

Elsevier Editorial System(tm) for Parallel Computing

Manuscript Draft

Manuscript Number:

Title: Relation-based computations in a monadic BSP model

Article Type: Research Paper

Section/Category:

Keywords: bulk synchronous parallel programs;
distributed relation-based computations;
specification, parallel program derivation;
algorithmic skeletons;
Haskell, C++

Corresponding Author: Dr. Nicola Botta,

Corresponding Author's Institution: PIK, Potsdam Institute for Climate Impact Research

First Author: Nicola Botta, Dr.

Order of Authors: Nicola Botta, Dr.; Cezar Ionescu, MSc

Manuscript Region of Origin:

Abstract: We propose a Haskell monadic model of bulk synchronous parallel programs and apply it to the analysis of relation-based computations.

Relation-based computations are simple but general patterns found in scientific computing applications. They are easy to implement sequentially, but difficult to parallelize.

We use the model to give high-level specifications of distributed relation-based algorithms and outline how to obtain testable parallel implementations from these specifications via equational reasoning.

We sketch the architecture of a C++ library of components for distributed relation-based computations. We argue that the model can be used to provide a concise and consistent library documentation.

Relation-based computations in a monadic BSP model

N. Botta C. Ionescu

*Data & Computation, Potsdam Institute for Climate Impact Research (PIK),
Telegrafenberg A 31, D-14473 Potsdam, Germany*

Abstract

We propose a Haskell monadic model of bulk synchronous parallel programs and apply it to the analysis of relation-based computations.

Relation-based computations are simple but general patterns found in scientific computing applications. They are easy to implement sequentially, but difficult to parallelize.

We use the model to give high-level specifications of distributed relation-based algorithms and outline how to obtain testable parallel implementations from these specifications via equational reasoning.

We sketch the architecture of a C++ library of components for distributed relation-based computations. We argue that the model can be used to provide a concise and consistent library documentation.

Contents

| | | |
|-----|---|----|
| 1 | Introduction | 2 |
| 2 | Relation-based algorithms and relation-based computations | 4 |
| 2.1 | Relation-based algorithms | 4 |
| 2.2 | Relation-based computations | 8 |
| 3 | A BSP monadic model | 10 |
| 3.1 | Haskell and pseudo-code | 10 |
| 3.2 | The local computations stage | 12 |
| 3.3 | The communication stage | 14 |
| 3.4 | Completing the model | 16 |

| | | |
|-----|--|----|
| 4 | Relation-based algorithms | 18 |
| 5 | Relation-based computations | 26 |
| 5.1 | Relational composition | 26 |
| 5.2 | Converse | 28 |
| 6 | C++ implementation and Haskell documentation | 31 |
| 7 | Conclusions | 35 |
| | Acknowledgements | 36 |
| | References | 36 |

1 Introduction

In this paper we apply a new monadic model of bulk synchronous parallel (BSP, [3,5]) programs to the analysis of relation-based computations (RBC).

The notions of BSP program, of monad and the new monadic model of BSP programs are presented in section 3 and discussed in full detail in a companion paper [7].

Relation-based algorithms (RBA) are simple but general computational patterns found in many scientific computing application domains. Examples of relation-based algorithms are: the computation of neighbor elements on a grid; the computation of geometrical properties of grid elements, e.g., element center, area, boundary integrals; sparse matrix-vector multiplications. While relation-based algorithms can be easily written for a sequential, single program single data (SPSD) computational environment, they are often difficult to implement in a parallel, single program multiple data (SPMD) distributed case.

The notion of relation-based algorithm and of relation-based computations are introduced and discussed in detail in section 2.

The work presented in this paper has grown within a project aiming at developing a framework of software components for distributed relation-based computations, SCDRC, see [10], <http://www.pik-potsdam.de/~botta/>. As shown in the architecture sketched in figure 1, SCDRC is a thin layer above message passing libraries but below applications. It is the basis on which application dependent software components, for instance software components

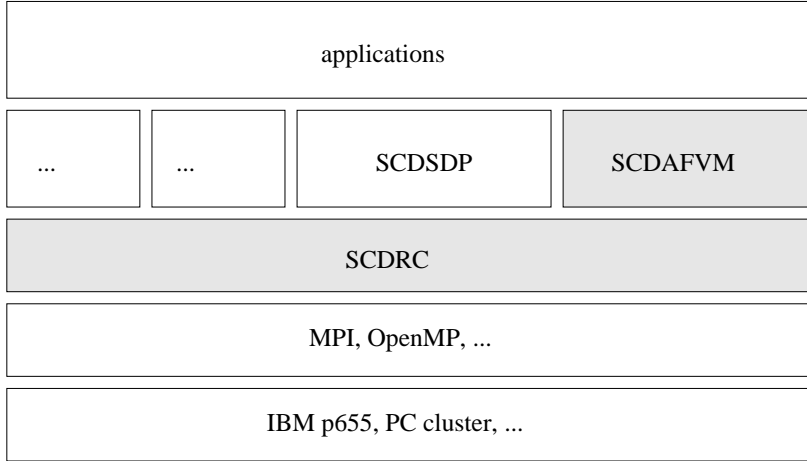


Fig. 1. Software architecture for distributed scientific computing applications.

for distributed adaptive finite volumes methods (SCDAFVM), are written.

In this architecture, the role of SCDRC is to provide application independent data structures and functionalities for structuring parallel implementations of the RBCs emerging in specific application domains: application dependent software components are implemented *in terms of* SCDRC components.

The architecture allows two approaches for implementing application dependent software components *in terms of* SCDRC while avoiding memory demanding temporary data and time consuming data type conversions. One approach is to require applications to use a pre-defined set of concrete data structures, e.g., for arrays, relations, grids. In this approach, such data structures are provided by SCDRC. The second approach is to design SCDRC as a set of *generic* software components.

The first approach is hardly practicable because of the natural variability of the data types used even in a single application domain. We have therefore chosen to implement SCDRC as a set of generic software components and to use C++ as programming language.

Among the programming languages offering the run-time performance required by scientific computing applications, C++ has the advantages of supporting both *type* and integral *value* genericity¹, of being standardized and well supported on most architectures.

Of course, C++ has well-known drawbacks. For our intended usage, the major drawbacks of C++ are, beside its complexity and arcaness, the problematic integration of support for static and dynamic polymorphism. Moreover, as

¹ in scientific computing applications, many data structures and functions are naturally parameterized w.r.t. an integral value representing, e.g., a number of coordinates, dimensions, equations etc.

most programming languages, C++ does not provide support for distributed data structures and for distributed computations.

Because of the above limitations, C++ is not a useful framework for discussing, explaining and reasoning about algorithms in general and about parallel algorithms for distributed relation-based computations in particular.

When developing parallel applications, however, it is essential to have a formal framework that allows one to distinguish between *distributed* and *local* data, to describe data *exchange* between remote *partitions*, to express the differences between single program, single data (SPSD) and single program, multiple data (SPMD) computations etc.

The monadic model presented in section 3 is such a framework. It has been conceived to fill the gap between informal descriptions of the problems arising in developing parallel SPMD applications and of the correspondent solution algorithms on one hand and C++ implementations on the other hand.

The model has been implemented as an Haskell module [1,11,4] and allows one to write parallel algorithms in a familiar, imperative *do* notation. It can be seamlessly used to specify parallel problems and to implement and run parallel solution algorithms.

In sections 4 and 5, we show how to use the monadic model introduced in section 3 to formulate elementary problems arising in parallel SPMD applications. The model is simple and expressive enough to formulate, compare and test solution algorithms.

In section 6 we outline how the monadic model presented in section 3 can be used to *explain* and document a C++ prototype implementation of SCDRC. For a detailed description of this SCDRC prototype and for a comparison between SCDRC and other approaches, we refer the reader to [10]. o

2 Relation-based algorithms and relation-based computations

2.1 Relation-based algorithms

Relation based algorithms are simple but general computational patterns. They can be expressed in terms of two functions h and f , of a relation R and of an array of positive natural numbers js :

Algorithm 1 : relation-based algorithm

```

for  $j$  in  $js$ 
  compute  $h([f(i) \mid i \text{ in } R(j)])$ 

```

Here j is drawn from js and h is applied to the array of values f takes at those indexes i which are in relation R with j . These are represented by the array $R(j)$. The notation $[f(i) \mid i \text{ in } R(j)]$ is an instance of an array comprehension, which generalises in the natural way the familiar set comprehension, and which is found in many programming languages, among which are Python, Haskell, Perl6.

