BSP Constraint Programming

Olivier Ballereau, Gaétan Hains, Arnaud Lallouet
Université d'Orléans — LIFO
BP 6759 — 45067 Orléans cedex 2
{Firstname.Name}@lifo.univ-orleans.fr

Abstract. We propose a new formalism for declarative parallel programming: constraint programming on vectors, whose positions are interpreted as explicit processes. We introduce the vector domain \( \mathcal{V}^* \) based on an underlying domain \( \mathcal{V} \) and propose a set of primitive constraints on this domain: scalar predicates are extended pointwise while communication is modeled by the new global constraint \( \equiv \) interpreted as a bulk-synchronous communication phase. We present two complete solvers on this domain but show that their distributed implementation is troublesome. To avoid this problem we also introduce an incomplete but more efficient parallel solver. The incomplete solver is 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 [10] 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 [9] are constraints between array elements and implicit processes.
1.1 Parallelism in logic and constraint programming

Task- and data-parallelism have been identified 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 optimize resources allocation is a non-trivial problem [18, 23]. Compilation towards a data-parallel execution model has been studied in the Reform model [20] or in the Multilog system [27].

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 [26] or with [25] constraints. Explicit data-(AND-)parallelism has been considered in Logic programming by two different approaches: the first one is a specialization of the &-Prolog system [12, 13, 11] where & is an annotation (possibly automatically determined) used to prescribe the parallel execution of two goals. The second one from which the current proposal inherits is our DP-LOG project [16, 15] and makes explicit use of vectors. Almost all these parallelization ideas have been lifted to CLP [29, 7, 8].

Most of the works in parallel constraint satisfaction consist in parallelizing a specific algorithm, like backtracking search [30] or arc-consistency computation [22]. Other recent works tend to give parallel constraint propagation a strong theoretical basis [21] but the data distribution problem is not really tackled. A promising approach for parallelization of arc-consistency that deserves further attention is [24]. Although the problem is P-complete [14], some acceleration should be observable on the average case. [24] also shows the importance of load balance on this highly irregular problem.

1.2 Data-parallelism and the BSP model

The basic data-parallel model proposes to compute on vectors instead of scalars, and thus provides a single centralized thread of computation. Computation is seen as a sequence of vector transformations and communication is simply 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 hand, the compiler has to fill the gap between the synchronous programming model and an asynchronous and efficient execution.

Traditionally, 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 [4]. This may be a 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) [28] and 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 execution model for our proposal.

A BSP computer (i.e., the BSP view of a parallel system) 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 \( h \).

![Fig. 1. The BSP scheme](image)

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 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:

\[
\text{Time}(s) = \max_{i: \text{processor}} w_i^{(s)} + \max_{i: \text{processor}} h_i^{(s)} \ast g + l
\]

where \( w_i^{(s)} \) is local processing time on processor \( i \) during superstep \( s \) and \( h_i^{(s)} = h_i^{(s)} + h_i^{(s)} \) 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 \( \sum_{s \in S} \text{Time}(s) \) of a BSP program composed of \( S \) supersteps is therefore a sum of 3 terms: \( W + H \ast g + S \ast l \) 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 distribution. 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 secondary here but can be found in [17]. 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^t the domain of vectors of elements of X. On a p processors machine, processor locations are integers between 0 and p−1 and actual vectors are cut up to the length of p. There are no vectors of vectors.

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 latter allow to define different computations at different process locations. 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 and synchronization.

The two types of constraints correspond to the two types of phases in BSP computing: asynchronous local and bulk-synchronous global communication. 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: \( \Rightarrow (X,L,Y) \) (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-processor machine, the constraint \( \Rightarrow (X,[1,0,0],Y) \) denotes the following equalities: \( Y_0 = X_1, Y_1 = X_0 \) and \( Y_2 = X_0 \) (figure 2). This encompasses the traditional 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 which it is enforced determines the solver. It is understood that more complex vector constraints may be predefined or built from this one.

