STRUCTURING DISTRIBUTED RELATION-BASED COMPUTATIONS WITH SCDRC

Nicola Botta, Cezar Ionescu, Ciaron Linstead, Rupert Klein
In this report we present a set of software components for distributed relation-based computations (SCDRC). We explain how SCDRC can be used to structure parallel computations in a single-program multiple-data computational environment.

First, we introduce relation-based algorithms and relation-based computations as generic patterns in scientific computing. We then discuss the problems that have to be solved to parallelize such patterns and propose a high-level formalism for specifying these problems.

This formalism is then applied to derive parallel distributed relation-based computations. These are implemented in the C++ library SCDRC. We present language independent elements of SCDRC and discuss C++ specific aspects of its design and architecture.

Finally, we discuss how to use SCDRC in a simple application and provide preliminary performance figures.
1. Introduction

1.1. What is SCDRC? SCDRC is a set of software components for structuring distributed relation-based computations.

Relation-based computations are simple but general patterns found in many scientific computing domains. In climate research, they are at the core of grid-based numerical methods for partial differential equations (ocean and atmosphere models), of inference algorithms for Bayesian networks and of viability kernel algorithms (viability studies). They also arise in data interpolation between regular and irregular grids (pre-processing, model coupling).

Relation-based computations are often nested in expensive, iterative programs. These programs could, in principle, take advantage of distributed parallel architectures to speed up computations.

In climate research, faster computations allow simulations on longer time scales, improved resolution and more representative sets of realizations in uncertainty studies.
In practice, however, computational tools for climate research do not take full advantage of parallel computers. Although based on a small set of common computational patterns, climate models are traditionally developed for specific domains in a sequential computational environment. They are neither easy to parallelize nor cheap to adapt to other domains. If those common patterns could be organized in a generic library, which can be used across different domains, substantial development efforts could be saved and the degree of parallelism could be increased.

SCDRC is a prototype of such a generic library.

1.2. What are relation-based computations? The notion of relation-based computations is introduced and discussed in detail in the next section. Examples of relation-based computations 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.

Relation-based computations which can be easily implemented in a sequential, single program single data (SPSD) computational environment are often difficult to implement in a parallel, single program multiple data (SPMD) distributed case.

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 sequential computation of the centers could read:

\[
\text{Algorithm 1: triangle centers}
\]

\[
\text{for } j \text{ in } \{0 \ldots \text{size}(vt)\} \text{ do}
\]

\[
\text{compute } (1/3) \ast (x(vt(j)(0)) + x(vt(j)(1)) + x(vt(j)(2)))
\]

\[
\text{end for}
\]

In algorithm 1, \([a \ldots b]\) represents the interval of natural numbers \( a, a + 1, \ldots, b - 1 \). It is not obvious how to implement the above rule on a parallel computer with distributed memory. If one requires the implementation to be reasonably efficient, one has to answer, among others, the following questions:

1. How are \( vt \) and \( x \) distributed among remote partitions?\(^1\)
2. Which vertex coordinates are needed on the local partition which are stored on remote partitions?
3. Which vertex coordinates stored on the local partition are needed by which remote partition?
4. How can these coordinates be exchanged between partitions?

A few remarks are appropriate here: if the efficiency requirement can be neglected, a parallel implementation of algorithm 1 can be easily derived by just duplicating the whole \( x \) on all partitions. This approach is, for most practical problems, unaffordable. Moreover, it raises the non-trivial question of how to ensure the consistency of duplicated data.

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

\(^1\)we do not attempt at defining the concept of a partition here: in a SPMD (single program multiple data) distributed computational environment, different partition may correspond to remote memory spaces.
parallel computation. In particular, the answers to question 2 and 3 essentially depend on how question 1 is answered.

Of course, structuring rules or guidelines cannot be given in general but only for certain classes of computations or computational patterns. Relation-based computations are a family of such patterns.

1.3. **Who can take advantage from SCDRC?** As a set of components for structuring distributed relation-based computations, SCDRC is a software layer above message passing libraries but below applications. It is not meant to be directly used by application developers. Instead, SCDRC is designed to be the basis on which application dependent software components are written. In other words, applications are expected to use SCDRC indirectly via application dependent abstractions.

As an example, consider a triangulation class supporting the implementation of finite element discrete differential operators. This is an application dependent abstraction (in the sense that it will implement, among others, methods which are specific to finite element computations) which could be written on top of SCDRC. A finite element program for approximating incompressible flows is an example of an application.

Being a low-level software layer (w.r.t. applications), SCDRC does not attempt at hiding the communication steps which are needed in relation-based computations. Communication steps which are conceptually complementary but distinct are represented by distinct data structures or function calls.

This means that developers have a high degree of control over communication and can take advantage of such control for optimizations. However, communication is structured in a set of primitives which have been specifically designed for relation-based computations. In particular, SCDRC users do not have direct access to standard message passing (MPI) primitives and do not need to care about synchronization, mutual exclusion, deadlock or race condition problems. They can develop application dependent software components on the top of SCDRC which hide communication or leave it visible to the user.

1.4. **How does SCDRC compare to other approaches?** A discussion of the many and different approaches towards introducing abstraction layers between message passing libraries and scientific computing applications goes beyond the scope of this report.

For an overview of the role of subroutine libraries and of frameworks in generic software components for scientific computing we refer the reader to [4]. A comprehensive discussion of domain specific languages, frameworks and toolkits from the point of view of domain engineering can be found in [9]. The concepts of grid and of algorithm oriented design of software components for grids and geometries are discussed in [1], [3] and [5].

SCDRC has been designed around patterns which are found in numerical methods for partial differential equations (PDE) and in other applications domain: adaptive stochastic sequential decision processes and Bayesian network inference are two prominent examples. Up to now, the most significant efforts towards developing frameworks of generic, reusable software components have been done in the PDE application domain. In the following, we point out differences and similarities between SCDRC and well established frameworks for solving PDEs.
A very elementary difference between SCDRC and frameworks like POOMA\(^2\), Overture\(^3\), Amatos\(^4\) and OpenFoam\(^5\) is in terms of size. SCDRC is a very small, thin software layer: at the present stage, sloccount\(^6\) counts about 12000 source lines of code in the main directory tree of SCDRC. As a comparison, OpenFoam version 1.2 consists of about one million lines of code!

