UNIVERSITE D'ORLEANS
Faculté des Sciences
LIFO

Laboratoire d'Informatique Fondamentale d'Orléans
4, rue Léonard de Vinci, BP 6759
F-45067 Orléans Cedex 2
FRANCE

Rapport de Recherche

www : http://www.univ-orleans.fr/LIFO/

A System for the High-level Parallelization and Cooperation of Constraint Solvers

Laurent GRANVILLIERS†, Gaétan HAINS†,
Quentin MILLER‡, Nicolas ROMERO†

† LIFO, Université d'Orléans
BP 6759, 45067 Orléans Cedex 2, FRANCE
‡ Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD, U.K.

Rapport N^o **98-07**

Abstract

A general system, Bs-Solve, is presented for the high-level programming and reuse of cooperative and parallel constraint solvers. The parallelization of two common algorithms is outlined : Groebner-basis computation for polynomial systems and Waltz consistency algorithm for continuous domains.

Keywords

Constraint solvers, cooperation, parallel strategies.

Résumé

Le système Bs-Solve permet l'écriture de stratégies parallèles de coopération de résolveurs de contraintes. La programmation déclarative des stratégies et la possibilité de réutilisation des résolveurs existants sont illustrées par l'implantation de deux algorithmes classiques : le calcul de bases de Groebner d'ensembles de polynômes et l'algorithme de consistance de Waltz sur des domaines continus.

Mots-clés

Résolveurs de contraintes, coopération, stratégies parallèles.

Work supported by a grant from AITEC of Japan: project CCOPS at the Universities of Nantes, Orléans, Oxford and INRIA Rocquencourt. The authors thank the project manager of the CCOPS project, Frédéric Benhamou, and the other collaborators, Philippe Codognet and Arnaud Lallouet

1 Introduction

Constraint programming emerges as a widely used technology in industrial or engineering applications involving optimization or constraint solving issues. In particular, various problems from robotics, physics or economics can be expressed as numerical constraints. Several constraint programming languages and systems addressing numerical constraints are based on combinations and/or cooperations of different methods. Among others, one can cite `Prolog IV` [8], `CLP(BNR)` [5], `Newton` [4] and `Numerica` [13]. The most significant motivation for combining different constraint solvers is to solve problems that none of the solvers can handle separately. By doing so, solvers can either act independently and dynamically share their results or some solvers can achieve some preprocessing for the others.

The intensive computations generated by solvers make them an ideal application for parallel computing. But parallelization of solvers raises the problems usually associated with parallel symbolic computing, dynamic data distribution and balancing. Moreover, a new and specific problem appears in this context, related to a loss of information when narrowing operators are applied in parallel.

To investigate these issues, we have designed a general system for the high-level programming of cooperative and parallel solvers. It realizes more than a purely declarative interface to the parallel architecture: it lets the programmer write customized strategies for parallel and cooperative computations.

In this paper, we describe the mathematical background, high-level software specification, architecture and software functionality of the system. We then describe the parallelization of two common constraint-solving algorithms: Gröbner-basis computation and Waltz's algorithm. The sequential algorithms are known to allow a trivial but effective form of interaction (pre-processing) and we apply this scheme to our parallel versions.

Preliminary execution tests highlight the major problems facing parallel constraint solvers: methods which are already data-dependent in the sequential case become even more unstable and converge slower when parallelized.

2 Cooperative Constraint Solvers over $\mathbb{R}$

Solvers of numerical constraints can be divided in two categories. Symbolic solvers manipulating for example Gröbner bases [6] transform the constraint expressions. Numerical solvers contract the variable domains using techniques from Numerical Analysis or Artificial Intelligence. The aim of this section is to summarize a framework inspired of [1] where cooperative solvers implementing symbolic and numerical operations can be described as repeated applications of closure operators on constraints and domains.

2.1 Preliminaries

Let Σ be a structure $< \mathbb{R}, \mathcal{O}, \mathcal{R} >$ where $\mathbb{R}$ is the set of real numbers, $\mathcal{O}$ a set of function symbols and $\mathcal{R}$ a set of relation symbols. Let $\mathcal{V}$ be a set of variables taking their values over $\mathbb{R}$. This set is assumed to be large enough to build the constraints described here. *Terms* in the constraint language are syntactic expressions built in the usual way from constants, variables and operations. *Constraints* are syntactic expressions representing relations made from terms and relation symbols. The set of constraints made on Σ is denoted $\mathcal{C}_\Sigma$. $\mathcal{P}(\mathbb{R})$ denotes the power set of $\mathbb{R}$.

Definition 1 (Constraint System) *A* constraint system *is a 3-tuple* $\varepsilon =< \mathcal{C}, \mathcal{X}, \mathcal{D} >$ *made of a set of variables* $\mathcal{X} = \{x_1, \ldots, x_n\} \subset \mathcal{V}$*, a vector of domains* $\mathcal{D} = (D_1, \ldots, D_n)$ *where each* D_i *is an element of* $\mathcal{P}(\mathbb{R})$ *and represents the possible values for* x_i*, and a set of constraints* $\mathcal{C} = \{c_1, \ldots, c_m\}$ *over* $\mathcal{X}$*.*
In the following, the set of variables will be omitted since it is implicit.

The *search space* of ε is the Cartesian product $D_1 \times \cdots \times D_n$ of variable domains. The *solution space*, denoted $\varepsilon^\star$, is composed of all the tuples of $\mathbb{R}^n$ included in the search space and satisfying all the constraints. Two constraint systems are said to be *equivalent* if their solution spaces are equal.

