SVP – a Model Capturing Sets, Streams, and Parallelism

D. Stott Parker *
University of California, Los Angeles

Eric Simon and Patrick Valduriez
Projet Rodin, INRIA, Rocquencourt

Stott@cs.ucla.edu, Eric.Simon@inria.fr, Patrick.Valduriez@inria.fr

Abstract

We describe the SVP data model. The goal of SVP is to model both set and stream data, and to model parallelism in bulk data processing. SVP also shows promise for other parallel processing applications.

SVP models *collections*, which include sets and streams as special cases. Collections are represented as ordered tree structures, and divide-and-conquer mappings are easily defined on these structures. We show that many useful database mappings (queries) have a divide-and-conquer format when specified using collections, and that this specification exposes parallelism.

We formalize a class of divide-and-conquer mappings on collections called SVP-*transducers*. SVP-transducers generalize aggregates, set mappings, stream transductions, and scan computations. At the same time, they have a rigorous semantics based on continuity with respect to collection orderings, and permit implicit specification of both independent and pipeline parallelism.

1 Introduction

Achieving parallelism in bulk data processing is a relatively old problem, which has recently enjoyed a resurgence of interest. This paper proposes a new approach to addressing the problem. Since many of the issues involved are complex, we begin with first principles.

*This research supported by NSF grant IRI-8917907.

Proceedings of the 18th VLDB Conference
Vancouver, British Columbia, Canada 1992

1.1 Parallel Programming

Parallel programming aims at exploiting high-performance multiprocessor systems. An important objective is to be able to express the parallelism available in an application. There are essentially three ways to accomplish this:

- automatically detect parallelism in programs written with a sequential language (e.g., Fortran, OPS5);
- augment a language with explicit parallel constructs that exploit the computational capabilities of a parallel architecture (e.g., C* [17], Fortran90);
- create a new language in which parallelism can be expressed in an architecture-independent manner.

The first approach can be practical in the short-term, but is faced by many difficult problems. Among these, development of a parallelizing compiler is a major challenge. Methods for automatic program restructuring, and the parallelization of serial programs can produce good results for some programs (e.g., certain scientific programs), but most of the time the resulting speed-up is quite limited. For instance, experiments conducted with the OPS5 rule-based language revealed that in practice, the true speed-up achievable from parallelism was less than tenfold [7]. A related serious problem with this approach is that, in the final analysis, the serial programming paradigm does not encourage the use of parallel algorithms.

The second approach enables the programmer to express parallel constructs such as task creation and inter-task synchronization, thereby providing leverage over parallelism. Although this approach can lead to high-performance, it is generally too low-level and difficult for the programmer. Furthermore, the large variety of parallel architectures result in distinct, architecture-specific extensions to the original language.[1] In order

[1]Linda [4] is a notable exception of 'coordination language' with simple, language-independent parallel constructs, which can mate easily with many non-parallel languages.

to achieve efficient program execution, the programmer must first become acquainted with the programming paradigm dictated by the architecture of the target machine.

The third approach can combine the advantages of the other two. It can ease the task of programming while allowing the programmer to express non-sequential computation in a high-level way [16]. Once the programmer has specified the algorithmic aspects of his program using high-level programming constructs, automatic or semi-automatic methods can be used to derive a mapping from the computational requirements of the program to parallel hardware. The basis for this mapping is data partitioning (also called data-parallelism), whereby program data can be divided into fragments on which either the same instructions can be executed in parallel (with the SIMD computation model) or different instructions are executed in parallel (with the MIMD computation model). The regularity of the data structures available in the language permits exploitation of different forms of parallelism, such as independent and pipeline parallelism [9].

In this paper, we follow the third approach, and propose a model for parallel database programming where the primary sources for parallelism are parallel set and stream expressions. Parallel programming environments that follow this approach have recently been proposed. For example, in Paragon [5], the primary source for parallelism is parallel array expressions. Paragon is targeted to scientific programming applications and offers the essential features of parallel Fortran languages. Our model is targeted at database applications, and bulk data processing.

1.2 Parallelism for Bulk Data Processing

There are various forms of parallelism. Figure 1 shows four simple kinds of parallelism graphically.

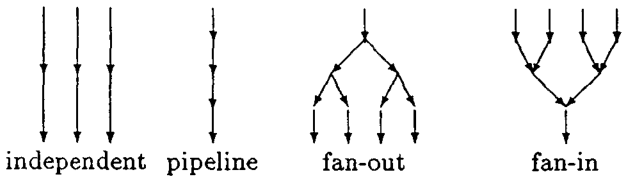

Figure 1: Types of Parallelism

A few key ideas can be derived from studying this figure, and applying the parallelism structures there to problems in bulk data processing:

- Division of problems is the essence of parallelism. Dividing into independent subproblems gives independent parallelism, while dividing into incremental computations gives pipeline parallelism. Set mappings naturally expose independent parallelism (a given instruction is independently applied to each element of a set) while stream mappings expose pipeline parallelism (some instructions are successively applied to each element of a stream). Thus, sets and streams suggest a divide-and-conquer format for specifying mappings which is implicitly also a format for specifying parallelism.

- Divide-and-conquer computations can be represented as series-parallel graphs. Series-parallel graphs [15] are defined recursively as graphs having one input and one output that can be constructed using two combination rules: series or parallel composition of the inputs and outputs. A typical series-parallel graph is shown in Figure 2. It models a situation where 1 and 2 are performed in parallel before 3, and 3 is performed before the parallel execution of 4, 5, and (6 followed by 7).

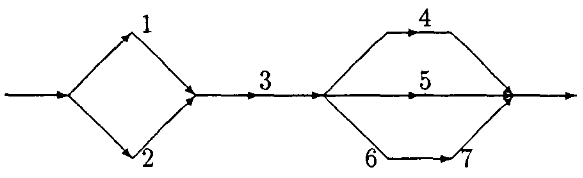

Figure 2: A Series-Parallel Graph

These graphs use only the constructs in Figure 1. Dividing a problem is represented by fan-out nodes in the graph, while conquering gathers results into a set (with independent parallelism), a stream (with pipeline parallelism), and/or an aggregate (with fan-in parallelism). Thus, divide-and-conquer solutions of problems often directly correspond to these four kinds of parallelism.

- Database applications provide excellent opportunities for parallel processing. The set-oriented nature of the relational model makes exploitation of independent parallelism natural [19]. In fact, set operators such as the relational algebra operators can often be naturally expressed as divide-and-conquer computations, as we will show in section 2.

These ideas raise hope for a parallel bulk data processing system that rests upon divide-and-conquer techniques. However, such a system must deal with several important technical issues to be viable.