Another important difference between SCDRC and computational frameworks for PDE lies in the level of abstraction. The central concepts in SCDRC are relations and relation-based computations. The main problems addressed by SCDRC are how to represent distributed relations and how to implement parallel relation-based computations.

In numerical frameworks for PDEs, grid concepts play an outstanding role. Grids are much more complex concepts than relations. One can think of relations in a couple of different ways – as sets of pairs, as characteristic functions or as functions – and one can distinguish between different kinds of relations: regular relations, irregular relations, etc. This complexity, however, is very little when compared to the complexity of grid concepts. The grids needed in computational frameworks for PDEs have geometrical and topological aspects. The latter are described by a whole set of grid relations. Grid representations depend on the choice of a coordinate system, on the number of dimensions of the geometrical space in which they are embedded, on a number of grid coordinates. One can distinguish between structured and unstructured grids, between regular and irregular grids, between rectangular, skew and curvilinear grids. Adaptive grids, hierarchical grids, overlapping and non-overlapping grids are other aspects of different grid taxonomies.

The development of SCDRC is an attempt at tackling the problem of structuring algorithm-oriented parallel computations on the basis of the smallest common concept and of the simplest computational patterns found in a wide class of scientific computing problems: relations and relation-based computations.

Since it tackles the parallelization problem at an elementary level, the SCDRC approach is more similar to the algorithmic skeletons or to POOMA’s stencil-based approach than to grid-based domain decomposition approach. As it will become clear in the following sections, relation-based algorithms and relation-based computations are, in fact, non-trivially parallelizable data parallel algorithmic skeletons in the sense of [14].

Special kind of relations – symmetric, anti-reflexive graphs – play a fundamental role in graph partitioning algorithms such as those implemented in the Metis [12] and ParMetis [8] library. As we will see in the next section, SCDRC provide an interface to these libraries. The interface allows one to apply partitioning algorithms to SCDRC relations. The application domains of Metis and ParMetis – graph and grid partitioning – on the one side and of SCDRC on the other side are complementary but clearly separated.

As a set of software components for structuring distributed relation-based computations, SCDRC provides a subset of the functionalities provided by the Janus framework, see [10]. In fact, the initial phase of the SCDRC development has been done in collaboration with Dr. J. Gerlach, the main developer of Janus. As we will show

---

\(^2\)http://acts.nersc.gov/pooma
\(^3\)http://acts.nersc.gov/overture
\(^4\)http://www.amatos.info
\(^5\)http://www.opencfd.co.uk/openfoam
in section 3, SCDRC shares with Janus (and with ParMetis) the conceptual model of representing distributed functions and relations. There are, however, important differences between SCDRC and Janus. These differences are both in implementation independent aspects and in the implementation design.

A major implementation independent difference is architectural. In contrast to Janus, the architecture of SCDRC is based on the formal specification of a small set of problems. These problems are informally introduced at the end of the next section and problem specifications are discussed in detail in section 3.

Another difference between SCDRC and Janus is in how solution algorithms for the problem specifications of section 3 have been derived. In SCDRC, this has been done on the basis of a single communication primitive in the spirit of the BSP (bulk synchronous parallel processing) model, see [7], [6]. This communication primitive is discussed in detail in 3 and 4. In contrast to SCDRC, Janus algorithms are not explicitly designed around a single communication primitive, and attempt to hide the distinction between parallel and sequential execution.

A third major difference between Janus and SCDRC is in the approach towards constructing relations. Janus supports incremental construction with a very flexible (albeit non trivial) two-phase model. SCDRC takes a more straightforward approach and does not support incremental construction.

1.5. What is the state of development of SCDRC? SCDRC is in a prototypical stage. Its sources are available under the GPL licence but have not been released (please contact botta@pik-potsdam.de). SCDRC has been compiled with gcc version 3.3.6 and 4.0.2 and tested on a linux-cluster under lam-mpi and mpich and on a 240 CPU IBM p655 cluster. At the present, no application dependent software components have been built on the top of SCDRC but a few simple examples are provided.

1.6. Outline. The rest of this report is organized as follows. In section 2 we introduce and discuss relation-based algorithms and relation-based computations following the triangle center example outlined above. Section 3 describes implementation independent design elements of SCDRC. In this section we discuss, among others, how distributed functions and distributed relations are conceptually represented – remember question 1) above – and which aspects of this representation are visible to SCDRC users. In section 4 we discuss implementation dependent aspects and architecture of SCDRC. In the last section we comment a simple application and discuss preliminary results.

2. Relation-based algorithms and relation-based computations

2.1. Relation-based algorithms. Let’s go back to algorithm 1 introduced in section 1 to represent a triangle center computation. In this rule we have used \(x(i)\) (for \(i\) equal to \(vt(j)(0)\), \(vt(j)(1)\) and \(vt(j)(0)\)) to represent the \(i\)-th element of the array \(x\).

We use the notation \(x(i)\) — in contrast to the more usual \(x[i]\) — to underline the fact that, for the purpose of expressing the triangle center computation, the way the vertex coordinates are obtained is immaterial. In concrete implementations, \(x\) does not need to be an array and one can easily think of triangulations in which the vertex coordinates are given by analytical expressions.