The limited precision of machine arithmetic generally prevents the representation of the exact solution space of a constraint system. A remedy is to compute approximations of solutions over an *approximate domain.*

Definition 2 (Approximate domain) *An* approximate domain A *over* $\mathbb{R}$ *is a subset of* $\mathcal{P}(\mathbb{R})$*, closed under intersection, such that* $\mathbb{R} \in \mathrm{A}$ *and for which the inclusion is a well-founded ordering.*

Every approximate domain is a complete lattice for the set inclusion ordering denoted $\subseteq$. The set of constraint systems built from Σ with domains in A is denoted $\mathcal{E}_{(\Sigma,A)}$. A well known example of approximate domain using in a wide variety of solvers is the set of intervals whose bounds are floating point numbers.

2.2 Abstract Constraint Solving

A cooperative solver is a collection of operations for transforming constraint systems together with a cooperation strategy to apply them. The operations are closure operators, called *constraint narrowing operators (CN operators)*, either contracting domains or rewriting constraints according to algebraic laws. Intuitively, three main problems have to be handled:

- *Correctness* of computations, in the sense that the solution space of constraint systems is included in the variable domains. The correctness of each CN operator is a sufficient condition.
- *Termination* is addressed by the definition of a well-founded ordering $\preceq$ on sets of constraints and the ordering $\sqsubseteq$ on constraint systems as:
  $< \mathcal{C}', \mathcal{D}' > \sqsubseteq < \mathcal{C}, \mathcal{D} >$ if $\mathcal{D}' \subseteq \mathcal{D}$ and $C' \preceq \mathcal{C}$. As a consequence, $\sqsubseteq$ is well-founded.
- *Confluence* generally depends on the cooperation strategy.

Definition 3 (CN Operator) *A* CN operator *over* A *is a mapping*
$N : \mathcal{E}_{(\Sigma,A)} \rightarrow \mathcal{E}_{(\Sigma,A)}$ *such that for all* $\varepsilon, \varepsilon' \in \mathcal{E}_{(\Sigma,A)}$*, the following properties hold:*

$$\begin{array}{ll} N(\varepsilon) \sqsubseteq \varepsilon & (Contractance), \\ N(\varepsilon)^{\star} = \varepsilon^{\star} & (Correctness), \\ \varepsilon' \sqsubseteq \varepsilon \text{ implies } N(\varepsilon') \sqsubseteq N(\varepsilon) & (Monotonicity). \end{array}$$

Since a composition of CN operators is still a CN operator, there are generally several granularity levels to describe a constraint solver. For example, a solver in the *black-box model* is defined by one CN operator which hides its internal mechanisms while it corresponds to several CN operators interacting together in the *glass-box model* (see [11] for more details). This is an important property for parallel applications where "parts" of constraint solvers can be distributed.

Finally, the cooperation of solvers over different approximate domains is handled by defining a new approximate domain made of the union of the previous ones. For example, rationals and intervals have to be merged when implementing Gröbner bases and interval tools in a same cooperative system.

We will now describe the BSP parallel programming model and the design of a high-level system for parallelizing and combining constraint solvers in BSP mode.

3 The BSP model

Bulk Synchronous Programming or BSP is a parallel programming model introduced by Valiant [12] to offer a high degree of abstraction like PRAM models and yet allow portable and predictable performance on a wide variety of architectures. It assumes as execution model a *BSP computer* consisting of a set of processor-memory pairs, a communication network allowing inter-processor delivery of messages and a global synchronization unit which executes collective requests for a synchronization barrier. Its performance is characterized by 3 parameters (expressed as multiples of the local processing speed): the number of processor-memory pairs p, the time l required for a global synchronization and the time g for (collectively) delivering a 1-relation (communication phase where every processor receives/sends at most one word). The network can deliver an h-relation in time gh for any arity h.

A BSP program is executed as a sequence of *supersteps*, each one divided into (at most) three phases. In the first phase each processor uses its local data (only) to perform sequential computations and to request data transfers to/from other nodes. In the second phase the network delivers the requested data transfers and in the third phase a global synchronization barrier occurs, making the transferred data available for the next superstep. The execution time of a superstep s is thus the sum of the maximal local processing time, of the data delivery time and of the global synchronization time:

$$Time(s) = \max_{0 \leq i < p} W_i^{(s)} + \max_{0 \leq i < p} h_i^{(s)} * g + l$$

where $W_i^{(s)}$ = local processing time on processor i, $h_i^{(s)}$ = the number of words transmitted/received by processor i. The execution time of a BSP program composed of S supersteps is therefore a sum $W + hg + Sl$ where W, h and S are functions of p and of the size of data n. To minimize execution time the BSP algorithm designer must jointly minimize the number S of supersteps and the total volume h (resp. W) of step-wise unbalance $h^{(s)} = \max_i h_i^{(s)}$ (resp. $W^{(s)} = \max_i W_i^{(s)}$) in communication (resp. in local computation).

In summary, BSP programs use global synchronizations, asynchronous local computations, non-blocking communications and have an explicit portable execution cost function.

3.1 BSP programming and subset synchronization

It is known [?] that BSP can be used as a machine-independent framework for the design and implementation of parallel algorithms while retaining control of their actual execution costs. The current proposal for a standard library [?] is based on 18 core operations callable from C or FORTRAN programs. Its application to regular and dense problems is now well understood, but the importance of irregular or sparse data and data-dependent algorithms would seem to imply that global synchronization barriers pose intrinsic limitations on the expressive power of the model. The requirement for global barriers apparently makes BSP programs ineffective in the presence of data skew.

This observation has led some researchers to study notions of nested BSP programs and/or synchronizations for subsets of the network, so-called *subset synchronizations*. Yet, faced with this possibility the authors of the proposed standard [?] have chosen to exclude subset synchronization from BSP libraries, for it would create serious semantic problems while making execution costs less predictable [?]. Accordingly, our BS-Solve system assumes the standard "flat" BSP execution mode and has been built on top of BSP-Lib.

