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/SCIENCES/LIFO/

Parallel Constraint Programming over BSP Vectors

Arnaud Lallouet and Gaétan Hains
Université d'Orléans, LIFO

Rapport N^o **99-11**

Parallel Constraint Programming over BSP Vectors

Arnaud Lallouet and Gaétan Hains
Université d'Orléans — LIFO
BP 6759 — 45067 Orléans cedex 2
{Arnaud.Lallouet | Gaetan.Hains}@lifo.univ-orleans.fr
http://www.univ-orleans.fr/SCIENCES/LIFO/Members/{lallouet | ghains}

June 16, 1999

Abstract

We propose and motivate a new formalism for declarative parallel programming: constraint programming on vectors, whose positions are interpreted as explicit processes. We introduce the vector domain $\mathcal{X}^*$ based on an underlying domain $\mathcal{X}$ and propose a set of primitive constraints on this domain: scalar predicates are extended pointwise while communication is modeled by the new global constraint $=_?$ interpreted as a bulk-synchronous communication phase. We present two families of complete solvers on this domain but show that their parallel implementation is troublesome. To avoid this problem we also introduce a family of more efficient but incomplete parallel solvers. The incomplete solvers are shown to support the declarative programming of data-parallel algorithms with explicit processes in the *bulk-synchronous parallelism* (BSP) style which ensures reliable and predictable execution costs.

1 Introduction

Writing a parallel program is a notoriously difficult task. Using C with a library like MPI [8] forces the programmer to handle low-level details that obscure the clarity of the algorithm while the use of concurrent primitives yields a program difficult to validate. As parallel machines become more and more available, there is a need for high-level languages that allow more abstraction but nevertheless maintain good performances. We strongly believe that the declarative programming paradigm is useful to address this general problem. In this paper, we focus on constraint programming with two ideas in mind:

1. Parallelism can accelerate constraint programming. Problems addressed by constraint programming are hard and often give rise to large computation times. Using more than one processor can lead to some speed-up, opening the way to new applications or allowing the resolution of larger problems.

2. Constraint programming can make parallel programming easier. By abstracting some details up to the right level, constraint programming can help to solve many of the tricky problems raised in the design of a parallel algorithm. Indeed, the placement directives of data-parallel languages like HPF [7] are constraints between array elements and explicit processes.

1.1 Parallelism in logic and constraint programming

The distinction between task- and data-parallelism has been made for years as two main sources of parallelism. Some programming languages propose to hide parallelism and leave to the compiler the difficult extraction of different threads of computations. This "implicit" parallelism has been noticed since the beginning in logic programming (see [6] for an extensive survey) because the simplicity of the logical semantics makes it apparently easy to extract. However, limiting the amount of extracted parallelism in order to maximize resources allocation is a non-trivial problem [16]. Compilation towards a data-parallel execution model has been studied in the Reform model [13] or in the Multilog system [19].

In contrast, explicit parallel languages allow the programmer to describe precisely the parallel execution of his program by specifying independent computations, communications or even task and memory allocation. A large number of concurrent logic languages exploit this form of parallelism, without [18] or with [17] constraints. Up to our knowledge, only one attempt has been made in order to exploit explicit data-(AND-)parallelism in logic programming: our DP-LOG project [11, 9, 10].

Most of the works in parallel constraint satisfaction consist in parallelizing a specific algorithm, like backtracking search [21], Gröbner basis computation [4] or arc-consistency [15]. Only recent works tend to give parallel constraint propagation a strong theoretical basis [14].

1.2 Data-parallelism and the BSP model

The basic data-parallel model proposes to compute on vector instead of scalars, and thus provides a single centralized thread of computation. Computation is seen as a sequence of vector transformations and communication is just a special (non-local) transformation. Advantages of this model are well-known since the programmer does not have to deal with complex interaction of different processes and concurrent access to the resources. On the other side, the compiler has to fill the gap between the synchronous programming model and an asynchronous and efficient execution.

Traditionnally, vectors are of arbitrary length and are mapped on a given architecture by a mechanism called "virtualization". Moreover, this length is subject to change during the computation when new vectors are created, like for example in the language NESL [3]. This may be the source of inefficiency. Therefore, a current trend in data-parallelism focuses on a static mapping between processes and actual processors called "explicit processes". The *Bulk Synchronous Parallelism* (BSP) [20] and the *LogP* [5] models rely on this feature to propose a cost model that allows a portable performance prediction. Here is a short presentation of BSP as it is chosen as target model for our proposition.

A BSP computer contains 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 (the last two 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* (cf figure 1), each one divided into (at most) three successive and logically disjoint phases. In the first phase each processor uses its local data (only) to perform sequential computations

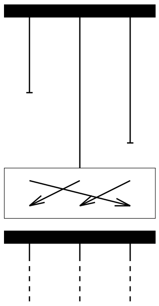

Figure 1: The BSP scheme

and to request data transfers to/from other nodes. In the second phase the network delivers the requested data 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 :

$$\mathrm{Time}(s) = \max_{i:processor} w_i^{(s)} + \max_{i:processor} h_i^{(s)} * g + l$$

where $w_i^{(s)}$ = local processing time on processor i during superstep s and $h_i^{(s)} = \max\{h_{i+}^{(s)}, h_{i-}^{(s)}\}$ where $h_{i+}^{(s)}$ (resp. $h_{i-}^{(s)}$) is the number of words transmitted (resp. received) by processor i during superstep s. The execution time $\sum_{s\in S} \mathrm{Time}(s)$ of a BSP program composed of S supersteps is therefore a sum of 3 terms: $W+H*g+S*l$ where $W = \sum_s \max_i w_i^{(s)}$ and $H = \sum_s \max_i h_i^{(s)}$. In general W, H and S are functions of p and of the size of data n, or of more complex parameters like data skew and histogram sizes. To minimize execution time the BSP algorithm design must jointly minimize the number S of supersteps and the total volume h (resp. W) and imbalance $h^{(s)}$ (resp. $W^{(s)}$) of communication (resp. local computation).

