TR-0750

Parallel Constraint Logic Programming Language GDCC and its Parallel Constraint Solvers

by

S. Terasaki, D. Hawley, H. Sawada, K. Satoh,
S. Menju, T. Kawagishi, N. Iwayama
& A. Aiba

March, 1992

Parallel Constraint Logic Programming Language GDCC and its Parallel Constraint Solvers

Satoshi Terasaki, David J. Hawley*, Hiroyuki Sawada, Ken Satoh, Satoshi Menju, Taro Kawagishi, Noboru Iwayama and Akira Aiba

Institute for New Generation Computer Technology
4-28, Mita 1-chome, Minato-ku, Tokyo 108, Japan

Abstract

Parallelization of a constraint logic programming (CLP) language can be considered at two major levels: the execution of an inference engine and a solver in parallel, and the execution of a solver in parallel. GDCC is a parallel CLP language that satisfies this two level parallelism. It is implemented in KL1 and is currently running on the Multi-PSI, a loosely coupled distributed memory parallel machine. GDCC has multiple solvers and a *block* mechanism that enables meta-operation to a constraint set. Currently there are three solvers: an algebraic solver for nonlinear algebraic equations using the Buchberger algorithm, a boolean solver for boolean equations using the Boolean Buchberger algorithm, and a linear integer solver for mixed integer programming. 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 machine, but nevertheless shows consistently good performance and speedups for a number of standard benchmarks from the literature.

1 Introduction

Constraint logic programming (CLP) is an extension of logic programming that introduces a facility to write and solve constraints in a certain domain, where constraints are relations among objects. The CLP paradigm was proposed by Colmeraure[Colmerauer 87], and Jaffar and Lassez[Jaffar and Lassez 87]. A similar paradigm (or languages) was proposed by the ECRC group [Dincbas *et al.* 88]. A sequential CLP language CAL (*Contrainte avec Logique*) was also developed at ICOT[Aiba *et al.* 88].

The CLP paradigm is a powerful programming methodology that allows users to specify what (declarative knowledge) without specifying how (procedural knowledge). This abstraction allows programs to be more concise and more expressive. Unfortunately, the generality of constraint programs brings with it a higher computational cost. Parallelization is an effective way of making CLP systems efficient. There are two major levels of parallelizing CLP systems. One is the execution of an inference engine and constraint solvers in parallel. The other is the execution of a constraint solver in parallel.

Several works have been published on extending this work from the sequential to the concurrent frame. Among them are a proposal of ALPS[Maher 87] that introduces constraints into committed-choice language, a report on some preliminary experiments in integrating constraints into the PEPSys parallel logic system[Hentenryck 89], and a framework for a concurrent constraint (cc) language to integrate constraint programming with concurrent logic programming languages[Saraswat 89].

GDCC[Hawley 91b], Guarded Definite Clauses with Constraints, that satisfies two level parallelism, is a parallel CLP language that introduces the framework of cc into a committed-choice language KL1[Ueda and Chikayama 90], and is currently running on the Multi-PSI, a loosely coupled distributed memory parallel logic machine. GDCC has multiple solvers to enable a user to easily specify a proper solver for a domain: they are an algebraic solver, a boolean solver and a linear integer solver. The incremental evaluation facility is very important to CLP language solvers. That is, a solver must consider cases where constraints are dynamically added to it during execution, not only those cases where all are given statically prior to execution.

The algebraic solver is used to solve non-linear algebraic equations, and can be applied to fields such as computational geometries and handling robot design problems[S. Sato and Aiba 90]. The solver uses the Buchberger algorithm [Buchberger 83, Buchberger 85] that is a method of solving multi-variate polynomial equations. This algorithm is widely used in computer algebra, and also fits reasonably well into the CLP scheme since it is incremental and (almost) satisfaction-complete as shown in [Aiba *et al.* 88, Sakai and Aiba 89]. Re-

*Current office:: Compuflex Japan Inc. 12-4, Kasuya 4-chome, Setagaya-ku, Tokyo 157 Japan cently, there have been several attempts made to parallelize the Buchberger algorithm, with generally disappointing results[Ponder 90, Senechaud 90], except for shared-memory machines[Vidal 90, Clarke *et al.* 90]. An interesting parallel logic programming approach implemented in Strand88[1] on Transputers was reported by Siegl[Siegl 90], with good speedups on the small examples shown, but absolute performance was only fair. We parallelize the Buchberger algorithm, emphasizing on absolute performance and incrementability rather than deceptive parallel speedups.

The boolean solver is used to solve boolean equations and can be applied to a wide range of applications such as logic circuit design. It uses the Boolean Buchberger algorithm [Y. Sato and Sakai 88]. It is different from the original Buchberger algorithm in load-balance of the internal processes, although they are basically similar. We implemented the parallel version of this algorithm, based on behavior analyses, using some example problems.

The target problems for the linear integer solver are combinatorial optimization problems such as scheduling problems, that obtain the minimum (or maximum) value with respect to an objective function in a discrete value domain under a certain constraint set. There are many kinds of formalization to solve the optimization problem, among them an integer programming that can be widely used for various problems. Integer programming still offers many methods of increasing search speed depending on the structures of problems, even if we focus on solving strictly optimized solutions only. The Branch-and-Bound method can apply to wide extent of problems independently to problem structures. We developed a parallel Branch-and-Bound algorithm, aiming to implement a high-speed constraint solver for large problems, and to perform experiments for describing parallel search problem in KL1.

The rest of this paper is organized as follows. We first mention the GDCC language and its system, and describe its parallel constraint solvers. Then, program examples in GDCC are shown using simple problems.

2 Parallel CLP Language

We will present a brief summary of the basic concepts of cc[Saraswat 89]. 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 (consistent) global database of constraints called the *store*. Constraints occurring in program text are classified by whether they are querying or asserting information, into the *Ask* and *Tell* constraints as shown in Figure 1.

[1]Strand88 is similar to KL1, although somewhat less powerful in that it does not support full unification.

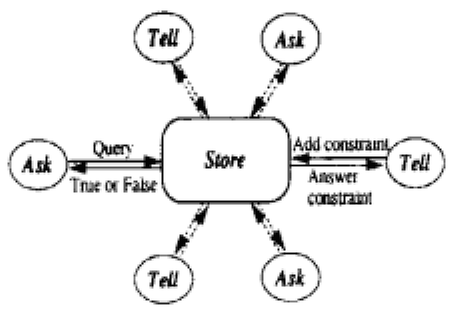

Figure 1: The cc language schema

This paradigm is embedded in a guarded (conditional) reduction system, where guards contain the *Ask* and *Tell*. Control is achieved by requiring that the *Ask* constraints in a guard are true (entailed), and that the *Tell* constraints are consistent (satisfiable), with respect to the current state of the *store*. Thus, this paradigm has a high affinity with KL1.

2.1 GDCC Language

GDCC is a member of the cc language family, although it does not support *Tell* in a guard part. The GDCC language includes most of KL1 as a subset: KL1 builtin predicates and unification can be regarded as the constraints of distinguished domain HERBRAND[Saraswat 89].

Now we define the logical semantics of GDCC as follows. 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 define the following notations.

- We write $v : s$ if variable v has sort s,
- $f : s_1 s_2 \dots s_n \rightarrow s$ if functor f has signature $s_1 s_2 \dots s_n$ and sort s, and
- $p : s_1 s_2 \dots s_n$ if predicate or constraint symbols p has signature $s_1 s_2 \dots s_n$.

We require that terms be well-sorted, according to the standard inductive definitions. An *atomic constraint* is a well-sorted term of the form $c(t_1, t_2, \dots, t_n)$ 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 and g,h range over atoms. We can now define the four relations *entails*, *accepts*, *rejects*, and *suspends*. Let x_g be the variables in constraints c and c_l.

Definition 2.1.1 c *entails* $c_l \stackrel{\text{def}}{=} \Delta \models (\forall x_g)(c \Rightarrow c_l)$

Definition 2.1.2 c *accepts* $c_l \stackrel{\text{def}}{=} \Delta \models (\exists)(c \wedge c_l)$

Definition 2.1.3 c *rejects* $c_l \stackrel{\text{def}}{=} \Delta \models (\forall x_g)(c \Rightarrow \neg c_l)$

Note that the property *entails* is strictly stronger than *accepts*, and that *accepts* and *rejects* are complementary.

Definition 2.1.4 c *suspends* c_l
$\stackrel{\text{def}}{=} c$ *accepts* $c_l \wedge \neg$ (c *entails* c_l).

A GDCC program is comprised of clauses that are defined as tuples (head, ask, tell, body), where "head" is a term having unique variables as arguments, "body" is a set of terms, "ask" is said to be *Ask constraint*, and "tell" is said to be *Tell constraint*. The "head" is the head part of the KL1 clause, "ask" corresponds to the guard part[2], and "tell" and "body" are the body part.

A clause (h, a, c, b) is a candidate for goal g in the presence of *store* s if $s \wedge g = h$ *entails* a. A goal g *commits* to candidate clause (h, a, c, b), by adding $t \cup c$ to the *store* s, and replacing g with b. A goal fails if the all candidate clauses are *rejected*. The determination of *entailment* for multiple clauses and *commitment* for multiple goals can be done in parallel.

Below is a program of pony_and_man written in GDCC.

```
pony_and_man(Heads,Legs,Ponies,Men) :- true |
    alg# Heads= Ponies + Men,
    alg# Legs=  4*Ponies + 2*Men.
```

Where, **pony_and_man(Heads,Legs,Ponies,Men)** is the head of the clause, "|" is the commit operator, **true** is an *Ask* constraint, equations that begin with **alg#** are *Tell* constraints. **alg#** indicates that the constraints are solved by the algebraic solver. In a body part, not only *Tell* constraints, but normal KL1 methods can also be written. In a guard part, we can only write read-only constraints that never change the content of the *store*, in the same way as the KL1 guard where active unification that binds a new value/structure to an undefined variable is inhibited.