4 Building Parallel and Cooperative Solvers

We now describe the design of a system for the explicit cooperation and parallelisation of constraint solvers. It uses the BSP (bulk-synchronous processing) paradigm of parallel programming: computation alternates between p independent computations and global operations, where p is the number of available processors. The resulting system called BS-Solve is available as a library where existing sequential solvers can be re-used.

4.1 High-level design

4.1.1 Programming model

The BSP realisation of of concurrent-cooperating parallel solver uses the following correspondences.

1. Resolution strategies are written as computation phases of p independent sequential computations. Each one is driven by an *agent*. An agent is either an existing solver or a program issuing calls to the BS-Solve system.

2. Cooperation strategies are written as so-called *contractions* and *expansions*, alternating with computation phases. These operations trigger all the communications and synchronisations in global (bulk-synchronous) mode. In other words, no subset of the network may communicate or synchronise without global synchronisation.

In this execution model, the parallel implementation of a solver is simply the explicit cooperation of its agents. The model thus smoothly integrates solver cooperation and parallelism: cooperation is realized by control parallelism when different agents are used in the same supersteps; data parallelism occurs when the same agent is used everywhere. Moreover, agents may change from one superstep to the next.

4.1.2 Variables, values, constraints

Agents coordinate the effect of operators which transform the basic objects of constraint solving: variables, their domains (or *values*) and constraints relating them.

The following definitions are needed to describe BS-Solve structures and operations.

Loc is the set of processor names (network *locations* for objects, hence the name Loc), a finite set of distinct names of cardinality p, usually $\{0, \ldots, p-1\}$.

X is the set of possible variable names. There are no scope rules and then variables are global. Each variable is stored at one node and a temporary local copy is created if it is used by another node. The coherence of variable's values is enforced by contraction operations which centralize and unify the different copies of a given variable. In this context, unification amounts to intersecting their domains. The semantics of constraint systems thus ensures the correctness of parallel operator applications to multiple copies of a single variable, a major simplification by comparison with parallel logic programming systems.

Val is the set of possible values for variables. The available types are integers, rationals, floats, intervals.

C is the set of all possible constraints on variables. They are symbolic formulas relating variables values, usually equality and ordering relations.

4.1.3 High-level system design

Computation phases are specified as p independent *local* state transformations. The set of local states is defined as

$$\mathrm{LocState} = \mathcal{P}_{\mathrm{fin}}(C) \times (X \xrightarrow{\mathrm{fin}} \mathrm{Val})$$

where elements of $\mathcal{P}_{\mathrm{fin}}(C)$ are finite sets of constraints and the arrow $\xrightarrow{\mathrm{fin}}$ denotes partial finite functions. So a local state σ is composed of a set of constraints and an environment binding variables to *domains*.

Local state transformers are called *operators* and may correspond to pre-existing solvers, or to a user program written with the BS-Solve system. Their type is:

$$\mathrm{Op} = \mathrm{LocState} \to \mathrm{LocState}$$

An operator is thus a function and moreover a total function. Its proofs of termination and deterministic behaviour are left to the programmer. The operator may be written as a concurrent program but the BS-Solve system executes it on a single processor and assumes that it is deterministic and eventually stops. Under those conditions, the BSP execution is also guaranteed to stop and to be I/O-deterministic in the sense of yielding the same answer for the same input data and value of p, independent of the relative processor speeds.

Because of the possibility of interleaving, operators are themselves under control of *agents*. An agent realises a local resolution strategy

$$\mathrm{Agent} = \mathcal{P}_{\mathrm{fin}}(\mathrm{Op}) \to \mathrm{LocState} \to \mathrm{LocState}$$

which is equivalent to

$$\mathrm{Agent} = \mathcal{P}_{\mathrm{fin}}(\mathrm{Op}) \to \mathrm{Op}.$$

An agent is a generalised operator composition or control structure. It takes a set of operators and turns them into a local state transformer, usually by applying them in a specific sequence.

Once a computation phase is completed, it is followed by a *contraction* and then an *expansion* operation. The intermediate object generated by the contraction and input to the expansion is called a *global state*. The type of global states is:

$$\mathrm{GlobState} = \mathcal{P}_{\mathrm{fin}}(C \times \mathrm{Loc}) \times (X \xrightarrow{\mathrm{fin}} \mathrm{Val} \times \mathrm{Loc}) \times \mathcal{P}_{\mathrm{fin}}(\mathrm{Op} \times \mathrm{Loc})$$

A global state Σ contains a set of constraints with locations and a set of operators with locations. Both allow for repetitions: a single constraint or operator may be present at several locations. Σ also contains an environment binding variables with domains and locations. A variable is thus globally unique in value and location.

The work of merging local state variables and constraints into a global state is called a contraction.

$$\mathrm{Contraction} = (\mathrm{Loc} \to \mathrm{LocState}) \to \mathrm{GlobState}$$

Notice that it "forgets" about operators. To ensure the global coherence of variables, contraction operations update each variable's value by the intersection of values for all local references to it.

In the high-level model, operators are provided anew before the next phase. But in the implementation and user programs, operators need not be discarded. Similarly, is global state is only logically "centralized". It is in fact a coherent but distributed structure (as specified by its location components).

To complete data exchanges and resume the application of operators, an expansion operation is used. It generates from a global state a Loc-indexed vector of agents, operators and local states.

$$\mathrm{Expansion} = \mathrm{GlobState} \to (\mathrm{Loc} \to (\mathrm{Agent} \times \mathcal{P}_{\mathrm{fin}}(\mathrm{Op}) \times \mathrm{LocState}))$$

The next computation phase is then ready to begin by using each agent (possibly unchanged, or locally pre-processed) with its local operators to modify the associated local state.

4.2 Architecture

The *BS-Solve system* is a prototype based on the above high-level design.
Its architecture is organized into several components. The kernel and solving components form the basis of the system, whereas the strategies component implements the user-control of resolution mechanisms.