1.3 Vectorial constraint programming

To reach the objectives stated above, we propose a programming scheme where constraint programs are given a BSP interpretation. This proceeds in five steps, each one realizing a minimal extension of constraint programming required by the objective of predictable parallel performance.

1. Domain extension. Data such as constants, numbers, variables or terms are promoted from scalar to vectors : for example, a vectorial variable X is a vector of variables (i.e. a family $(X_i)_{i\in\mathbb{N}}$). A careful definition of vectorial term also allows to identify it to a vector of terms. This point is developped in section 2.1. These vectorial entities are linked together by relations called “global constraints”. For example, the constraint $p(a, X)$ operates on two vectors : a and X. The domain of vectors is defined to represent explicitly the mapping of data to processors. For a given domain X, we denote by X^* the domain of vectors of elements of X. On a p processors machine, processor locations are integer between 0 and $p-1$ and actual vectors are cut up to the length of p.

2. Vectorial constraints. We define two types of constraints : constraints on vectors called *global* or *vectorial* constraints and constraints on vector elements called *local* or *scalar* constraints. The latters allow to define different computations at different process location. They are defined classically from a set of constraint predicates. Then, one way to define a global constraint is to extend pointwise a scalar constraint predicate. For example, $leq(X, Y)$ denotes the vector of constraints $\forall i \in \mathbb{N}, X_i \leq Y_i$. Equality between vector is defined similarly. In addition, new predicates can be used to define vectorial constraints that do not have a simple local counterpart. This feature is used to model communication.

The two types of constraints correspond to the two types of phases in BSP computing: asynchronous local and bulk-synchronous global computation. Parallel execution is thus made internal to the solver's semantics i.e. the knowledge of a parallel solver's algorithm, combined with the straightforward BSP semantics of our constraints determines a realistic execution time (as a function of the execution time of sequential solvers).

3. Communication as remote equality. Communication is defined as a global constraint between three vectors: $=_?(X, L, Y)$ (pronounced "equal-get") where X and Y are data vectors and L is a vector of processor locations. The intended meaning is to state that X is equal to Y modulo an indirection denoted by L. For example, on a three processors machine, the constraint $=_?(X, [1, 0, 0], Y)$ denotes the following equalities: $Y_0 = X_1, Y_1 = X_0$ and $Y_2 = X_0$. This is shown in figure 2. This encompasses the traditionnal view of communication in a relational way. This also means that, semantically speaking, communication is no more an event but an equality and the precise method with whose it is enforced determines the solver. It is understood that more complex vector constraints may be predefined or built from this one.

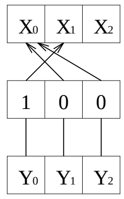

Figure 2: A simple communication

Here is an example with a single constraint to express the broadcast of a data to all processes: $=_?(X, 0, Y)$. When processed, all Y_i unify with X_0. Another (toy) example is given in the proof of proposition 3.4. A more complex example is the parallel prefix sum which computes the prefix sum of all elements of a vector of size n in time $\mathcal{O}(\log_2 n)$ described in section 4. It needs a host language to state dynamically the constraints.

4. Parallel solvers. A parallel solver is then defined as the combination of a set of operators on constraints and a strategy for applying them. Since the domain X^* is built on top of a given domain X, we suppose that a solver for X is available. Recent works [2, 1] tend to give constraint propagation over heterogeneous domains a formal and uniform view, based on a fixpoint computation of some domain reduction functions. We take up this framework and describe our solvers in term of this kind of operators.

We present three solvers that differ by the way they treat communication, since the other global constraints are just distributed pointwise:

- a centralized solver called CentralSolv, which enforces full equality and is thus complete with regards to parallelism. Unfortunately, we show that its implementation on a distributed system is troublesome;

- a complete and distributed solver called CopySolv, but with a deplorable complexity;

- an incomplete but efficient solver called MATCHSOLV, which replaces unification by matching. Communication recovers its event status and opens the way to an efficient implementation. Moreover, this allows to give a BSP interpretation of vectorial constraint solving. In contrast, as it can be seen in logic programming for functional arithmetics, using a incomplete solver needs a knowledge of the solver's strategy.

Nevertheless and despite its incompleteness, our last solver deserves further attention for two reasons: firstly it takes place in a broader framework and the existence of a complete and yet efficient solver is an open question. Moreover, optimizations may be found to reduce the degree of incompleteness. And secondly, the indirection vector (the second parameter of $=_?$) is also subject to be constrained and can possibly be non-ground. This means that we can specify a class of parallel algorithms with constraints, and let the system guess for the right communication pattern. This can be compared with owning a parallelizing compiler at execution time. But this search space is large and these constraints have to be carefully stated.

5. Host language issue. Any kind of host language can be chosen, either declarative or imperative, either parallel or sequential. Here we present a sequential pseudo-pascal extended with primitives allowing to add a vectorial constraint to the store or a scalar constraint to a specific location of the store. Parallel computation is then done when the constraints are solved. We are currently implementing a solver using C++ as the host language and MPI [8] as the communication library. The combination allows to state the constraints in a data-parallel fashion. We also plan to define a CLP language based on our previous work for logic programming [9].

In the rest of the paper, we define the domain of vectors, we present our three solvers and we show that their parallel semantics is an instance of the chaotic iteration framework. We then prove completeness results for them and give two strategies for our last solver.