A first problem is that the relational model offers no way to talk about order among data (e.g., sorted relations, or ordered tuples). Relational languages are therefore inadequate for specifying 'stream processing', in which ordered sequences of data are processed sequentially [13]. Pipeline parallelism is generally used, transparently to the user, in lower-level languages implementing relational algebra (e.g., PLERA [2], or PFAD [8]). However, higher-level relational interfaces do not permit streams to be exploited, preventing specification of stream computations and also pipeline parallelism.

A second problem is that parallel data processing requires effective data partitioning capabilities. Typically, a relational query (select-project-join expression) is translated into a low-level form of relational algebra with explicit (low-level) parallel constructs [2]. Data partitioning is used to spread the computation of relational algebra operators among parallel processors [1]. This partitioning is typically defined during the physical database design and then exploited by a compiler. Most of the time, a partitioned computation requires that processors exchange intermediate results in order to compute the final result.

In our view, data partitioning must be expressible by the programmer within a parallel database language. Specifying parallel computations over relations often requires specifying how data partitioning (fan-out parallelism) will be done and how distributed results will be collected (fan-in parallelism). This view is supported by recent results on data reduction for Datalog programs [21], in which rules are replaced by their per-processor specializations. These specialized rules include appropriate hash functions that capture partitioning information. This approach is very interesting in that it incurs no communication costs between processors. However, determining the appropriate hash functions to perform data reduction is still an open problem, known to be undecidable in some cases. It seems unlikely that database systems will be able to completely automate partitioning decisions.

Database models have been developed before that permit expression of both ordering among tuples and data partitioning. For example, the FAD language has operators that express various forms of fan-out and fan-in parallelism [6]. FAD is a strongly-typed set-oriented database language based on functional programming and relational algebra. It provides a fixed set of higher-order functions to aggregate functions, like the pump parametrized aggregate operator and the grouping operator. The pump operator applies a unary function to each element of a set, producing an intermediate set which is then 'reduced' to a single datum using a binary function that combines the intermediate set elements. Indeed, pump naturally expresses a special case of fan-out and fan-in parallelism. At the same time, the group operator permits set partitioning.

1.3 Goals of the Paper

Based on the observations above, our main goal is to develop a data model, called SVP, that supports both:

- ordered and unordered (stream and set) data representations;
- a formal semantics for divide-and-conquer computations on sets and streams to express independent (set) and pipeline (stream) parallelism.

This model is intended to serve as a formal foundation for defining parallel database languages in which parallelism is specified at a high-level.

The SVP data model has the following features:

- SVP values either are *collections* (a generalization of sets and streams), or are tuples of SVP values. Collections are represented as ordered binary tree structures. Intuitively, lists can represent streams, balanced trees can represent sets, and ordered binary trees can represent either.
- SVP allows restricted divide-and-conquer mappings on SVP values. In this paper these mappings are specified with recursive functional equations. They generalize other specification techniques, including restricted higher-order mappings like the reduction operator in APL [10] and the pump operator in FAD [6], list comprehensions and elegant variants thereof [20], and series-parallel computation graphs [15].
- Parallelism in the dividing and conquering is specified using both the structure of the data, and the structure of the divide-and-conquer mapping: dividing-parallelism is specified by the data, and conquering-parallelism is specified by the mapping. Partitioning can always be used to modify data structure, and thus affect dividing-parallelism.

The paper is organized as follows. Section 2 investigates the relationships between set and stream processing, and demonstrates with examples how divide-and-conquer mappings are important for data processing. Section 3 presents the SVP model and defines SVP values, types, and mappings. Section 4 then gives examples of SVP mappings for expressing relational algebra operators, grouping and aggregate operators. Finally, Section 5 concludes the paper and summarizes the contributions of the SVP model. A more comprehensive presentation of SVP is given in [14].

2 Set and Stream Processing

Let us clarify first what set processing and stream processing are, and then study how they might be integrated in a parallel processing model.

2.1 Sets and Streams

For the purposes of this paper, we will rely on similar formulations of sets and streams.

Given a set of values D, we will write 2^D to denote the finite or infinite sets on D, and write $D^\bullet$ to denote the finite or infinite streams on D. Sets use the following notation:

1. $\{\}$ is a set (the empty set);
2. $\{x\}$ is a set, for any value x;
3. Finite sets are written with set braces, as with: $\{1,2,3\}$.
4. The union $S_1 \cup S_2$ is a set, if S_1 and S_2 are sets. (We use the symbol '$\cup$' for *disjoint* set union in this paper, except where indicated otherwise.)
5. The *cardinality* $\|S\|$ of any set S is the number of values in the set.

Streams analogously use the following notation:

1. $[\,]$ is a stream (the empty stream);
2. $[x]$ is a stream, for any value x;
3. Finite streams are written with square braces, as with: $[1,2,3]$.
4. The concatenation $S_1 \bullet S_2$ is a stream, if S_1 and S_2 are streams. (We use the symbol '$\bullet$' for stream concatenation ('append') in this paper.)
5. The *length* $|S|$ of any stream S is the number of values in the stream.

As usual, set union is associative and commutative, where stream concatenation is only associative.

Although streams are formalized here like strings, with a concatenation operator, they are accessible like *lists*. Specifically, every nonempty stream S satisfies

$$S \;=\; (h \cdot T)$$

where h is the *head* of S, and T is the *tail* of S. Here h will be a value, and T will be a stream. The constructor symbol '$\cdot$' ('cons') can be viewed as an operator that combines a value and a stream into a stream. The single-element stream $[x]$ is actually a shorthand for $(x \cdot [\,])$, and $[1,2,3]$ is a shorthand for $(1 \cdot 2 \cdot 3 \cdot [\,])$. All finite streams are terminated explicitly with $[\,]$.

One more bit of notation will be useful. We use parentheses to set off *tuples* (vectors). Thus

$$(a, 1, b)$$

denotes a 3-tuple (tuple with 3 elements).

2.2 Set and Stream Mappings

Consider the following mappings, using the formalization of sets and streams given above. We would like to be able to formalize these mappings in our model.

The equations

$$\begin{array}{lcl} count(\{\}) & = & 0 \\ count(\{x\}) & = & 1 \\ count(S_1 \cup S_2) & = & count(S_1) + count(S_2) \end{array}$$

define a set mapping (in this case an aggregate) recursively. This definition reflects parallelism that can be obtained by computing cardinalities of subsets independently. For example, in the computation

$$\begin{array}{lcl} count(\{a,b,c\}) & = & count(\{a,b\}) + count(\{c\}) \\ & = & count(\{a\}) + count(\{b\}) \\ & & \;+ count(\{c\}) \\ & = & 1 + 1 + 1 \\ & = & 3 \end{array}$$

we have ultimately three independent parallel threads that are 'fanned-in' to an aggregate.