We only consider relations between zero-based intervals of natural numbers. As discussed in detail in section 3 of [10], this is a convenient choice when dealing with finite relations. We therefore think of R as of a subset of $[0 \dots m) \times [0 \dots n)$ where m and n are the sizes of the *target* and of the *source* of R , respectively.

In this paper we use a left-from-right notation when giving the signature of relations and functions:

$$R :: [0 \dots m) \longleftarrow [0 \dots n)$$

$$R(\cdot) :: \text{subarrays}([0 \dots m)) \longleftarrow [0 \dots n)$$

As mentioned in the introduction, relation based algorithms are commonly found in many scientific computing algorithms. Consider, for instance, the problem of computing the centers of the triangles of a triangulation. Let the triangulation be represented by an integer table vt : the j -th row of vt contains the three indexes of the vertexes of the j -th triangle. Given vt and an array x of vertex coordinates, a computation of the centers can be written as:

Algorithm 2 : triangle centers

```

for  $j$  in  $[0 \dots \text{size}(vt))$ 
  compute  $\text{center}(x(vt(j)(0)), x(vt(j)(1)), x(vt(j)(2)))$ 

```

In a Cartesian system of coordinates and for a plane triangulation, `center` would simply add up the vertex coordinates $x(vt(j)(0))$, $x(vt(j)(1))$ and $x(vt(j)(2))$ and divide the result by three. In a different context, for instance in the case of spherical triangles, `center` is a less straightforward function. Algorithm 2 is obviously a special RBA with $h = \text{center}$, $f = x$ and with the relation R expressed in terms of the vertex-triangle table vt i.e. $R(j) = [vt(j)(0), vt(j)(1), vt(j)(2)]$.

Let us generalize a little bit RBAs by allowing f to take as argument pairs in $[0 \dots m) \times [0 \dots n)$:

Algorithm 3 : relation-based algorithm

```

for  $j$  in  $js$ 
  compute  $h([f(i, j) \mid i \text{ in } R(j)])$ 

```

In this form, relation-based algorithms can be specialized to represent matrix-vector multiplications. Let A be a sparse matrix and c , e and p be a CRS (compact row storage, see [6]) representation of A that is, c , e and p satisfy the following equivalence:

$$A(j, i) \neq 0 \equiv \exists! k \text{ in } [p(j) \dots p(j+1)] : i == c(k) \wedge A(j, i) == e(k) \quad (1)$$

An efficient representation of the computation of the product between A and a suitably sized vector b reads:

Algorithm 4 : sparse matrix vector multiplication

```

for  $j$  in  $[0 \dots n)$ 
  compute  $\text{sum}([e(k) * b[c(k)] \mid k \text{ in } [p(j) \dots p(j+1)]])$ 

```

Using 1, this rule can be written as a relation-based algorithm with

$$\begin{aligned}
 h &= \text{sum} \\
 f(i, j) &= e(k(i, j)) * b[i] \\
 &\quad \textbf{where } k(i, j) = p(j) + \text{index_of}(i, R(j)) \\
 R(j) &= [c(k) \mid k \text{ in } [p(j) \dots p(j+1)]]
 \end{aligned}$$

In the above expression we have used the function `index_of` which computes the index of a given element in an array:

$$k == \text{index_of}(s, ls) \equiv s == ls[k]$$

Of course, `index_of` is a function only for array arguments which are *nubbed* that is, contain no duplicates. We impose this requirement on any relation representation $R(\cdot)$.

Implementations of sparse matrix-vector multiplications as relation-based algorithms are useful only if the data structures that implement relations provide efficient ways of computing $R(j)$ and $\text{index_of}(i, R(j))$. This suggests that relation-based implementations of sparse matrixes might be quite different from relation-based implementations of, e.g., Cartesian grids. This is an example of the data structure variability mentioned in the introduction: data structures representing a given concept – in this case that of a finite relation – might require quite different implementations in different contexts.

In a sequential SPSD computational model, relation-based algorithms can be easily expressed in the pseudo-code notation used in algorithm 1, 2 and 4. In

this model, relation-based algorithms can be easily implemented in standard programming languages.

Unfortunately, relation-based algorithms are not, in general, trivially parallelizable. In particular, they are not trivially parallelizable whenever the following conditions occur:

- the function f is represented by storing *disjoint* subsets of the graph of f in, say, np partitions². In scientific computing it is often the case that f -values are stored in arrays.
- The relation R is such that any disjoint splitting of its source in partial relations $R_1 \dots R_{np-1}$ yields non-disjoint ranges $\text{ran}(R_p), \text{ran}(R_q)$ for some distinct $p, q < np$.

For relation-based algorithms which are not trivially parallelizable local computations require, no matter how f and R are distributed among remote partitions, f -values which are stored on remote partitions. Of course, any non-trivially parallelizable RBA can be turned into a trivially parallelizable one. A simple approach is to duplicate the whole f on every partition. Duplicating data persistently, however, is inefficient and raises the non-trivial problem of how ensure the consistency of the duplicated data.

Parallelizing relation-based algorithms in a SPMD computational environment requires, among others, answering the following questions:

- (1) How are f and R distributed among remote partitions ?
- (2) Which f -values are needed on the local partition which are stored on remote partitions?
- (3) Which f -values stored on the local partition are needed by which remote partition?
- (4) How can these f -values be exchanged between partitions?

Message passing libraries, e.g. MPI, provide efficient and portable answers to question 4: how to exchange data between partitions. However, they cannot provide answers to questions 1-3: these questions are concerned with the *structuring* of the parallel computation. In particular, the answers to question 2 and 3 essentially depend on how question 1 is answered.

A first difficulty that has to be faced when dealing with the above questions is that of representing the co-existence of partial representations of f and R on distinct partitions. To do this, the notation used in the above algorithms has obviously to be extended.

² we do not attempt at defining the concept of a *partition* here: in a SPMD (single program multiple data) distributed computational environment, different partition usually correspond to remote memory spaces.

Beside representing the co-existence of partial representations of functions and relations and expressing the relationships between local and global data, a model for expressing distributed RBA also has to support expressing data exchange between partitions.

The monadic model introduced in the next section fulfills these requirements and allows us to formulate and answer the above questions in a formal framework. This is done in sections 4 and 5. In the remain of this section we introduce some elementary relation-based computations. Together with RBAs, these generic computational patterns allow one to represent a wide class of algorithms commonly found in scientific computing applications.

2.2 *Relation-based computations*

While being powerful patterns, RBAs are certainly not enough for structuring complex applications. To understand why, let's go back to the triangle centers example and assume, for concreteness, that the table vt and the vertex coordinates array x are initially stored in a file (vt and x can be seen as a minimal representation of a triangulation). Before a parallel computation of the triangle centers can take place, the following steps have to be done:

- (1) read vt and x from the file,
- (2) compute a partitioning of the source of vt ,
- (3) compute a partitioning of the source of x ,
- (4) distribute vt and x according to these partitionings.

Let's focus on steps 2 and 3. Computing a partitioning of the source of vt means associating a unique partition number to each triangle of vt .

Of course, one would like to partition the rows (triangles) of vt in such a way that the subsequent parallel computation of the triangle centers is done efficiently. This boils down to requiring that all partitions contain approximately the same number of rows (or a number of rows proportional to the computational capacity associated with the partitions) and that the total number of *edge-cuts* is minimal and equally distributed among partitions. Notice that the number of edge-cuts – pairs (v, t) in vt such that v and t belong to different partitions – can only be computed if a partitioning of the vertexes is already known (or computed together with the partitioning of the triangles). Minimizing the number of edge-cuts means minimizing the number of vertex coordinates that have to be exchanged between partitions in the triangle centers computation.

Grid and relation (graph) partitioning is a well-established research area and very efficient graph partitioning algorithms and libraries are available. Two

outstanding ones are Metis [8] and ParMetis [6].

Of course, different partitioning algorithms put different requirements on their argument relations. Metis and ParMetis, for instance, require such relations to be *symmetric* and *anti-reflexive*. This means that

$$(iRj \equiv jRi) \wedge (iRj \Rightarrow i \neq j)$$

The vertex-triangle relation of our example is certainly non symmetric. This means that, in order to take advantage of Metis and ParMetis for computing a partitioning of vt , one has to construct a symmetric, anti-reflexive auxiliary relation, say avt , that represents vt “well”.