The user interaction with BS-Solve is very flexible, and depends on his intentions. All levels of interactions are permitted, from resolution-only with pre-existing solvers and strategies, to fine scheduling of the user's solvers with his own strategies using low-level primitives.

The system has been implemented in `C` and `KLIC`, a portable version of the `KL1` language [7]. `KL1` is a concurrent logic language developped at Institute for New Generation Computer Technology originally as part of the Fifth Generation Computer national project of Japan.

4.2.1 The kernel

The lowest level of the system is the kernel. it does not depend on the user, while both solving and strategies components are implemented by the user. It provides :

- *communication and synchronization mechanisms*, through Quentin Miller's BSP-KLIC library [10],
- *foreign-language (usually C) solvers integration facilities*,
- a *toolbox* to write strategies (agents).

4.2.2 The BSP-KLIC library

BS-Solve communication operations are implemented with the BSP-KLIC library, a restricted protocol calling the BSPLib library [9] for BSP programming.

The BSP-KLIC library is a module of KLIC predicates which implements BSP communication among a set of KLIC processes executing the same program. Exchange of datas can be performed either by message-passing or by direct memory-to-memory copying.

4.2.3 Solvers integration and cooperation

BS-Solve allows integration of external solvers through an uniform interface between the system and `C` or `KLIC` solving programs. Cooperation strategies between solvers are expressed using predefined data structures, ready-to-use predicates with system control and monitoring abilities, and low-level predicates to implement new resolution schemes. The two following sub-sections describe these features in more detail.

4.3 Connecting a solver

BS-Solve provides `C` and `KLIC` interface libraries and permits the use of pre-existing solvers, through standard domain representations and a large set of operations.

4.3.1 Domains

Domains can be manipulated as generic or typed values, whether the user knows the type or not. The generic form is dynamically typed and can evolve through the computation among a fixed set of types. Available types are integer, rational, float, infinity, and interval. Operations on values permit :

- allocation, deallocation, copy, instantiation of values,
- intersection between domains,
- basic arithmetical and numerical computations,
- comparison,
- printing,
- translation from and to fully-documented byte- and string-based representation, to make easier moving between processors and quick interfacing with external solvers.

Solvers can also have their own internal representations of domains, provided that they come with translating functions which permit to convert external representations into standard representations.

4.3.2 Variables and Constraints

Other basic structures are records for variables and constraints, allowing for external representations to coexist with the standard one. When a solver reduces a variable's value, it is responsible for setting the name of the most recent modifier to the name of its own representation. A later use of the value by an other solver or by the system itself will require the translation of this representation into BS-Solve's own. Such a mechanism is not needed for constraints since their representations never move.

The whole set of variables and constraints is partitioned among all processors, and solvers can locally manage a given structure with specialized functions.

4.3.3 Operators

Our description of solvers needs to be clarified. A solver can be defined as a set of solving mechanisms on domains and constraints, together with a resolution strategy. From the *black-box* point of view, a BS-Solve program embedding cooperative solving mechanism is a solver. However, this scheme does not allow to study cooperative strategies. We have preferred the *glass-box* approach, where the user can act on resolution and cooperation strategies. Therefore, we call solvers the actual implementation of CN operators. Their complexity determines the granularity of the system : one resolution process can be seen (and implemented) either as a single solver (as an opaque glass-box) or as the combination of several solvers in a same step of computation (transparent glass-box).

Operators are the remaining principal structures of the system. Their implementation consists of a piece of executable code written either in `KLIC` or `C` and references to variables and constraints whose values are needed locally for the operator's execution.

4.4 Cooperation between solvers

The aim of the BS-Solve system is to study resolution processes using concurrent and cooperating solvers. The control on execution is achieved by the writing of KLIC programs implementing resolution strategies.

As BS-Solve uses the BSP model, resolution is divided into sequential computations running concurrently on each node and global synchronizing barriers where exchanges of datas can be performed. During the expansion, operators to apply are selected among the existing ones, and copies of non-local values and constraints referenced are automatically retrieved. Thus, the user who wants to control all the aspects of a resolution can act on :

- distribution of datas, including initial positioning and dynamic rebalancing of operators,
- dynamic choice of operators to apply during one computation phase,
- cooperation between different solvers by interleaving of phases involving them.

As was said above, the BS-Solve representation can be seen as canonical : it is the common denominator of all numerical solvers, thus cooperation is mainly performed by standard intersection of computed values.

5 Application

We now present the parallelization in BS-Solve of two important algorithms: Buchberger's [6] and Waltz's [14] algorithms for computing Gröbner bases and solving constraint systems respectively. They were chosen for their effective interaction (observed in the setting of sequential computation [2]) and as representatives of symbolic- and interval-based algorithms. The BS-Solve system allows the user to parallelize and mix the two algorithms in complex ways through interleaving. Experience in the setting of sequential computing shows however that for the case of a Gröbner-base procedure and Waltz's algorithm, useful interaction occurs when (and apparently only when) the former is applied, once, before the latter. We have therefore tested this combination of the corresponding parallel algorithms.

5.1 Design of BS-Gröbner

Gröbner bases are an important tool borrowed from Computer Algebra which can be seen as a kind of Gaussian elimination for non-linear polynomial systems. Among other things, they can be used to simplify sets of polynomials before their solution with numerical techniques. We first present the (sequential) Buchberger's algorithm and then its BS-Solve parallel version.