But, bi-directionality in the evaluation of constraints, the important characteristic of CLP, is not spoiled by this limitation. For example, the query

```
?-  pony_and_man(5,14,Ponies,Men).
```

will return **Ponies=2, Men=3**. Thus, we can evaluate a constraint bi-directionally as *Tell* constraints have no limitations like *Ask*.

2.2 GDCC System

The GDCC system supports multiple plug-in constraint solvers with a standard stream-based interface, so that users can add new domains and solvers.

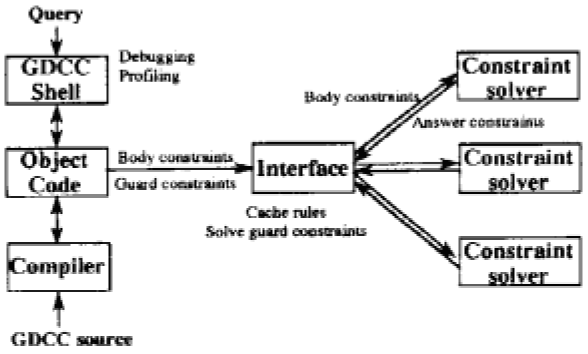

Figure 2: System Construction

The system is shown in Figure 2. The components are concurrent processes.

Specifically, a GDCC program and the constraint solvers may execute in parallel, "synchronizing" only and to the extent necessary, at the program's guard constraints.

The GDCC system consists of:

(i) Compiler
Translates a GDCC source program into KL1 code.

(ii) Shell
Translates queries and provides rudimentary debugging facilities. The debugging facilities comprise the standard KL1 trace and spy 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.

(iii) Interface
Interacts with a GDCC program (object code), sends body constraints to a solver and checks guard constraints using the results from a solver.

(iv) Constraint Solvers
Interact with the interface module and evaluate body constraints.

The decision of entailment using a constraint solver is described in each solver's section, as it differs from each algorithm adopted by a solver.

2.3 Block

A handling robot design support system [S. Sato and Aiba 90] has been used as an experimental application of our CLP systems for a few years. In applying GDCC to this problem, two problems arose. These were the handling of multiple contexts and the synchronization between an inference engine and solvers.

[2] "ask" contains constraints in the HERBRAND domain, that is, it includes the normal guards in KL1.

To clarify the backgrounds to these problems, we explain the handling of multiple contexts in sequential CLP language CAL. CAL has a function to compute approximated real roots in univariate non-linear equations. For instance, it can obtain values $X = \pm\sqrt{2}$ from $X^2 = 2$. Using this facility, the handling robot design support system can solve a given problem in detail. In this example, there are two constraint sets, one that includes $X = \sqrt{2}$, and another that includes $X = -\sqrt{2}$. CAL selects one constraint set from these two and solves it. Then the other is computed by backtracking (i.e., the system forces a failure). In other words, CAL handles these two contexts one-by-one, not simultaneously. In committed-choice language GDCC, however, we cannot use backtracking to handle multiple contexts. There are same problems in implementing hierarchical CLP language[K. Satoh and Aiba 90, K. Satoh 90b] in GDCC.

The other problem is the synchronization between an inference engine and solvers. It is necessary to describe to the timing and the target constraints to execute a function to find approximated real roots. In a sequential CLP, it is possible to control where this description is written in a program. While in GDCC, we need another kind of mechanism to specify a synchronization point, as a clause sequence in a program does not relate to the execution sequence. A similar situation occurs when a meta operation to constraint sets is required, such as computing a maximum value with respect to a given objective function.

Constraint sets in GDCC are basically treated as global. Introducing local constraint sets, however, independence of the global ones, can eliminate these problems. Multiple contexts are realized by considering each local constraint as one context. An inference engine and solvers can be synchronized at the end point of the evaluation of a local constraint set.

Therefore, we introduced a mechanism, called *block*, to describe the scope of a constraint set. We can solve a certain goal sequence with respect to a local constraint set. The block is represented in a program by a builtin predicate `call`, as follows.

`call(` *Goals* `) using` *Solver-Package* `for` *Domain*
`initial` *Input-Con* `giving` *Output-Con*

Constraints in goal sequence *Goals* are computed in a local constraint set. "`using` *Solver-Package* `for` *Domain*" denotes the use of *Solver-Package* for *Domain* in this block. "`initial` *Input-Con*" specifies the initial constraint set. "`giving` *Output-Con*" indicates that the result of computing in the block is *Output-Con*.

Both local variables and global variables can be used in a block where the local variables are only valid within the block and the global ones are valid even outside the block. Local variables are specified by the builtin predicate `alloc/2` that assigns variables to a block. Variables that are not allocated in a block are assumed to be global.

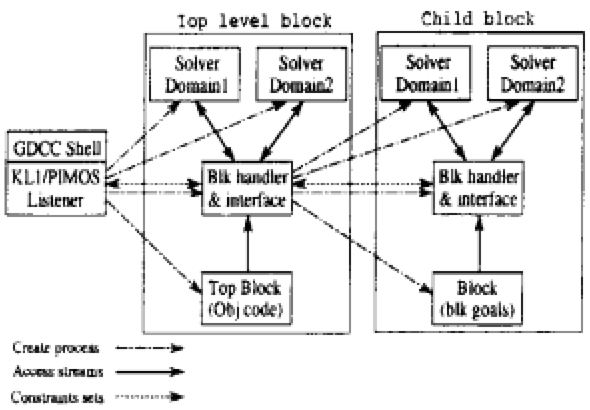

Figure 3: Implementation of *block* in GDCC

A block is executed by evaluating *Goals* with respect to *Input-Con*. The result of *Output-Con* is a local constraint set, that is, it is never merged with the global ones unless specified explicitly by a user.

Let us consider the next program.

```
test:- true |
    alloc(200,A),
    alg#A=-1,
    call( alg#A=1 ) initial nil giving C0,
    call( alg#A=0 ) initial nil giving C1.
```

This program returns the constraint set $\{A = 1\}$ as `C0` and the constraint set $\{A = 0\}$ as `C1`.

The block mechanism is implemented by the three modules shown in Figure 3: an inference engine(block), a block handler and constraint solvers. To encapsulate failure in a block, the *shoen* mechanism of PIMOS[Chikayama *et al.* 88] is used. The block handler creates a block process, sends constraints from a block to a constraint solver, and goals to other processors. Each GDCC goal has a stream connecting to the block handler to which the goal belongs.

3 Parallel Constraint Solvers

3.1 Algebraic Solver

3.1.1 Domain of Constraint

A constraint system that is the target domain of the algebraic solver is generally called a *nonlinear algebraic polynomial equation*. According to the definitions in Section 2.1, this can be formalized as the constraint system $(\Sigma = F \cup C \cup P, \Delta, V, C)$, where:

$$S = \{\mathbf{A}\}$$
$$F = \{\times : \mathbf{AA} \rightarrow \mathbf{A}, + : \mathbf{AA} \rightarrow \mathbf{A}\} \cup \{\text{fraction} :\rightarrow \mathbf{A}\}$$

$$
\begin{aligned}
C &= \{=\} \\
P &= \{\text{string starting with a lowercase letter}\} \\
V &= \{\text{string starting with an uppercase letter}\} \\
\triangle &= \text{axioms of complex numbers}
\end{aligned}
$$

with the structure

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

3.1.2 Gröbner Basis and Buchberger Algorithm

Below is a brief introduction to some notation and definitions needed to explain Gröbner bases and the Buchberger algorithm. Then, the sequential version of the Buchberger algorithm, on which the parallel version is based, is presented.

Definition 3.1.1 (Power product, monomial) *Power product is a product comprised of nonzero and finite number of variables, that is,*

$x_1 x_2 \ldots x_n$ *($n \geq 0$, each x_i are variable).*

Monomial is a product of a coefficient ($\in$rational number) and a power product.

A power product that contains no variable is written as "1".

Definition 3.1.2 (Admissible order) *An ordering $\prec$ is admissible when it satisfies the next properties. For all power products p, q, r,*

(i) $1 \prec p$, and

(ii) $p \prec q \Rightarrow pr \prec qr$.

Examples of admissible ordering that are often used in the Buchberger algorithm are *total degree lexicographic ordering* and *total degree reverse lexicographic ordering*. Let us represent the power product $x_1^{\alpha_1} x_2^{\alpha_2} \cdots x_n^{\alpha_n}$ by the vector $(\alpha_1, \alpha_2, \ldots, \alpha_n)$, where the variables are arranged in lexicographic order. We define the *total degree lexicographic order* $\prec_{dl}$ as follows.

$$
\begin{aligned}
&(\alpha_1, \alpha_2, \ldots, \alpha_n) \prec_{dl} (\beta_1, \beta_2, \ldots, \beta_n) \\
&\quad \Leftrightarrow \sum_{i=1}^{n} \alpha_i < \sum_{i=1}^{n} \beta_i, \quad \text{or,} \\
&\quad \Leftrightarrow \sum_{i=1}^{n} \alpha_i = \sum_{i=1}^{n} \beta_i, \quad \exists i \;\; \alpha_i < \beta_i, \; \alpha_j = \beta_j \; (j < i).
\end{aligned}
$$

That is, the order $\prec_{dl}$ determines a greater monomial by comparing the vector elements in lexicographic order, when the total degree is the same between the two monomials. On the other hand, the *total degree reverse lexicographic order* $\prec_{drl}$ is defined by:

$$
\begin{aligned}
&(\alpha_1, \alpha_2, \ldots, \alpha_n) \prec_{drl} (\beta_1, \beta_2, \ldots, \beta_n) \\
&\quad \Leftrightarrow \sum_{i=1}^{n} \alpha_i < \sum_{i=1}^{n} \beta_i, \quad \text{or,} \\
&\quad \Leftrightarrow (\sum_{i=1}^{n} \alpha_i, -\alpha_n, \ldots, -\alpha_2) \prec_{dl} (\sum_{i=1}^{n} \beta_i, -\beta_n, \ldots, -\beta_2)
\end{aligned}
$$

When the total degree of two monomials is equal, this order compares the subtotal degree by removing the last elements from both vectors.