Here is an example with a single constraint to express the broadcasting of a data to all processes: \(= \land (X, 0, Y)\). When processed, all \(Y_i\) unify with \(X_0\). Another (toy) example is given in the proof of proposition 3. 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 \(O(\log_2 n)\) 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 [1, 3] 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 distributed pointwise and involve no communication:

- a centralized solver called \texttt{CentralSolv}, which enforces full equality and is thus complete with respects to parallelism. Unfortunately, we show that its implementation on a distributed system is troublesome;
- a complete and distributed solver called \texttt{CopySolv};
- an incomplete but efficient solver called \texttt{MatchSolv}, which replaces unification by matching. Communication recovers the status of an event and opens the way to a BSP implementation. Programming with an incomplete solver is well-known in Prolog, for example with the functional arithmetic predicate “\(\text{is}\)”. The counterpart is that the programmer needs to be aware of the solver’s strategy to avoid errors at run-time.

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 \(= \land\)) is also subject to be constrained and can possibly
be non-ground. This means that we can specify a class of parallel algorithms with communications expressed as constraints, and let the system search for the right communication pattern. This can be compared with the availability of a parallelizing compiler at execution time. But its 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 [10] 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 [15].

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 [3] and similar to [1]). Let $\Sigma$ be a signature composed of a denumerable set of function symbols of all arities $F$, $\mathcal{V}$ be a finite set of variables $\mathcal{V} = \{V_1, \ldots, V_{n_v}\}$ large enough to represent the given problem and of a set of constraint predicates $\Pi$ including $\text{true}$, $\text{false}$ and $\text{=}$, As in [3], 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 $\Sigma$-structure composed of a domain (a complete lattice) $\mathbb{D}$ and of a set of functions and relations interpreting the symbols of $\Sigma$. The domain membership of a variable $v$ is included in $\mathbb{D}$. 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 (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 with the set of variables $\mathcal{V}$.

**Definition 2 (Solutions of a CSP).** A solution of a CSP $< \mathcal{C}, \mathcal{D} >$ is a tuple of $\mathcal{D}$ which satisfies the constraints.
The search space corresponds to sets of tuples and is ordered by the product ordering of the domains orderings, thus forming a complete lattice. Following [1], we can model the semantics of a solver as the greatest common fixed-point (gfp) of a set of reduction operators (i.e., correct, monotone and contractant operators). This gfp can be computed as the limit of a chaotic iteration of these operators [1].

2 Vectorial constraints

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

A solver for a vectorial constraint can be expressed by two types of rules:
- a \textit{vectorial} rule: \( \mathcal{AC} \rightarrow 2^{\mathcal{AC}} \);
- a \textit{localization} rule: \( \mathcal{AC} \rightarrow \mathcal{AC}^* \).

User-defined constraints are likely to belong to the first type. Since we want to model a 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 \( \mathcal{P} \) the set of pointwise extensions of scalar constraints plus a minimal set \( \mathcal{P}_0 \) of vectorial predicates that do not have a scalar counterpart:

\[
\mathcal{P} = \{ \overbar{c} = [c]_{i \in \mathbb{N}} \mid c \in \Pi \} \cup \mathcal{P}_0
\]

We will study \( \mathcal{P}_0 = \{ =? \} \) (=?, 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 \( L \). We have in mind that a real language should provide a \texttt{put} operation too.

A solver consists in providing localization rules for the vectorial constraints of \( \mathcal{P} \) 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:

\textit{Example 1.} Let us consider a constraint symbol \( p \in \mathcal{P} \) of arity 1. Thus \( p(X) \) is a valid vectorial atomic constraint formula if \( X = [X_i]_{i \in \mathbb{N}} \in \mathcal{V} \). 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:

\textbf{Definition 3 (Distributed object).} A \textbf{vectorial object} \( o \) is \textbf{distributed} if each projection on an index of its localization \( \text{loc}(o) \) 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 CEN\_TR\_SOLV: 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 CEN\_TR\_SOLV 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 COPY\_SOLV: 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 it has the worst space complexity of all, proportional to the cartesian product of the locations concerned by a communication. COPY\_SOLV 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 MATCH\_SOLV: this last solver models a highly desirable property: atomicity of communication but at the cost of the solver's incompleteness because equality is replaced by the weaker notion of term-matching. MATCH\_SOLV is designed to use a distributed memory and to avoid redundant computation. It is the best candidate for a BSP interpretation. It is interesting to relate this observation to the fact that syntactic unification is P-complete [7] but syntactic term-matching can be efficiently parallelized [7].