1.4 Basic notations and definitions

We first recall the basic framework in which our work takes place (taken from [2] and similar to [1]). Let Σ be a signature composed of a denumerable set of function symbols of all arities F, $\mathcal{V}$ be a finite set of variables $\mathcal{V} = \{V_1, \ldots, V_{n_v}\}$ we assume large enough to represent the given problem and of a set of constraint predicates Π including $true, false$ and $=$. As in [2], we extend each n-ary relation ($n \leq n_v$) in a n_v-ary relation. We call $T(\Sigma)$ the set of closed terms and $T(\Sigma, \mathcal{V})$ the set of terms. An atomic constraint is of the form $p(t_1, \ldots, t_n)$ with $p \in \Pi$ and $t_i \in T(\Sigma, \mathcal{V})$. We denote by $\mathcal{AC}(\Sigma, \mathcal{V}, \Pi)$ the set of atomic constraint formulae. Let assume that the constraints are given by a first-order language closed under variable renaming, conjunction and existential quantification.

The interpretation of constraints is given by a Σ-structure composed of a domain $\mathbb{D}$ and of a set of functions and relations interpreting the symbols of Σ. The domain membership of a variable v is included in $\mathbb{D}$. To be more precise, we can consider following [2] that it is a subset A of $\mathcal{P}(\mathbb{D})$ called approximate domain. A simple example is the set of intervals bounded by IEEE floating point numbers used to achieve error-free computations. In [1], the same notion is called family of subsets based on $\mathbb{D}$. When no confusion is possible, we shall write the same symbol to denote a constraint and its interpretation: a constraint can be seen as a subset of $\mathbb{D}^{n_v}$.

Definition 1.1 (CSP) *A Constraint Satisfaction Problem (in short CSP) is a couple $< \mathcal{C}, \mathcal{D} >$ where $\mathcal{C}$ is a set of constraints or store, and $\mathcal{D} = D_1 \times \ldots \times D_{n_v} \in A^{n_v}$ denotes the domain associated to the set of variables $\mathcal{V}$.*

Definition 1.2 (Declarative semantics of a CSP) *The declarative semantics of a CSP is given by:*

$$[\![< \mathcal{C}, \mathcal{D} >]\!]_{decl} = \bigcap_{c_i \in \mathcal{C}} c_i \cap \mathcal{D}$$

The set of CSPs can be equipped with the following ordering (information ordering): let $SC_1 =< \mathcal{C}_1, \mathcal{D}_1 >$ and $SC_2 =< \mathcal{C}_2, \mathcal{D}_2 >$ be two CSPs, we set $SC_1 \sqsubseteq SC_2 \Longleftrightarrow \mathcal{C}_1 \supseteq \mathcal{C}_2$ et $\mathcal{D}_1 \subseteq \mathcal{D}_2$. This set is a complete lattice. Still following [2], we can model the semantics of a solver as the greatest common fixpoint of a set of narrowing operators (a correct, monotone and contractant operator). This result is made more precise in [1] as the limit of a chaotic iteration of these operators.

An operator o associated to a constraint ρ is complete if it is guaranteed to detect inconsistency when applied on a given "box" domain, ie if $\forall d \in A^{n_v}, o(d) = \emptyset \Longleftrightarrow \rho \cap d = \emptyset$. A solver made of complete operators is also complete.

This framework assumes that the solving of a CSP can be modeled by iteration of monotone operators. However, some domains like finite domains need additionnal search capabilities. This can be modeled by a rule for enumeration, or a variation on this idea like balanced divide-and-conquer. According to [1], these two techniques have to be mixed.

2 Vectorial constraints

2.1 Vectors and global naming

Let E be a set. We call *vector* on E a family $a = [a_i]_{i \in \mathbb{N}}$ of elements of E. The set of vectors on E is denoted by E^*, and $\forall i \in \mathbb{N}$, $a|_i$ denotes the i-th projection of a, which is intended to be the part of he object stored on processor i of a parallel computer.

We call $\overline{T}$ the set of vectorial terms over a signature Σ. Construction of these terms is straightforward on scalar domains like integers or reals. An extension to an arbitrary signature is defined hereafter by the mean of two mappings *loc* and *cent* that associates a vectorial term to its corresponding vector of scalar terms.

In the following, Σ and V are sets of functions and variables symbols we call *scalar* symbols and upon which we build the sets $T(\Sigma)$ (the set of closed terms) and $T(\Sigma, \mathcal{V})$ (the set of terms). To be more precise, a term is a finite tree defined by a set of *node positions* (a standard domain for nodes is the set of finite sequences of integers) and a labeling function associating a symbol to a given position. For a term t and a position p, we denote by $t{\downarrow_p}$ the symbol of t located at the position p.

We want to define the set of vectorial terms such that there exists a correspondence between every vector of terms and a vectorial term, i.e. to define a correspondence between an element of $T(\Sigma, V)^*$ and a vectorial term. For this, we need to define the symbol sets from which the vectorial terms are built. The main problem is that a vector of terms may contain terms of different depth and/or arity. For example, compare the homogeneous vector of figure 3(a) to the heterogeneous one in figure 3(b). Intuitively, we want to use as vectorial functions symbols the sequence of symbols appearing at a given position in every term of the vector. For example, we want to denote the first term by $[f, f, f]([a, a, a], [b, b, b])$, the sequence

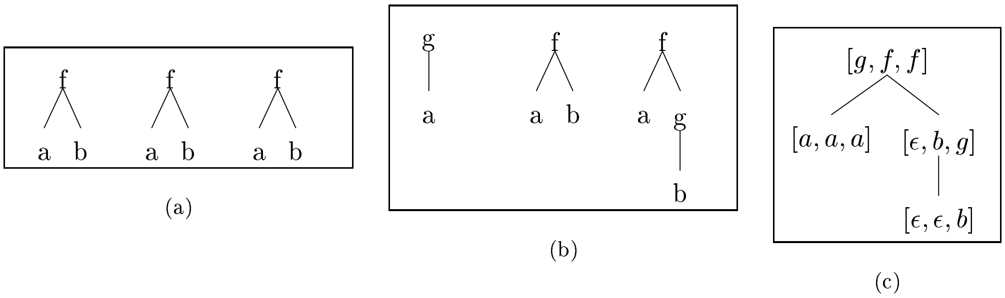

Figure 3: Homogeneous and heterogeneous vectors of terms

$[f, f, f]$ being a vectorial function symbol. In order to treat both cases equally, we introduce a new symbol ϵ ($\epsilon \notin \Sigma$ and $\epsilon \notin V$) intended to model a "hole" somewhere in the vector at a given position. And since our vector are potentially of any length, we also fill all non-defined positions with ϵ.