Consider now the stream mapping

$$\begin{array}{lcl} \mathit{diffs}([\,]) & = & [\,] \\ \mathit{diffs}(x \cdot [\,]) & = & [\,] \\ \mathit{diffs}(x \cdot y \cdot S) & = & (y - x) \cdot \mathit{diffs}(y \cdot S) \end{array}$$

This yields a stream of the differences between adjacent elements in the input stream. For example:

$$\begin{array}{ll} \multicolumn{2}{l}{\mathit{diffs}(98 \cdot 99 \cdot 97 \cdot 97 \cdot 99 \cdot 96 \cdot [\,])} \\ = & +1 \cdot \mathit{diffs}(99 \cdot 97 \cdot 97 \cdot 99 \cdot 96 \cdot [\,]) \\ = & +1 \cdot -2 \cdot \mathit{diffs}(97 \cdot 97 \cdot 99 \cdot 96 \cdot [\,]) \\ = & +1 \cdot -2 \cdot \;0 \cdot \mathit{diffs}(97 \cdot 99 \cdot 96 \cdot [\,]) \\ = & +1 \cdot -2 \cdot \;0 \cdot +2 \cdot \mathit{diffs}(99 \cdot 96 \cdot [\,]) \\ = & +1 \cdot -2 \cdot \;0 \cdot +2 \cdot -3 \cdot \mathit{diffs}(96 \cdot [\,]) \\ = & +1 \cdot -2 \cdot \;0 \cdot +2 \cdot -3 \cdot [\,]. \end{array}$$

This mapping implements a kind of 'automaton', or 'transducer', that scans the stream of values and translates it to a stream of pairwise differences. These transducer mappings are important in analyzing streams, but are (at best) quite challenging to implement with a set-oriented model.

2.3 Composition of Set and Stream Mappings

Functional mappings can be composed naturally. We consider a simple example that illustrates how composition of set and stream mappings allows us to answer arbitrary queries by composing a few elementary mappings.

Example: Areas of Convex Polygons

We are given a convex polygon as a stream of points in (x, y)-coordinate form that trace out the boundary of the polygon, and the problem is to compute the total area of the polygon.

This problem can be solved by triangulating the polygon, i.e., cutting the polygon into triangles, and computing the total area of the triangles. Specifically we can transform the stream of points of the polygon

$$[(x_1, y_1),\ (x_2, y_2),\ \cdots\ (x_n, y_n)]$$

into a set of triangles (triples of points)

$$\begin{array}{ll} \{ & ((x_1,y_1),(x_2,y_2),(x_3,y_3)), \\ & ((x_1,y_1),(x_3,y_3),(x_4,y_4)),\ \cdots \\ & ((x_1,y_1),(x_{n-1},y_{n-1}),(x_n,y_n))\ \} \end{array}$$

and then compute the sum of the areas of the triangles.

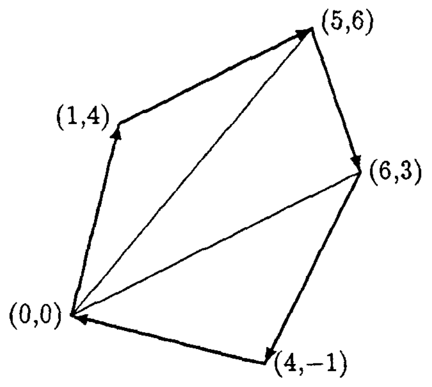

Figure 3: Triangulation of a Convex Polygon

For example the polygon given by the stream of points

$$[(0,0),(1,4),(5,6),(6,3),(4,-1)]$$

corresponds to the set of triangles

$$\begin{array}{ll} \{ & ((0,0),(1,4),(5,6)), \\ & ((0,0),(5,6),(6,3)), \\ & ((0,0),(6,3),(4,-1))\ \} \end{array}$$

having respective areas[2]

$$\{\ 7.0,\ 10.5,\ 9.0\ \}$$

and a total area of 26.5. See Figure 3. This is expressible as

$$\begin{array}{lcl} polygon & = & [(0,0),(1,4),(5,6),(6,3),(4,-1)] \\ total\_area & = & sum(\ areas(\ ts(\ polygon\ )\ )\ ) \end{array}$$

where we define ts (triangles) with

$$\begin{array}{lcl} ts([\,]) & = & \{\} \\ ts(p_0 \cdot [\,]) & = & \{\} \\ ts(p_0 \cdot p_1 \cdot [\,]) & = & \{\} \\ ts(p_0 \cdot p_1 \cdot p_2 \cdot S) & = & \{(p_0,p_1,p_2)\} \cup ts(p_0 \cdot p_2 \cdot S). \end{array}$$

and the aggregate functions needed are:

$$\begin{array}{lcl} sum(\{\}) & = & 0 \\ sum(\{x\}) & = & x \\ sum(S_1 \cup S_2) & = & sum(S_1) + sum(S_2). \end{array}$$

$$\begin{array}{lcl} areas(\{\}) & = & \{\} \\ areas(\{(p_0,p_1,p_2)\}) & = & \{area(p_0,p_1,p_2)\} \\ areas(S_1 \cup S_2) & = & areas(S_1) \cup areas(S_2). \end{array}$$

$$dist((x_1,y_1),(x_2,y_2)) = \sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$$

$$\begin{array}{lcl} area(p_0,p_1,p_2) & = & \sqrt{s(s-a)(s-b)(s-c)} \\ \text{where:} & & a = dist(p_0,p_1) \\ & & b = dist(p_1,p_2) \\ & & c = dist(p_2,p_0) \\ & & s = (a+b+c)/2. \end{array}$$

The example here hopefully makes two points: First, a model based on composing mappings on sets and streams is sufficient to develop expressive database systems – significantly more expressive than standard DBMS. Although the example problem above is not easy to solve with standard DBMS, the structures involved (sets of streams, etc.) are easy to understand, and the queries are easy to state, and easy to state mathematically.

Second, the structure of the data (sets and streams) directly reflects parallelism in the data processing required. Both pipeline and independent parallelism are crucial in data processing, and these kinds of parallelism can be made evident by the stream or set structure of the data.

2.4 Perspective: Divide-and-Conquer Mappings

Our goal is to develop a formal data model that will support all of the mappings shown earlier. The challenge comes in developing a model that encourages

[2] The Heron formula for the area of a triangle whose sides have respective lengths a, b, c is given by

$$\sqrt{s(s-a)(s-b)(s-c)}$$

where $s = (a+b+c)/2$.

optimization and extraction of parallelism and supports at least the set and stream mappings shown earlier.