5.1.1 Gröbner bases

The computation of Gröbner bases essentially depends on the following three notions:

1. Every polynomial is ordered using a *total ordering on monomials* $\succ$ which is well-founded and compatible with the multiplication on monomials. The maximal monomial of a polynomial q with respect to $\succ$ is called the *leading term* of q denoted $LT(q)$.
2. The *reduction process* of a polynomial q with respect to a list of polynomials Q computes an irreducible polynomial obtained as a remainder after iteratively dividing q by the elements of Q. The final polynomial, denoted $Reduce(q, Q)$, depends on the order of the elements in Q. This algorithm is the multivariate counterpart of the Euclid's algorithm for univariate polynomials.
3. The *S-polynomial* of two polynomials q and r, denoted $Spoly(q, r)$, is a combination of those polynomials that cancels their leading terms:

$$Spoly(q,r) = \frac{Lcm(Lt(q), Lt(r))}{Lt(q)} q - \frac{Lcm(Lt(q), Lt(r))}{Lt(r)} r$$

   where $Lcm(m, m')$ is the least common multiple of the monomials m, m'.

The algorithmic characterization of Gröbner bases proposed by Buchberger in [6] is the following: a set of polynomials $Q = \{q_1, \ldots, q_m\}$ is a Gröbner basis if and only if for all pairs $(i, j) \in \{1, \ldots, m\}$, $Reduce(Spoly(q_i, q_j), Q) = 0$. A basic algorithm consists in successive computations of reduced S-polynomials until no non-zero S-polynomial can be computed:

```
Gröbner ( in: Q = {q_1, ..., q_m} ; out: GB(Q) )
begin
  B ← {(q_i, q_j) | 1 ≤ i, j ≤ m, i ≠ j}           % initialization
  k ← m + 1
  GB(Q) ← Q
  while B ≠ ∅ do
    Remove (q_i, q_j) from B                        % selected pair not used anymore
    q_k ← Reduce(Spoly(q_i, q_j), GB(Q))            % make reduced S-polynomial
    if q_k ≠ 0 then
      B ← B ∪ {(q_k, q_h) | 1 ≤ h ≤ k − 1}         % create the new pairs
      GB(Q) ← GB(Q) ∪ {q_k}                         % add non-zero S-polynomial
      k ← k + 1                                     % to the Gröbner basis
    endif
  endwhile
end
```

The algorithm stops in finite time and the output is a Gröbner basis for the input set of polynomials Q.

5.1.2 Gröbner bases in BS-Solve

We describe now the BS-Solve implementation of the Gröbner algorithm in the *glass-box model*. It maintains the iterative structures B and $GB(Q)$ of the sequential algorithm but parallelizes the loop body in the following sense. Multiple S-polynomials are created in parallel, reduction is then processed by a collective operation before new pairs of polynomials are created in parallel. We now present the details of the Gröbner parallel algorithm.

5.2 Embedding Buchberger's algorithm in BS-Solve

The BS-Solve implementation of Gröbner bases maintains the iterative structure of the sequential algorithm but parallelizes the loop body in the following sense. Multiple S-polynomials are created in parallel, reduction is then processed by a collective operation before new pairs of polynomials are created in parallel. We now present the details of the BS-Gröbner parallel algorithm. The question of dynamic cooperation strategies with numerical constraint solvers will be treated later.

It is not realistic to duplicate dynamic data on any important subset of the processors because a quadratic blow-up of space would follow. It was chosen instead to distribute the data structures Gb and B on processors $0, \ldots, p-1$, creating thus the partitions $Gb(Q) = [Gb(Q)(0), \ldots, Gb(Q)(p-1)]$ and $B = [B(0), \ldots, B(p-1)]$. The algorithm is an iterative process of

1. local S-polynomial computations on processor j for all pairs in $B(j)$,
2. global reduction of S-polynomials since the *Reduce* operation needs the whole list $Gb(Q)$ of polynomials,
3. global creation of the new pairs to be added in B, composed of each new non-zero reduced S-polynomial and each pre-existing polynomial in $Gb(Q)$,

until B becomes empty. This strategy faces at least two types of inefficiencies:

- The second operation seems too costly as described above, since $Gb(Q)$ must be known on each processor and may be very large. The chosen strategy is to distribute the S-polynomials rather than $Gb(Q)$. The reduction process is then a sequence of local reduction using $Gb(Q)(j)$ on every processor j and communication of the partially reduced polynomials to other processors.

- The third operation creates the new list B to be processed by the next global iteration. This introduces the problem of dynamic balancing of B in order to minimize computation time for the next iteration.

We now describe in detail the strategies chosen to cope with both problems.

Reduction of S-polynomials

Let $SP(j)$ be the list of new S-polynomials computed by processor j during a given iteration. We describe the parallel reduction process of the S-polynomials in $SP(j)$. The list of reduced S-polynomials with respect to $Gb(Q)$ on processor j is denoted $FinalSP(j)$, initialized to the empty list. The following three steps are iterated until $SP(j)$ becomes empty for every j.

1. For all $s \in SP(j)$, send s to all processors. There are now p copies of s ($s(j)$ is the master copy) denoted $s(0), \ldots, s(p-1)$.
2. On each processor i, compute $s'(i) = Reduce(s(i), Gb(Q)(i))$ for every s just received.
3. For every $s \in SP(j)$, compute the parallel reduction
$$s' = min(s'(0), \ldots, s'(p-1))$$
with resulting value on processor j (This operation is realized as the concurrent execution of one *fold* operation for every s and is called a *multi-fold*). The minimum always exists since the deletion of terms in polynomials is an ordered process from the maximal terms. Then s is removed from $SP(j)$. If $s' = s$ then it is added to $FinalSP(j)$. If $s' = 0$ nothing is done. Otherwise s' is added in $SP(j)$ and the first step is executed again.

Balancing

The balancing of the partition of $Gb(Q)$ seems to be a necessary condition for an efficient parallel implementation, although it does not affect the correctness of the algorithm. Because the result of each new polynomial's reduction is unknown in advance, any global iteration may create some imbalance. Yet it is not clear whether the cost of re-balancing is always smaller than the speedup it provides in subsequent iterations.

Faced with this dilemma we have designed two versions of the algorithm: with and without dynamic balancing, so that performance tests may help to compare them. Moreover, the algorithm with balancing is driven by a threshold value so as to explore even further the tradeoff between more/less re-balancing. We now present the dynamic balancing algorithm.