Let $Lt(f)$ denote the maximal monomial of a polynomial f with respect to a certain admissible ordering, and $Rest(f)$ mean the remaining monomials of f. Let the power product and coefficient of $Lt(f)$ be $Lp(f)$ and $Lc(f)$ respectively.

For each polynomial f $(= Lc(f)Lp(f) + Rest(f))$, we define a rewriting rule $\Rightarrow_f$ over polynomials as follows.

Definition 3.1.3 (Rewriting) $g \Rightarrow_f h$, *if a monomial of a polynomial p is a multiple of $Lp(f)$ then the monomial is replaced with $\frac{-Rest(f)}{Lc(f)}$, and the result of calculation by the replacement is h. For a finite set of polynomials G, $g \Rightarrow_G h$ if $\exists f \in G$ and $g \Rightarrow_f h$.*

Definition 3.1.4 (Irreducible) *The irreducible form of a polynomial g w.r.t. $\Rightarrow_G$ is the polynomial which cannot be rewritten by $\Rightarrow_G$ any more after applying the rewriting rule set G finitely many (or zero) times. The irreducible form of g is denoted by $g\downarrow_G$.*

Let $R[x_1, \ldots, x_m]$ be a polynomial ring in n variable of $x_1, \ldots, x_m$ over the rational number field, and $f_1, \ldots, f_n$ be elements of it. A polynomial ideal $\mathcal{I}$ generated by $f_1, \ldots, f_n$ is a polynomial set defined by the following.

Definition 3.1.5 (Polynomial ideal)

(i) $\mathcal{I} \neq \phi$, $f, g \in \mathcal{I} \Rightarrow f - g \in \mathcal{I}$ (property of modules)

(ii) $f \in \mathcal{I} \Rightarrow h \cdot f \in \mathcal{I}$ for any $h \in R[x_1, \ldots, x_m]$

With no loss of generality, we can assume that all polynomial equations are in the form $f = 0$. Let $E = 0$ be a system of polynomial equations $\{f_1 = 0, \ldots, f_n = 0\}$. The following close relation between the solutions of $E = 0$ and the elements of $\mathcal{I}(E)$ of the ideal generated by E is well known.

Theorem 3.1.1 (Hilbert zero point theorem) *Let f be a polynomial. Every solution of $E = 0$ is also a solution of $f = 0$, iff there exists a natural number s such that $f^s \in \mathcal{I}(E)$.*

Corollary 3.1.1 *E has no solution iff $1 \in \mathcal{I}(E)$.*

Thus, the problem of solving given polynomial equations is reduced to that of deciding whether a polynomial belongs to the ideal. Buchberger introduced the notion of Gröbner bases, and devised an algorithm to determine the membership relations of a polynomial and to the ideal [Buchberger 83, Buchberger 85].

Let there be an admissible ordering among monomials and let a system of polynomial equations $E = 0$ be given.

A rough sketch of the algorithm is as follows. In the system of E, each equation can be considered as being a rewriting rule as defined in Definition 3.1.3. When the left hand sides $Lp(f_1)$ and $Lp(f_2)$ of two rewrite rules f_1 and f_2 are not mutually prime, the least common multiple of their left hand sides can be rewritten in two different ways according to these two rules. The pair resulting from this rewriting is called a critical pair. If further rewriting does not succeed in converging a critical pair, the pair is said to be divergent. To get a confluent rewriting system, equations made from such critical pairs, S-polynomials, are added to the system of equations. By repeating this procedure, we can eventually obtain a confluent rewriting rule set. This confluent rewriting rule set is called a Gröbner basis of E.

Definition 3.1.6 (Gröbner basis [Buchberger 83]) *The Gröbner basis $G(E)$ is a finite set that satisfies the following properties.*

(i) $\mathcal{I}(E) = \mathcal{I}(G(E))$

(ii) For all f, g, $f - g \in \mathcal{I}(E)$ iff $f\downarrow_G = g\downarrow_G$, ***especially**, $f \in \mathcal{I}(E)$* iff $f\downarrow_G = 0$, *and,*

(iii) G is reduced *if every element of the basis is irreducible w.r.t. all the others.*

From Theorem 3.1.1, the reduced $G(E)$ can be regarded as being the canonical form of the solution of $E = 0$, because the reduced Gröbner basis with respect to a given admissible ordering is unique. Moreover, when $E = 0$ does not have a solution, $\{1\} \in G(E)$ is deduced from Corollary 3.1.1.

Definition 3.1.7 (Critical pair, S-polynomial) *If two rewriting rules f_1, f_2 are not mutually prime, that is $Lp(f_1)$ and $Lp(f_2)$ have a greatest common divisor other than 1, the pair f_1, f_2 is called the* critical pair, *and the polynomial made from this critical pair in the following way:*

$$Lc(f_2)\frac{lcm(f_1, f_2)}{Lp(f_1)} \cdot f_1 - Lc(f_1)\frac{lcm(f_1, f_2)}{Lp(f_2)} \cdot f_2$$

is called S-polynomial *and denoted by $Spoly(f_1, f_2)$, where, $lcm(f_1, f_2)$ is the least common multiple of $Lp(f_1)$ and $Lp(f_2)$.*

Figure 4 shows the sequential version of the Buchberger algorithm. E denotes the input polynomial equation set, and R is the output Gröbner basis. Line (4) indicates the rewriting process using R. Lines (7), (8) and (9) are the subsumption test in which the old rule set is updated by the newly generated rule. If the left hand side of an old rule is rewritten by the new rule, the rewritten rule goes back to equation set F. Line (12) is the S-polynomial generation.

```
(1)  input  $F := E$, $R := \emptyset$
(2)  while  $F \neq \emptyset$
(3)    choose  $f \in F$
(4)    $F := F - \{f\}$,  $f' := f\downarrow_R$
(5)    if  $f' \neq 0$ then
(6)      for  every $p \in R$
(7)        if  $Lt(p) \Rightarrow_{f'} lt(p')$
(8)          then $F := F \cup \{lt(p') + Rest(p)\}$,  $R := R - \{p\}$
(9)          else $R := (R - \{p\}) \cup \{Lt(p) + Rest(p)\downarrow_{R \cup \{f'\}}\}$
(10)       endif
(11)     endfor
(12)     $F := F \cup Spoly(f', R)^{\dagger}$,   $R := R \cup \{f'\}$
(13)   endif
(14) endwhile
(15) output  $R$ ($R$ is $G(E)$)
```

†: *Spoly*(f', R) is to be generated by S-polynomials between polynomial f' and all elements in rule set R.

Figure 4: Sequential Buchberger algorithm

3.1.3 Satisfiability, Entailment

Based on the above results, we could determine satisfiability by using the Buchberger algorithm to incorporate the polynomial into the Gröbner bases as per Corollary 3.1.1. But the method of Definition 3.1.6(ii) is incomplete in terms of deciding entailment, since the relation between the solutions and the ideal described in Theorem 3.1.1 is incomplete. For example, the Gröbner basis of $\{X^2 = 0\}$ is $\{X^2 \rightarrow 0\}$, and rewriting using this Gröbner basis cannot show that $X = 0$ is entailed. There are several approaches solving the entailment problem:

(a) Use the Gröbner basis of the radical of the generated ideal, $\mathcal{I}$, i.e. $\{p | p^n \in \mathcal{I}\}$. Although it is theoretically computable, efficient implementation is not possible.

(b) As a negation of $p = 0$, add $p\alpha$ to the Gröbner basis and use the Buchberger algorithm, where α is a new variable. Iff 1 is included in the new Gröbner basis, $p = 0$ is held in the old Gröbner basis. This has the unfortunate side-effect of changing the Gröbner basis.

(c) Find n such that p^n is rewritten to 0 by the Gröbner basis of the generated ideal. Since n is bounded[Cangilia *et al.* 88], this is a complete decision procedure. The bound, however, is very large.

When there are a lot of resources to compute, and no more computation can be done, according to the method described in (c) we may adopt the incremental solution of repeatedly raising p from a small positive integer power and rewriting it by the Gröbner basis. On the other hand, the total efficiency of the system is greatly affected by the computation time in deciding entailment. Therefore, we determine the entailment by rewriting using a Gröbner basis from the view point of efficiency, even though this method is incomplete. This decision procedure runs on the interface module parallel with the solver execution, as shown in Figure 2. Whenever a new rule is generated, the solver sends the new rule to the interface module via a communication stream. The interface determines entailment while storing (intermediate) rules to a self database. The interface updates the database by itself whenever a new rule from the solver arrives. It can also handle constraints such as inequalities in the guard parts, if they can be solved by passive evaluation.

3.1.4 Parallel Algebraic Solver

There are two main sources of parallelism in the Buchberger algorithm, the parallel rewriting of a set of polynomials, and the parallel testing for subsumption of a new rule against the other rules. Since the latter is inexpensive, we should concentrate on parallelizing the coarse-grained reduction component for the distributed memory machine. 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.