The mappings above are all 'divide-and-conquer' mappings, of three kinds:

1. *Aggregates*
   Aggregates can be described as functions of sets with the format
   $$\begin{array}{lcl} f(\{\}) & = & id \\ f(\{x\}) & = & h(x) \\ f(S_1 \cup S_2) & = & f(S_1)\ \theta\ f(S_2) \end{array}$$
   where θ is an associative, commutative operator whose identity is *id*, and h is a function that yields values of the type taken by θ.

2. *Set Mappings*
   Set mappings have the divide-and-conquer form
   $$\begin{array}{lcl} f(\{\}) & = & \{\} \\ f(\{x\}) & = & h(x) \\ f(S_1 \cup S_2) & = & f(S_1) \cup f(S_2) \end{array}$$
   where h is a set-valued function.

3. *Stream Transducers*
   Stream mappings like *diffs* and *triangles* are naturally characterized as 'automata' that incrementally translate their input. We will call this kind of mapping a *transducer.*

   In general form, we define a stream transducer f in terms of two function parameters, δ and h, and an iterative control structure F:
   $$\begin{array}{lcl} f(S) & = & F(q_0, S) \\ F(q, []) & = & h(q, []) \\ F(q, x \cdot S) & = & h(q, x) \bullet F(\delta(q, x), S). \end{array}$$
   Here intuitively there is a set of 'states', q_0 is the 'initial state', δ is a 'state transition function' that maps a (state,input)-pair to a new state, and $h(q, x)$ is the output stream produced in state q with input x. So, in particular, $h(q, [])$ is the output stream produced in state q when no input remains. Thus F maps a (state,stream)-pair into a stream.

   We call f a stream transducer because its definition directly mirrors the definition of a finite state transducer – a finite automaton that produces output given its current input symbol and current state.

An obvious question facing us now is:

What is a useful generalization of aggregates, set mappings, and stream transducers, that can be applied successfully in parallel data processing?

The SVP model described next offers one answer to this question.

3 The SVP Model

The goals of SVP require a model in which collections (both stream collections and (multi-)set collections) can be expressed, and mappings on these collections can be defined. For simplicity, and without loss of generality, we limit ourselves to a value-based model – i.e., objects are not handled by the model currently.

3.1 SVP Values

SVP models two kinds of values: *atomic values*, and *constructed values.* Constructed values represent complex structures, or nested values, and can be either *tuples* or *collections.* Tuples are typically heterogeneous structures with a small number of elements, while collections are typically homogeneous structures with a large number of elements.

Values are recursively defined as follows:

- Any atom is a SVP value.
- Any finite tuple $(v_1, \ldots, v_n)$ of SVP values $v_1, \ldots, v_n$ is a SVP value. A tuple with one atom is called a 1-tuple, a tuple with two atoms is called a 2-tuple, etc.
- Any collection is a SVP value.

In SVP, *collections* are recursively defined as follows:

- $\langle\rangle$ is the empty collection.
- $\langle v\rangle$ is a unit collection if v is a SVP value.
- $S_1 \diamond S_2$ is a collection if S_1 and S_2 are *nonempty* SVP collections. Collections are forbidden to properly contain the empty collection.

This definition allows SVP collections to model many structures of interest, including:

- *sets*
  The SVP-collection $(\langle 1\rangle \diamond \langle 2\rangle) \diamond (\langle 3\rangle \diamond \langle 4\rangle)$ represents the set {1,2,3,4} as a balanced binary tree.
- *streams and sequences*
  A stream is a *sequence* (right-linear tree) that, if finite, is terminated with []. The SVP-collection $\langle 1\rangle \diamond (\langle 2\rangle \diamond (\langle 3\rangle \diamond (\langle 4\rangle \diamond [])))$ represents the stream [1,2,3,4] as a list-like structure.

3.2 SVP Types

Database systems support homogeneous collections of data. SVP does also, resting on a simple polymorphic type system that defines the following *value types*:

- **atom**

- **tuple**$(T_1, \ldots, T_n)$ is a constructed value type, if each T_i is a value type.
- **collection**(T) is a homogeneous collection type, if T is a value type.

Thus the following are homogeneous collection types: **collection(atom)**, **collection(collection(atom))**, **collection(tuple(atom,collection(atom)))**, etc.

3.3 SVP Mappings

Data models typically specify how all permissible mappings can be constructed. We take a different approach. SVP imposes few restrictions on atomic value mappings – essentially any mapping on atomic values is permitted. However, SVP requires *all* collection mappings to be SVP-*transducers*. This class of mappings is powerful, and suited to bulk data processing on homogeneous collections. At the same time SVP-transducers are restrictive enough to permit optimization and extraction of parallelism.

3.3.1 Basic SVP Mappings

SVP explicitly provides the following basic mappings:

- *Constructors*
  - *tupling* $(\langle\cdots\rangle)$
    If $x_1, \ldots, x_n$ are values of types $T_1, \ldots, T_n$, then $(x_1, \ldots, x_n)$ is of type **tuple**$(T_1, \ldots, T_n)$.
  - *collection* $(\diamond)$
    If S_1 and S_2 are of type **collection**(T), $S_1 \diamond S_2$ is also. The constructor $\diamond$ is used both as a constructor and as an operator that guarantees its result is a properly-formed collection, so that $\langle\rangle \diamond S \;=\; S \diamond \langle\rangle \;=\; S$, but otherwise $S_1 \diamond S_2$ yields the ordered binary tree with left subtree S_1 and right subtree S_2. That is, the *expression* $S_1 \diamond S_2$ evaluates to the *structure* $S_1 \diamond S_2$ precisely when S_1 and S_2 are $\langle\rangle$-free collections. This may be slightly confusing at first, but avoids introducing a new operator.
- *Deconstructors*
  SVP provides the following type-membership predicates for SVP values v:
  - atom(v) – whether v is an atomic value.
  - tuple(v) – whether v is a tuple.
  - collection(v) – whether v is a collection.
  - emptycollection(v) – whether v is $\langle\rangle$.
  - unitcollection(v) – whether v is $\langle x\rangle$ for some x.

  Furthermore, the following functions are provided:

  - unitcollectionvalue(S) – x, if $S = \langle x\rangle$.
  - arity(t) – number n of elements in a tuple t.
  - $t[i]$ – tuple subscripting. If t is a tuple $(x_1, \ldots, x_n)$ of type **tuple**$(T_1, \ldots, T_n)$, and i is an integer between 1 and n, then $t[i]$ yields x_i, of type T_i.

  Note only tuple deconstructors are allowed to appear in user-defined mappings. Deconstructors are not provided for collections. *The only construct for iterating over collections is the SVP-transducer.*

SVP collections can be regarded as an abstract data type, whose only defined operations are the collection constructor, the limited collection deconstructors just defined, and SVP-transducers introduced next.