For this, let us consider $\Sigma_1 = (\Sigma \cup \epsilon)^*$ and $V_1 = (V \cup \epsilon)^*$ the sets of finite sequences of functions and variables symbols, plus eventually ϵ. Let $\simeq$ be the relation on Σ_1 and V_1 defined by $\{(u, v) \mid \exists w \in \{\epsilon\}^* \text{ such that } u.w = v\}$. We now consider as the set of *vectorial* functions symbols $\overline{\Sigma}$ and *vectorial* variables symbols $\overline{V}$ the quotients $\overline{\Sigma} = \Sigma_1/_\simeq$ and $\overline{V} = V_1/_\simeq$. The canonical representant of a class is the sequence $[a_i]_{i\in\mathbb{N}}$ but we only represent the smallest prefix whose last element is different from ϵ. The arity of such a symbol is given by the maximal arity of the scalar symbols it contains.

A first idea is to take $T(\overline{\Sigma}, \overline{V})$ as the set of vectorial terms. But it is possible to build a term in $T(\overline{\Sigma}, \overline{V})$ that does not correspond to a vector of terms if the arity of scalar symbols does not match. For this reason, we define the set $\overline{T}$ of vectorial terms as being a subset of $T(\overline{\Sigma}, \overline{V})$. We want to model the bijection between the set of vectorial terms $\overline{T}$ and the set of vectors of terms $T(\Sigma, V)^*$ as a couple of functions called *localization* and *centralization*:

$$\overline{T} \quad \underset{\text{centralization}}{\overset{\text{localization}}{\rightleftarrows}} \quad T(\Sigma, V)^*$$

The set of vectorial terms $\overline{T}$ is the image set of the centralization function. With this restriction, we ensure that all vectorial term corresponds to an effective vector of terms. For a scalar term t, we extend the labeling function of the tree to all positions by setting $t{\downarrow_p} = \epsilon$ if p was not defined. Let $[t_i]_{i\in\mathbb{N}}$ be a vector of terms. The vectorial term $cent([t_i]_{i\in\mathbb{N}})$ is defined at each position p by the symbol $[t_i{\downarrow_p}]_{i\in\mathbb{N}}$. Of course $t_i{\downarrow_p} \in \Sigma \cup V \cup \{\epsilon\}$. Our example of figure 3(b) yields the term in figure 3(c).

The localization function is defined recursively as follows:

- $loc([\epsilon]_{i\in\mathbb{N}}) = [\epsilon]_{i\in\mathbb{N}}$;
- $\forall t = f(t_1, \ldots, t_n) \in \overline{T}, loc(t) = [loc(f)|_i(loc(t_1)|_i, \ldots, loc(t_n)|_i)]_{i\in\mathbb{N}}$.

The term obtained at a location may still contain ϵ at a function position. For example, the localisation at index 0 of the term in figure 3(c) yields to $g(a, \epsilon(\epsilon))$. A normalization step is thus necessary to get rid of this superfluous symbol. It uses the closure of the following rule: $f(t_1, \ldots, t_n, \epsilon) \longrightarrow f(t_1, \ldots, t_n)$.

In our running example, we get at index 0 : $g(a, \epsilon(\epsilon)) \longrightarrow g(a, \epsilon) \longrightarrow g(a)$. Arities are matching by construction of the term with the centralization function.

Vectors are implemented on a p-processors machine by taking their prefix up to a width of p. Thus two different programs may show the same behavior if they agree on a prefix of length p.

2.2 Vectorial constraints and solvers

Let Π be a set of (scalar) constraint predicates symbols and let AC be the set of atomic constraints. Let $\overline{\Pi}$ be a set of (vectorial) constraint symbols. The set $\overline{AC} = \{c(t_1, \ldots, t_n) \mid c \in \overline{\Pi} \text{ and } \forall i \in [1..n], t_i \in \overline{T}\}$ is the set of primitive vectorial constraints.

A solver for a vectorial constraint can be expressed by two types of rules:

- a *vectorial* rule: $\overline{AC} \longrightarrow 2^{\overline{AC}}$;
- a *localization* rule: $\overline{AC} \longrightarrow AC^*$.

User-defined constraints are likely to belong to the first type. Since we want to model the minimal language to express parallelism, we only provide rules of the second type for a minimal set of vectorial constraints as follows. We choose to take as $\overline{\Pi}$ the set of pointwise extensions of scalar constraints plus a minimal set $\overline{\Pi_0}$ of vectorial predicates that do not have a scalar counterpart:

$$\overline{\Pi} = \{\overline{c} = [c]_{i \in \mathbb{N}} \mid c \in \Pi\} \cup \overline{\Pi_0}$$

Let $\overline{\Pi_0} = \{=_?\}$ ($=_?$ is of arity 3). The intended meaning of $=_?$ (X, L, Y) is to model a global communication, as informally defined in section 1, i.e. the remote equality between elements of X and Y directed by the location L.