Since avt is to be used to compute a partitioning of the source of vt , its source has to coincide with the source of vt . Moreover, partitionings of (the source of) avt which satisfy minimal edge-cut constraints should lead to minimal or almost minimal edge-cuts for vt as well.

Grid-relations like vt are commonly found in many application domains. They often describe coverings of 1- 2- or 3-dimensional manifolds or neighborhood relationships on such coverings. A common way of computing an auxiliary relation for grid relations like vt is the following:

- (1) compute the *converse* of vt , vt° .
- (2) compute $tvt = vt^\circ \cdot vt$.
- (3) compute $avt = tvt - \text{id}_{[0 \dots \text{source_size}(vt)]}$

We use R° to denote the converse of a relation (or of a function) R . If $R : [0 \dots m) \longleftarrow [0 \dots n)$, then $R^\circ : [0 \dots n) \longleftarrow [0 \dots m)$ and $jR^\circ i \equiv iRj$.

Notice that tvt is symmetric and represents a *neighborhood* relationship: $tvt(j)$ provides, for the j -th triangle, the indexes of those triangles that share at least one vertex with the j -th triangle. Neighborhood relationships naturally arise, among others, as *stencils* of discrete differential operators in finite volumes, finite elements and finite differences methods for the numerical approximation of partial differential equations.

The relation avt is symmetric and anti-reflexive and its source coincides with the source of vt . Thus avt can be used to compute a partitioning of (the source of) vt with the Metis library. Given a partitioning function sp , a “suitable” partitioning tp of the target of vt – the source of x in our example – can be easily computed by considering the relation $sp \cdot vt^\circ$. This relation associates to each vertex the partition numbers of those triangles that share the given vertex. A suitable way of partitioning the target of vt (the source of vt°) is then to pick-up, for each vertex i in the source of $sp \cdot vt^\circ$, one of the partition numbers that appears most frequently in the array $(sp \cdot vt^\circ)(i)$. This choice,

the fact that tvt is a neighborhood relation and the edge-cut properties of the partitioning of avt computed by Metis, guarantee that the partition number of most vertexes will coincide with the partition number of the triangles it belongs to. This, in turn, means that the number of edge-cuts is almost minimal.

Notice also that the computation of tp described above is itself a relation-based algorithm with $R = vt^\circ$, $f = sp$ and $h = \text{most_frequent}$. Here most_frequent is a function that takes an array of natural numbers and returns a natural number such that no other array element appears more frequently.

The above discussion shows that, in order to apply relation-based algorithms in realistic distributed parallel applications, a minimal set of elementary relational operations – in the example discussed above composition and conversion – are needed. We use the term relation-based computations (RBC) to broadly denote relation-based algorithms together with such a set of elementary relational operations.

3 A BSP monadic model

3.1 Haskell and pseudo-code

In this section, we present a formalisation of BSP-like parallel computations, which has been implemented in the programming language Haskell. On one hand, this leads to a notation for the specification of parallel computations which is consistent, complete and easy to use. On the other hand, we can execute Haskell programs which implement the given specifications, simulating the parallel execution of the computation and avoiding potential errors that might otherwise be apparent only after the costly stage of coding.

Consider the main computational pattern we are interested in, expressed by algorithm 1 in the previous section. This can be implemented in Haskell in the following way:

```
[ h [ fs!!i | i <- rss!!j ] | j <- js ]
```

subject to the following remarks.

- (1) The renaming of identifiers, from f to \mathbf{fs} and from R to \mathbf{rss} , has been done partly because Haskell reserves uppercase identifiers for type constructors, and partly because postfixing an identifier name with “s” or “ss” is the traditional way of denoting a list or a list of lists, respectively.
- (2) We use in Haskell lists instead of arrays. The operation of indexing above

is available for both structures, as are all other operations we use here. Lists are one of the most important data structures in functional programming and Haskell provides special notation for them (such as list comprehension) and a large number of predefined combinators (such as `map`, used below). The main advantage of arrays is the efficiency of the indexing operation (constant time versus linear time for lists), but that is not a main concern for a specification language. In implementing the design expressed by the Haskell programs, we will, of course, use arrays where appropriate.

- (3) List comprehension is a way of constructing a list by combining the elements of existing lists, in much the same way as set comprehensions are used to define sets from existing ones. In the implementation above, we see two *nested* list comprehensions. The inner one constructs the list of all the values of `fs` at the indices `i` which are in the relation `rss` with `j`. The outer list comprehension constructs the list of results of applying `h` to each such list (one for each `j` in the list `js`).

- (4) While translating the `for` loop of algorithm 1 with a list comprehension is the common idiom, it is not the only way. We could also have written

```
map h [ [ f!!i | i <- rss!!j ] | j <- js ]
```

`map` is one of the combinators provided for lists: it takes two arguments, a function and a list, and applies the function to every element of the list. A third version, very close to the imperative one, is

```
do j <- js
   return h [ f!!i | i <- rss!!j ]
```

where `for` has been replaced with `do`, `in` with `<-` and `compute` with `return`. We are going to use the `do` notation in a different context, and so, to avoid confusion, prefer here one of the other notations. We have mentioned this alternative here to underline the very short distance between our pseudo-code notation and actual Haskell code.

Since they are so similar, it is not surprising that both notations must be extended in order to be used to specify parallel computations. The extensions have to make explicit which steps are local, and which require communication and synchronisation. Naive approaches can fail to provide an adequate representation of the parallel computation (e.g. by using the same notation for functions which require communication and for those which do not), or be very cumbersome (e.g. by requiring the indexing with the partition number of data which is guaranteed to be the same across all partitions).

The Haskell-based model presented below is, we think, both correct and usable. Moreover, the specifications can be easily refined to yield working Haskell programs which can be directly executed (on small data sets) to simulate the parallel execution and thus help find potential errors.

3.2 *The local computations stage*

In order to give precise specifications of our problems and to derive, or at least prove the correctness of algorithms for the solution of these problems, we need a mathematical model of parallel computations. The model we have adopted is based on the Bulk-Synchronous Parallel (BSP) computational model, see [12,2,5]. In particular, Miller ([9]) presents a Haskell implementation of a BSP library which is structured using monads, and by extension, a monadic model for BSP programs.

BSP belongs to the class of SPMD models and is characterised by considering only the case of a fixed number of processors, each with its own memory, and one data exchange primitive, similar to the MPI `MPI_Alltoall` function. The computer can be in one of two states: either every processor is performing some computation using only the data in its own memory space and its own “processor id”, or every processor is involved in the data exchange operation. A local computation stage followed by a data exchange is called a “superstep”, and a BSP program can be viewed as a sequence of supersteps.

In order to formalise this intuitive description, we start with the local computation stage. Each processor acts on its own, local data: we need a description of the data associated with each processor, which suggests that this data is described by a function of type

$$d : A \leftarrow \text{Proc}$$

where `Proc` is the set of processors (usually represented by $\{0, 1, \dots, np - 1\}$).

All the data that is going to be involved in parallel computations is going to be of this kind. In Haskell, we could interpret this by defining distributed data of some generic type `a` to be a `newtype`.

```
newtype D a = D (Proc -> a)
```

As a simple example, the distributed data value which associates to each processor its id is described by

```
pid :: D Proc
pid = D id
```

Distributed values of type `list` or `array` are often considered to be representations of a non-distributed, global `list` or `array`. There are many ways in which the distributed value can be related to the global one, but typically it will

involve some “gluing” together of the elements of the distributed collection in a certain order to obtain the global value. The simplest such “gluing” is described by the following function:

```
dist2global    :: D [a] -> [a]
dist2global dxs = concat [ dxs p | p <- [0 .. (np-1)]]
```

This “gluing” implicitly defines the notion of distributed data adopted throughout this paper (for list-like data structures): `dxs` is a distributed representation of `xs` iff `xs = dist2global dxs`. This notion is compatible with the description of distributed data given in the previous section. As mentioned before, it is trivial to replace the list data structure with an array. One could also make the `dist2global` function polymorphic, so that it applies to every data structure which has an adequate `concat` function.

The function `dist2global` is important in simplifying specifications, but it is essential to understand that it cannot be used in writing parallel programs: it is neither a local computation nor a communication step. It has the same status as the function *setify* used in the specification of programs in terms of relations between mathematical sets which are not datatypes of any programming language.

In the local computation stage, a new distributed value is obtained from existing ones by applying on each processor a function to the local data. The `do` notation provided by Haskell allows us to express this in a natural fashion. Informally, we have that

```
do  x1 <- dx1
    x2 <- dx2
    ...
    xm <- dxm
    return f (x1, x2, ..., xm)
```

represents a distributed value which has on a processor `p` the value obtained by applying a function `f` to the `m` values on processor `p` of the distributed values `dx1`, `dx2`, ..., `dxm`.