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 $\mathbf{c} = [c]_{i \in \mathbb{N}}$ be the vectorial pointwise extension of the scalar constraint predicate $c$, and $t_1, \ldots , t_n$ be vectorial terms.

**Rule 21 (Pointwise extension)**

\[
\frac{\mathbf{c}(loc(t_1))_i, \ldots , loc(t_n)_i}_{i \in \mathbb{N}} \quad \vdash \quad \mathbf{c}(t_1, \ldots , t_n)
\]

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 non-ground case is discussed below.

- CEN\_TR\_SOLV: 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 22 (CentralSolv COM)**

\[
\frac{\text{loc}(X)_{\text{loc}(L)_{k}} = \text{loc}(Y)_{i}}{\Rightarrow (X, L, Y)} \in \mathbb{N}
\]

As explained above, a BSP interpretation is difficult because local computation is not decoupled from communication. For example, let us consider the following constraint: \(\Rightarrow ([X_0, X_1, X_2], [1, 0, 0], [Y_0, Y_1, Y_2])\). 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 23 (CopySolv COM)**

\[
\frac{\{\text{loc}(X)_{\text{loc}(L)_{k}} = \text{loc}(Y)_{i} \mid k \in \mathbb{N}, \text{loc}(L)_{k} = \text{loc}(L)_{i} \text{ or } i = \text{loc}(L)_{k}\}}{\Rightarrow (X, L, Y)} \in \mathbb{N}
\]

The equation \(X_{L_k} = Y_i\) 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:

<table>
<thead>
<tr>
<th>(i)</th>
<th>(X_0)</th>
<th>(X_1)</th>
<th>(X_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(X_0 = Y_0)</td>
<td>(X_1 = Y_1)</td>
<td>(X_2 = Y_2)</td>
</tr>
<tr>
<td>1</td>
<td>(X_0 = Y_1)</td>
<td>(X_1 = Y_0)</td>
<td>(X_2 = Y_2)</td>
</tr>
<tr>
<td>2</td>
<td>(X_0 = Y_2)</td>
<td>(X_1 = Y_0)</td>
<td>(X_2 = Y_1)</td>
</tr>
</tbody>
</table>

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 occurrences 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 feasibility 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 \(\sigma\) such a substitution.

**Rule 24 (MatchSolv COM)**

\[
\frac{\sigma(\text{loc}(X)_{\text{loc}(L)_{k}}) = \text{loc}(Y)_{i}}{\Rightarrow (X, L, Y)} \in \mathbb{N}
\]

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-reversible 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 finite domain variable, 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 responsibility 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 \(VC \subseteq \mathcal{C}\) is a store of vectorial constraints, \(SC \in (2^{\mathcal{C}})^*\) 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 4 (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 computations and the asynchronous local ones. Namely, a VCO models a BSP communication and synchronization phase, 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 parallelism does not introduce more incompleteness than the underlying solver.

**Proposition 1.** Central\textsc{Solv} 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 \( \text{ CENTRAL SOLV}\), we get at each index \( i\) a set of scalar constraints \( SC_i\). Thus at index \( i\), \( SC_i \cup RC_i\) is a scalar store. Since all bindings are preserved, \(< VC, SC, D >\) and \(< \emptyset, [SC_i \cup RC_i]_{i \in N}, D >\) have the same solutions. Then completeness of the scalar solver implies completeness of its vectorial extension by \( \text{ CENTRAL SOLV}\).