Let’s take a critical view at algorithm 1: what if the triangulation covers a sphere and the triangles themselves are spherical? In this case, algorithm 1 yields triangle
centers that do not lie on the surface of the sphere. This is probably not what a triangle centers algorithm is meant to compute. Algorithm 1 can be easily modified to avoid triangle shape over-specification:

**Algorithm 2**: triangle centers

```plaintext
for j in [0...size(vt)) do
  compute center(x(vt(j)(0)), x(vt(j)(1)), x(vt(j)(2)))
end for
```

The new rule delegates the computation of the centers to the center function. This function is now assumed to know whether plane of spherical triangles are at stake in any particular case. In fact, algorithm 2 can be easily generalized to compute whatever function of the coordinates of the triangle vertexes, for instance the triangle areas. This is, in contrast to center, a non-linear function. We can go one step further and think of algorithm 2 as a particular instance of a computational pattern in which `j` is drawn from some array `js` of positive natural numbers and a function `h` is applied to the array of values some function `f` takes at those indexes `i` which are in relation with `j`:

**Algorithm 3**: relation-based algorithm

```plaintext
for j in js do
  compute h([f(i) | i in R(j)])
end for
```

Here `R(j)` is an array of indexes which are in relation `R` with `j`, in the triangle center example `R(j) = [vt(j)(0), vt(j)(1), vt(j)(2)]`. The notation `[f(i) | i 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 call the above computational pattern a relation-based algorithm (RBA). Relation-based algorithms are defined in terms of two functions `h` and `f` and of a relation `R`.

In relation-based algorithms and, in general, in SCDRC, we will only consider relations between zero-based intervals of natural numbers. This restriction is discussed in detail in section 3. For the moment, let’s accept this restriction and think of a relation `R` as a subset of `[0, m) × [0, n)` where `m` and `n` are the sizes of the target and of the source of `R`, respectively. In this report we will always use a left-from-right notation when giving the signature of relations and functions:

```plaintext
R :: [0...m) ←− [0...n)
R(·) :: subarrays([0...m)) ←− [0...n)
```

If we allow `f` to take as argument pairs in `[0, m) × [0, n)`:}

**Algorithm 4**: relation-based algorithm

```plaintext
for j in js do
  compute h([f(i, j) | i in R(j)]),
end for
```
relation-based algorithms can be easily specialized to represent matrix-vector multiplications. Let $A$ be a sparse matrix and $c$, $e$ and $p$ be a CRS (compact row storage, see [8]) representation of $A$ that is, $c$, $e$ and $p$ satisfy the following equivalence:

$$A(j, i) \neq 0 \equiv \exists k \in [p(j) \ldots p(j+1)): i == c(k) \land A(j, i) == e(k)$$

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

**Algorithm 5**: sparse matrix vector multiplication

```plaintext
for j in [0 \ldots n) do
    compute sum([e(k) * b[c(k)] | k in [p(j) \ldots p(j+1)]]),
end for
```

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

$$h = \text{sum}$$

$$f(i, j) = e(k(i, j)) * b[i]$$

where $k(i, j) = p(j) + \text{index}(i, R(j))$

$$R(j) = [c(k) | k \in [p(j) \ldots p(j+1)]]$$

In the above expression we have used the function $\text{index}(\cdot)$ which computes the index of a given element in an array:

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

Of course, $\text{index}(\cdot)$ 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 $\text{index}(i, R(j))$. We will address this problem in section 4. For the moment let us summarize the results of the above analysis in the following observations:

- Relation-based algorithms are computational patterns (algorithmic skeletons) commonly found in many application domains; we have seen two examples: grid computations and linear algebra. Examples in numerical methods for PDEs and other application domains can be easily made.
- 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 $f$ values in memory. 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 $np$ partial relations $R_1 \ldots R_{np-1}$ yields non-disjoint ranges $\text{ran}(R_p), \text{ran}(R_q)$ for distinct $p, q < np$.

Unfortunately, many interesting computations, among others the examples discussed above, are not trivially parallelizable.

In section 4 we show how distributed relation-based algorithms can be defined in SCDRC by specializing a generic $RBA$ rule with concrete types for the functions $h$, $f$ and for the relation $R$. In SCDRC, concrete $RBA$ objects can be constructed by
passing concrete distributed objects representing \( h, f \) and \( R \) to \( RBA \) constructors. Distributed \( f \) objects can be, in turn, distributed relation-based algorithms. This provides a natural scheme for composing distributed relation-based algorithms to define complex parallel computations.

2.2. Relation-based computations. While being powerful patterns, RBAs are certainly not enough for structuring even simple distributed computations like the triangle centers example introduced in section 1. Let’s go back to this 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).

If we think of \( vt \) as of the relation \( R \) and of \( x \) as of the function \( f \) of a relation-based algorithm, then the following steps have to be done before a parallel computations of the triangle centers can take place:

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 partitioning.

We will discuss in detail what it means to distribute a relation and an array according to a given partitioning in the next sections. For the moment, let’s consider steps 2 and 3. Computing a partitioning of the source of \( vt \) simply means associating a unique partition number to each triangle of \( vt \).

Of course, one would like to partition the triangles of \( vt \) in such a way that the subsequent parallel computation of the centers is done efficiently. This boils down to requiring that all partitions contain approximately the same number of triangles (or a number of triangles 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 SC-DRC does not attempt at providing new solutions in this field. Instead, SCDRC provides an interface to Metis [12] and ParMetis [8]. These are very efficient graph partitioning libraries. The SCDRC interface could be easily extended to other partitioning algorithms.

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

\[
(iR_j \equiv jR_i) \land (iR_j \Rightarrow i \neq j)
\]

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 $vt$, $vt^\circ$.
2. compute $tvt = vt^\circ \cdot vt$.
3. compute $avt = tvt - id_{[0...source\_size(vt)]}$

We use $R^\circ$ to denote the converse of a relation (or of a function) $R$. If $R : [0...m] \longleftarrow [0...n]$, then $R^\circ : [0...n] \longleftarrow [0...m]$ 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 such partitioning, say $sp$, 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 $vt^\circ$) is then to pick-up, for each vertex $i$ in the source of $sp \cdot vt^\circ$, the partition number 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 $h = most\_frequent$. 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.

### 2.3. Core problems.

In this section we have introduced relation-based algorithms as computational patterns. We have seen that, in order to apply such patterns in a distributed parallel computational environment, other relation-based computations – among others composition and conversion – are needed. Of course, one would like these computations too to run in parallel and on distributed data.

In developing SCDRC, we have focused our attention on a few core problems. In order to implement distributed relation-based algorithms, these problems have to be solved no matter which programming languages and data structures are used for the implementation. Of course, concrete implementations will require the solution of more additional problems.

We close this section by listing the core problems informally, as they have been formulated at the beginning our analysis. In the next two sections we will introduce a
more formal specification, discuss the most important elements of the SCDRC architecture and show how SCDRC components can be combined to implement distributed relation-based computations.