As an example, consider the *apply* function given in [5]. The local computation stage is described there as the operation of applying a “distributed function” to a “distributed value”:

$$\begin{aligned} \text{apply} &: DA \longleftarrow DB \longleftarrow D(A \longleftarrow B) \\ \text{apply } df \text{ } db &= da \textbf{ where } da \text{ } p = (df \text{ } p) (db \text{ } p) \end{aligned}$$

In the `do` notation, this translates to:

```
apply df db = do f <- df
```

```

    b <- db
    return (f b)

```

This is quite similar to imperative programming, and is thus a suitable choice for a specification language for algorithms that are going, in the end, to be implemented in some imperative language.

In Haskell, the `do` notation can be used only for datatypes which are instances of the typeclass *Monad*, see [1]; thus using it presupposes that the datatype of distributed values has been installed as such an instance. A number of mathematical results have been formulated in terms of the `do` notation, and we can take advantage of them in reasoning about the specifications of parallel programs. The interested reader can find more about this in our companion paper, [7], here we shall use only elementary results in a less rigorous way.

3.3 The communication stage

In the communication stage, every processor may send or receive one or more elements of some datatype *A* to any other processor. This can be expressed formally in several equivalent ways. We give here a description based on lists.

```

exch :: D [(a, Proc)] -> D [(a, Proc)]
exch (D dxs) = D dys
    where
    dys p = [(x, p') | p' <- [0 .. np-1],
                      (x, p'') <- dxs p',
                      p'' == p]

```

The SCDRC implementation of `exch` is very different from this description, involving MPI function calls and using data structures more suitable to the interface of these functions. The most important difference is that `exch` is constrained to act on types *a* which are instances of “Plain Old Data” types (see [10]). We ignore such complications here.

Together with local application as expressed in the `do` notation, `exch` allows us to write programs with a pre-determined number of communication stages. Here is, for instance, a program that computes on every partition the list of *offsets* of a distributed list.

Given a distributed list `dxs :: D [a]`, the list of offsets `offs :: [Nat]` helps establish a relation between the elements of the “global” list `xs = dist2global dxs` and the elements of *Proc* in the following way: the *i*th element of `xs` is going to be found on the partition *p* iff `offs!!p ≤ i < offs!!(p+1)`. Such a list can be easily obtained if we know the individual

sizes of the local parts of the list:

```
mk_offs  :: [Nat] -> [Nat]
mk_offs  = map sum . inits
```

`inits` is a predefined function which returns the list of initial segments of a given list, informally:

```
inits [x0, x1, ..., xn]
=
[[], [x0], [x0, x1], ..., [x0, x1, ..., xn]]
```

and therefore

```
mk_offs [x0, x1, ..., xn]
=
[0, x0, x0+x1, ..., (x0+x1+ ...+xn)]
```

In order to compute the list of offsets, we can collect, on each partition, the list of the sizes of the local parts and apply `mk_offs`. The list of sizes can be obtained by extracting, using the library function `fst`, the first elements of the pairs of values obtained after a communication step:

```
offs dxs = do sizes <- dsizes
           return (mk_offs (map fst sizes))
           where dsizes = exch (do xs <- dxs
                                return (repl (length xs)))
```

The `repl` function represents a typical idiom for parallel programming, used when a processor is to send the same data to all others:

```
repl      :: a -> [(a, Proc)]
repl x    = [(x, p) | p <- [0 .. np]]
```

`exch` has a number of properties, among which is the fact that it is idempotent, that is

```
exch . exch = id
```

Another important result is that the distributed value obtained by

```
exch (do {x <- dx; return repl x})
```

has the same local value on all partitions. Using this result we can show that `offs dxs` defined above has the same value on all partitions.

3.4 Completing the model

If we now tried to write a recursive program with the elements we have, we would fail. This is because the terminating condition of the recursion cannot be written as a local operation (inside a `do` clause), nor is it something that simply results from an `exch`, and those are the only operations we have on distributed datatypes. Therefore, something more is needed. In [5], this is expressed in the form of a global conditional. We adopt a slightly different approach, by providing a partial function which extracts the value of type `a` from a constant distributed value of type `D a`. The result of this operation is not defined if the distributed value is not constant.

```
-- val specification
val    :: D a -> a
val dx  = dx 0, if dx is constant
         undefined, otherwise
```

For our Haskell model we can implement `val` to issue an error message if the input is not constant.

With `val` we have now all the elements needed to write recursive parallel programs. As a simple example, consider a program to search for an element satisfying a given predicate in a distributed list.

For instance, in a list of triangles, we might want to ensure that each triangle satisfies some non-degeneracy condition. In order to check this condition, each element of the (distributed) list has to be checked, but the checking can stop after the first degenerate triangle has been found. Depending on the sizes of the local lists it might be profitable to interrupt the checking process after some number of steps and see if the other processors have found an invalid element in the meantime. For simplicity, we make this verification here after each local step.

In one step of local checking we determine if we have found an invalid element or if we have to continue checking more, or if we have checked all elements:

```
data State a = OK | Checking | Error a

chk_fst :: [a] -> (State, [a])
chk_fst [] = (Ok, [])
chk_fst (x:xs) = if invalid x then (Error x, xs)
                  else (Checking, xs)
```

The following function applies this local step to all processors:

```

d_chk_fst :: D [a] -> D (State, [a])
d_chk_fst dxs = do xs <- dxs
                return (chk_fst xs)

```

Next, every processor sends the resulting state to every other processor. The function that prepares this exchange uses `repl`:

```

d_send_chk    :: D (State, [a]) -> D [(State, Proc)]
d_send_chk dxs = do xs <- dxs
                  return (repl (fst xs))

```

After exchanging this information, every processor can compute whether an invalid element has been found, or whether no more elements need to be examined on every partition, by applying the function `chk_list` defined below:

```

d_get_chk     :: D [(State, Proc)] -> D (State, Proc)
d_get_chk dxs = do xs <- dxs
                  return (chk_list xs)

chk_list      :: [(State, Proc)] -> (State, Proc)
chk_list []   = (OK, 0)
chk_list (Error x, p):xs = (Error x, p)
chk_list (Checking, p):xs = if s' == OK
                              then (Checking, p)
                              else (s', p')
                              where (s', p') = chk_list xs
chk_list (OK, p):xs      = chk_list xs

```

The result of `d_get_chk` is a constant distributed value if its argument is a constant distributed value. Therefore we can apply `val` and determine whether another step must be performed or not, resulting in the following program:

```

check        :: D [a] -> D String
check dxs = if fst (val dc) /= Checking then report dc
            else check dxs'
            where
              dxs1 = d_chk_fst dxs
              dxs2 = d_send_chk dxs1
              dxs3 = exch dxs2
              dc   = d_get_chk dxs3
              dxs' = do xs <- dxs1
                      return (snd xs)

```

4 Relation-based algorithms

As shown in section 3, sequential relation-based algorithms can be implemented in Haskell with nested list comprehensions:

```
[ h [ fs!!i | i <- rss!!j ] | j <- js ]
```

In the above expression `js` is a list of indexes. In Haskell, lists of indexes are conveniently represented with objects of type `[Int]`.

We have explained in section 2 why, when dealing with relation-based algorithms, we only consider relations between zero-based intervals of natural numbers. Throughout this paper, we represent relations as set-valued functions.

In this section we derive an implementation of a parallel generic distributed relation-based algorithm in the monadic BSP model introduced in section 3. Of course, we consider relation-based algorithms which are not trivially parallelizable. In non trivially parallelizable RBAs, the function f is represented by storing its graph (i.e. f is not defined in terms of some analytic expression). This can be conveniently expressed in Haskell by representing f by means of generic lists of elements of the type of the target of f , say `a`. Similarly, set-valued function representations of relations are implemented in our Haskell based BSP model, as lists of lists of integers.

In this type framework `h` has to be of (generic) type `[a] -> b`. We can make these type constraints explicit by explicitly introducing an `rba` function:

```
rba :: [[Int]] -> [a] -> ([a] -> b) -> [Int] -> [b]
rba rss fs h js = [ h [ fs!!i | i <- rss!!j ] | j <- js ]
```

According to this definition, `rba` can be seen as a function that takes a relation `rss` of type `[[Int]]`, a function `fs` of type `[a]`, a function `h` of type `[a] -> b` and returns a function of type `[Int] -> [b]`.