Three different architectures have been implemented: namely, a pipeline, a distributed, and a master-slave architecture. The distributed architecture was already reported in [Hawley 91a, Hawley 91b], however, it has been greatly refined since then. The master-slave architecture also offers comparatively good performance. Thus, we touch on the distributed and master-slave architectures in the following sections.

Distributed architecture

The key idea underlying the distributed architecture is that of sorting a distributed set of polynomials. Each processor contains a complete set of rewriting rules and polynomials, and a load-distribution function ω that logically partitions the polynomials by specifying which processor "owns" which polynomials. The position in the output rule sequence of each polynomial is calculated by its owning processor, based on an associated key (the leading power product), identical in every processor, and which does not change during reduction. A polynomial is output once it becomes the smallest remaining. The S-polynomials 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 when there are no polynomials remaining to be rewritten.

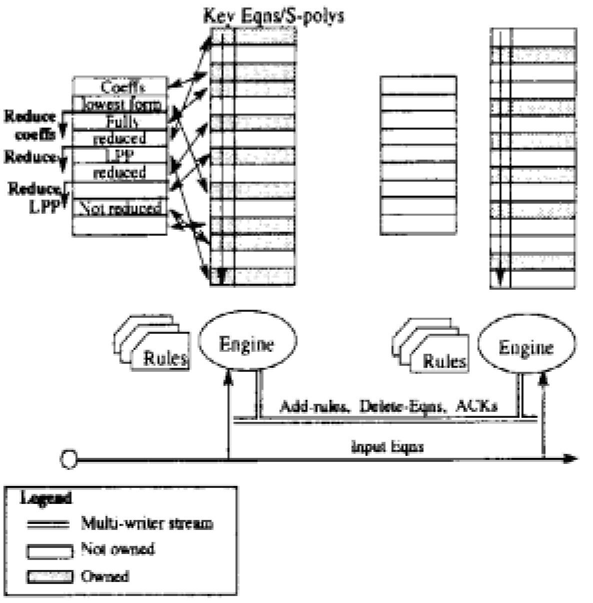

Figure 5: Architecture of distributed type solver

Figure 5 shows the architecture. The central data structures in the implementation are two work item lists: the global list and the local list. The global list, that contains all polynomials including both owned and not owned polynomials, is used to decide the order in which a processor can output a new rule based on the keys of polynomials. On the other hand, the local list consists of owned polynomials only. Items in the local list are rearranged by each processor to maintain increasing key order, whenever an owned polynomial is rewritten.

There will be a situation where, when a processor is busy rewriting polynomials, another processor outputs a new rule. In such a case, any processor that receives a new rule must quit the current task as soon as possible to check subsumption and to update the old rule set. Continuing tasks while using the old rule set without interruption increases the number of useless tasks. To manage such interruption and resumption of rewriting, the complete execution of one piece of work is broken down into a three-stage pipeline: first polynomials are rewritten until the leading power products can be reduced no further, they are fully reduced, and thirdly the coefficients are reduced by taking the greatest common divisor among all coefficients of a polynomial. Based on this breakdown, we pipeline the execution of the entire list, giving us maximum overlap between communication and local computation.

Table 1 shows the results of benchmark problems to show the performance of this parallel algorithm, the benchmark problems are adopted from [Boege *et al.* 86, Backelin and Fröberg 91]. The monomial ordering is degree reverse lexicographic, and low level bignum (mul-

Table 1: Timing (sec) and speedup obtained with distributed architecture

	Number of processors				
Problems	1	2	4	8	16
Katsura-4	9.86	7.48	5.34	4.82	5.94
	1	1.32	1.85	2.05	1.66
Katsura-5	94.89	62.43	48.20	39.95	40.52
	1	1.52	1.97	2.38	2.34
Cyc5-roots	37.24	33.33	20.02	22.52	29.73
	1	1.12	1.86	1.65	1.25
Cyc6-roots	1268.96	1396.37	1555.58	817.07	3266.68
	1	0.909	0.816	1.55	0.388

tiple precision integer) support on PIMOS is used for coefficient calculation. The method of detecting unnecessary S-polynomials proposed by [Gebauer and Möller 88] is implemented. Examples and their variable ordering are shown below.

Katsura-4: $(U_0 < U_1 < U_2 < U_3 < U_4)$

$$U_0^2 - U_0 + 2U_1^2 + 2U_2^2 + 2U_3^2 + 2U_4^2 = 0$$
$$2U_0U_1 + 2U_1U_2 + 2U_2U_3 + 2U_3U_4 - U_1 = 0$$
$$2U_0U_2 + 2U_1^2 + 2U_1U_3 + 2U_2U_4 - U_2 = 0$$
$$2U_0U_3 + 2U_1U_2 + 2U_1U_4 - U_3 = 0$$
$$U_0 + 2U_1 + 2U_2 + 2U_3 + 2U_4 - 1 = 0$$

Katsura-5: $(U_0 < U_1 < U_2 < U_3 < U_4 < U_5)$

$$U_0^2 - U_0 + 2U_1^2 + 2U_2^2 + 2U_3^2 + 2U_4^2 + 2U_5^2 = 0$$
$$2U_0U_1 + 2U_1U_2 + 2U_2U_3 + 2U_3U_4 + 2U_4U_5 - U_1 = 0$$
$$2U_0U_2 + 2U_1^2 + 2U_1U_3 + 2U_2U_4 + 2U_3U_5 - U_2 = 0$$
$$2U_0U_3 + 2U_1U_2 + 2U_1U_4 + 2U_2U_5 - U_3 = 0$$
$$2U_0U_4 + 2U_1U_3 + 2U_1U_5 + U_2^2 - U_4 = 0$$
$$U_0 + 2U_1 + 2U_2 + 2U_3 + 2U_4 + 2U_5 - 1 = 0$$

Cyclic 5-roots: $(X_1 < X_2 < X_3 < X_4 < X_5)$

$$X_1 + X_2 + X_3 + X_4 + X_5 = 0$$
$$X_1X_2 + X_2X_3 + X_3X_4 + X_4X_5 + X_5X_1 = 0$$
$$X_1X_2X_3 + X_2X_3X_4 + X_3X_4X_5 + X_4X_5X_1 + X_5X_1X_2 = 0$$
$$X_1X_2X_3X_4 + X_2X_3X_4X_5 + X_3X_4X_5X_1 + X_4X_5X_1X_2 + X_5X_1X_2X_3 = 0$$
$$X_1X_2X_3X_4X_5 = 1$$

Cyclic 6-roots: $(X_1 < X_2 < X_3 < X_4 < X_5 < X_6)$

$$X_1 + X_2 + X_3 + X_4 + X_5 + X_6 = 0$$
$$X_1X_2 + X_2X_3 + X_3X_4 + X_4X_5 + X_5X_6 + X_6X_1 = 0$$
$$X_1X_2X_3 + X_2X_3X_4 + X_3X_4X_5 + X_4X_5X_6 + X_5X_6X_1 + X_6X_1X_2 = 0$$
$$X_1X_2X_3X_4 + X_2X_3X_4X_5 + X_3X_4X_5X_6 + X_4X_5X_6X_1 + X_5X_6X_1X_2 + X_6X_1X_2X_3 = 0$$
$$X_1X_2X_3X_4X_5 + X_2X_3X_4X_5X_6 + X_3X_4X_5X_6X_1 + X_4X_5X_6X_1X_2 + X_5X_6X_1X_2X_3 + X_6X_1X_2X_3X_4 = 0$$
$$X_1X_2X_3X_4X_5X_6 = 1$$

Sometimes parallel execution is slower than sequential execution. Moreover a serious drawback occurs in the case of "cyclic 6-roots". The reasons are; first, redundant tasks increase in parallel since updating a rule set, generating S-polynomials and detecting unnecessary S-polynomials are overlapped with every processor, second, the selection criteria of the next new rule is only a rough approximation as the keys of not owned polynomials are never updated during rewriting.

Master-slave architecture

In the distributed architecture, if the keys of other polynomials are updated according to their rewriting such that the global smallest polynomial can be found, then much communication between the processors is required. One simple way of avoiding such communication overhead is to have each processor output the local minimum polynomial and another processor decide the global minimum among them. Our third trial, therefore, is the master-slave architecture shown in Figure 6.

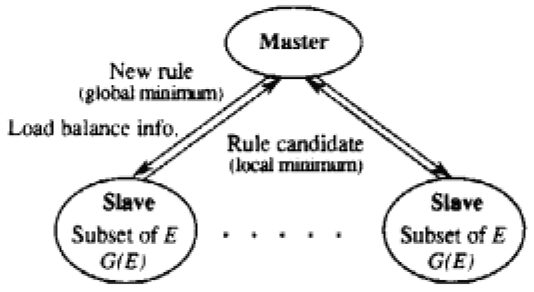

Figure 6: Architecture of master-slave type solver

The set of polynomials E is physically partitioned and each slave has a different part of them. The initial rule set of $G(E)$ is duplicated and assigned to all slaves. New input polynomials are distributed to the slaves by the master. The reduction cycle proceeds as follows.

Each slave rewrites its own polynomials by the $G(E)$, selects the local minimum polynomial from them, and sends its leading power product to the master. The master processor awaits reports from all the slaves, and selects the global minimum power product. The minimum polynomial can be decided only after all the slaves have reported to the master. Those that are not minimums can be decided quickly, however. Thus, the *not-minimum* message is sent to the slaves as soon as possible, and the processors receive the *not-minimum* message reduce polynomial by the old rule set while waiting for a new rule. On one hand, the slave that receives the *minimum* message converts the polynomial into a new rule and sends it to the master, the master sends the new rule to all the slaves except the owner. If several candidates are equal power products, all candidates are converted to rules by owner slaves and they go to final selection by the master.