3.3.2 SVP-Transducers

SVP-transducers specify mappings of collections as:

1. the mapping of the elements in the input collection;
2. the collecting of the resulting mapped input elements into an output.

SVP-transducers are capable of implementing all the example mappings shown earlier.

A mapping f on SVP collections is an *SVP-transducer* if it is the composition of one or more functions, each of which can be written in the following divide-and-conquer form:

$$\begin{aligned} f(S) &= F(Q_0, \rho(S)) \\ F(Q, \langle\rangle) &= id_\theta \\ F(Q, \langle x\rangle) &= h(Q, x) \\ F(Q, S_1 \diamond S_2) &= F(Q, \rho(S_1)) \\ &\quad\ \theta\ F(\delta(Q, S_1), \rho(S_2)). \end{aligned}$$

Here Q_0 is an arbitrary fixed value, ρ is either the identity mapping or an SVP-transducer, and h, θ, and δ are arbitrary SVP mappings of two arguments. We have written θ as a binary operator.

We also permit f, F, and h to take additional arguments not shown explicitly here; in particular f can be a function of other parameters besides the collection S (including other collection parameters). Also, the Q argument can be omitted if it is not used by h or δ.

The mapping $\rho(S)$ typically performs *data partitioning* on the collection S. Common values for $\rho(S)$ include just S (the identity mapping, with no repartitioning), and the operator partition(P,S), in which P is a predicate defining a splitting of S into two parts $S_1 \diamond S_2$, the first for which P yields the value **true**, and the latter the value **false** (assuming both are nonempty). Partitioning operators will be investigated later.

The operator θ must be of type $T \times T \rightarrow T$, for some SVP type T which must be declared. For example, the $\diamond$ collector here is restricted to work on operands of type **collection**, and produce a **collection**. This type also restricts the values produced by the h function. For example, when θ is '$\diamond$', h must produce a **collection**.

When θ is a complete operator with a left identity id_θ, we call θ a *collector*. The table below gives examples of collectors. Parallel evaluation of associative collector expressions is sometimes called *parallel prefix computation* [12]. Other properties of collectors (such as *C*ommutativity, *I*dempotency) can be exploited to obtain greater parallelism.

θ	*Result Type*	id_θ	*Properties*
$\diamond$	**collection**	$\langle\rangle$	—
$\star$	**collection**	$\langle\rangle$	A
$+$	**atom**	0	A, C
$*$	**atom**	1	A, C
max	**atom**	$-\infty$	A, C, I
min	**atom**	$+\infty$	A, C, I

Here $\diamond$ is the collection-forming operator, and $\star$ is the append operator for collections. Thus $\langle\rangle$ is the identity for $\star$, and when S and T are nonempty $S \star T$ is the collection consisting of S but with the rightmost leaf $\langle x\rangle$ of S replaced by $\langle x\rangle \diamond T$.

In the general SVP-transducer, θ is permitted to be an arbitrary operator, and id_θ an arbitrary value. However, we shall mainly deal in the rest of the paper with transducers in which θ is a collector.

The important restriction this form imposes is that the computation over the collection be performed by a divide-and-conquer traversal. Basically, SVP-transducers provide a ('large') control structure that directs a function on values ('small' data manipulation) to be applied as needed. This control structure can be viewed as a generalized scan over a collection, together with gathering of scan results, where both the scan and the final gathering may be performed using parallel techniques.

3.3.3 Definition of SVP Mappings

Further mappings can be built using the basic mappings and SVP-transducers defined above. However, SVP mappings are restricted in the following ways:

- All mappings arguments are typed, and all mappings must be well-typed. In particular, constructors and deconstructors can be applied only to operands of the appropriate type.
- The only operators that can be applied to collections are SVP-transducers, and the constructor and deconstructors defined earlier.
- An SVP mapping used as a parameter can invoke at most a bounded number of SVP-transducers.

These restrictions limit the power of SVP-mappings. Without this restriction, transducers can be used (for example) to implement arbitrary iterative deconstructors or Turing machines. General query optimization is infeasible. With the restriction, transducers are restricted to implement only bounded networks of transductions on collections, and optimization is feasible.

3.4 Properties of the SVP Model

The SVP model was designed to address the goals given at the outset. This section has provided a variety of examples using SVP that will help motivate its being the way it is. To help clarify, however, below are perceptions about SVP that shaped its current form.

1. The definition of SVP transducers is a direct generalization of the earlier definitions of set mappings, aggregates, and stream transducers. Furthermore, it is a modest generalization, covering essentials only.

2. SVP collections are ordered binary trees because this is enough to let them represent both recursive problem division, and also sets and streams. The ordering of the leaves in the tree is the stream ordering. Otherwise, the actual topology of the tree is important precisely in that it determines problem division (and conquering).

3. When θ is an associative operator, the expression
$$x_1 \; \theta \; x_2 \; \theta \; \cdots \; \theta \; x_n$$
gives the same result regardless of the way it is parenthesized, i.e., regardless of the topology of the expression tree. The result is affected only by the ordering of the x_i. Thus *associative operators are naturally stream mappings*. Furthermore, when θ is associative and commutative, the result is the same regardless of the ordering of the x_i. Thus *associative, commutative operators are naturally set mappings.*

4. The definition of SVP transducers leads to a very nice theory, based on the idea of *structure preservation*. Divide-and-conquer techniques work only when the data can be divided in a way that represents some underlying 'structure'. SVP collections allow us to model various kinds of structure important in data processing, including sort ordering and physical data partitioning. It is also possible to generalize the classic work of Kahn for continuous functions on sequences [11] to work for continuous functions on collections. The basic idea is

that prefix-continuous functions on sequences[3] are exactly those functions that yield pipeline parallelism. Stream-continuity gives pipeline parallelism, and set-continuity gives independent parallelism.

5. SVP is a *model*. It is not intended as a full database system and query language, but rather the sketch of a larger, full-featured system. It permits many practical extensions, including n-ary trees (not just binary) for representing collections, permitting n-ary problem division, and if-then-else constructs, permitting early termination of a scan over a collection. Earlier versions of SVP permitted these extensions explicitly, but the result was a more complicated model. The current model is simple, and encourages a transducer style with more parallelism.

4 Examples of SVP-Transducers

To illustrate the power of SVP, we show how basic data processing primitives can be expressed as SVP transducers. These examples have all been implemented and execute correctly in an SVP prototype written in Bop, a rewrite rule language developed at UCLA.