1. Given a distributed representation of a function \( f \) and of a partitioning of its source, compute a new distributed representation of \( f \) consistent with the given partitioning.

2. Given a distributed representation of a relation \( R \) and of a partitioning of its source, compute a new distributed representation of \( R \) consistent with the given partitioning.

3. Given a distributed representation of a function \( f \) and given, on each partition, a subset of \( \text{dom}(f) \), compute, on each partition, the correspondent values of \( f \).

4. Given a distributed representation of a relation \( R \) and of a partitioning of its target, compute a distributed representation of \( R^c \) consistent with the given partitioning.

5. Given consistent, distributed representations of relations \( S \) and \( T \), compute a consistent, distributed representation of \( S \cdot T \).

3. Implementation independent elements

3.1. Set, function and relation representations. As mentioned in the previous section, SCDRC relations are defined between zero-based intervals of natural numbers. Very often, such relations represent relations between finite sets. In our triangulation example, for instance, \( \text{vt} \) is understood to be a representation of a vertex-triangle relation \( \text{vt} \) into a vertex set \( V \) from a triangle set \( T \):

\[
\begin{align*}
V & \quad \text{vt} \quad T \\
\rho_V & \quad \rho_T \\
[0 \ldots m) & \quad [0 \ldots n) \\
\end{align*}
\]

Most applications only deal with representations of finite functions and relations: neither the sets \( V \) and \( T \), nor the relation \( \text{vt} \) or the representation functions \( \rho_V \) and \( \rho_T \) do appear in algorithm 1. One can think of any table \( \text{vt}' \) obtained via a permutation of the rows of \( \text{vt} \) as another representation of \( \text{vt} \) having the same legitimacy as \( \text{vt} \).

For applications that only deal with representations of finite functions and relations, natural numbers (Nat) and zero-based Nat intervals are very convenient abstractions for set elements and finite sets. They are computationally cheap (a zero-based Nat interval is described by a single Nat) and naturally lead to representations of finite functions in terms of arrays. For instance, the vertex coordinates of our example are represented by a simple one-dimensional array \( x \) of size \( m \) (\( nd \) here is the number of dimensions of the space in which the triangulation is embedded, typically 2 or 3, and \( V(\text{Real}, nd) \) is the set of \( nd \)-dimensional real-valued vectors):

\[
\begin{align*}
\mathbb{R}^{nd} & \quad x \quad V \\
\rho_{\mathbb{R}^{nd}} & \quad \rho_V \\
V(\text{Real}, nd) & \quad [0 \ldots m) \\
\end{align*}
\]
In turn, arrays of some generic type X, A(X), are, for many computational purposes, very efficient representations of finite functions.

An alternative approach for representing finite sets of a generic type X is by means of a parameterized data structure: Set(X). Representations of finite sets-based on parameterized types are, of course, more powerful than representations-based on zero-based Nat interval abstractions. They can distinguish between sets of different types but same cardinality. On the other hand, parameterized relation and function representations based on parameterized set representations – data structures of the kind Rel(Set(X), Set(Y)) or Fct(V(Real, nd), Set(Y)) in place of A(V(Real, nd)) for functions – make functions and relations dependent on application-specific, possibly inefficient representations of set element types.

Notice that frameworks like Janus support parameterized representations of finite sets but rely on relations between zero-based Nat intervals. In SCDRC we only consider sets, functions and relations which are finite. We adopt the Janus approach for relations but we do not support parameterized representations of finite sets. Of course, there are situations in SCDRC in which sets (most probably of Nats) have to be explicitly represented. In these cases we use suitable containers like lists or arrays.

The analysis presented in this section is based on a simple conceptual representation of finite functions and relations as arrays of some type X and as arrays of arrays of Nats, respectively. We stress the fact that this is a conceptual model. As we will see in section 4, SCDRC relations are implemented by means of specific data structures like CRS_Rel and Reg_Rel(n). While being isomorphic to arrays of arrays of natural numbers (both CRS_Rel and Reg_Rel(n) can be constructed in terms of such an array), SCDRC implementation of relations are defined in terms of an iterator-based interface which is very different from the array interface.

3.2. Distributed functions and distributed relations. In the list of problems presented at the end of the previous section, we have used the term distributed representation for functions and relations. In this paragraph we discuss such representations. We follow the approach outlined above and think of functions and relations as arrays of some type X and of type A(Nat), respectively. Let a be an array. A straightforward way of distributing a on np partitions is:

(1) Cut a into np chunks.
(2) Assign the first chunk to the first partition, the second chunk to the second partition and so on.

For this partitioning scheme, the function \( pa : [0 \ldots np) \rightarrow [0 \ldots \text{size}(a)) \) that associates a partition number in \([0 \ldots np)\) to each element of a is non-decreasing. As usual, we represent \( pa \) with an array of Nats. Therefore, if \( \text{size}(a) = \text{size}(pa) >> np \), \( pa \) can be more economically represented by an array of offsets (again of Nats) of size \( np + 1 \). In fact, any non-decreasing array \( ofs \) satisfying:

\[
\begin{align*}
\text{size}(ofs) &= np + 1 \\
ofs(0) &= 0 \\
ofs(np) &= \text{size}(a)
\end{align*}
\]

represents the non-decreasing partition function \( pa : [0 \ldots np) \rightarrow [0 \ldots ofs(np)) \):

\[
ofs(p) \leq k < ofs(p + 1) \equiv pa(k) = p
\]

Conversely, given \( pa \) non-decreasing, the corresponding \( ofs \) can be easily computed:
Algorithm 6: offsets

Require: is\_not\_decreasing(pa) \&\& max(pa) < np
1: ofs = make\_array(np + 1, 0)
2: for k in [0...size(pa)] do
3:   p = pa(k)
4:   ofs(p + 1) = 1 + ofs(p + 1)
5: end for
6: for p in [0...np] do
7:   ofs(p + 1) = ofs(p) + ofs(p + 1)
8: end for

Ensure: size(ofs) == np + 1 \&\& ofs(0) == 0 \&\& ofs(np) == size(pa)

At line 1, ofs is initialized as an array of size np + 1 with elements equal to zero. At the end of the first loop, ofs(p + 1) contains the number of indexes of pa whose partition number is p. The pre-condition max(pa) < np guarantees that, inside loop 1, p + 1 < np + 1 always holds. Thus, no array bound violation can occur at line 4 and each entry of pa is counted exactly one time in exactly one ofs entry. Therefore, at the end of the first loop, the sum of the entries of ofs is equal to the size of pa. In the second loop this sum is stored in ofs(np).