The building of a solver consists to provide localization rules for the vectorial constraints of $\overline{\Pi}$ and a strategy. But it is important to notice that localization does not imply distribution. In fact, it just defines how the global symbols are interpreted at the local level. Here is an illustration:

Example 2.1 *Let us consider a constraint symbol $p \in \overline{\Pi}$ of arity 1. Thus $p(X)$ is a valid vectorial atomic constraint formula if $X = [X_i]_{i \in \mathbb{N}} \in \overline{V}$. Assume now that the solver for p rewrites $p(X)$ to $[p(X_{i-1})]_{i \in \mathbb{N}}$. At each location, the vector of constraints contains a non-local symbol.*

In general, this may not be a desirable feature in a distributed case, because it yields shared variables or a communication overhead. We define the notion of distributed vectorial object:

Definition 2.2 (Distributed object) *A vectorial object o is* distributed *if each projection on an index of its localization $loc(o)$ does not include any reference to a sub-object located at another index.*

For example, the following vectorial term is not distributed: $[g, g]([X, X])$ because of the variable X.

Here we present three possible families of parallel solvers which differ by the way they treat communication. At this point we will leave their strategies open and for convenience we will refer to "the solver" for a typical member of each family.

- The centralized solver CENTRALSOLV : this first solver preserves remote bindings and yields non-distributable computations. Its main interest is to give a declarative meaning to the computation. The implementation of CENTRALSOLV requires a shared memory because the management of non-local bindings may require an unpredictable number of additional communication and is thus not scalable to very large parallel systems.

- The distributed-by-copy solver COPYSOLV : this solver copies every equality constraint on every concerned location. Variables with the same name on different locations are considered as distinct. This is a complete and distributed solver but has the worst space complexity of all, proportional to the cartesian product of the locations concerned by a communication. COPYSOLV removes the requirement for a shared memory by repeating (in parallel) the processing of every constraint on every concerned processor. Clearly, speedup relative to sequential processing is not guaranteed in this case.

- The term-matching solver MATCHSOLV : this last solver models a highly desirable property : atomicity of communication but at the cost of the solver's completeness because equality is replaced by the weaker notion of term-matching. MATHSOLV is designed to use a distributed memory *and* to avoid redundant computation. It is the best candidate for a BSP interpretation.

These solvers agree on the way they reduce the pointwise extensions of scalar constraints. They replace the vectorial constraint by the corresponding vector of scalar constraints. Let $\overline{c} = [c]_{i\in\mathbb{N}}$ be the vectorial pointwise extension of the scalar constraint predicate c, and $t_1, \ldots, t_n$ be vectorial terms.

Rule 2.3 (Pointwise extension)

$$\frac{[c(loc(t_1)|_i, \ldots, loc(t_n)|_i)]_{i\in\mathbb{N}}}{\overline{c}(t_1, \ldots, t_n)}$$

Here follows the rule defining communication in each of these three solvers. Let X, L and Y be vectorial terms. In order for the rule to be applicable, L must be ground and bound to an integer vector. The general case will be discussed later.

- CENTRALSOLV : this solver does not ensure the decoupling of the store. A communication is not atomic, since, in a distributed setting, the update of remote bindings may force new communications.

  Rule 2.4 (CENTRALSOLV **COM)**

  $$\frac{[loc(X)|_{loc(L)|_i} = loc(Y)|_i]_{i\in\mathbb{N}}}{=_? (X, L, Y)}$$

  As explained above, a BSP interpretation is difficult because local computation is not decoupled from communication. For example, let us consider the following constraint : $=_? ([X_0, X_1, X_2], [1, 0, 0], [Y_0, Y_1, Y_2])$. This constraint is rewritten as $[\{X_1 = Y_0\}, \{X_0 = Y_1\}, \{X_0 = Y_2\}]$ by the rule. The first equation contains a reference to X_1, which is located on processor 1. Subsequent solving on processor 0, for example the binding of Y_0, may modify X_1, requiring a communication invisible in the syntax.

- COPYSOLV : this solver duplicates an equation at every location where one of its variables is used.

Rule 2.5 (CopySolv **COM)**

$$\frac{[\{loc(X)|_{loc(L)|_i} = loc(Y)|_k \mid \exists k \in \mathbb{N} : loc(L)|_k = loc(L)|_i \;\; or \; i = loc(L)|_k\}]_{i\in\mathbb{N}}}{=_? (X, L, Y)}$$

The equation $X_{L_i} = Y_i$ is added to the index i, to the index L_i and moreover to every index where X_{L_i} has to be communicated. Using the same example as above, the following table describes the constraints added to each index:

0	1	2
$X_1 = Y_0$	$X_0 = Y_1$	$X_0 = Y_1$
$X_0 = Y_1$	$X_0 = Y_2$	$X_0 = Y_2$
$X_0 = Y_2$	$X_1 = Y_0$	

In order to ensure independence, remote copies of a variable arising from this operation are tagged by their new local index. Note that the global coherence is not easy to ensure since different occurences of the same variable have to agree in order to get a valid result. The purpose of this solver is only to show the theoretical feasability of a complete and distributed solver with atomic communication, but its high cost makes it not worth to implement.

- MatchSolv: this solver implements term-matching between X_{L_i} and Y_i, rather than full unification. For this, we rename the free variables of X_{L_i} with fresh ones which do not appear in X_{L_i} or Y_i, or anywhere else in the global vectorial constraint. Let us call σ such a substitution.

  Rule 2.6 (MatchSolv **COM)**

  $$\frac{[\sigma(loc(X)|_{loc(L)|_i}) = loc(Y)|_i]_{i\in\mathbb{N}}}{=_? (X, L, Y)}$$

  The resulting absence of variable sharing yields the atomicity of communication and distributability of the store, ensuring the possibility of a BSP interpretation: the application of the communication rule is a non-revocable action. It does not create any dependences between indices once the renamed terms are communicated.