Assuming that B is correctly balanced at the beginning of the algorithm, the re-balancing operation is a change in the partition
$$FinalSP = [FinalSP(0), \ldots, FinalSP(p-1)]$$
after each reduction phase. Perfect balancing is obtained by computing an additive fold [1] of the whole number m of polynomials in *FinalSP*, to broadcast the result to each processor and then to move for each processor j the additional polynomials if the cardinality of $FinalSP(j)$ is more than $MinSP = \lceil \frac{m}{p} \rceil$. However, optimal balancing may be too costly and it may be desired to have more flexibility. For this purpose, we offer the user a parameter $ExcessSP(j)$ such that $ExcessSP(j) + MinSP$ is the maximal number of polynomials which are allowed to stay on processor j.

The value of $ExcessSP(j)$ determines the cost and quality of balancing. A low value of $ExcessSP(j)$ implies a good but costly balancing, in particular optimal balancing is obtained when $ExcessSP(j) = 0$ for all $j \in \{0, \ldots, p-1\}$. On the contrary, a high value is a condition for moving fewer polynomials at the cost of a greater imbalance. The details of the balancing operation are as follows. For simplicity, let us first describe it when $ExcessSP(j) = 0$ everywhere.

- An additive *fold* operation computes the global sum m of the cardinality of the lists $FinalSP(i)$ over all processors i.

[1] parallel sum operation

- A *broadcast* sends the above value m to all processors.
- Each processor j computes its relative excess from the average load

$$\Delta_i = FinalSP(i) - \lceil \frac{m}{p} \rceil.$$

- A total exchange (every processor sends one value to every other) of the p values Δ_i gives every processor the complete histogram of excess loads.
- From this histogram, every processor holding positive excess computes a minimal set of messages it must send as follows. By scanning the histogram, it determines the global ranks of its excess polynomials within the set of all those is local excess. It then computes the numbers of the processors holding the corresponding ranks among negative Δ_i.
- Finally, the excess polynomials are sent to the their target processors in an exact global operation.

After re-balancing, for each new non-zero reduced S-polynomial s placed on processor j the following operations are performed.

- s is sent to all processors i and the pairs (s, p) for all $p \in Gb(Q)(i)$ are added to $B(i)$,
- s is added to $Gb(Q)(j)$.

5.3 BS-Waltz

The Waltz's algorithm is a consistency technique from Artificial Intelligence for handling constraint systems. Intuitively, it aims at contracting as much as possible the variable domains by removing some values which do not satisfy the constraints. Recently, Benhamou [1] described a Waltz's algorithm for solving constraints over intervals using CN operators which only modify the domains (not the constraints).

The idea is to define abstractions of solvers implementing consistency techniques by means of CN operators which modify domains of the variables appearing in one constraint. The algorithm then iterates application of a CN operator and propagation of the domains which have been modified to all the constraints sharing the corresponding variables, until no more reduction occurs. The input to Waltz is a constraint system ε and a set of couples (C_i, N_i) made of a CN operator N_i associated to a constraint C_i. The output is a new set of domains from which some values not satisfying the constraints have been discarded. In particular, if any output domain is empty then the initial constraint system has no solution.

The sequential algorithm is as follows:

```
Waltz ( inout : ε =< C, D > ; in {(c_1, N_1), ..., (c_m, N_m)} )
begin
   Queue ← {c_1, ..., c_m}
   while Queue ≠ ∅ and D ≠ ∅ do
      Select c_i from Queue          % choice phase
      < C, D' >← N_i(ε)              % contraction of domains
      if D' ≠ D then                 % propagation phase following dependencies
         Queue ← Queue ∪ {c_j | ∃x_k ∈ var(c_j) ∧ D'_k ≠ D_k}
         D ← D'
      endif
      Queue ← Queue \ {c_i}          % selected constraint not used anymore
   endwhile
end
```

and has the following properties:

1. If the computation of the CN operators terminates, the algorithm terminates due to the fact that set inclusion is a well-founded ordering for approximate domains.

2. The algorithm is correct in the sense that the solutions of the initial constraint system are included in the ouput domains. In particular, if this Cartesian product is empty, the constraint system is unsatisfiable.
3. The algorithm is confluent in the sense that the output is independent of the order in which the constraints are chosen.

The above properties lead to the following observations pertaining to parallel implementation.

- Confluence with respect to the order of application of the CN operators is essential for parallel algorithms. It allows the simultaneous application of several CN operators.
- The contraction of CN operators is generally obtained by intersection of the computed domain with the initial domain. A parallel algorithm applying several CN operators for reducing the same variable domain during a given iteration, must assign to this variable the intersection of all its new domains. The parallel implementation must therefore realize an intersection algorithm.
- The propagation phase adds in $Queue$ every CN operators depending on a variable whose domain has just changed. Intuitively, a CN operators depends on one variable if its associated constraint contains this variable. Now, if the constraints and CN operators are distributed over several processors, this phase needs to inform all the processors containing CN operators depending on variables when domains are contracted. This is realized as a global communication operation.

We describe now the BS-Solve implementation of the Waltz algorithm in the *glass-box model.*

5.4 Embedding Waltz's algorithm in BS-Solve

The parallel algorithm BS-Waltz is designed to maintain a distributed $Queue$ of active constraints while repeatedly selecting and applying in parallel a set of operators.

It proceeds in four steps, the last three being the body of the iteration to be repeated. The name of the steps correspond to the high-level model of BS-Solve computations defined earlier.