Since SCDDRC functions and relations are conceptually represented in terms of arrays, it is natural to conceptually represent functions and relations which are distributed on np partitions as np-tuples of arrays, as described above. The cutting up of the array in np pieces induces an array of offsets. Alternatively, we can view the np-tuple as resulting from the non-distributed array according to the array of offsets. In either case, we can assume that both the tuple \((f_0, \ldots, f_{np-1})\) and the offsets ofs are present: they constitute a valid representation of f if

\[
\begin{align*}
    f_0 ++ f_1 ++ \cdots ++ f_{np-1} &= f \\
    ofs(k + 1) - ofs(k) &= size(f_k) \&\& ofs(0) &= 0
\end{align*}
\]

The ++ operator “glues” the chunks together. The two conditions are equivalent to the following ones, which are more useful in practice since they provide an explicit “point-wise” characterization of the elements involved.

\[
\begin{align*}
    ofs(np) &= size(f) \\
    f(j) &= f_p(j')
\end{align*}
\]

where

\[
\begin{align*}
    j' &= j - ofs(p) \\
    p : ofs(p) &< j < ofs(p + 1) \\
    ofs : ofs(0) &= 0 \&\& ofs(q + 1) &= \sum_{k=0}^{k<q+1} size(f_k), \ q \ in \ [0...np)
\end{align*}
\]

Notice that \(ofs(np) = \sum_{k=0}^{k<np} size(f_k)\). Therefore the first equation in 4 can be written as

\[
size(f) = \sum_{k=0}^{k<np} size(f_k)
\]
which states an obvious size-consistency condition between \( f \) and \((f_0 \ldots f_{np-1})\). Similarly, \((R_0 \ldots R_{np-1})\) is a distributed representation of \( R \) if:

\[
\text{oofs}(np) == \text{source} \cdot \text{size}(R) \\
iR_j == iR_p j'
\]

where

\[
j' = j - \text{oofs}(p) \\
p : \text{oofs}(p) <= j < \text{oofs}(p+1) \\
ofs : \text{oofs}(0) == 0 \\
\wedge
\]

\[
\text{oofs}(q + 1) == \sum_{k=0}^{k=q+1} \text{source} \cdot \text{size}(R_k), \ q \ \text{in} \ [0 \ldots np]
\]

A caveat: just like the model for non-distributed functions and relations discussed in the previous paragraph, the model discussed here for distributed functions and relations is a conceptual one. It is essential for understanding the formal specification of the problems introduced in the next section. This model, however, does not mean that there are data structures, in SCDRC, representing a function or a relation together with its corresponding offset or partitioning function. In much the same way, you will not find, in SCDRC, functions that formally take tuple arguments that represent distributed functions or relations.

The scheme described above for distributing a one-dimensional array \( a \) on \( np \) partitions and the corresponding conceptual representation of distributed functions and relations is not new. This scheme is used in ParMetis where it is referred to as distributed CRS format. In Janus, distributed relations are equipped with “descriptors” which contain, among others, informations about sizes and offsets of tuple representations.

One can argue that there are many other ways of distributing an array \( a \) on \( np \) partitions and some of them might be better than the scheme presented here. For instance, if some “communication” relation is defined between the chunks of \( a \) and between the partitions (these could be arranged, for a certain computational architecture, according to some hardware “topology” that makes communication between some partitions faster that between others), one might want to “fit” the structure of \( a \) to that of the computing architecture. Beside simplicity and minimality, another consideration supports the conceptual model presented here: the problem of partitioning the source and the target of a relation for efficient distributed computations of relation-based algorithms is not trivial. As mentioned in the previous section, SCDRC delegates the solution of this problem to external libraries like Metis and ParMetis. It is in the solution of the partitioning problem that additional, architecture specific partitioning constraints can and should be naturally accounted for.

3.3. Problem specification. In this paragraph we give a formal specification of the problems informally introduced at the end of section 2. This specification rests on the conceptual representation of function and relations discussed in 3.1 and 3.2.

3.3.1. Problem 1. Given a distributed representation of a function \( f \) and of a partitioning of its source, compute a new distributed representation of \( f \) consistent with
the given partitioning.

Given:
\[(f_0 \ldots f_{np-1}), f_p :: A(X)\]
\[(pf_0 \ldots pf_{np-1}), pf_p :: A(Nat)\]
such that:
   i) size\((f_p) == size(pf_p)\)
   ii) size\((pf_p) == 0 \lor \text{max}\_\text{elem}(pf_p) < np\)
find:
\[(f'_0 \ldots f'_{np-1}), f'_p :: A(X)\]
such that:
   iii) \[f'_p \cdot \text{perm}_p =\{ f(i) | i \in [0 \ldots \text{size}(f)]) \), pf(i) == p\]
where
\[
\text{perm}_p :: [0 \ldots m_p) \leftarrow [0 \ldots m_p) \text{ is bijective}

m_p == \text{size}([f(i) | i \in [0 \ldots \text{size}(f)]) \), pf(i) == p]).
\]

Let us comment on this specification: as discussed in 3.2, \(f\) is represented by a tuple of arrays, one array for each partition. On each partition, a partition function \(pf_p\) specifies how the elements of \(f_p\) have to be redistributed among \(np\) partitions. As usual, \(pf_p\) is represented with an array. This has to have the same size as \(f_p\) and has to take values in \([0 \ldots np)\). The solution of problem 1 is a new distributed function \(f'\). Condition iii) requires \(f'_p\) to contain exactly those values \(f(i)\) of \(f\) such that \(pf(i) == p\). Because of the permutation \(\text{perm}_p\), these values can appear in any order in \(f'_p\).

