Var4,Var5,Var6,Var7,Var8],
                [alg#tell(=,expr(-1*(Var3*Var5)+(-1*(Var4*Var6)
                                  +(-1*Var7+1*Var8))))])].
mid(Var0,Var1,Var2,XA) :-
    Var0=p(Var3,Var4),
    Var1=p(Var5,Var6),
    Var2=p(Var7,Var8) |
    XA=[suspend([Var3,Var5,Var7],[alg#tell(=,expr(-1*Var3+(-1*Var5+2*Var7)))]),
        suspend([Var4,Var6,Var8],[alg#tell(=,expr(-1*Var4+(-1*Var6+2*Var8)))])].
create_points(Var0,XA) :- Var0=[] | XA={} .
create_points(Var0,XA) :- Var0=[Var1|Var2] |
    create_point(Var1,Var3),
    create_points(Var2,Var4),
    XA={Var3,Var4} .
mid_points(Var0,Var1,XA) :- Var0=[Var2], Var1=[] |
    XA={} .
mid_points(Var0,Var1,XA) :- Var0=[Var2,Var3|Var4], Var1=[Var5|Var6] |
    mid(Var2,Var3,Var5,Var7),
    mid_points([Var3|Var4],Var6,Var8),
    XA={Var7,Var8} .
create_point(Var0,XA) :- Var0=Var1  |
    Var1=p(Var2,Var3),
    XA=[new_variable(Var2,100),new_variable(Var3,100)|{}] .
```