We can now define what it means to derive a parallel generic distributed relation-based algorithm `d_rba` in the BSP model of section 3. This is equivalent to implementing a function

```
d_rba :: D [[Int]] -> D [a] -> ([a] -> b) -> D [Int] -> D [b]
```

that represents `rba` in the sense that

```
-- specification 1
dist2global (d_rba drss dfs h djs) = rba rss fs h js
where rss = dist2global drss
```

```

fs = dist2global dfs
js = dist2global djs

```

for arbitrary `drss`, `dfs`, `h` and for arbitrary `djs` *compatible* with `drss`. We say that a distributed list of integers `djs` is compatible with a distributed relation `drss` if, on each partition `p`, all the elements of `djs p` are in `[os!!p .. os!!(p+1)- 1]` where `os` are the offsets of `drss`.

In this framework, the fact that relation-based algorithms are generally not trivially parallelizable means that a naive implementation of `d_rba` as

```

d_rba drss dfs h djs =
do p <- pid
  rssp <- drss
  fsp <- dfs
  jsp <- djs
  ros <- offs drss
  fos <- offs dfs
  return [ h [ fsp!!(i-fos!!p) | i <- rssp!!(j-ros!!p)] |
          j <- jsp ]

```

does not satisfy specification 1. As explained in section 3, this is because, in general, `rssp` contains indexes `i` which are smaller than `os!!p` or greater or equal to `os!!(p+1)`. These yield `fsp` indexes `i - os!!p` which are smaller than zero or greater or equal to `length fsp` and therefore violate a precondition of the `!!` operator.

According to the definition of `offs` given in the previous section, `i < os!!p` means that `(dist2global dfs)!!i` is stored on a partition `p' < p`. Conversely, `i >= os!!(p+1)` means that `(dist2global dfs)!!i` is stored on `p' > p`. In fact, `dist2global` and `offs` imply the following identity:

```

(dist2global dxs)!!i = (dxs q)!!(i - os!!q)
where os = val (offs dxs)
      q = offs2part os i

```

where `offs2part` is a function that computes the partition associated to an index `i` according to the offsets `os`. `offs2part` fulfills

```

(q = offs2part os i) == (os!!q <= i && i < os!!(q+1))

```

and can be implemented via binary search. In our BSP model *non-local* values have to be accessed via a communication stage. With the function `offs2part` and given `os`, one can easily compute, for any `i` in the domain of `dist2global dfs`, the partition on which `(dist2global dfs)!!i` is stored. In particular, one can compute, on each partition `p` and for each `i` in the range of

`drss p` and in the domain of `dist2global dfs`, the partition `q` on which `(dist2global dfs)!!i` is stored.

Therefore, one can think of implementing a distributed relation-based algorithm by first computing, for each `j` in `js` and for each `i` in `rss!!(j - os!!p)`, the partition `q` on which `(dist2global dfs)!!i` is stored. If `q == p` then `(dist2global dfs)!!i` is on the local partition and can be accessed as `fsp!!(i-os!!p)`. Otherwise, partition `q` is requested to evaluate `(dist2global dfs)!!i` and send it to the local partition.

This approach requires two calls to `exch` – one for sending the index of the required data, a second one for receiving the data itself – for each element of `concat rssp`. In fact it requires exactly

$$2 \max_{p \in [0, np-1]} (\text{length} (\text{concat}(\text{drss } p)))$$

calls to `exch`. This approach minimizes the amount of data that have to be temporarily stored to implement a parallel distributed relation-based algorithm. It requires a number of communication stages that increases linearly with the size of the distributed data. Moreover, it forces a synchronization of the “parallel computation” at each step of the innermost loop over `rssp!!j` and for all `j` in `jsp`.

For most scientific computing applications, this approach is prohibitively expensive and is conveniently replaced by one that minimizes the communication costs. As we will show, this requires extra temporary storage of the order of the size of `dist2global drss`.

This second approach can be derived by writing a distributed relation-based algorithm as a distributed application of `rba` on modified distributed data `dfs'`, `drss'` and `djs'`:

```
d_rba drss dfs h djs =
do rssp' <- drss'
  fsp' <- dfs'
  jsp' <- djs'
  return rba rssp' fsp' h jsp'
where drss' = ...
      dfs' = ...
      djs' = do p <- pid
              jsp <- djs
              os <- offs drss
              return (map (add os!!p) jsp)
```

where `dfs'` and `drss'` are constructed in such a way that, for any partition `p` in `[0 .. np-1]`, the application of the distributed relation-based algorithm

is equivalent to applying the sequential counterpart on `djs p`:

```
-- specification 2
(d_rba drss dfs h djs) p = rba rss fs h (djs p)
where rss = dist2global drss
      fs = dist2global dfs
```

As usual, this condition has to hold for arbitrary `drss`, `dfs`, `h` and for `djs` *compatible* with `drss`. It is a quite natural condition: it requires the computational work to be done in parallel on the partitions to be proportional to the local sizes of the distributed data `djs`.

Specification 2 is sufficient (but, of course, not necessary) for `d_rba` to fulfill specification 1:

```
dist2global (d_rba drss dfs h djs)
= {definition of dist2global}
concat [ (d_rba drss dfs h djs) p | p <- [0 .. (np-1)] ]
= {hypothesis}
concat [ rba rss fs h (djs p) | p <- [0 .. (np-1)] ]
where rss = dist2global drss
      fs = dist2global dfs
= {definition of rba}
concat [ [ h [ fs!!i | i <- rss!!j ] | j <- (djs p) ] |
        p <- [0 .. (np-1)]
      ]
where rss = dist2global drss
      fs = dist2global dfs
= {concat properties}
[ h [ fs!!i | i <- rss!!j ] |
  j <- concat [(djs p) | p <- [0 .. (np-1)]] ]
where rss = dist2global drss
      fs = dist2global dfs
= {definition of dist2global}
[ h [ fs!!i | i <- rss!!j ] | j <- (dist2global djs) ]
where rss = dist2global drss
      fs = dist2global dfs
= {definition of rba, introduction of js}
rba rss fs h js
where rss = dist2global drss
      fs = dist2global dfs
      js = dist2global djs
```

The problem of implementing a `d_rba` function that fulfills specification 1 can therefore be formulated as that of computing `drss'` and `dfs'` for `d_rba` to

fulfill specification 2. To tackle this problem, let us start with a remark.

A necessary condition for `d_rba` to fulfill specification 2 is that, on any partition `p`, `dfsp'` contains all the values of `dist2global dfs` corresponding to the range of `rssp`.

This can be easily seen by considering the above incomplete definition of `d_rba` together with specification 2. The latter has to hold for any subsequence `jsp = djs p` of `[os!!p .. os!!(p+1)-1]`. In particular, it has to hold for `jsp = [0 .. (length rssp - 1)]`. For this case the computation of `d_rba` on partition `p` requires the evaluation of `dist2global dfs` on the range of `rssp`. Therefore `fsp'` has to contain the values

```
[fs!!i | i <- nub (concat rssp), i < os!!p || i >= os!!(p+1)]
where fs = dist2global dfs
      rssp = drss p
      os = val (offs dfs)
```

where `nub :: [a] -> [a]` is a standard function that eliminates duplicates from a list. Accordingly

```
length fsp' ≥ length (nub (concat rssp))
where fsp' = dfs' p
      rssp = drss p
```

This remark suggests to compute `dfs'` as a *minimal extension* of the distributed data `dfs`. This can be done by appending the correspondent missing data to the projections of `dfs`:

```
dfs' p = dfs p ++ [fs!!i | i <- nub (concat rssp),
                      i < os!!p || i >= os!!(p+1)]
where fs = dist2global dfs
      rssp = drss p
      os = val (offs dfs)
```

The above expression cannot be used to compute `dfs'` in our BSP model. This is, among others, because `dist2global` cannot be used in this model to write parallel programs: as explained in section 3, it is neither a local computation nor a communication stage. In fact, the above expression is a specification and our next obligation is to write a program that computes `dfs'` and fulfills such specification. To do this, consider the following auxiliary functions:

```
d_rqs :: D [[Int]] -> D [a] -> D [(Int,Proc)]
d_rqs drss dfs = do p <- pid
                  rssp <- drss
                  return [(i,q) | i <- nub (concat rssp),
```

```

                                q <- [0 .. np-1],
                                q /= p,
                                fos!!q <= i &&
                                i < fos!!(q+1)]
    where fos = val (offs dfs)