In the above specification, we have assumed that non-empty arrays of Nats are equipped with a function \(\text{max}\_\text{elem} :: \text{Nat} \leftarrow A(\text{Nat})\) that computes the maximal element. \(\text{max}\_\text{elem}\) and \(\text{size}\) are examples of functions whose implementation depends on the computational environment. The implementation of such functions in SCDRC’s SPSD and SPMD-distributed environments is discussed at the beginning of section 4. Let’s turn the attention to the second problem introduced at the end of section 2. The specification of problem 2 is completely analogous to that of problem 1:

3.3.2. Problem 2. Given a distributed representation of a relation \(R\) and of a partitioning of its source, compute a new distributed representation of \(R\) consistent with the given partitioning.

Given:
\[(R_0 \ldots R_{np-1}), R_p :: A(A(Nat))\]
\[(pR_0 \ldots pR_{np-1}), pR_p :: A(Nat)\]
such that:
   i) source\_size\((R_p) == size(pR_p)\)
   ii) size\((pR_p) == 0 \lor \text{max}\_\text{elem}(pR_p) < np\)
find:
\[(R'_0 \ldots R'_{np-1}), R'_p :: A(A(Nat))\]
such that:
   iii) \[R'_p \cdot \text{perm}_p =\{ R(i) | i \in [0 \ldots \text{source}\_\text{size}(R)]) \), pR(i) == p\]
where
\[
\text{perm}_p :: [0 \ldots m_p) \leftarrow [0 \ldots m_p) \text{ is bijective}
\]
Notice that, even in the case in which $\text{perm}_p$ is taken to be the identity permutation on all partitions both in problem 1 and in problem 2, $f'$, $R'$ are not, in general, equal to $f$ and $R$. This is because of the fact that $pf$ and $pR$ have not been required to be non-decreasing. In other words, the repartitioning is not required to be a re-cutting.

In section 3.2, we have motivated a conceptual representation of distributed functions and relations that rests on non-decreasing partition functions. From this point of view, the specifications given above seem to be far too general: a (generic) program $\text{redistribute}$ implementing these specifications, allows one to write a SPMD distributed parallel program to:

1. on all partitions $p$ in $[0...np]$, do: initialize an empty vertex coordinates array $x_p$ and an empty vertex-triangles relation $vt_p$.
2. on partition 0, do: read a vertex coordinates array $x$ and a vertex-triangle relation $vt$ from some file into $x_0$, $vt_0$.
3. on partition 0, do: compute suitable\(^7\) partition functions $px_0$, $pvt_0$ of the sources of $x_0$, $vt_0$; on all partitions $p \neq 0$ initialize empty $px_p$, $pvt_p$.
4. on all partitions, do: redistribute $(x_0...x_{np-1})$, $(vt_0...vt_{np-1})$ according to $(px_0...px_{np-1})$, $(pvt_0...pvt_{np-1})$.

These steps rephrase steps 1-4 of section 2.2 in terms of the notation introduced in this section for distributed representations of functions and relations. Steps 1-4 of section 2.2 have been introduced to set up a SPMD parallel computation of the centers of the triangles of the triangulation represented by the vertex coordinates array $x$ and by the vertex-triangle relation $vt$. In this section we are going to use this computation as an example of a SPMD parallel relation-based algorithm. We will come back to this example over and over again to motivate and refine formal specifications for the problems introduced at the end of section 2.

Consider steps 1 and 2 above. These yield distributed representations $(x_0...x_{np-1})$, $(vt_0...vt_{np-1})$ of $x$, $vt$ with $x_0 == x$, $vt_0 == vt$ and $x_p == vt_p == []$, $p_0 in [1...np]$. Similarly, step 3 provides distributed representations $(px_0...px_{np-1})$, $(pvt_0...pvt_{np-1})$ of $px$, $pvt$ with $px_0 == px$, $pvt_0 == pvt$ and $px_p == pvt_p == []$, $p_0 in [1...np]$. If $px$ and $pvt$ are not non-decreasing, step 4 yields distributed representation $(x_0'...x_{np-1}')$, $(vt_0'...vt_{np-1}')$ of arrays $x'$, $vt'$ such that, in general (i.e. for arbitrary $x$, $vt$), $x' \neq x$ and $vt' \neq vt$. The analysis raises two questions:

Q1 Why do we put forward specifications of redistribution problems that seem to imply more general conceptual representations of distributed functions and relations than the one introduced in section 3.2?

Q2 If we stick to the conceptual representation of distributed arrays of section 3.2 (and restrict ourselves to partition functions that can be represented by arrays of offsets), isn’t it very inefficient to represent $px$, $pvt$ with tuples of arrays $(px_0...px_{np-1})$, $(pvt_0...pvt_{np-1})$ (of total size $\text{size}(x)$, $\text{source-size}(vt)$) instead of offsets-based representations of size $np + 1$ independent of the size of the represented function?

To answer these questions and refine the problem specifications presented above, let us discuss steps 1)-4) in a concrete case. Consider a very simple triangulation in

\[ m_p == \text{size}([ R(i) | i \in [0...\text{source-size}(R)] ]), pR(i) == p ] \]
which $x$ and $vt$ are:

\begin{align}
  x &= \begin{bmatrix} 0.0, 0.0 \end{bmatrix}, \begin{bmatrix} 1.0, 0.0 \end{bmatrix}, \begin{bmatrix} 1.0, 1.0 \end{bmatrix}, \begin{bmatrix} 0.0, 1.0 \end{bmatrix} \\
  vt &= \begin{bmatrix} 0, 1, 3 \end{bmatrix}, \begin{bmatrix} 1, 2, 3 \end{bmatrix}
\end{align}

and represent the triangulation of the unit square sketched above. Assume $np == 2$. Then, at the end of step 2, our distributed representation of the triangulation is:

\begin{align}
  (x_0, x_1) &= \begin{bmatrix} [0, 0, 0, 0], [1, 0, 0, 0], [1, 0, 1, 0], [0, 0, 1, 0] \end{bmatrix}, [ ] \\
  (vt_0, vt_1) &= \begin{bmatrix} [0, 1, 3], [1, 2, 3] \end{bmatrix}, [ ]
\end{align}

where $x_0 == x$, $vt_0 == vt$ and $x_1$, $vt_1$ are still empty (remember step 1). Assume that, in step 3, the following partitionings have been computed:

\begin{align}
  (px_0, px_1) &= \begin{bmatrix} [1, 1, 0, 0], [ ] \end{bmatrix} \\
  (pvt_0, pvt_1) &= \begin{bmatrix} [1, 0], [ ] \end{bmatrix}
\end{align}