As we noted above, the index of communication has to be exactly known in order to execute the remote binding $=_? (X, L, Y)$. If not, the variable L would be treated as any variable on finite domain, having to be enumerated before the communication can commit. However, since L is a vectorial variable, the enumeration would have to be done in a distributed setting, i.e. not independently. It is under the responsability of the solver to provide a good traversal of the search space. In our current study we do not focus on this more general approach, but we aim to highlight this important aspect. With this feature, a program could choose its actual parallelization at run-time. The obvious expected benefit could make the high cost of a distributed backtracking acceptable.

3 BSP chaotic iteration

Now we define an abstract framework for chaotic iteration on vectors. A *vectorial CSP*, or vCSP is a triple $< VC, SC, D >$ where $VC \subseteq \overline{AC}$ is a store of vectorial constraints, $SC \in (2^{AC})^*$ is a vector of scalar stores and $D = D_1 \times \ldots \times D_{n_v}$

denotes the domain of the variables. Since scalar variables are embedded into vectorial terms, we only need to focus on the former at this level of description, namely that of the parallel solver and not the host language.

In the intended distributed implementation of a vCSP, each processor owns every vectorial constraints (they correspond in a sense to a sequential calculus replicated on every node), the projection of SC on its name and the variables present at its location. It can be either scalar variables from SC or the projection of the variables of VC on its location name.

Chaotic iteration over a vCSP needs a proper notion of operator. A vectorial operator models a global reduction on a vector:

Definition 3.1 (vectorial operator) *A* vectorial operator *belonging to a vCSP is either an operator associated with a vectorial constraint in VC, called* vectorial constraint operator *(VCO), or a vector of sets of scalar operators, called* vectorial asynchronous operator *(VAO).*

The approximate semantics of a vCSP is then given by a chaotic iteration of its associated vectorial operators.

Since we want to provide a BSP interpretation of constraint programming, this definition allows to model both the synchronous global computation and the asynchronous local ones. Namely, a VCO models a BSP superstep, namely a communication and synchronization phase (or even an embedded predefined BSP algorithm), and a VAO models an asynchronous computation phase.

Since they do not discard any binding, the first two solvers are complete with respect to the vectorial part of the computation. We name this property *vector completeness*. It means that they are complete if this property holds for the solver of the underlying domain. However, this is not true for the last solver because a communication prevents any future bindings between two locations to occur.

Proposition 3.2 CENTRALSOLV *is vector-complete.*

Proof Let consider a vCSP $< VC, SC, D >$. As a vectorial store, VC contains communication constraints of the form $=_? (X, L, Y)$ and vectorial extensions of scalar constraints. By rewriting according to the rules of CENTRALSOLV, we get at each index i a set of scalar constraints RC_i. Thus at index i, $SC_i \cup RC_i$ is a scalar store. Since all bindings are preserved, $< VC, SC, D >$ and $< \emptyset, [SC_i \cup RC_i]_{i \in \mathbb{N}}, D >$ have the same solutions. Then completeness of the scalar solver implies completeness of its vectorial extension by CENTRALSOLV. ⊛

Proposition 3.3 COPYSOLV *is vector-complete.*

Proof The proof is analogous, except that at an index i, the set $SC_i \cup RC_i$ does not contain any reference to a remote object. Moreover, using different strategies on different nodes can lead to incoherent bindings. ⊛

Proposition 3.4 MATCHSOLV *is incomplete.*

Proof An incomplete solver is not always able to detect insatisfiability of the store. Here is a counter-example over the scalar domain of finite trees (Herbrand domain). Let S be the following vCSP on the indexes 0 and 1:

$$S =< \{=_? (X, [1, 0], Y)\}, [\{X_0 = a, Y_0 = a\}, \{X_1 = b, Y_1 = b\}], D >$$

with $D = \{X_0 \in T(\Sigma, V), X_1 \in T(\Sigma, V), Y_0 \in T(\Sigma, V), Y_1 \in T(\Sigma, V)\}$. In this example, the domain of X_0 for example has not yet been reduced with its local constraint. Let assume that the strategy we choose is to reduce vectorial constraints first. Let S' be the vCSP obtained by rewriting the vectorial constraint with CentralSolv and S'' the one obtained by the same operation using MatchSolv. We use the standard subsumption ordering on terms, extended to CSP by its cartesian product across variables according to section 1.4 and to vCSP by cartesian product over indices.

$$\begin{tikzcd} & S \arrow[dl, "\textsc{CentralSolv}"'] \arrow[dr, "\textsc{MatchSolv}"] & \\ S' & \sqsubseteq & S'' \end{tikzcd}$$

By using CentralSolv, we get:

$$S' =< \emptyset, [\{X_0 = a, Y_0 = a, Y_0 = X_1\}, \{X_1 = b, Y_1 = b, Y_1 = X_0\}], D >$$

After application of obvious local domain reduction rules, we get the domain $D' = \{X_0 \in \{a\}, X_1 \in \{b\}, Y_0 \in \emptyset, Y_1 \in \emptyset\}$. By using MatchSolv, we get:

$$S'' =< \emptyset, [\{X_0 = a, Y_0 = a, Y_0 = X_1'\}, \{X_1 = b, Y_1 = b, Y_1 = X_0'\}], D >$$

This leads to a domain $D'' = \{X_0 \in \{a\}, X_1 \in \{b\}, Y_0 \in \{a\}, Y_1 \in \{b\}\}$. Hence, $S' \sqsubseteq S''$, because $D' \subseteq D''$ and $C' \supseteq C''$. Here MatchSolv could not detect insatisfiability. It is to be remarked that a different strategy would have worked better in this case: choosing local constraints first would have yield $S' = S''$. ⊛

The last crucial point concerning the solver is to define its strategy. Since MatchSolv is an interesting solver with regards to parallelism, choosing a good strategy could eventually provide more accurate results (CentralSolv and CopySolv results are insensitive to strategy). An interesting point is that vectorial constraints obtained by pointwise extension of scalar ones do not need any synchronisation since they are independent: we can treat them as local constraints. We propose the following strategies:

- the *locally eager* strategy:

  1. reduce pointwise extensions of scalar constraints. This point can be done locally, i.e. without synchronization;
  2. apply local operators at each location up to reach a fixpoint;
  3. reduce $=_?$ constraints. This step involves communications and synchronizations. By delaying as long as possible the application of these constraints, we hopefully communicate full computation results instead of partial ones.