4.1 Three Basic Transducers

SVP offers essentially three functionals:

- collect (bottom-up accumulations)

$$\begin{aligned}
\mathsf{collect}(\theta,\ id,\ \langle\rangle) &= id\\
\mathsf{collect}(\theta,\ id,\ \langle x\rangle) &= x\\
\mathsf{collect}(\theta,\ id,\ S_1 \diamond S_2) &= \mathsf{collect}(\theta,\ id,\ S_1)\\
&\qquad \theta\ \mathsf{collect}(\theta,\ id,\ S_2).
\end{aligned}$$

- transduce (general transductions on collections)

$$\begin{aligned}
\mathsf{transduce}(h,\ \delta,\ Q,\ \langle\rangle) &= \langle\rangle\\
\mathsf{transduce}(h,\ \delta,\ Q,\ \langle x\rangle) &= h(Q,x)\\
\mathsf{transduce}(h,\ \delta,\ Q,\ S_1 \diamond S_2) &=\\
&\mathsf{transduce}(h,\ \delta,\ Q,\ S_1)\\
&\quad \diamond \mathsf{transduce}(h,\ \delta,\ \delta(Q,S_1),\ S_2).
\end{aligned}$$

- restructure (top-down reorganization)

$$\begin{aligned}
\mathsf{restructure}(\rho,\ S) &= \mathsf{R}(\rho,\ \rho(S))\\
\mathsf{R}(\rho,\ \langle\rangle) &= \langle\rangle\\
\mathsf{R}(\rho,\ \langle x\rangle) &= \langle x\rangle\\
\mathsf{R}(\rho,\ S_1 \diamond S_2) &= \mathsf{R}(\rho,\ \rho(S_1)) \diamond \mathsf{R}(\rho,\ \rho(S_2)).
\end{aligned}$$

[3]Prefix-continuous functions on sequences are functions that are monotone with respect to the sequence prefix ordering, so giving the function more input cannot result in the function's producing less output, and also cannot wait indefinitely before producing an output. Fixed-point results for continuous functions lead to a rigorous fixed point semantics for networks of SVP-transducers, even for cyclic networks.

Most SVP transducers are definable in terms of collect, transduce, and restructure. Note that whenever $\delta(Q,S) = \delta'(Q, \mathsf{restructure}(\rho,S))$ for all Q and S, the SVP transducer expression $f(S)$ with the defining recursion

$$\begin{aligned}
f(S) &= F(Q_0, \rho(S))\\
F(Q,\ \langle\rangle\,) &= id_\theta\\
F(Q,\ \langle x\rangle\,) &= h(Q,x)\\
F(Q,\ S_1 \diamond S_2\,) &= F(Q,\ \rho(S_1))\\
&\qquad \theta\ F(\ \delta(Q,S_1),\ \rho(S_2)\ ).
\end{aligned}$$

is equivalent to the expression

$$\mathsf{collect}(\theta,\ id_\theta,\ \mathsf{transduce}(h,\ \delta',\ Q_0,\ \mathsf{restructure}(\rho,\ S))).$$

4.2 Restructuring, Partitioning and Grouping

Often it is useful to transform of one structure to another. Reorganization can be done both with restructure and collect in many ways. For example, sequence — a transducer that maps an input collection to a flattened sequence — is definable as

$$\mathsf{sequence}(S) \quad = \quad \mathsf{restructure}(\,\mathsf{first\_rest},\ S\,)$$

where first_rest is a transducer that partitions a collection into first and remaining elements. Alternatively:

$$\mathsf{sequence}(S) \quad = \quad \mathsf{collect}(\ \star,\ \langle\rangle,\ S\,)$$

— collections can be flattened by recursive appending.

The restructure functional is useful for *top-down* reorganization of collections. For example, if ρ splits a tree into two trees of equal cardinality, then $\mathsf{restructure}(\rho,S)$ produces a balanced version of S. Also, if ρ partitions a tree into two subtrees by comparing with a median-estimate key value, then $\mathsf{restructure}(\rho,S)$ sorts a tree S by that key.

We can restructure any collection into a balanced collection (balanced tree) by repeated halving:

$$\begin{aligned}
\mathsf{balance}(S) &= \mathsf{restructure}(\mathsf{split}, S)\\
\mathsf{split}(S) &= \mathsf{pcoll}(\ \mathsf{halves}(1, \mathsf{count}(S),\\
&\qquad\qquad \mathsf{sequence}(S))\,)\\
\mathsf{halves}(i,n,\langle\rangle) &= (\,\langle\rangle,\ \langle\rangle\,)\\
\mathsf{halves}(i,n,\langle x\rangle) &= \textbf{if } i \le n/2 \textbf{ then } (\,\langle x\rangle,\ \langle\rangle\,)\\
&\qquad \textbf{else } (\,\langle\rangle,\ \langle x\rangle\,)\\
\mathsf{halves}(i,n,S_1 \diamond S_2) &= \mathsf{halves}(i,n,S_1)\\
&\qquad \mathsf{pcomb}\ \ \mathsf{halves}(i+1,n,S_2).
\end{aligned}$$

Here we need several operators on partitions:

$$\begin{aligned}
\mathsf{pcoll}(\,(S_1,S_2)\,) &= S_1 \diamond S_2\\
(P_1,P_2)\ \mathsf{pcomb}\ (Q_1,Q_2) &= (P_1 \diamond Q_1,\ P_2 \diamond Q_2).
\end{aligned}$$

Note pcomb is a binary operator with identity $(\langle\rangle, \langle\rangle)$, and is thus a collector.

Collecting is useful for *bottom-up* restructuring of collections. Partitioning can be performed with collecting:

$$\begin{aligned}&\mathsf{partition}(P, S) \;=\\ &\qquad \mathsf{pcoll}(\,\mathsf{collect}(\,\mathsf{pcomb},\ (\langle\rangle, \langle\rangle),\ \mathsf{parts}(P, S)\,)\,)\end{aligned}$$

$$\begin{aligned}\mathsf{parts}(P, \langle\rangle) &= \langle\rangle\\ \mathsf{parts}(P, \langle x\rangle) &= \textbf{if } P(x) \ \textbf{then } (\,\langle x\rangle,\ \langle\rangle\,)\\ &\qquad\qquad\ \ \textbf{else } (\,\langle\rangle,\ \langle x\rangle\,)\\ \mathsf{parts}(P, S_1 \diamond S_2) &= \mathsf{parts}(P, S_1) \diamond \mathsf{parts}(P, S_2).\end{aligned}$$

This partitions by splitting collection S into two subcollections (S_1, S_2) according to predicate P, and using pcoll to recombine these into a collection.