To make load balance during rewriting, each slave reports the number of polynomials it owns, piggybacked onto leading power product information. The master sorts these numbers into increasing order and decides the order in which to distribute S-polynomials. After applying the unnecessary S-polynomial criterion, each slave generates the S-polynomials it should own corresponding to the order decided by the master. Subsumption test and rule update are done independently by each slave.

Table 2 lists the results of the benchmark problems. The monomial ordering, bignum support and variable ordering are same as for the distributed architecture. Both absolute performance and speedup are improved compared with the distributed architecture. Speedup appears to become saturated at 4 or 8 processors except for "cyclic 6-roots". However, these problems are too small to obtain a good speedup because it takes about half a minute until all the processors become fully operational as the unnecessary S-polynomial criteiorn works well.

Table 2: Timing and speedup of the master-slave architecture

	Number of processors				
Problems	1	2	4	8	16
Katsura-4 (sec)	8.90	7.00	5.83	6.53	9.26
	1	1.27	1.53	1.36	0.96
Katsura-5 (sec)	86.74	57.81	39.88	31.89	36.00
	1	1.50	2.18	2.72	2.41
Cyc.5-roots (sec)	27.58	21.08	19.27	19.16	25.20
	1	1.31	1.43	1.44	1.10
Cyc.6-roots (sec)	1430.18	863.62	433.73	333.25	323.38
	1	1.66	3.30	4.29	4.42

3.2 Boolean Constraint Solver

An algorithm called the Boolean Buchberger algorithm [Y. Sato and Sakai 88] has been proposed for boolean constraints. Boolean constraints are handled differently from algebraic constraints in the following points.

(i) Multiplication and addition are logical-and and exclusive-or, respectively, in boolean constraints.

(ii) Coefficients are boolean values, that is, 1 and 0. So, a monomial is a product of variables.

(iii) The power of a variable is equal to the variable itself ($X^n = X$). So, a monomial is actually a product of distinct variables.

From the property (iii), the theorem of a boolean polynomial that corresponds to Theorem 3.1.1 is as follows.

Theorem 3.2.1 (Zero point theorem) *Let f be a boolean polynomial. Every solution of $E = 0$ is also a solution of $f = 0$, iff $f \in \mathcal{I}(E)$.*

Therefore, the relation between an ideal and solution and the relation between a solution and a Gröbner basis is complete in a boolean polynomial. Thus, *entailment* can be decided by rewriting a guard constraint by a Gröbner basis.

The Boolean Buchberger algorithm differs from the (algebraic) Buchberger algorithm in the following points. That is, we have to consider *self-critical pairs* as well as critical pairs, where a *self-critical pair polynomial* (SC-polynomial) of boolean polynomial f is defined as $Xf + f$ for every variable X of $Lp(f)$. As shown (ii) above, the coefficient calculation in the boolean solver is much cheaper than the algebraic solver, while *self-critical pairs* have to be considered. Thus, the load-balance of this algorithm is completely different from that of the algebraic solver.

3.2.1 Analysis of Sequential Algorithm and Parallel Architecture

The sequential Boolean Buchberger algorithm is shown in Figure 7. Here *EQlist* is a list of input boolean constraints and *GB* is a Boolean Gröbner basis. Numbers (1) to (6) indicate the step number of the algorithm.

From Figure 7 we can see that the following are possible for parallel execution:

(i) polynomial rewriting in step 6.

(ii) monomial rewriting (lower granularity of (i)),

(iii) subsumption test in step 4.

(iv) SC-polynomial generation in step 5, and

(v) S-polynomial generation in step 5.

Since there is a communication overhead in the distributed memory machine, we have to exploit the most coarse-grained parallelism. To design a parallel execution model, we measured the execution time in each step in Figure 7 using two kinds of example program. One is a logic circuit problem for a counter circuit that counts the number of 1's in a three-bit input and outputs the results as a binary code. The other is the n-queens problem where 4 queens have 80 equations with 16 variables, 5 queens have 165 equations with 25 variables, and 6 queens have 296 equations with 36 variables. The time ratio for each step is shown in Table 3.

Table 3: Time ratio of each step (%)

	Step number						
Problem	1	2	3	4	5	6	Total(sec)
4queens	25.8	3.2	8.4	17.3	25.4	19.0	1.8
5queens	6.4	3.4	22.3	3.7	14.5	50.4	53.5
6queens	1.0	1.5	15.0	2.0	2.6	77.7	2240.0
circuit	2.1	4.2	8.2	3.3	8.0	74.0	70.7

```
      input $EQlist$, $GB$
      $EQlist := \{p \in EQlist \mid p \downarrow_{GB} \neq 0\}$

      while $EQlist \neq \emptyset$
(1)     $q := \min\{Lp(p) \mid p \in EQlist\}$
        choose $e \in \{p \in EQlist \mid Lp(p) = q\}$
        $EQlist := EQlist - \{e\}$
(2)     $r = e \downarrow_{GB}$, $RWlist := \emptyset$
(3)     for every $p \in GB$
          if $Lp(p) \Rightarrow_r p'$
            then $GB := GB - \{p\}$
                 $RWlist := RWlist \cup \{p' + Rest(p)\}$
            else $GB := (GB - \{p\})$
                        $\cup \{Lp(p) + Rest(p) \downarrow_{GB \cup \{r\}}\}$
          endif
        endfor
        $GB := GB \cup \{r\}$
(4)     for every $p \in EQlist$
          if $Lp(p) \Rightarrow_r p'$
            then $EQlist := EQlist - \{p\}$
                 $RWlist := RWlist \cup \{p' + Rest(p)\}$
          endif
        endfor
(5)     $RWlist := RWlist \cup SCpoly(r)^{\dagger} \cup Spoly(r, GB)$
(6)     while $RWlist \neq \emptyset$
          choose $p \in RWlist$
          $RWlist := RWlist - \{p\}$
          if $p \neq 0$
            then if $Lp(p) \Rightarrow_{GB} p'$
              then $RWlist := RWlist \cup \{p' + Rest(p)\}$
              else $EQlist := EQlist \cup \{p\}$
            endif
          endif
        endwhile
      endwhile
      output $GB$
```

† : $SCpoly(r)$ indicates the set of all self-crirical pair polynomials for r.

Figure 7: Booelan Buchberger algorithm

We can consider another parallel execution model by modifying the algorithm. Although Figure 7 shows all the reducible polynomials lumped together and rewritten in step 6, this reduction may be distributed to steps 3, 4 and 5. Moreover, reduction may be done in each step independently. Let steps 3', 4' and 5' denote the modified steps 3, 4 and 5. If execution times of steps 3', 4' and 5' are balanced after applying the modification to the algorithm, this model is also a good parallel execution model. However, as shown in Table 4, the times are not balanced. So, we can discard this possibility of parallelization.

From the above analysis, it becomes clear that step 6 is the largest part of the execution, the other parts being small. Therefore, we can determine the master-slave parallel execution model to make the best use of parallelism in step 6, as shown in Figure 8.

Table 4: Time ratio in modified algorithm (%)

Problem	Step number 3'	4'	5'
4queens	12.1	23.5	36.4
5queens	24.7	11.2	54.0
6queens	15.5	39.9	41.9
circuit	8.2	33.5	51.7

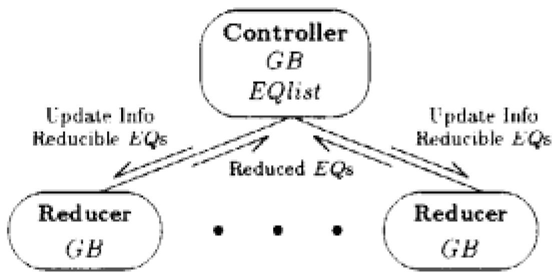

Figure 8: Parallel execution model

The controller (master) is in charge of step 1 to step 5 in the algorithm and the other reducers (slaves) reduce polynomials by GB. The message from the controller to the reducers consists of update information for GB and the polynomials to be rewritten. After receiving the message, the reducer first updates its current GB according to the update information, rewrites the polynomials from the controller, and finally sends the results of the reduction to the controller. As the controller becomes idle after sending the message, the controller also acts as a reducer during the reduction process. The number of polynomials sent to each reducer is kept as equal as possible to balance the loads for each processor.

3.2.2 Implementation and Evaluation

Having implemented the above parallel execution model in KL1, the following improvement was made.

Improvement 1 We can remove redundant equations from $EQlist$, produced by deleting rules in step 3, prior to their distribution. Although this removal can be done in each reducer, the distributed tasks may not be well balanced since the removal of tasks is much less involved than reduction.

Improvement 2 We can distinguish rules of the form "$x = A$" ("A" is variable) from other rules since these rules express assignments only and we need not consider SC-polynomials nor S-polynomials for these rules. These rules are stored differently in the controller and, if a new equation is input, we first apply these assignments in the controller to the equation.

By this application, reducers do not have to store such rules and the time needed to generate an SC-polynomial and S-polynomial can be saved.

Improvement 3 If the right hand side (RHS) of a rule is 0, then no SC-polynomial can be produced. If both RHSs of two rules are 0, then an S-polynomial cannot be produced. Therefore, the RHS of a rule is checked first. This technique is also effective for the sequential version.

Table 5 lists the execution times and the improvement ratio for the 6 queens problem.

Table 5: Timing and improvement ratio

Number of PEs	1	2	4	8	16
Original version (sec)	3735	2400	1745	1539	1262
Improved version (sec)	2489	1706	1223	1142	1092
Improvement ratio(%)	66.6	71.1	70.1	74.2	86.5

Let a purely sequential part in the parallel execution model be a and its parallel executable part be b. Then, we can approximate the execution time for n PEs as $(a+b)/n$. By calculating a and b from the data, we obtain $a = 1130$, $b = 2590$ for the original version, and $a = 930$, $b = 1540$ for the improved version. This means that the parallel executable part constitutes 70% to the entire execution for the original version and 62% for the improved version. Since we parallelized the sequential algorithm to obtain the original version, 70% is a satisfactory ratio for parallel execution since this ratio is very near to the upper bound value calculated from the analysis of the sequential algorithm. The difference is caused by the task distribution overhead. In the improved version, the ratio of the parallel executable part is decreased because of the increase in the number of controller tasks. However, this result is encouraging since the overall performance is improved.