```

```

d_mrqs :: D [(Int,Proc)] -> D [a] -> D [(a,Proc)]
d_mrqs drqs dfs = do p <- pid
                    rqsp <- drqs
                    fsp <- dfs
                    return [(fsp!!(i-fos!!p),q) |
                            (i,q) <- rqsp]
                    where fos = val (offs dfs)

```

The function `d_rqs` takes a distributed relation `drss` and a distributed function `dfs`. It computes, on each partition `p`, the list of those pairs (i,q) such that i is in the range of `drss p` and $(\text{dist2global } \text{dfs})!!i$ is stored on partition $q \neq p$. The result can be conveniently interpreted as a distributed *list of requests*. On partition `p`, a pair (i,q) of `d_rqs drss dfs` represents a request *to* partition `q` to provide the value of $(\text{dist2global } \text{dfs})!!i$; conversely, a pair (i,q) of `exch (d_rqs drss dfs)` represents a request *from* partition `q` and any such pair satisfies:

$$\text{os}!!p \leq i < \text{os}!!(p+1) \wedge q \neq p$$

This is equivalent to say that $(\text{dist2global } \text{dfs})!!i$ is stored on partition `p` (in $(\text{dfs } p)!!(i - \text{os}!!p)$) and i is in `concat (drss q)`.

The function `d_mrqs` takes a distributed list of requests `drqs` and a distributed function `dfs` and computes, for each (i,q) in `drqs p`, the pair $((\text{dist2global } \text{dfs})!!i,q)$. The resulting distributed list of type `D [(a,Proc)]` represent the answer of the local partition to the requests issued by remote partitions. It is important to understand the mechanics of `d_mrqs` and `d_rqs`. One has:

```

(exch (d_mrqs (exch (d_rqs drss dfs)) dfs)) p
=
[(fs!!i,q) | i <- nub (concat (drss p)),
             q <- [0 .. np-1],
             q /= p,
             os!!q <= i && i < os!!(q+1)]
where fs = dist2global dfs

```

Thus, combining `d_rqs` and `d_mrqs`, it is easy to write a program that computes `dfs'` given `drss` and `dfs`:

```

complete :: D [[Int]] -> D [a] -> D [a]
complete drss dfs = do fsp <- dfs
                    mrqsp <- dmrqs
                    return fsp ++ (map fst mrqs)
                    where dmrqs = exch (d_mrqs drqs dfs)
                          drqs = exch (d_rqs drss dfs)

```

Two remarks are at place here. The first one is that a call to `complete` requires exactly two communication stages: these are the two calls to `exch` in the computation of `dmrqs` and `drqs`.

The second remark is that, for the computation of the distributed list of requests `drqs`, one does not actually need the values of `dfs`: in `d_rqs`, `dfs` is only used to compute the offsets `os`. This means that, for a given distributed relation, the same list of requests can be used for all functions distributed according to the same offsets.

This suggests that, for applications in which distributed relation-based algorithms are called in some iterative procedure – for instance to compute a matrix-vector multiplication in some iterative method for solving systems of equations – it might be convenient to compute (and store) `drqs` outside the iteration and implement a version of `complete` that takes `drqs` as an extra argument. This approach allows one to reduce the number of calls to `exch` per iteration from 2 to $(n + 1)/n$ where n is the number of iterations. For many scientific computing applications this means, in practice, halving the communication costs and can lead to significant improvements in the speed of the overall computation.

Using `complete`, one can compute the modified distributed data `dfs'` as

```
dfs' = complete drss dfs
```

In order to finalize the definition of `d_rba` we have now to compute the *access table* `drss'`. Obviously, `drss'` is distributed as `drss`:

```
offs drss' = offs drss
```

Moreover, it has the same *shape* as `drss`:

```
map length (drss' p) = map length (drss p)
```

The distributed relation `drss'` has to provide, on each partition `p`, the access to the values of `(dist2global dfs)!!i` for each `i` in `concat (drss p)`. Therefore it has to fulfill

```
(concat (drss' p))!!k = if (os!!p <= i < os!!(p+1))
```

```

then i - os!!p
else lfsp +
    elemIndex i (map fst rqsp)
where i = (concat (drss p))!!k
      lfsp = length (dfs p)
      os = val (offs dfs)
      rqsp = (d_rqs drss dfs) p

```

where `elemIndex` is a standard function of the `List Haskell` module that computes the position (the index) of a given element in a list. In much the same way as for `dfs'`, we have to write a program for computing `drss'` that fulfills the above specification. This can be easily done on the basis of the functions already introduced. We first introduce a function that rescales a global index `i` according to the offsets `os` and to the list of requests `rqs`:

```

rescale :: Int -> [Int] -> Int -> [(Int,Proc)] -> Int
rescale i os q rqs = if (os!!q <= i && i < os!!(q+1))
    then (i-os!!q)
    else (fromJust
        (elemIndex i (map fst rqs))
        ) + os!!(q+1) - os!!q

```

Using `rescale` it is easy to write a function `d_acc` that computes the distributed access table `drss'` from `drss` and `dfs`:

```

d_acc :: D [[Int]] -> D [a] -> D [[Int]]
d_acc drss dfs = do p <- pid
    rssp <- drss
    rqsp <- d_rqs drss dfs
    return [ [ rescale i os p rqsp |
                i <- rssp!!j ] |
                j <- [0 .. (length rssp) - 1] ]
    where os = val (offs dfs)

```

In much the same way as for `d_rqs`, one does not actually need the values of `dfs` in order to apply `d_acc`: in `d_acc`, `dfs` is only used to compute the offsets `os`. In the above implementation, `os` are computed twice: in the computation of `d_rqs drss dfs` and in `d_acc` itself. This suggests that it might be convenient to compute access and exchange tables together.

With `d_acc` and `complete` it is now easy to implement a distributed relation based algorithm:

```

d_rba :: D [[Int]] -> D [a] -> ([a] -> b) -> D [Int] -> D [b]
d_rba drss dfs h djs = do p <- pid
    rssp' <- d_acc drss dfs

```

```

fsp' <- complete drss dfs
jsp <- djs
return (rba rssp' fsp' h jsp)

```

5 Relation-based computations

In 2.2 we have explained the need for additional operations on relations, roughly classified as “relation-based computations”. Here, we reprise the examples of relation composition and converse, starting from their set-theoretical definitions, going to their Haskell specifications for the sequential case, and using the monadic model in order to specify them in the distributed context.

5.1 Relational composition

The composition of two relations $R_1 : [0 \dots m) \longleftarrow [0 \dots n)$ and $R_2 : [0 \dots n) \longleftarrow [0 \dots p)$ gives a relation $R_1 \cdot R_2 : [0 \dots m) \longleftarrow [0 \dots p)$ defined as

$$\begin{aligned}
i(R_1 \cdot R_2)k &\equiv \exists j \in [0 \dots n) \text{ such that} \\
&iR_1j \wedge jR_2k
\end{aligned}$$

When we use the isomorphism of relations to set-valued functions, as explained in the beginning of the previous section, relational composition does not translate to functional composition (that would be impossible due to the type mismatch) but to the following operation:

$$\begin{aligned}
R_1 &: P [0 \dots m) \longleftarrow [0 \dots n) \\
R_2 &: P [0 \dots m) \longleftarrow [0 \dots p) \\
&\Rightarrow \\
R_1 \triangleleft R_2 &: P [0 \dots m) \longleftarrow [0 \dots p) \\
(R_1 \triangleleft R_2)k &= \bigcup \{R_1(j) \mid j \in R_2(k)\}
\end{aligned}$$

Here and in the sequel, we overload the names of the relations to stand for the corresponding set-valued functions.