**Proposition 2.** \( \text{ COPY SOLV} \) 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.** \( \text{ MATCH SOLV} \) 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:

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

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 us assume that the strategy is to reduce vectorial constraints first. Let \( S'\) be the vCSP obtained by rewriting the vectorial constraint with \( \text{ CENTRAL SOLV}\) and \( S''\) the one obtained by the same operation using \( \text{ MATCH SOLV}\). 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.

\[
\begin{array}{c}
S \\
\text{ CENTRAL SOLV} \\
\downarrow \\
S' \\
\text{ MATCH SOLV} \\
\uparrow \\
S''
\end{array}
\]

By using \( \text{ CENTRAL SOLV}\), we get:

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

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 \( \text{ MATCH SOLV}\), we get:

\[
S'' = \langle \emptyset, \{X_0 = a, Y_0 = a, Y_0 = X_1\}, \{X_1 = b, Y_1 = b, Y_1 = X_0\}\}, D >
\]
This leads to a domain \( D'' = \{X_0 \in \{a\}, X_1 \in \{b\}, Y_0 \in \{a\}, Y_1 \in \{b\}\} \) and we have \( D' \subset D'' \). 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 respect 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 can be done locally, i.e. without synchronization;
  2. apply local operators at each location up to a fixpoint;
  3. reduce \( =_l \) 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 corresponds to a call of the solver.

- the *data-flow* strategy:
  1. reduce the constraint \( =_l (X, L, Y) \) 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 expressive 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, vectors 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 \( \text{Nprocs} \)) 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 SPMD language to build the vCSP and to control the solving process. It uses a special statement called \texttt{begin LOCAL} (i) which binds the variable \texttt{i} to the current location (an explicit version of the constant \texttt{This}). We introduce here three basic operations on the store $<VC, SC, D>$:

- \texttt{AddVC(vc)} takes a vectorial constraint \texttt{vc} and adds it to the vectorial part \texttt{VC} of the store;
- \texttt{AddSC(sc)} takes a scalar constraint \texttt{sc} and adds it to the scalar store \texttt{SC} at the current location bound in the \texttt{begin LOCAL} statement. Constraints can be different at each location in order to model different asynchronous computations.
- \texttt{Solve(S)} starts the solver and can be parameterized by a strategy \texttt{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 \texttt{for} and \texttt{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.

\textit{Example 2.} The \texttt{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 3). The underlying domain of computation is \textit{FD} with the usual constraints ($+,-,\ldots$) and we suppose that the construction \texttt{vect of X} denotes the type of a vector of objects of type \texttt{X}. The program in figure 4 implements the \texttt{scan} algorithm. It uses an auxiliary recursive procedure \texttt{sc}. The meaning of the different \texttt{parts} is the following:

- \texttt{A} is the starting vector and \texttt{B} the result, \texttt{N} is used to count the recursion depth;
- the first \texttt{for} loop is used to compute the index of the next communication, i.e. at distance $\texttt{N}/2$ 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 3). 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 made unnecessary the call to \( \text{Solve}() \) in the procedure \( \text{sc} \). This is because the binding of \( C \) coming out of the return from the recursive call to \( \text{sc} \) would launch the communication and synchronization phase.

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 a simple performance model and a reasonably 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 MATCHSolve forces the programmer to have a knowledge of the evaluation strategy. This should not be understood as a weakness of this approach since a whole generation of Prolog programmers had to deal with an incomplete solver for arithmetics (the so-called is predicate) before the rise of CLP. The problem is similar here since the programmer has to design carefully the parallel structure of his algorithm and can take advantage of constraints for the sequential parts. This approach can also be viewed as a BSP coordination scheme for parallel solvers.

The requirement for a ground index of communication makes the language close to a functional one (see the language BS\( \lambda \) [19] for example), at least with regards to parallelism. But our language 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:

- a non-ground index of communication allows the programmer to leave some freedom in the placement of data, and opens the way to dynamic load balancing controlled by the program itself.
- the relational aspect of \( = \) is a challenging problem in order to preserve efficiency (see [24] on irregular problems for example). There could exist a compromise between communication and duplication (with the implicit migration of variables for example);
An implementation of these concepts as a C++ library is in progress [2].

Acknowledgments. The authors wish to thank an anonymous referee for pointing out many valuable references.

References


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 )
    begin LOCAL (i)
      if ( i >= M ) then
        AddSC ( L = i-M )
      else
        AddSC ( L = i )
      end if
    end LOCAL
    AddVC ( =?(C,L,D) )
    Solve( locally eager )
    begin LOCAL (i)
      if ( i >= M ) then
        AddSC ( E = D )
      else
        AddSC ( E = 0 )
      end if
    end LOCAL
    AddVC ( B = C+E )
  end

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

Fig. 4. The scan program