3.3 Integer Linear Constraint Solver

The constraint solver for the integer linear domain checks the consistency of the given equalities and inequalities of rational coefficients, and gives the maximum or minimum values of the objective linear function under these constraint conditions. The integer linear solver utilizes the rational linear solver for the optimization procedure to obtain the evaluation of relaxed linear problems created as part of the solution. A rational linear solver is realized by the simplex algorithm. The purpose of this constraint solver is to provide a fast solver for the integer optimization domain by achieving a computation speedup by incorporating the search process into a parallel program.

These solvers can determine satisfiability and entailment. Satisfiability can be easily checked by the simplex algorithm. Entailment is equivalent to negation failure with respect to a constraint set.

In the following we discuss the parallel search method employed in this integer linear constraint solver. The problem we are addressing is a mixed integer programming problem, to find a maximum or minimum value of a given linear function under integer linear constraints. The method we use is the Branch-and-Bound algorithm.

The Branch-and-Bound algorithms proceed by dividing the original problem into two child problems successively, producing a tree-structured search space. If a certain node gives an actual integer solution (that is not necessarily optimal), and if other search nodes are guaranteed to have lower objective function values than that solution, then the latter nodes need not be searched. In this way, this method prunes sub-nodes through the search space to effectively cut down computation costs, but those costs still become quite high for large-scale problems, since the costs increase in an exponentially with the size of the problem.

As a parallelization of the Branch-and-Bound algorithm, we distribute the search nodes created through the branching process to different processors, and let these processors work on their own sub-problems sequentially. Each sequential search process communicates with other processes to prune the search nodes. Many search algorithms utilize heuristics to control the schedule of the order of the sub-nodes to be searched, thus reducing the number of nodes needed to obtain the final result. Therefore it is important, in parallel search algorithms, to balance the distributed load among processors, and to communicate information for pruning as quickly as possible between these processors. We adopted one of the best search heuristics used in sequential algorithms.

3.3.1 Formulation of Problems

We consider the following mixed-integer linear optimization problems.

Problem – *ILP*
Minimize the following objective function of real variables x_j and integer variables y_j,

$$z = \sum_{i=1}^{n} p_i x_i + \sum_{i=1}^{m} q_i y_i$$

under the linear constraint conditions:

$$\sum_{i=1}^{n} a_{ij} x_j + \sum_{i=1}^{m} b_{ij} y_j \geq e_j \quad \text{for } 1 \leq j \leq l$$
$$\sum_{i=1}^{n} c_{ij} x_j + \sum_{i=1}^{m} d_{ij} y_j = f_j \quad \text{for } 1 \leq j \leq k$$

where

$$x_i \in \boldsymbol{R} \text{ and } x_i \geq 0 \text{ for } 1 \leq i \leq n$$
$$y_i \in \boldsymbol{Z} \text{ where } l_i \leq y_i \leq u_i \text{ and } l_i, u_i \in \boldsymbol{Z} \text{ for } 1 \leq i \leq m$$
$$a_{ij},\ b_{ij},\ c_{ij},\ d_{ij},\ e_i,\ f_i \text{ are real constants.}$$

In practical situations integer variables y_j often take only 0, 1, but here we consider the general case.

3.3.2 Sequential Branch-and-Bound Algorithm

As a preparation to solve the above mixed-integer linear problems $\boldsymbol{ILP}$, we consider the continuously-relaxed problem $\boldsymbol{LP}$.

Problem – $\boldsymbol{LP}$
Minimize the following objective function of real variables x_j, y_j,

$$z = \sum_{i=1}^{n} p_i x_i + \sum_{i=1}^{m} q_i y_i$$

under the linear constraint conditions:

$$\sum_{i=1}^{n} a_{ij} x_j + \sum_{i=1}^{m} b_{ij} y_j \geq e_j \quad \text{for} \quad 1 \leq j \leq l$$
$$\sum_{i=1}^{n} c_{ij} x_j + \sum_{i=1}^{m} d_{ij} y_j = f_j \quad \text{for} \quad 1 \leq j \leq k$$

where

$$x_i \in \boldsymbol{R} \text{ and } x_i \geq 0 \text{ for } 1 \leq i \leq n$$
$$y_i \in \boldsymbol{R} \text{ where } l_i \leq y_i \leq u_i \text{ and } l_i, u_i \in \boldsymbol{Z} \text{ for } 1 \leq i \leq m$$
$$a_{ij},\ b_{ij},\ c_{ij},\ d_{ij},\ e_i,\ f_i \text{ are real constants.}$$

$\boldsymbol{LP}$ can be solved by the simplex algorithm. If the values of original integer variables are exact integers, then it also gives the solution of $\boldsymbol{ILP}$. Otherwise, we take a non-integer value $\bar{y}_s$ for the solution of $\boldsymbol{LP}$, and impose two new interval constraints $\bar{y}_s$, $l_s \leq y_s \leq [\bar{y}_s]$ and $[\bar{y}_s]+1 \leq y_s \leq u_s$, where y_s is an integer variable, and obtain two child problems (Figure 9). Continuing this procedure, called branching, we continue to divide the search space to produce more constrained sub-problems as we proceed deeper into the tree structured search space. Eventually this process leads to a sub-problem having a continuous solution that is also an integer solution to the problem. Also we can select the best integer solution from those found in the process.

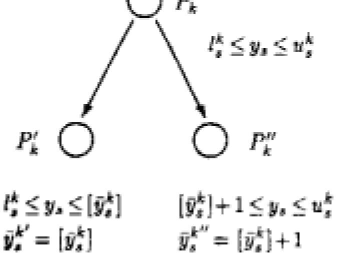
P_k
$l_s^k \leq y_s \leq u_s^k$
P_k'
P_k''
$l_s^k \leq y_s \leq [\bar{y}_s^k]$
$\bar{y}_s^{k'} = [\bar{y}_s^k]$
$[\bar{y}_s^k]+1 \leq y_s \leq u_s^k$
$\bar{y}_s^{k''} = [\bar{y}_s^k]+1$

Figure 9: Branching of nodes

While the above branching process can only enumerate integer solutions, if we have a means of guaranteeing that a sub-problem cannot have a better solution than the already obtained integer solutions in terms of the optimum value of the objective function, then we can skip these sub-problems and need only search for the remaining nodes. For mixed-integer linear problems we can use the solutions for continuously relaxed problems as a criterion for pruning. Continuously relaxed problems always have a better optimum value for the objective function than the original integer problems. Sub-problems whose continuously relaxed problems have no better optimum than the already obtained integer solution cannot give a better optimum value, hence it becomes unnecessary to search further (bounding procedure).

Branch-and-Bound methods repeat branching and bounding in this way to obtain the final optimum. These sub-problems obtained through the branching process denote search nodes.

Sequential algorithm

Step 0 Initial setting
Let $\boldsymbol{ILP}_0$ mean the original problem $\boldsymbol{ILP}$, and $\mathcal{N}$ mean the set of search nodes. Set $\mathcal{N}$ to $\{\boldsymbol{ILP}_0\}$, and solve a continuously relaxed problem $\boldsymbol{LP}_0$. If an integer solution is obtained go to Step5. Otherwise set the incumbent solution $\hat{z}$ to ∞ and go to Step1.

Step 1 Selecting branching node
If $\mathcal{N} = \emptyset$, then go to Step5.

If $\mathcal{N} \neq \emptyset$, then select the next branching node $\boldsymbol{ILP}_k$ out of $\mathcal{N}$ following the heuristics, and go to Step2.

Step 2 Selecting branching variable and branch
Select the integer variable y_s to be used for the branching process to work on $\boldsymbol{ILP}_k$ according to the heuristics, and branch with respect to it. Let the resulting two nodes be $\boldsymbol{ILP}_{k'}$, $\boldsymbol{ILP}_{k''}$
Go to Step3.

Step 3 Continuously relax two nodes
Solve two continuously relaxed problems $\boldsymbol{LP}_{k'}$ and $\boldsymbol{LP}_{k''}$ by the simplex algorithm. Go to Step4.

Step 4 Fathom two children nodes
If relaxed problem $\boldsymbol{LP}_{k'}$ does not have a solution, or gives a solution $\bar{z}_{k'}$ that is no better than the incumbent solution, in other words $\bar{z}_{k'} > \hat{z}$, then stop searching (bounding operation).

If the point $(\bar{x}^{k'}, \bar{y}^{k'})$ to achieve a solution $\bar{z}_{k'}$ has integer value $\bar{y}$ and moreover gives a better solution than the incumbent solution obtained so far, in other words $\bar{z}_{k'} < \hat{z}$, then let $\hat{z} = \bar{z}_{k'}$, $\hat{x} = \bar{x}^{k'}$ and $\hat{y} = \bar{y}^{k'}$ (revision of the incumbent).

If $(\bar{x}^{k'}, y^{k'})$ is not an integer solution and gives a better optimum value than the incumbent, then add this node, $\mathcal{N} := \mathcal{N} \cup \{ILP_{k'}\}$ (Addition of a node).

Do the same thing to $\boldsymbol{ILP}_{k''}$, and go to Step1.

Step 5 End step

If $\hat{z} \neq \infty$, then let the incumbent $(\hat{x}, \hat{y})$ be the optimum solution.