A more computational version of this specification is easily obtained by replacing sets by their list representations and set-specific operations by their implementational equivalents:

```

after :: [[Int]] -> [[Int]] -> [[Int]]
(rss1 'after' rss2)!!k = (nub.concat) [rss1!!j | j <- rss2!!k]

```

where the following conditions represent the typing information:

```

length (rss1 'after' rss2) == length rss2
length rss1 > maximum (concat rss2)

```

Since the length of the resulting list is known, we can apply the following property of lists:

```

xs!!i = e(i) and length xs == n
≡
xs     = [e(i) | i <- [0 .. n-1]]

```

(2)

and thus arrive at the following specification of sequential relational composition:

```

rss1 'after' rss2 = [(nub . concat) [rss1!!j | j <- rss2!!k] |
                    k <- [0 .. p-1]]
                    where p = length rss2

```

A distributed version of `after` is a function

```

d_after :: D [[Int]] -> D [[Int]] -> D [[Int]]

```

which fulfills the following specification:

```

-- specification 3
dist2global (drss1 'd_after' drss2) = rss1 'after' rss2
where rss1 = dist2global drss1
      rss2 = dist2global drss2

```

The implementation of such a function is much simplified by the observation that relational composition is, in fact, a relation-based algorithm. Indeed:

```

rss1 'after' rss2 = rba rss2 rss1 (nub.concat) doms
                    where doms = [0 .. (length rss2)-1]

```

Therefore, we can use the results of the previous section to arrive at:

```

drss1 'd_after' drss2
=
d_rba drss2 drss1 (nub.concat) ddoms
where ddoms = do rss <- drss2

```

```
return [0..(length rss)-1]
```

From the specification of distributed relation-based algorithms, we see that the requirement for `d_after` is satisfied if

```
doms = dist2global ddoms
```

i.e.

```
[ 0 .. (length rss2)-1 ]
=
dist2global (do rss <- drss2
              return [0 .. length(rss)-1])
where rss2 = dist2global drss2
```

which is trivial, if tedious.

5.2 Converse

Another basic operation on relations is *converse*, defined in 2.2 as

$$jR^\circ i \equiv iRj$$

where $R^\circ : [0 \dots n) \longleftarrow [0 \dots m)$, $R : [0 \dots m) \longleftarrow [0 \dots n)$

Using the isomorphism between relations and set-valued functions, we can translate this definition to

$$j \in R^\circ(i) \equiv i \in R(j)$$

or, a more “constructive” version:

$$R^\circ(i) = \{j | j \in [0 \dots n), i \in R(j)\}$$

where the types involved are now

$$R^\circ : P[0 \dots n) \longleftarrow [0 \dots m)$$

$$R : P[0 \dots m) \longleftarrow [0 \dots n)$$

In order to obtain a specification for the sequential case, we just need to replace the function application operation with an indexing one, and the membership relation for sets with the one for the data structure used to implement sets, in our case lists:

```
j 'elem' rcss!!i == i 'elem' rss!!j
```

where the typing conditions are expressed by

```
rcss, rss :: [[Int]]
length rss == n
length rcss == m
maximum (concat rss) < m
maximum (concat rcss) < n
```

This specification is not directly implementable, which leads us to consider a new specification which corresponds to the constructive one given above, where we replace the set comprehension with a list comprehension:

```
rcss!!i = [j | j <- [0 .. n-1], i 'elem' rss!!j]
```

Applying property 2, we obtain:

```
rcss = [[j | j <- [0 .. n-1], i 'elem' rss!!j |
        i <- [0 .. m-1]]
```

This leads immediately to the executable specification:

```
converse :: [[Int]] -> Int -> [[Int]]
converse rss m = [[j | j <- [0 .. n-1], i 'elem' rss!!j |
                  i <- [0 .. m-1] ]
                where n = length rss
```

Notice that the size of the target of `rss` cannot be deduced from `rss`, and therefore has to be supplied as an argument to `converse`.

The following remarks, while referring to the specifications above, point out more general aspects of the road from mathematical descriptions to implementations:

- (1) while the set-based definitions of `converse` are equivalent, the non-constructive specification is not equivalent to the constructive one (due to the ordering imposed by the list comprehension). This illustrates the more general fact that translating equivalent definitions into specifications does not result in equivalent specifications, i.e. a program may satisfy one of the specifications, but not the other.
- (2) the program defining `converse` is very inefficient, and should not be understood to be the final point in the implementation of a sequential converse operator. Using the properties of standard Haskell data structures and functions, it is possible to derive a more efficient version (which will provably satisfy the initial specification). Such a version has been taken

as the starting point for SCDRC implementations.

Let us now turn to the problem of specifying a distributed version of `converse`. It is tempting to start by requiring

```
d_converse :: D [[Int]] -> D [[Int]]
dist2global (d_converse drss) = converse rss
where rss = dist2global drss
```

but that will, of course, not work: the sequential version needs an additional argument. We could, therefore, try

```
d_converse :: D [[Int]] -> Int -> D [[Int]]
dist2global (d_converse drss m) = converse rss m
where rss = dist2global drss
```

but now we face the following difficulty: the specification does not distinguish between different results corresponding to the same global list. In particular, a program which first collects all data on a single partition and then applies the sequential `converse` will satisfy this specification. If we want to avoid this possibility, we must specify a partitioning of the resulting relation, e.g. by supplying the offsets of this relation.

```
d_converse :: D [[Int]] -> D [Int] -> D [[Int]]
d_converse drss dos = drcss
```

≡

```
offs drcss == dos
dist2global drcss == converse rss m
where rss = dist2global drss
      m = last (val dos)
```

Deriving an executable version of `d_converse` is complicated by the fact that we cannot use the sequential version in a program of the form

```
d_converse drss dos = do rss' <- drss'
                      p    <- pid
                      os   <- dos
                      return (converse rss' m)
                      where
                        m = os!!(p+1) - os!!p
                        drss' = ...
```

because, due to our representational choice, the source of `rss'` above consists of integers in `[0 .. length(rss')-1]`, therefore the target of the `converse`

of `rss` will also consist of integers in the same interval, while the elements of `(drcss p)` can lie outside this interval. Obtaining the information required to scale these elements requires as much computational effort as computing converse itself.

The solution, which we cannot present here in detail (but see ???), is to compute on every partition a list of pairs which are going to make up the converse relation, together with the partition where this information will be stored:

```
d_pairs drss dos = do rss <- drss
                    os  <- dos
                    return [((j, i), q) |
                              j <- [0 .. (length rss)-1],
                              i <- rss!!j,
                              q <- [offs2part os i]]
```

The resulting distributed value can then be used by `exch`. After exchange, every partition has locally all the necessary information to compute the correct local segment of the converse relation. `;`

6 C++ implementation and Haskell documentation

In the previous sections we have introduced relation-based computations and we have presented a new model of BSP computations. The model, implemented in Haskell, has been used for the specification of relation-based computations and for simulating parallel distributed SPMD implementations of simple sequential algorithms.

In this section we show how the new BSP model can be used to document and understand a simple C++ program for computing the offsets of a distributed array. The program uses very few SCDRC components and is simple enough to keep the presentation compact. The C++ implementation can be easily understood without advanced C++ expertise. In discussing the offsets computation example, we outline a few elements of the design and of the architecture of SCDRC. For a detailed description of SCDRC we refer to [10].

SCDRC provides support for offsets computation through a template void function

```
template<class A>
void
offsets(Array<Nat, A>& res, const Nat sz);
```

This function is defined in the file

`spmd_distr_ops/src_cc/SPMD_Distr_ops.h`

We say that `offsets` is implemented in the `spmd_distr_ops` component. In SCDRC, software components are associated with directory names. Thus, `spmd_distr_ops` contains programs that implement standard operations (suffix `ops`) in the single program multiple data (SPMD) computational environment.

SCDRC supports two computational environments: a single program single data (SPSD) environment for sequential computations (prefix `SPSD`) and an SPMD environment for distributed parallel computations (prefix `SPMD_Distr`).

In SCDRC, computational environments are implemented as namespaces. Thus, `offsets` is embedded in the `SPMD_Distr` namespace. In [10] we motivate this design and compare it to other approaches, e.g. to singleton type implementations.

Using SCDRC, a simple C++ program for computing the offsets of a distributed array could be written as follows:

```
1  #include <numeric_types/src_cc/Nat.h>
2  #include <spmd_distr_ops/src_cc/SPMD_Distr.h>
3  #include <spmd_distr_ops/src_cc/SPMD_Distr_ops.h>
4  #include <spmd_distr_array/src_cc/SPMD_Distr_Array.h>
5  #include <spmd_distr_array/src_cc/SPMD_Distr_Array_ops.h>
6  using namespace std;
7  using namespace SPMD_Distr;
8  using local::size;
9  using local::operator<<;
10
11 int main(int argc, char **argv) {
12     initialize(argc, argv);
13     Array<Nat> pjs(p() +1, 0);
14     Array<Nat> os;
15     offsets(os, size(pjs));
16     finalize();
17     return 0;
18 }
```

The program explicitly uses three SCDRC components via `#include` directives. The first component, `numeric_types`, provides a type `Nat` for representing natural numbers. In SCDRC, `Nat` has been implemented with a simple `typedef` and does not depend on the computational environment. The other two components explicitly refer to the SPMD distributed computational environment.