Grouping can also be expressed as collecting. The grouping operation takes a set S and a characteristic function h (say a hash function or a key function) as input, and produces as output a set of 2-tuples (k, S_i), $1 \leq i \leq p$, where k is the value obtained by applying h to any member of the set S_i, and S is partitioned by the S_i subsets. This can be implemented as an SVP-mapping that applies h to the x values in the input and accumulates the resulting (k, x) values into buckets [14].

4.3 Algebraic Operators

In FAD [6], the parameterized aggregate operator pump(h,θ,id_θ,S) is defined to yield

$$\begin{array}{ll} id_\theta & \text{if } S = \{\}\\ h(x_1)\ \theta\ \ldots\ \theta\ h(x_n) & \text{if } S = \{x_1, \ldots, x_n\}\end{array}$$

where θ is an associative, commutative binary operator, with identity id_θ. It is definable as an SVP transducer:

$$\begin{aligned}\mathsf{pump}(h, \theta, id_\theta, \langle\rangle) &= id_\theta\\ \mathsf{pump}(h, \theta, id_\theta, \langle x\rangle) &= h(x)\\ \mathsf{pump}(h, \theta, id_\theta, S_1 \diamond S_2) &= \mathsf{pump}(h, \theta, id_\theta, S_1)\\ &\quad\ \theta\ \ \mathsf{pump}(h, \theta, id_\theta, S_2).\end{aligned}$$

The list1 operator in [3] is similar. The APL reduction operator [10] allows non-associative, non-commutative operators. In particular, if θ is a binary operator and $S = (x_1, x_2, \ldots, x_n)$ is a vector, the APL reduction of S by θ is $\theta/S \ = \ ((\cdots (x_1\ \theta\ x_2)\ \theta\ \cdots)\ \theta\ x_n)$. This is an aggregation that reflects the ordering of the input. It can also be written as a SVP-transducer, assuming the input is in left-linear form:

$$\begin{aligned}\mathsf{APLreduction}(\theta, \langle\rangle) &= \langle\rangle\\ \mathsf{APLreduction}(\theta, \langle x\rangle) &= x\\ \mathsf{APLreduction}(\theta, S_1 \diamond S_2) &= \mathsf{APLreduction}(\theta, S_1)\\ &\quad\ \theta\ \mathsf{APLreduction}(\theta, S_2).\end{aligned}$$

Furthermore, any collection can be restructured to left-linear form with a simple SVP-mapping [14].

4.4 Joins

Surprisingly, important n-ary operations like joins can be implemented with transducers! In fact, interesting join algorithms can be developed.

Let us define a general join algorithm. One general specification for joins would be something like:

$$\begin{aligned}combine(R, S) \ = \ \{\, &\mathsf{RESULT}(r, s) \ \mid\\ &\mathsf{TEST}(r, s)\ \wedge\ r \in R\ \wedge\ s \in S\ \}.\end{aligned}$$

Most join algorithms use a simple definition for RESULT (e.g., tuple concatenation) and TEST (e.g., testing equality of key values). The join partitions the cross product $R \times S$ into equivalence classes. The kind of equivalence classes used are determined by the join algorithm, and can be used to introduce 'groups' over which the join is to be done – for example grouping the tuples with equal key values.

With this in mind, we can produce a generalized join mapping, in which R has 'groups' R_i, S has 'corresponding groups' S_i, and we join over groups:

$$\begin{aligned}combine(R, S) \ = \ \{\, &\mathsf{RESULT}(r, s) \ \mid\\ &\mathsf{TEST}(r, s)\ \wedge\ r \in R_i\ \wedge\ s \in S_i\\ &\wedge\ R_i = \mathsf{MAP1}(P)\ \wedge\ P \in R\\ &\wedge\ S_i = \mathsf{MAP2}(P, S)\ \}.\end{aligned}$$

This generalized join mapping could be implemented in 'macro'-like pseudocode as follows:

```
combine($R$,$S$) = $T$ where
{
$T = \emptyset$;
for $P$ in $R$
    {
      $R_i$ = MAP1($P$);
      $S_i$ = MAP2($P$,$S$);
      for $r$ in $R_i$
          for $s$ in $S_i$
              if (TEST($r$,$s$))
              then $T = T \cup$ RESULT($r$,$s$)
    }
}
```

Here P is a 'part' of R (such as a (key value, group)-pair), and R_i and S_i are the actual groups the join is to be done over. MAP1 and MAP2 are arbitrary functions that convert groups to a suitable representation. Groups give what is needed for joins to deal with multiple occurrences of join keys (or even of tuples); they capture join equivalence classes.

This definition implements various join algorithms according to the structure chosen for R and S, and the choice of parameters.

- If R and S are collections of tuples, MAP1 maps a tuple P in R to the collection $R_i = \langle P \rangle$, and MAP2 simply takes S_i to be the entire collection S, then we obtain the *nested loops* algorithm.

- If R and S are groups (collections of collections) of elements with the same join key value, R_i is the group of R with join key i, S_i is the group of S with join key i, we obtain a general indexed join algorithm. Specifically, if the groups are collections of elements with the same hash key value, then the groups represent hash buckets, and we have the *parallel hash join* algorithm. The algorithm is parallel in that all groups can be joined in parallel.

- The parameterized set map operator filter(h,$S_1,\ldots,$ S_m), in FAD [6] yields the value of h applied to each tuple in the cross product of the sets $S_1, \ldots, S_m$ (for $m > 0$):

$$\mathsf{filter}(h, S_1, \ldots, S_m) \;=\; \{\, h(x_1, \ldots, x_m) \mid x_1 \in S_1,\ \ldots,\ x_m \in S_m \,\}.$$

  We can implement filter as a cascade of $m - 1$ *combines* implementing nested loops joins, where the final *combine* in the cascade applies h as its RESULT mapping.

The generalized join operator described above can be implemented with cascaded transductions:

$$\begin{array}{lcl}
\mathsf{combine}(R,S) & = & \mathsf{combine1}(S,R) \\[4pt]
\mathsf{combine1}(S,\langle\rangle) & = & \langle\rangle \\
\mathsf{combine1}(S,\langle P\rangle) & = & \mathsf{combine2}(\mathsf{MAP2}(P,S), \\
 & & \qquad \mathsf{MAP1}(P)) \\
\mathsf{combine1}(S,P_1 \diamond P_2) & = & \mathsf{combine1}(S,P_1) \\
 & & \diamond\ \mathsf{combine1}(S,P_2) \\[4pt]
\mathsf{combine2}(S_i,\langle\rangle) & = & \langle\rangle \\
\mathsf{combine2}(S_i,\langle r\rangle) & = & \mathsf{combine3}(\langle r\rangle,S_i) \\
\mathsf{combine2}(S_i,R_{i1} \diamond R_{i2}) & = & \mathsf{combine2}(S_i,R_{i1}) \\
 & & \diamond\ \mathsf{combine2}(S_i,R_{i2}) \\[4pt]
\mathsf{combine3}(\langle r\rangle,\langle\rangle) & = & \langle\rangle \\
\mathsf{combine3}(\langle r\rangle,\langle s\rangle) & = & \textbf{if}\ \mathsf{TEST}(r,s) \\
 & & \textbf{then}\ \mathsf{RESULT}(r,s) \\
 & & \textbf{else}\ \langle\rangle \\
\mathsf{combine3}(\langle r\rangle,S_{i1} \diamond S_{i2}) & = & \mathsf{combine3}(\langle r\rangle,S_{i1}) \\
 & & \diamond\ \mathsf{combine3}(\langle r\rangle,S_{i2}).
\end{array}$$