If $\hat{z} = \infty$, then problem $\boldsymbol{ILP}$ has no solution.

3.3.3 Heuristics for Branching

The following two factors determine the schedule of the order in which the sequential search process goes through the nodes in the search space:

1. The priorities of sub-problems(nodes) to decide the next node on which the branching process operates.
2. Selection of a variable out of the integer variables with which the search space is divided.

It is preferable that the above selections are done in such a way that the actual nodes, searched in the process of finding the optimal, form as small a part as possible within the total search space. We adopted one of the best heuristics of this type from operations research as a basis of our parallel algorithm([Benichou *et al.* 71]).

Selection of sub-problems

We use a combination of depth-first strategy and best-first strategy(w.r.t. heuristic function). In each branching process, what is called the pseudo-costs $p_{up}(j)$, $p_{down}(j)$ of integer variables y_j are computed. These are the increase ratios of the optimum value of the continuously relaxed problem with regard to those integer variables. In the next heuristic function $h(ILP_k)$ of the node is calculated:
$h(ILP_k) = \hat{z}_k + \sum_{j=1}^{n_2} \min\{p_{up}(j)(1-f_j), p_{down}(j)f_j\}$,
$f_j = \bar{y}_j^k - [\bar{y}_j^k]$,
Suppose the node $\boldsymbol{ILP}_k$ is divided into $\boldsymbol{ILP}_{k'}$ and $\boldsymbol{ILP}_{k''}$.

n-i. When at least one of these two nodes is not yet terminated, select the one having a better(i.e., smaller) heuristic value $h(ILP)$ as the next branching node (depth-first).

n-ii. When both have terminated,

a. if no incumbent solution has yet been found, select the latest node to which branching has been done (depth-first).

b. if an incumbent solution has already been found, select the node having the best heuristic function value (best-first).

Selection of the branching variable

To select the branching variable when trying to branch at the node $\boldsymbol{ILP}_k$.

v-i. If no incumbent solution is found, select the variable y_j^k from those integer variables that do not take exact integer values in $(\bar{x}^k, \bar{y}^k)$, and which gives the greatest difference between the two increases in the heuristic value, namely the one to attain
$\max_j\{|p_{up}(j)(1-f_j) - p_{down}(j)f_j| : f_j \text{non-integer}\}$

v-ii. If an incumbent solution is found, select the variable y_j^k out of those integer variables that do not take exact integer values in $(\bar{x}^k, \bar{y}^k)$, and which gives the maximum of the minimum value of the left and right side heuristic values, namely that to attain
$\max_j\{\min\{p_{up}(j)(1-f_j), p_{down}(j)f_j : f_j \text{non-integer}\}$

3.3.4 Parallel Branch-and-Bound Method

The parallel algorithm derived from the above sequential algorithm is implemented on Multi-PSI. Our parallel algorithm exploits the independence of many sub-processes created through branching in the sequential algorithm, distributing these processes to different processors. What is necessary here is that the search space is divided as evenly as possible among processors to achieve good load balance, and that the pruning operation is performed by all the processors simultaneously. Also, incumbent solutions found in each processor need to be communicated between processors. The details of the parallel algorithm is described in the following.

Load balancing

One parent processor works on the sequential algorithm up to a certain depth d of the search tree. It then creates 2^d child nodes and distributes them to other processors as shown in Figure 10. These search nodes are allocated to different processors cyclically, where each of the processors works on these sub-problems sequentially. Therefore, load balancing is static in this case.

Distribution is done only at a certain depth of the search tree, to prevent the granularity of a node from being too small and to decrease the communication costs.

Heuristics for pruning

Each processor has a share of a certain number of subproblems assigned, and works on these nodes with the same heuristics of branching node selection and branching variable selection as those of the sequential case. For the node selection heuristics, we use the priority control facility of KL1, to assign priorities to the search nodes on which the best-first strategy with the heuristic function can depend. (See [Oki *et al.* 89] for details of this technique.)

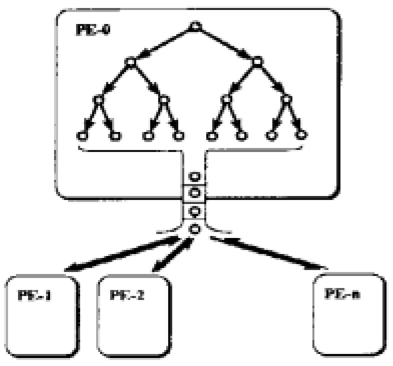

Figure 10: Generation of parallel processes

Transfer of global data
While the search space is distributed among different processors, if the information to prune nodes is not communicated well among them, then the processor has to work on unnecessary nodes, and the overall work becomes larger compared with the sequential version. This causes a reduction in the computation speed.

Therefore, incumbent solutions are transferred between processors to be shared so that each processor can update the current incumbent solution as soon as possible (Figure 11). This is realized by assigning a higher priority to the goal responsible for data transfer in the program.

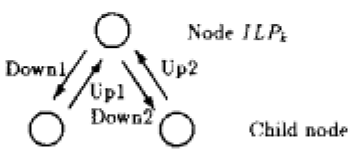

Figure 11: Report stream between nodes

3.3.5 Experimental Results

We implemented the above parallel algorithm in KL1, and experimented with job-shop scheduling problem. Table 6 shows a result of computation speedups for a 4job-3machine problem and the total number of searched nodes to get to the solution.

The situation often occurs where a processor visits an unnecessary node before the processor receives pruning information. This is because communication takes a time, and certainly cannot be instantaneous, in a distributed memory machine. Table 6 shows a case where this actually happens.

Table 6: Speedup

Processors	1	2	4	8
Speedup	1.0	1.5	1.9	2.3
Number of nodes	242	248	395	490

One of the problems in parallel search algorithms is how to decrease the growth of the size of the total search space compared with the sequential search algorithms.

4 GDCC Program Examples

Example 1 : integer programming

The following program is a simplified version of the integer programming used to find the integer solution that gives the minimum (or maximum) value of an objective function under given constraints. This program shows the basic structure of the Branch-and-Bound method.

```
:- module pseudo_integer_programming.
:- public integer_pro/3.

integer_pro(X,Y,Z):- true |
    call((simplex#X>=5,
          simplex#X+2*Y>=-3,
          simplex#X+Y-Z<=5)) initial nil giving Co,
    take_min(Co).

take_min(Co):- true |
    call(simplex#min(X+Y,Ans)) initial Co giving Co1,
    (Ans={minusinfinite,_} -> error;
     otherwise;
     Ans={_,[X=ValX|_]}     -> check(ValX,Co)).

check(ValX,Co):- kl1!integer(ValX) |
    solve_another_variables(Co).
otherwise.
check(ValX,Co):- true |
    floor(ValX,SupX,InfX),
    call(simplex#X=<InfX) initial Co giving Co1,
    take_min(Co1),
    call(simplex#X>=SupX) initial Co giving Co2,
    take_min(Co2).
```

The block in the clause **integer_pro** solves a set of constraints. The block in the clause **take_min** finds the minimum value of the given objective function. If the minimum value exists (not $-\infty$), **check** is called. In clause **check**, if the value of X, that gives the minimum value of the objective function is not an integer, two new constraints are added in order to the X become integer (for instance, if $X = 3.4$ then $X >= 4$ and $X <= 3$), and the minimum values with respect to the new constraints are solved again. Method **kl1!integer** decides whether the value X is an integer. Where, **kl1!** indicates KL1 method calling, a KL1 method is called from the GDCC program using this notation.

Synchronization between the inference engine and the solver to get the minimum value is achieved by the blocks in `integer_pro` and `take_min`. Multiple contexts are shown by the two blocks of `check`.

Example 2 : geometric problem

Next, we show how to use a function to find the approximated roots of uni-variate equations and how to handle multiple contexts using an example which is also used in [Aiba *et al.* 88].

```
:- module heron.
:- public tri/4, test1/4, test2/4.

tri(A,B,C,S) :- true |
    alloc(10,CA,CB,H),
    alg#C=CA+CB,
    alg#CA**2+H**2=A**2,
    alg#CB**2+H**2=B**2,
    alg#H*C=2*S.

test1(A,B,C,S) :- true |
    call( tri(A,B,C,S) ) initial nil giving GB,
    output1(GB).   % output to a window screen

test2(A,B,C,S) :- true |
    call( tri(A,B,C,S) ) initial nil giving GB,
    Err= 1/100000000,
    kl1!find:find(GB,Err,1,SubGB,UniEqs,UniSols),
    kl1!find:sol(SubGB,UniSols,Err,1,FGB),
    check(FGB, S).

check([], _) :- true | true.
check([FGB|FGBs], S) :- true |
    call( check_ask(S,Ans) ) initial FGB giving Sol,
    check_sub(Ans, Sol, FGBs, S).

check_sub(true, Sol, FGBs, S) :- true |
    output(Sol),  %  output to a window screen
    check(FGBs, S).
check_sub(false, _, FGBs, S) :-  check(FGBs, S).

check_ask(S, Ans) :- alg#S > 0 | Ans = true.
check_ask(S, Ans) :- alg#S =< 0 | Ans = false.
```

Figure 12 shows the meaning of the constraints set contained in clause `tri`, where `**` in equations indicates a power operation. `CA,CB,H` are local variables. `A`, `B`, `C` represents the three edges of a triangle, and `S` is its area. `alloc`(*Pre*, *Var1*,..., *VarN*) is a declaration to give precedence *Pre* to variables *Var1*,..., *VarN*. A monomial including a variable that has the highest `Pre` is the highest monomial, that is the precedence of variables is stronger than the degree in comparison.