1. Initialization. The system $\{(C_i, N_i)\}_i$ is partitioned among the p processors and its layout will remain fixed throughout the computation. The initial distribution is defined by the user of BS-Waltz either by loading a pre-existing system from a file, or by applying redistribution operations. If the system is taken from a KLIC program, it is assumed to be entirely held by processor 0 until any redistribution is applied. The two redistribution operations are `move` and `spread`.
   - The effect of `move` is to relocate a specified variable or constraint from its current processor to a new designated one.
   - The effect of `spread` is to distribute a given list of variables and constraints equally between all processors.

   As a final effect, the initialization step decorates every variable with a list of operators that depend on it. This information is used in the extraction step.
2. Expansion. This step prepares the asynchronous local computations to follow. During it, the operators that will be applied on each processor are first selected. Then the data describing them are collected exchanged so as to become available at their target location.
3. Local agents. During this step, the chosen operators are applied to reduce variable domains in parallel without any communications.
4. Contraction. During this step the modified domains are intersected and then the changes are propagated by updating $Queue$.

- For every variable whose domain has been contracted on some processor, all existing copies of its domain participate in a global intersection operation. If only one variable took part in the operation, the intersection could be implemented as a parallel `fold` algorithm reducing p values to one in $\log p$ supersteps. But since multiple variables are being manipulated, a new algorithm has been designed. It is called `multi-fold` and reduces concurrently the sets of domains of multiple variables.
- The *Queue* of active constraints is then incremented with all operators which depend on a variable whose domain has been contracted. The dependence information is readily available at the location of every variable because it has been precomputed during initialization. As in the sequential algorithm, the previously applied operators are removed from the *Queue*.

After an iteration of steps 2,3 and 4, control returns to step 2 unless the termination condition is reached.

There are two possible pre-defined modes for the user to specify this condition: a `for` loop where a fixed number of iterations or a `while` loop where iteration continues until the fixpoint is reached (i.e. until $Queue = \emptyset$).

The normal mode of execution is `while` but its effective use of BS-Waltz requires a balanced load among processors throughout execution. This depends heavily on the choice of initial layout and on the constraint system itself. As a result, the intended use of BS-Waltz is first through an *instrumented mode* so as to determined an efficient initial layout for the given constraint system, and then in normal mode with `while` execution. The instrumented mode uses `for` execution and provides the user with a symbolic trace of the computation. The trace lists the constraints, and variables with their network layout, the dependencies among them and for each iteration and for each processor the number of operators applied and the list of variable domains reduced. The user of BS-Waltz may thus select experimentally the appropriate initial data layout for his constraint system (without re-programming the algorithm) and then maximize speedup with `while` execution. Note that these two modes are provided as examples of using BS-Waltz, and can be easily extended or replaced by user-defined predicates.

Both of the above algorithms are completely implemented in BS-Solve, but large-scale tests have not yet been conducted. Preliminary tests highlight the algorithmic problems involved in programming scalable versions of the solvers. These issues are clearly illustrated on a well-known small example: the parabola constraint system.

6 Experiment

The *Parabola* system is composed of constraints determining the intersection points coordinates of a circle and a parabola:

$$\begin{cases} c_1 : X^2 + Y^2 = 2 \\ c_2 : Y = X^2 \\ d_x : X = [-1.0e8, 1.0e8] \\ d_y : Y = [-1.0e8, 1.0e8] \end{cases}$$

We will precisely describe the behaviours of BS-Waltz and BS-Gröbner algorithms and show some of the problems raised by parallel execution and cooperation of solving processes.

6.1 BS-Waltz on *Parabola*

First, CN operators are created, one per variable V in a constraint c. The operator $c|_V$ contracts only the domain of V by means of interval Newton methods (see [13] for more details). Second, the following steps are iterated until there is no CN operator to apply:

- Application of CN operators for domains contraction, intersection of the new domains for each variable, and then
- Awakening of the CN operators depending on constraints containing a variable whose domain has just changed.

Table 1 summarizes the results. The final domains are:

$$X = [-1.18920711500272169303116 3, +1.18920711500272169303116 3]$$
$$Y = [+0.76536686473017789822392 84, +1.41421356237309536751922 7]$$

	processor 0	processor 1
Distribution	X, d_y, c_2	Y, d_x, c_1
Operators	$d_y\|_Y$, $c_2\|_X$, $c_2\|_Y$	$d_x\|_X$, $c_1\|_X$, $c_1\|_Y$
Applied	$d_y\|_Y$, $c_2\|_X$, $c_2\|_Y$	$d_x\|_X$, $c_1\|_X$, $c_1\|_Y$
Reduced	X	Y
Applied	$d_y\|_Y$, $c_2\|_X$, $c_2\|_Y$	$d_x\|_X$, $c_1\|_X$, $c_1\|_Y$
Reduced	X	
Applied	$c_2\|_X$, $c_2\|_Y$	$d_x\|_X$, $c_1\|_X$, $c_1\|_Y$
Reduced		Y
Applied	$d_y\|_Y$, $c_2\|_X$, $c_2\|_Y$	$c_1\|_X$, $c_1\|_Y$
Reduced		

Table 1: Execution of BS-Waltz for the *Parabola* system.

More operators are applied in this computation than in the equivalent sequential computation. Moreover, the load is well balanced (although as explained earlier this is not guaranteed in general). Parallelization would thus appear to be effective. Yet, even on such a small trace, with respect to the equivalent sequential execution, convergence is slower. This is in fact due to parallelism itself: local computations, being independent, are less effective in contracting domains. Their intersection does not compute as small a domain as the serial application of the same contractions. This problem is probably the most difficult one to be overcome before obtaining scalable parallel solvers.

6.2 BS-Gröbner on *Parabola*

CN operators corresponding to pairs of constraints are dynamically created after the generation of each new constraint by combination with the existing ones. They are then applied only once and the computation ends if no non-zero polynomial has been created.