The tuples $(x_0, x_1)$, $(px_0, px_1)$ and $(vt_0, vt_1)$, $(pvt_0, pvt_1)$ fulfill the preconditions i) and ii) of the specifications for problems 1 and 2, respectively. Therefore we can apply step 4 using any program that implements these specifications. The distributed representation

\begin{align}
  (x'_0, x'_1) &= \begin{bmatrix} [0, 0, 1, 0], [1, 0, 1, 0], [0, 0, 0, 0], [1, 0, 0, 0] \end{bmatrix} \\
  (vt'_0, vt'_1) &= \begin{bmatrix} [1, 2, 3], [0, 1, 3] \end{bmatrix}
\end{align}

is a legitimate outcome of step 4: with $perm_0 = [1, 0]$, $x'_0 \cdot perm_0$ is indeed equal to $\{x(i) | i \in \{0 \ldots \text{size}(x)\}, px(i) == 0\}$. The remaining requirements are all satisfied with identity permutations.

In the framework of the conceptual representation of distributed arrays introduced in section 3.2, $(x'_0, x'_1)$, $(vt'_0, vt'_1)$ is obviously not a distributed representation of the triangulation represented by $(x_0, x_1)$, $(vt_0, vt_1)$. As sketched in the figure, the triangles of $x'$, $vt'$ are now overlapping! The inconsistency between $(x_0, x_1)$, $(vt_0, vt_1)$ on one side and $(x'_0, x'_1)$, $(vt'_0, vt'_1)$ on the other side comes into place because, in our example

1. $px$ is not non-decreasing.
2. $perm_p$ is not the identity.

The effect of non non-decreasing partitioning functions $px$ and non identical permutations $perm_p$ is obviously that of modifying the order in which the elements of $x$ appear in $x'$: for arbitrary $px$ and $perm_p$ steps 1)-4) yield vertex coordinates $x'$ which are permutations of $x$:

\begin{align}
  x' &= x \cdot perm^o
\end{align}

In our example, $perm^o = [3, 2, 0, 1]$. The reason why we denote the permutation with the converse symbol will become clear in the following analysis. Notice that, if $px$ were non-decreasing and $perm_p$ were the identity, $perm^o$ would be the identity permutation and we had $x' == x$.

Of course, we can always choose $perm_p$ to be the identity. However, $px$ is obtained, in SC DCRC, from Metis or from other graph partitioning algorithms: it is not possible to require $px$ to be non-decreasing.

We can keep our conceptual representation of distributed functions and relations and allow for arbitrary partition functions $px$ if we modify steps 1)-4). To understand how this has to be done, consider equation (8). This can be interpreted in two different
ways. On one hand, one can think of \( x'(i) \) as of the “new” position of the \( i \)-th vertex under the (musical chairs like) motion described by \( \perm \). On the other hand, one can think of \( \perm \) as of a renumbering of the vertex set. Under such renumbering, the \( i \)-th vertex stays at its own position but gets a “new” index \( \perm(i) \). Conversely, \( \perm^{-1} \) takes a new index argument of \( x' \) into an old index argument of \( x \). This motivates the converse symbol used with equation (8).

The second interpretation of equation (8) suggests how to allow for arbitrary partition functions while keeping our simple conceptual representation of distributed arrays. If equation (8) represents a renumbering of the source of \( x \) then we have to account for this renumbering in \( vt \) as well because the target of \( vt \) coincides with the source of \( x \). This means that we have to replace the “old” indexes of \( vt' \) with the “new” indexes:

\[
\text{vt}'(j)(i) \leftarrow \perm(\text{vt}'(j)(i))
\]

If we do this replacement in equation (7) we obtain (with \( \perm = \llbracket 2, 3, 1, 0 \rrbracket \), \( \perm \cdot \perm^{-1} = \iddots_{\llbracket 0, 1 \rrbracket} \)):

\[
\begin{align*}
(x'_0, x'_1) &= (\llbracket 0.0, 1.0 \rrbracket, \llbracket 1.0, 1.0 \rrbracket, \llbracket 0.0, 0.0 \rrbracket, \llbracket 1.0, 0.0 \rrbracket) \\
(\text{vt}'_0, \text{vt}'_1) &= (\llbracket 3.1, 0 \rrbracket, \llbracket 2, 3, 0 \rrbracket)
\end{align*}
\]

The new triangulation is now, up to a renumbering of the vertexes and of the triangles, identical to the original one. Notice that the triangle set has been renumbered as well (\( \perm \) is not, in our example, non-decreasing).

It is now easy to see how steps 1)-4) have to be modified to allow for arbitrary partition functions \( px \), \( pet \) while keeping the conceptual representation of distributed functions and relations introduced in section 3.2. What we have to do is compute permutations \( \perm x'_0 \), \( \perm vt'_0 \) of the sources of \( x_0 \), \( vt_0 \) such that \( px_0 \cdot \perm x'_0 \) and \( pvt_0 \cdot \perm vt'_0 \) are non-decreasing and modify \( x_0 \) and \( vt_0 \) accordingly. Then, we redistribute the modified representations with implementations that fulfill our problem 1 and problem 2 specifications and with \( \perm d \) equal to the identity. The modified procedure can be described as follows:

(1) on all partitions, do: initialize an empty vertex coordinates array \( x_p \) and an empty vertex-triangles relation \( vt_p \).

(2) on partition 0, do: read a vertex coordinates array \( x \) and a vertex-triangles relation \( vt \) from some file into \( x_0 \), \( vt_0 \).

(3) on partition 0, do: compute suitable partition functions \( px_0 \), \( pvt_0 \) of the sources of \( x_0 \), \( vt_0 \).