Many other tricks are possible here. For example, we can implement merge scans on streams (linear collections) with SVP-transducers. Merging of two streams is accomplished by making one of the streams the initial state, and incrementally consuming this state while simultaneously consuming the other stream [14].

5 Summary

To review what is new about SVP:

- SVP models information with *collections*. Collections include many interesting special cases, including streams, multisets, and groups, and combine sets and streams neatly in a single model.

- Collections are represented as *trees*. Where there have been many attempts to develop data processing models using functional operators on sets or on lists, SVP uses trees. This not only permits us to handle sets and streams in the same model, but also gives an explicit way to represent divide-and-conquer processing and parallel processing.

- SVP permits a natural characterization of *structure preserving mappings* on collections, and these mappings have important properties that yield parallelism and performance in data processing.

- SVP is *simple*. It does not rely on sophisticated algebraic concepts, or a powerful higher-order function framework, but on divide-and-conquer and functional composition. Earlier versions of the model experimented with greater sophistication (in fact a sizeable running prototype was written that treated transducers as higher-order functions), but ultimately were discarded in favor of simplicity.

In SVP, database mappings (queries) are formalized as transducers. These mappings have important properties:

1. SVP-transducers implement many useful bulk data operations: scan computations, relational algebra operators, arbitrary aggregate operators, including FAD's pump operator, arbitrary set mappings, including FAD's filter operator, and many stream mappings (specifically, stream transductions). More generally, SVP-transducers implement divide-and-conquer mappings that appear useful in bulk data processing.

2. SVP-transducers provide a natural means of specifying both independent and pipeline parallelism. At the same time, they have a rigorous semantics based on continuity with respect to collection orderings, that supports both independent and pipeline parallelism. Rigorous fixed point semantics can be derived for networks of SVP-transducers, even for cyclic networks.

The objective of a database model is to find a class of structures and mappings on those structures that: permit conceptualization of complex problems; permit adaptation and extensibility for new situations; permit efficient implementation; are rigorously defined; are generally useful. We feel the SVP model meets these essential criteria, and in addition offers insights on parallel data processing.

Acknowledgement

We are indebted to Serge Abiteboul, Paolo Atzeni, Dennis Shasha, and the referees for important critical comments and suggestions. We also want to thank Dave Martin, Ed Stabler, Cliff Walinsky, Members of the POP Laboratory at UCLA, particularly Shen-Tzay Huang and Xin Wang, and Members of the Rodin group at INRIA for fruitful discussions.

References

[1] D. Bitton, H. Boral, D. DeWitt, W. Wilkinson, Parallel Algorithms for the Execution of Relational Database Operations, *ACM Transactions on Database Systems*, **8**:3, 1983.

[2] P. Borla-Salamet, C. Chachaty, B. Dageville, Compiling Control into Database Queries for Parallel Execution, *Proc. Int. Conf. on Parallel and Distributed Information Systems*, Miami, Florida, Dec. 1991.

[3] W.H. Burge, *Recursive Programming Techniques*, Reading, MA: Addison-Wesley, 1975.

[4] N. Carriero, D. Gelertner, Linda in Context, *Communications of the ACM* **32**: 4, April 1989.

[5] C. Chase et al., Paragon: a Parallel Programming Environment for Scientific Applications Using Communications Structures, *Proc. Int. Conf. on Parallel Programming*, St. Charles, Illinois, Aug. 1991.

[6] S. Danforth, P. Valduriez, A FAD for Data-Intensive Applications, *IEEE Trans. Knowledge and Data Engineering*, **4**: 1, February 1992.

[7] A. Gupta, C. Forgy, A. Newell, High Speed Implementations of Rule-Based Systems, *ACM Transactions on Computer Systems*, **7**:2, May 1989.

[8] B. Hart, S. Danforth, P. Valduriez, Parallelizing FAD, a Database Programming Language, *Int. Symp. on Databases in Distributed and Parallel Systems*, Austin, Texas, Dec. 1988.

[9] W.D. Hillis, G.L. Steele, Data Parallel Algorithms, *Communications of the ACM*, 29:12, Dec. 1986.

[10] K.E. Iverson, *A Programming Language*, NY: J. Wiley, 1962.

[11] G. Kahn, The Semantics of a Simple Language for Parallel Programming, *Proc. IFIP 74*, North-Holland, 471–475, August 1974.

[12] R.E. Ladner, M.J. Fischer, Parallel Prefix Computation, *J. ACM* **27**:4, 831–838, 1980.

[13] D.S. Parker, Stream Data Analysis in Prolog, in L. Sterling, ed., *The Practice of Prolog*, MIT Press, 1990.

[14] D.S. Parker, E. Simon, P. Valduriez, SVP, a Model Capturing Sets, Streams, and Parallelism, Technical Report CSD-920020, Computer Science Department, UCLA, April 1992.

[15] I. Rival, ed., *Graphs and Order: the role of graphs in the theory of ordered sets and its applications*, Kluwer Academic Publishers, 1985.

[16] D.B. Skillikorn, Architecture-Independent Parallel Computation, *IEEE Computer* **23**:12, 38–50 (December 1990).

[17] Thinking Machines Corporation, *Programming in C**, Version 5.0, Cambridge, Ma, 1989.

[18] P. Valduriez, S. Khoshafian, Parallel Evaluation of the Transitive Closure of a Database Relation, *International Journal of Parallel Programming*, **17**: 1, 19–42, 1988.

[19] P. Valduriez, ed., *Data Management and Parallel Processing*, London: Chapman and Hall, 1991.

[20] P. Wadler, Comprehending Monads, *Proc. 1990 ACM Conf. on LISP and Functional Programming*, Nice, France, 61–78, June 1990.

[21] O. Wolfson, A. Ozeri, A New Paradigm for Parallel and Distributed Rule Processing, Proc. ACM SIGMOD Int. Conf., Atlantic City, May 1990.