Applications on the top of SCDRC are usually developed in one computational environment: either SPSD or SPMD distributed. It is possible but unusual to include both `spsd` and `spsd_distr` SCDRC components.

At lines 6 and 7 all the names belonging to the namespaces `std` and `SPMD_Distr` are injected in the local scope. Lines 8 and 9 inject the names `local::size` and `local::operator<<` in the local scope. As explained in detail in [10], `local` is a namespace nested in `SPMD_Distr` and all local operations are implemented, in the SPMD distributed computational environment, in `local`.

Line 12 and 16 in `main` initialize and finalize the SPMD distributed computational environment. Between line 12 and line 16, `n_p()` clones of the program run in parallel on `n_p()` partitions.

On the p -th partition, an array of natural numbers of size $p+1$ is allocated and filled with zeros at line 13. `Array` is a SCDRC data structure. It extends STL `vector` with bounds checking and with a set of computational environment dependent elementary operations.

At line 15 `local::size` is used (remember line 8) to query the size of `pjs`. This computation can be expressed, in our monadic BSP model, with the code fragment

```
do pjs <- djs
  return (length pjs)
```

The offsets of `djs` are computed at line 15 by calling the SCDRC function `offsets` with the local arguments `os` and `size(pjs)`.

It is easy to model the parallel program for computing the offsets of a distributed array in the monadic BSP model presented in section 3. The corresponding Haskell code would be:

```
1  import D
2  import D_ops
3
4  djs = do p <- pid
5        return [0 | i <- [0..p]]
6  dos = offs djs
```

Here `D` is the module that implements the new BSP model. `D_ops` implements a number of distributed operations, among others the `offs` computation presented in section 3. At lines 4-5 we compute the distributed list of natural numbers `djs`. The offsets of `djs` are then computed at line 6.

We believe that the Haskell implementation helps understanding and clarifying

the C++ program in different ways. First, it offers a complementary view of the parallel computation itself: while the C++ code emphasizes a *local* view (no distributed data appear explicitly in the C++ implementation), the Haskell program explicitly deals with distributed entities and with functions operating on distributed entities.

Second, the Haskell model offers a *consistent* view of the parallel offsets computation: the function `offs` is a polymorphic distributed function of type `D [a] -> D [Int]`. It takes a distributed list (of elements of arbitrary type `a`) and returns a distributed list of integers. The SCDRC `offsets` function, on the other side, formally acts on local data only. Obviously `offsets` has access to the sizes of `pjs` on remote partitions through a global communication step. Its signature, however, is not distinguishable from that of a `local` function.

The Haskell program represents the essence of the parallel computation implemented in the C++ code in a concise and effective way: it provides a form of documentation that scales well with increasing algorithmic complexity. While the C++ program for computing the offsets of a distributed array is fairly straightforward and easily understandable, the same is not necessarily true for more realistic parallel programs. As an example, consider the SCDRC implementation of `offsets` itself:

```
template<class A>
void
offsets(Array<Nat, A>& res, const Nat sz) {

    using local::breadth;
    using local::pos;
    using local::resize;
    using local::is_offsets;

    CRS<Nat> splitsz;
    resize(breadth(splitsz), n_p());
    resize(pos(splitsz), n_p() + 1);
    pos(splitsz)[0] = 0;
    for(Nat i = 0; i < n_p(); i++) {
        breadth(splitsz)[i] = sz;
        pos(splitsz)[i + 1] = pos(splitsz)[i] + 1;
    }

    CRS<Nat> splitszp;
    exch(splitszp, splitsz);

    resize(res, n_p() + 1);
    res[0] = 0;
```

```

for(Nat i = 0; i < n_p(); i++)
  res[i + 1] = res[i] + breadth(splitszp)[i];

ENSURE(is_offsets(res));
}

```

We do not provide a program dissection of `offsets` here although the C++ is only marginally more complex than the main program discussed above. Instead, we contrast the C++ program with the Haskell implementation presented in section 3:

```

offs dxs = do sizes <- dsizes
          return (map (sum . inits) (map fst sizes))
          where dsizes = exch (do xs <- dxs
                                return (repl (length xs)))

```

We believe that the monadic BSP model proposed here is a useful tool for documenting and explaining SCDRC implementations. It is concise and expressive enough to reason about parallel programs and to effectively develop and test prototype parallel versions of sequential algorithms.

7 Conclusions

We have presented in this paper an important computational pattern, the “relation-based algorithm”, together with some related functions, in a distributed, BSP setting. To this end, we have introduced a Haskell monadic model of BSP programs, allowing us to give high-level, executable specifications of distributed algorithms. We have also shown an example of the C++ implementation of one of the simpler (shorter) Haskell specifications.

There are a number of aspects which we could not present here without diluting the focus of the paper (and massively increasing its size). We have not given detailed proofs of many of the properties we have used in our analysis of Haskell specifications. We have glossed over the aspects of more general data structures, and their distributed representations, limiting ourselves to the use of lists.

The question of generic data structures becomes paramount the closer we get to implementations from specifications. Efficiency requirements motivate, for example, the usage of arrays instead of lists whenever random access is more frequent than insertion or deletion operations. Data is sometimes better represented not as something stored in a container, but as resulting from a computation. For example, in the case of an unstructured grid, coordinates of nodes

must be stored, but in the case of a structured grid, they can be computed. A generic implementation should ideally allow for an efficient treatment of all these cases. Since this question goes beyond that of specification of distributed algorithms, we have not been able to describe our (provisional) solutions in this paper.

This brings us to the issue of future work. The major task is to investigate the possibility of a principled approach to the derivation of implementations from specifications. We need to build a collection of representation and other high-level theorems, that ease the burden of proofs and raise the expressivity of our specification language. We will attempt to specify all the functionalities implemented in SCDRC using the Haskell monadic model, and try to shorten the way to the C++ implementations by, for instance, deriving efficient Haskell algorithms from the specifications. Finally, we aim to write specifications and implementations of applications in terms of the SCDRC computations.

Acknowledgements

It is our pleasure to thank R. Klein, C. Linstead, A. Priesnitz, S. Schupp and J. Gerlach for contributing to the development and to the benchmarking of prototype software components for distributed relation-based computations and for the many interesting discussions.

The work presented in this paper heavily relies on free software, among others on hugs, vi, the GCC compiler, Emacs, L^AT_EX and on the FreeBSD and Debian / GNU Linux operating systems. It is our pleasure to thank all developers of these excellent products.

This research was funded partly by R. Klein, Gottfried Wilhelm-Leibniz-Preis 2003.

References

- [1] R. Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, 1998.
- [2] R. H. Bisseling. *Parallel Scientific Computation: A Structured Approach using BSP and MPI*. Oxford University Press, Oxford, UK, 2004.
- [3] W. F. McColl D. B. Skillicorn, J. M. D. Hill. Questions and answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.

- [4] S. L. Peyton Jones et al. *Haskell 98 Language and Libraries: the Revised Report*. Cambridge University Press, 2003.
- [5] G. Hains F. Loulergue and C. Foisy. A calculus of functional BSP programs. *Science of Computer Programming*, (37):253–277, 2000.
- [6] K. Schloegel G. Karypis and V. Kumar. Parallel Graph Partitioning and Sparse Matrix Ordering Library. <http://www-users.cs.umn.edu/~karypis/metis/parmetis/files/manual.pdf>, 2003.
- [7] C. Ionescu. A new monadic model of BSP computations. Technical report, PIK Report No. XX, Potsdam Institute for Climate Impact Research, 2006. in preparation.
- [8] G. Karypis and V. Kumar. A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices. <http://www-users.cs.umn.edu/~karypis/metis/metis/files/manual.pdf>, 1998.
- [9] Q. Miller. BSP in a lazy functional context. In K. Hammond, editor, *Trends in Functional Programming*. 2002.
- [10] C. Linstead N. Botta, C. Ionescu and R. Klein. Structuring distributed relation-based computations with SCDRC. Technical report, PIK Report No. XX, Potsdam Institute for Climate Impact Research, 2006. in preparation.
- [11] J. Peterson P. Hudak and J. Fasel. A Gentle Introduction to Haskell, Version 98. <http://www.haskell.org/tutorial>, 2000.
- [12] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

Dear Editors,

we hereby submit the following manuscript: "Relation-based computations in a monadic BSP model".

sincerely,

Nicola Botta