If the goal,

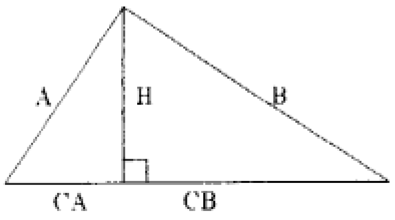

Figure 12: A triangle and its parameters

```
?- alloc(0,A,B,C),alloc(5,S),
   heron:test1(A,B,C,S).
```

is given, in which all parameters are free, this program outputs a Gröbner basis consisting of seven rules. Among them is the following rule that contains only `A`, `B`, `C` and `S`.

```
S**2= -1/16*C**4+1/8*C**2*B**2+1/8*C**2*A**2
      -1/16*B**4+1/8*B**2*A**2-1/16*A**4.
```

This is equivalent to Heron's formula. Of course, this program can be executed by a goal with concrete parameters. For example, when the goal,

```
?- alloc(5,S), heron:test1(3,4,5,S).
```

is given, the program produces `S**2= 36`.

However, the Buchberger algorithm cannot extract discrete values from this equation, as shown in section 3.1.2. Method `test2` approximates the real roots from a Gröbner basis, if the basis contains uni-variate equations. If the goal

```
?- alloc(5,S), heron:test2(3,4,5,S)
```

is given, first the constraint set is solved to obtain Gröbner basis `GB` using the `call` predicate, then uni-variate equations are extracted from `GB` using the method `find`:

```
kl1!find:find(GB,Err,1,SubGB,UniEqs,UniSols).
```

Where, `UniSols` contains the all combinations of solutions with precision `Err`. `UniEqs` is a set of the uni-variate equations extracted from Gröbner basis `GB`, and `SubGB` is the basis remaining after removing the uni-variate equations. The next method `sol` obtains a new Gröbner basis `FGB` by asserting the combinations of approximated solutions `UniSols` into `SubGB`. It is necessary to modify the Buchberger algorithm to handle approximated solutions, as explained in [Aiba *et al.* 91]. `FGB` contains plural Gröbner bases in list format, and these bases are filtered by the method `check`, which checks whether $S > 0$ is satisfied at the guard of the sub-block `check_ask`.

5 Conclusion

GDCC is an instance of the cc language and satisfies two levels of parallelism: the execution of an inference engine and solvers in parallel, and the execution of a solver in parallel. A characteristic of a cc language is that it is more declarative than sequential CLP languages, since the guard part is the only synchronization point between an inference engine and solvers. GDCC inherits this characteristic and, moreover, it has a block mechanism to synchronize meta-operations with constraints.

In the latest (master-slave) version of the parallel algebraic solver, the parallel execution of "cyclic 6-roots" with 16 processors is 4.42 times faster than execution with a single processor. With the boolean solver, parallel execution of the 6 queens problem with 16 processor is 2.28 times faster than with a single processor. We also show the realization of fast parallel search for mixed integer programming using the Branch-and-Bound algorithm.

The following items are yet to be studied. As shown in the program examples, current users must describe everything explicitly to handle multiple contexts. Thus, support faculties and utilities to handle multiple contexts are required. We will also improve the parallel constraint solvers to obtain both good absolute performance and better parallel speedup. The algebraic solver requires parallel speedup. The boolean solver needs to increase the parallel executable parts of its algorithm. The linear integer solver has to improve the ratio of pruning in parallel execution. Through these refinements and experiments using the handling robot design system, we can realize a parallel CLP language system that has high functionality in both its language facilities and performance.

6 Acknowledgments

We would like to thank Professor Makoto Amamiya at Kyushu University and the members of the CLP working group for their discussions and suggestions. We would also like to thank Dr. Fuchi, Director of the ICOT Research Center, and Dr. Hasegawa, Chief of the Fourth and Fifth Laboratory, for their encouragement and support in this work.

References

[Aiba *et al.* 88] A. Aiba, K. Sakai, Y. Sato, D. Hawley and R. Hasegawa. Constraint Logic Programming Language CAL. In *International Conference on Fifth Generation Computer Systems*, pages 263-276, 1988.

[Aiba *et al.* 91] A. Aiba, S. Sato, S. Terasaki, M. Sakata and K. Machida. CAL: A Constraint Logic Programming Language – Its Enhancement for Application to Handling Robots –. Technical Report TR-729, Institute for New Generation Computer Technology, 1991.

[Backelin and Fröberg 91] J. Backelin and R. Fröberg. How we proved that there are exactly 924 cyclic 7-roots. In S. M. Watt, editor, *Proc. ISSAC'91* pages 103–111. ACM, July 1991.

[Benichou *et al.* 71] M. Benichou, L. M. Gauthier, P. Girodet, G. Hentges, G. Ribiere and O. Vincent. Experiments in Mixed-Integer Linear Programming. In *Mathematical Programming* 1 pages 76–94, 1971.

[Boege *et al.* 86] W. Boege, R. Gebauer and H. Kredel. Some Examples for Solving Systems of Algebraic Equations by Calculating Groebner Bases. *Symbolic Computation*, 2(1):83–98, 1986.

[Buchberger 83] B. Buchberger. Gröbner bases:An Algorithmic Method in Polynomial Ideal Theory. Technical report, CAMP-LINZ, 1983.

[Buchberger 85] B. Buchberger. Gröbner bases:An Algorithmic Method in Polynomial Ideal Theory. In N. K. Bose, editor, *Multidimensional Systems Theory*, pages 184–232. D. Reidel Publishing Company, 1985.

[Cangilia *et al.* 88] L. Caniglia, A. Galligo and J. 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.

[Chikayama *et al.* 88] T. Chikayama, H. Sato and T. Miyazaki. Overview of Parallel Inference Machine Operationg System (PIMOS). In *International Conference on Fifth Generation Computer Systems*, pages 230–251, 1988.

[Clarke *et al.* 90] 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, October 1990.

[Colmerauer 87] A. Colmerauer. Opening the Prolog III Universe: A new generation of Prolog promises some powerful capabilities. *BYTE*, pages 177–182, August 1987.

[Dincbas *et al.* 88] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf and F. Bertheir. The Constraint Logic Programming Language CHIP. In *International Conference on Fifth Generation Computer Systems*, pages 693–702, 1988.

[Gebauer and Möller 88] R. Gebauer and H. M. Möller. On an installation of Buchberger's algorithm. *Symbolic Computation*. 6:275–286. 1988.

[Hawley 91a] D. J. Hawley. A Buchberger Algorithm for Distributed Memory Multi-Processors. In *The first International Conference of the Austrian Center for Parallel Computation*, Salzburg, September, 1991. Also in Technical Report TR-677 Institute for New Generation Computer Technology, 1991.

[Hawley 91b] D. J. Hawley. The Concurrent Constraint Language GDCC and Its Parallel Constraint Solver. Technical Report TR-713 Institute for New Generation Computer Technology, 1991.

[Hentenryck 89] P. Van Hentenryck. Parallel Constraint Satisfaction in Logic Programming: Prelimiary Results of CHIP within PEPSys. In *6th International Conference on Logic Programming*, pages 165–180, 1989.

[Jaffar and Lassez 87] J. Jaffar and J-L. Lassez. Constraint Logic Programming. In *4th IEEE Symposium on Logic Programming*, 1987.

[Li 86] G.-J. Li and W. W. Benjamin. Coping with Anomalies in Parallel Branch-and-Bound Algorithms. *IEEE Trans. on Computers*, 35(6): 568–573, June 1986.

[Maher 87] M. 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.

[Oki *et al.* 89] H. Oki, K. Taki, S. Sei, and M. Furuichi. Implementation and evaluation of parallel Tsumego program on the Multi-PSI. In *Proceedings of the Joint Parallel Processing Symposium (JSSP'89)*, pages 351–357. 1989. (In Japanese).

[Ponder 90] 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.

[Quinn 90] M. J. Quinn. Analysis and Implementation of Branch-and-Bound Algorithms on a Hypercube Multiprocessor. *IEEE Trans. on Computers*, 39(3):384–387, March 1990.

[Sakai and Aiba 89] K. Sakai and A. Aiba. CAL: A Theoritical Background of Constraint Logic Programming and its Applications. *Symbolic Computation*, 8(6):589–603, 1989.

[Saraswat 89] V. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Carnegie-Mellon University, Computer Science Department, January 1989.

[K. Satoh and Aiba 90] K. Satoh and A. Aiba. Hierarchical Constraint Logic Language: CHAL. Technical Report TR-592, Institute for New Generation Computer Technology, 1990.

[K. Satoh 90b] K. Satoh. Computing Soft Constraints by Hierarchical Constraint Logic Programming. Technical Report TR-610, Institute for New Generation Computer Technology, 1990.

[S. Sato and Aiba 90] S. Sato and A. Aiba. An Application of CAL to Robotics. Technical Memorandum TM–1032, Institute for New Generation Computer Technology, 1990.

[Y. Sato and Sakai 88] Y. Sato and K. Sakai. Boolean Gröbner Base. February 1988. LA-Symposium in winter, RIMS, Kyoto University.

[Senechaud 90] 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.

[Siegl 90] K. Siegl. Gröbner Bases Computation in STRAND: A Case Study for Concurrent Symbolic Computation in Logic Programming Languages. Master's thesis, CAMP-LINZ, November 1990.

[Takeda *et al.* 88] Y. Takeda, H. Nakashima, K. Masuda, T. Chikayama and K. Taki. A load balancing mechanism for large scale multiprocessor systems and its implementation. In *International Conference on Fifth Generation Computer Systems*, pages 978–986, 1988.

[Ueda and Chikayama 90] K. Ueda and T. Chikayama. Design of the Kernel Language for the Parallel Inference Machine. *Computer Journal*, 33(6):494–500, December 1990.

[Vidal 90] 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, August 1990.