  With this strategy, a BSP superstep correspond to a call of the solver.

- the *data-flow* strategy:

  1. reduce the constraint $=_? (X, L, Y)$ as soon as X and L are closed, then synchronize. This scheme is close to the delay mechanism of logic languages.

2. reduce pointwise extensions of scalar constraints and apply local operators at each location ;

With this strategy, a call of the solver may involve more than one BSP superstep.

4 A simple host language and an example

There is obviously no perfect solution to the problem of defining a strategy, but control can come from somewhere else. A solver has to be embedded in a host language used to define the constraints. Incrementality of the solver and control inherited from the host language gives a good expression power to solve a problem. It is out of the scope of this paper to define a good host language, but we nevertheless need a minimal set of primitives to present some examples.

Since an actual computation involves an actual parallel computer, vector are cut to a length of p corresponding to the number of available processors. In order for the program to be scalable, it is needed to know two basic parameters : the actual number of processors (which is here the vectorial constant `Nprocs`) and the actual index of computation. The latter is only useful for the asynchronous part of the computation and is determined by a distinguished variable when the constraints are added.

We propose to use a small Pascal-like sequential language to build the vCSP and to control the solving process. We introduce here three basic operations on the store $< VC, SC, D >$:

- `AddVC(`vc`)` takes a vectorial constraint vc and adds it to the vectorial part VC of the store ;
- `AddSC(`l`,`sc`)` takes a scalar constraint sc and adds it to the scalar store SC at the location l. Constraints can be different at each location in order to model different asynchronous computations.
- `Solve(`S`)` starts the solver and can be parameterized by a strategy S.

The result of the computation is obtained by examining the reduced domain of a set of variables of interest. We assume that the language features include *for* and *while* loops, procedures and recursion. Its exact properties are not important since the main purpose is to be able to define clearly and easily a vCSP. Here follows an example : the prefix sum of a vector of integers. It is meaningful because firstly it is a classical example used as a basic block to build more complex parallel algorithms ; secondly, it presents all aspects of our study : vectorial pointwise and communication constraints, local asynchronous constraints and resolution on the vector domain ; and then it illustrate the complex nesting between the host language and the solver, as well as the importance of the strategy.

Example 4.1 *The* scan *algorithm computes the prefix sum of a vector of integers in $O(\log_2 n)$ time by ordering computations into a binary tree (see figure 5). The underlying domain of computation is FD with the usual constraints ($+, -, \dots$) and we suppose that the construction* `vect of X` *denotes the type of a vector of objects of type* `X`*. The program in figure 4 implements the* scan *algorithm. It uses an auxiliary recursive procedure* `sc`*. The meaning of the different parts is the following :*

- *A is the starting vector and B the result, N is used to count the recursion depth ;*

```
procedure sc ( in A, out B : vect of int, N : int )
  M,L,C,D,E : vect of int
  i : int
begin
  if ( N = 1 ) then
    AddVC ( A=B )
  else
    AddVC ( M = N/2 )
    sc (A,C,M)
    for i := 1 to Nprocs-1 do
      if ( i >= M ) then
        AddSC ( i, L = i-M )
      else
        AddSC ( i, L = i )
      end if
    end for
    AddVC ( =?(C,L,D) )
    Solve( locally eager )
    for i := 1 to Nprocs-1 do
      if ( i >= M ) then
        AddSC ( i, E = D )
      else
        AddSC ( i, E = 0 )
      end if
    end for
    AddVC ( B = C+E )
end

procedure scan ( A,B : vect of int )
begin
  sc ( A,B,Nprocs )
  Solve( locally eager )
end
```

Figure 4: The scan program

- *the first* for *loop is used to compute the index of the next communication, i.e. at distance $N/2$ of the current index. But some indices (depending of the depth of the recursion) do not participate, for example the index* 0 *for* $N = 1$ *(see figure 5 in appendix). They are given a dummy index that does not add any communication cost (namely their own index).*
- *with the locally eager strategy, the call to the solver is needed to commit the communication. If it does not occur, the further binding of B by $C + E$ would be unknown, showing the incompleteness of the solver;*
- *the last for loop replaces the value to be added by 0 (neutral for addition) on non-concerned locations for the same reasons as above.*

Interestingly, the use of the data-flow strategy would have make unnecessary the call to `Solve()` *in the procedure* `sc`. *This is because the binding of C coming out*

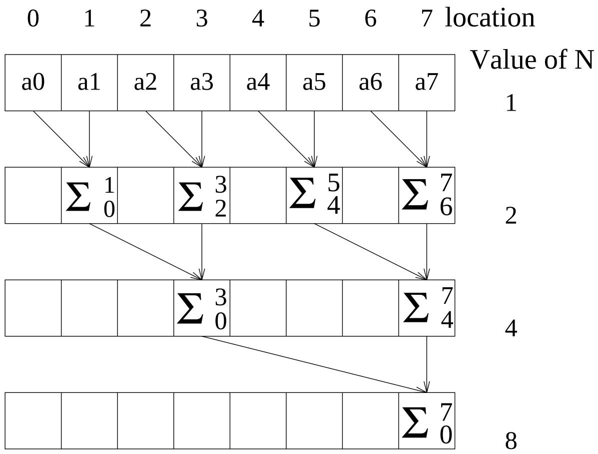

Figure 5: Communication graph for the last result

of the return from the recursive call to `sc` *would launch the communication and synchronization phase.*

The sequential nature of the host language presented here is not a real limitation because the *for* loops may be executed in parallel with a suitable data-parallel language (for example with a *forwhere* statement). One may eventually suppose they are pre-calculated and added at no cost. Moreover, scalar constants like N or $Nprocs$ are naturally extended to vectors and the sequential part is done on every node without any slow-down. Again, our purpose was only to illustrate the use of the solver.