Table 2 describes the resolution. The final set of constraints is:

$$\{X^2 + Y^2 = 2, Y = X^2, -Y^2 - Y + 2 = 0\}$$

	processor 0	processor 1
Distribution	c_2	c_1
Operators	(c_2, c_1)	
Created	$c_3 : -Y^2 - Y + 2 = 0$	
Operators	(c_2, c_3)	(c_1, c_3)
Created		

Table 2: Execution of BS-Gröbner for the *Parabola* system.

Here again, the example is small enough to avoid any imbalance. Execution yields the same result as the equivalent sequential process.

6.3 Cooperation between BS-Gröbner and BS-Waltz

The cooperation strategy adopted here is the following (adapted from [3]): BS-Gröbner computes all new constraints first, then BS-Waltz reduces the variables' domains using the system augmented with the Gröbner basis.

Table 3 presents an execution of BS-Gröbner followed by BS-Waltz for the *Parabola* system. The computed Gröbner basis and domains are:

$$\{X^2 + Y^2 = 2, Y = X^2, -Y^2 - Y + 2 = 0\}$$
$$X = [-1, +1.00000000000000044408921]$$
$$Y = [+1, +1]$$

	processor 0	processor 1
	BS-Gröbner	
Distribution	X, Y, d_y, c_2	d_x, c_1
Operators	(c_2, c_1)	
Created	$c_3 : -Y^2 - Y + 2 = 0$	
Operators	(c_2, c_3)	(c_1, c_3)
Created		
	BS-Waltz	
Operators	$d_y\|_Y$, $c_2\|_X$, $c_2\|_Y$, $c_3\|_Y$	$d_x\|_X$, $c_1\|_X$, $c_1\|_Y$
Applied	$d_y\|_Y$, $c_2\|_X$, $c_2\|_Y$, $c_3\|_Y$	$d_x\|_X$, $c_1\|_X$, $c_1\|_Y$
Reduced	X, Y	
Applied	$d_y\|_Y$, $c_2\|_X$, $c_2\|_Y$, $c_3\|_Y$	$d_x\|_X$, $c_1\|_X$, $c_1\|_Y$
Reduced	Y	
Applied	$d_y\|_Y$, $c_2\|_X$, $c_2\|_Y$ $c_3\|_Y$	$c_1\|_X$, $c_1\|_Y$
Reduced	X	
Applied	$c_2\|_X$, $c_2\|_Y$	$d_x\|_X$, $c_1\|_X$, $c_1\|_Y$
Reduced		

Table 3: Cooperative execution for the *Parabola* system.

As in sequential computation, the cooperative strategy converges more. The new execution of BS-Waltz is no more perfectly balanced. Moreover, here again, convergence is slower than the equivalent sequential computation because of the parallel decoupling of domains.

7 Conclusion

We have described the design of a high-level programming system for cooperative and parallelized constraint solvers. We have also designed and implemented in it, parallel versions of two important solver algorithms. Preliminary tests show that their cooperation is as effective as in the sequential case but that efficient parallelization is very difficult to attain. Apart from the usual load-balancing problems, parallel solvers face a particular problem: the parallel decoupling of domain-contraction operations possibly leading to slower convergence even in a well-balanced parallel execution. Future work will investigate possible solutions to this problem, the scalability of parallel solvers and cooperation strategies based on interleaving.

Despite the very tentative state of our algorithmic study, the software system shows that it is possible to write complex strategies at a relatively high level, while directly reusing sequential solvers. Such systems should allow the discovery of effective methods for optimizing parallel solvers and parallelizing existing cooperation strategies.

The BS-Solve system is available from the following URL:

http://www.icot.or.jp/AITEC/FGCS/funding/itaku-H9-index-E.html

under project CCOPS. More example executions of the BS-Gröbner and BS-Waltz algorithms can be found there.

References

[1] F. Benhamou. Heterogeneous Constraint Solving. In *Proceedings of ALP'96*, volume 1139 of *LNCS*, pages 62–76, Aachen, Germany, 1996. Springer-Verlag.

[2] F. Benhamou and L. Granvilliers. Application of Gröbner Bases for Numerical Constraint Solving. In *Proceedings of IMACS ACA'96*, Linz, Austria, 1996.

[3] F. Benhamou and L. Granvilliers. Automatic Generation of Numerical Redundancies for Non-Linear Constraint Solving. *Reliable Computing*, 3(3):335–344, 1997.

[4] F. Benhamou, D. McAllester, and P. van Hentenryck. CLP(Intervals) Revisited. In *Proceedings of ILPS'94*, pages 124–138, Ithaca, NY, 1994. MIT Press.

[5] F. Benhamou and W. J. Older. Applying Interval Arithmetic to Real, Integer and Boolean Constraints. *J. Logic Programming*, 32(1):1–24, 1997.

[6] B. Buchberger. Gröbner Bases: an Algorithmic Method in Polynomial Ideal Theory. In *Multidimensional Systems Theory*, pages 184–232. D. Reidel Publishing Company, 1985.

[7] T. Chikayama, F. Tetsuro, and S. Daigo. *KLIC User's Manual*. Inst. for New Generation Computer Technology, 1995, revised in 1997.

[8] A. Colmerauer. Specifications of Prolog IV. Draft, 1996.

[9] J. Hill, S. Donaldson, and A. McEwan. *Installation and User Guide for the Oxford BSP Toolset (v1.3) Implementation of BSPLib*. Oxford University Computing Laboratory, 1997.

[10] Q. Miller. *The BSP-KLIC Library*, January 1998.

[11] V. Saraswat. Concurrent constraint programming. In *POPL'90, ACM*, pages 232–245, 1990.

[12] L. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, pages 103–111, August 1990.

[13] P. van Hentenryck, L. Michel, and Y. Deville. *Numerica: a Modeling Language for Global Optimization*. MIT Press, 1997.

[14] D. L. Waltz. Generating Semantic Descriptions from Drawings of Scenes with Shadows. In P. H. Winston, editor, *The Psychology of Computer Vision*. McGraw Hill, 1975.