5 Discussion and conclusion

Let us summarise the main contributions of this paper : we propose a new framework for parallel constraint programming : constraint programming on vectors. This allows to take advantage of data-parallelism, especially of the BSP model that combines the simplicity to performance prediction and efficient execution. Communication is modeled as a global relation on vectors for which we provide solvers. Efficiency is obtained by relaxing the completeness of the solver. We describe the theoretical framework for a large class of parallel constraint languages adaptable to many programming paradigms (from logic to imperative).

The use of MatchSolv forces the programmer to have a knowledge of the evaluation strategy. The requirement for a ground index of communication makes the language close to a functional one (see the language BS-λ [12] for example), at least with regards to parallelism. But our theory is broader and a better solver is a topic deserving further attention. Moreover, this can be done in an incremental way, by choosing the features to add :

- the handle of a non-ground index of communication allows the programmer to leave some freeness in the placement of data, and opens the way to dynamic load balancing controlled by the program itself.
- the handle of the relational aspect of $=_?$ is a challenging work in order to preserve efficiency. There could exist a compromise between communication

and duplication (with the implicit migration of variables for example) ;

An implementation of these concepts is in progress as a C++ library.

References

[1] K. R. Apt. The essence of constraint propagation. *Theoretical Computer Science*, 1999. to appear.

[2] Frédéric Benhamou. Heterogeneous constraint solving. In Michael Hanus and Mario Rodríguez-Artalejo, editors, *Algebraic and Logic Programming, 5th International Conference, ALP'96*, volume 1139 of *lncs*, pages 62–76, Aachen, Germany, 25–27 September 1996. Springer.

[3] Guy E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, 1996.

[4] Soumen Chakrabarti and Katherine A. Yelick. On the correctness of a distributed memory gröbner basis algorithm. In Claude Kirchner, editor, *Rewriting Techniques and Applications*, number 690 in LNCS, pages 77–91, Montreal, CA, June 1993. Springer.

[5] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and V. E. Thorsten. Logp : Towards a realistic model of parallel computation. In *Principles and practice of parallel programming*, pages 1–12. ACM Sigplan, 1993.

[6] J. Chassin de Kergommeaux and P. Codognet. Parallel logic programming systems. *ACM Computing Surveys*, 26(3), September 1994.

[7] High Performance Fortran Forum. High performance Fortran language specification, version 1.0. *Scientific Programming*, 2(1), July 1993.

[8] MPI Forum. Mpi 1.1: A message-passing interface standard, 1995.

[9] Arnaud Lallouet. Semantics of a data-parallel logic language using the BSP execution model. In José L. Freire, editor, *APPIA-GULP-PRODE 1998, Joint Conference on Declarative Programming*, pages 1–13, Corunna, Spain, July 20-23 1998.

[10] Arnaud Lallouet. DP-Log : un langage de programmation logique data-parallèle. *Technique et Science Informatiques*, 18(2):233–261, February 1999.

[11] Arnaud Lallouet and Yann Le Guyadec. Contribution to the semantics of a data-parallel logic programming language. In Fernando Silva Vítor Santos Costa and Inês de Castro Dutra, editors, *Post International Logic Programming Symposium Workshop on Parallel Logic Programming Systems*, pages 32–41. Portland, Oregon, December 8 1995.

[12] F. Loulergue, G. Hains, and C. Foisy. A calculus of recursive-parallel BSP programs. In S. Gorlatch, editor, *CMPP'98 International Workshop on Constructive Methods for Parallel Programming*, Marstrand, Sweden, June 1998. Fakultät für Mathematik und Informatik, Univ. Passau, Germany, Technical Report MIP-9805.

[13] H. Millroth. Reforming compilation of logic programs. In *International Logic Programming Symposium*, San Diego, CA, 1991.

[14] Eric Monfroy and Jean-Hugues Réty. Chaotic iteration for distributed constraint propagation. In *Symposium on Applied Computing*, 1999.

[15] T. Nguyen and Yves Deville. A distributed arc-consistency algorithm. *Science of Computer Programming*, 30(1-2):227–250, January 1998.

[16] Enrico Pontelli, Gopal Gupta, Francesco Pulvirenti, and Alfredo Ferro. Automatic compile-time parallelization of prolog programs for dependent AND-parallelism. In *International Conference on Logic Programming*, pages 108–122, 1997.

[17] Vijay A. Saraswat. *Concurrent Constraint Programming*. ACM Doctoral Dissertation Awards. MIT Press, Cambridge, Massachusetts, 1993.

[18] Ehud Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):413–510, 1989.

[19] Donald A. Smith. MultiLog: Data or-parallel logic programming. In D.S. Warren, editor, *International Conference on Logic Programming*, pages 314–331, Budapest, Hungary, 1993.

[20] L. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[21] Makoto Yokoo, Edmund H. Durfee, Toru Ishida, and Kazuhiro Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 10(5):673–685, September/October 1998.