(4) on partition 0, do: compute \( \perm x'_0 \), \( \perm vt'_0 \) such that \( px_0 \cdot \perm x'_0 \), \( pvt_0 \cdot \perm vt'_0 \) are non-decreasing.

(5) on partition 0, do: replace \( x_0 \), \( px_0 \), \( vt_0 \), \( pvt_0 \) with \( x_0 \cdot \perm x'_0 \), \( px_0 \cdot \perm x'_0 \), \( vt_0 \cdot \perm vt'_0 \) and \( pvt_0 \cdot \perm vt'_0 \), respectively. Renumber the elements of \( vt_0 \) according to the rule

\[
\text{vt}_0(j)(i) \leftarrow \perm vt_0(\text{vt}_0(j)(i))
\]

(6) on all partitions, do: redistribute \( x_0 \ldots x_{np-1} \), \( vt_0 \ldots vt_{np-1} \) according to \( px_0 \ldots px_{np-1} \), \( pvt_0 \ldots pvt_{np-1} \) and to the specifications of problem 1 and problem 2 with \( \perm d = \iddots \).
An algorithm for computing a permutation \( \text{perm}_o \) such that \( \text{part} \cdot \text{perm}_o \) is non-decreasing for arbitrary partition functions \( \text{part} \) can be easily written in terms of relational operations:

**Algorithm 7**: order preserving permutation

1. \( \text{perm}_o = \text{breadth}(\text{converse}(\text{part})) \)

Ensure: is\_non\_decreasing(compose(\text{part}, \text{perm}_o))

At line 1 we first compute \( \text{part}^o = \text{converse}(\text{part}) \). This is, in general, a relation. It associates to each partition \( p \) in \( [0 \ldots np) \) those indexes \( i \) in \( [0 \ldots \text{size}(\text{part})) \) for which \( \text{part}(i) == p \). Notice that the breadth of \( \text{part}^o \) – the concatenation of \( \text{part}^o(0), \ldots \text{part}^o(np - 1) \) – is a permutation of \( [0 \ldots \text{size}(\text{part}))^o \). Because of the order of concatenation \( i < j \Rightarrow \text{part}(\text{perm}^o(i)) \leq \text{part}(\text{perm}^o(j)) \) that is \( \text{part} \cdot \text{perm}_o \) is non-decreasing. If we apply steps 1)-6) to our simple triangulation 5 with the partition functions given by 6, we obtain “the” following distributed triangulation:

\[
\begin{align*}
(x'_0, x'_1) &= ([1,0,1,0], [0,0,1,0], [0,0,0,0], [1,0,0,0]) \\
(vt'_0, vt'_1) &= ([3,0,1], [2,3,1])
\end{align*}
\]

Steps 1)-6) raise a question and a remark. The question is: how to proceed in case \( x \) and \( vt \) are already non-trivially distributed (that is, \( x_p, vt_p \) are not empty for \( p > 0 \), for instance as a result of a previous application of steps 1)-6)) and non-trivial distributed partitionings are computed, for instance with ParMetis?

The remark is that, after step 5, one can in fact redistribute \((x_0 \ldots x_{np-1}), (vt_0 \ldots vt_{np-1})\) according to an offset-based representation of \((px_0 \ldots px_{np-1}), (pvt_0 \ldots pvt_{np-1})\) and therefore to a less general specifications of problem 1 and problem 2.

We are not going to answer the above question in this report. We point out, however, that for solving the problem of redistributing non-trivially distributed functions and relations, the specifications of problem 1 and problem 2 presented in this section

---

\(^6\)this is because \( \text{part} \) is a function.

\(^3\)We have not provided a specification of converse that unambiguously defines \( \text{part}^o \) from \( \text{part} \). Up to this ambiguity, however, the outcome of steps 1)-6) is unique. Here we detail the steps of the computation. Using algorithm (7) and with an implementation of converse with ordered sub-arrays, step 4 yields the following permutations:

\[
\begin{align*}
\text{perm}_{x_0} &= [2,3,0,1] \\
\text{perm}_{vt_0} &= [1,0]
\end{align*}
\]

In the first part of step 5, we account for the renumbering of the vertex set in \( x_0, px_0 \) and of the triangle set in \( vt_0, pvt_0 \). This yields:

\[
\begin{align*}
 x_0 &= [[1,0,1,0],[0,0,1,0],[0,0,0,0],[1,0,0,0]] \\
 px_0 &= [0,0,1,1] \\
 vt_0 &= [[1,2,3],[0,1,3]] \\
 pvt_0 &= [0,1]
\end{align*}
\]

In the second part of step 5, we account for the renumbering of the vertex set in \( vt_0 \). This yields, with \( \text{perm}_x == \text{perm}_x^o == [2,3,0,1]: \)

\[
vt_0 = [[3,0,1],[2,3,1]]
\]
play an important role. This partially answers question Q1. For the purpose of implementing steps 1)-6), it is indeed meaningful to introduce less general specifications. This is what we are going to do next, thereby answering question Q2.

One can think of steps 1)-5) as of pre-processing steps that generate some triangulation file. This file contains a vertex coordinates array \( x \), a vertex-triangles relation \( vt \) and two offsets arrays \( ox \) and \( ovt \). Steps 1)-6) can be then rephrased as follows:

1. on all partitions \( p \) in \([0 \ldots np]\), initialize an empty vertex coordinates array \( x_p \), an empty vertex-triangles relation \( vt_p \) and empty offsets arrays \( ox_p \) and \( ovt_p \).
2. on partition 0, read a vertex coordinates array \( x_0 \), a vertex-triangles relation \( vt_0 \) and offsets arrays \( ox_0 \) and \( ovt_0 \) from some file into \( x_0 \), \( vt_0 \), \( ox_0 \) and \( ovt_0 \).
3. on all partitions, redistribute \((x_0 \ldots x_{np-1}) \), \((vt_0 \ldots vt_{np-1})\) according to \((ox_0 \ldots ox_{np-1})\), \((ovt_0 \ldots ovt_{np-1})\).

Implementations of Step 3) are now required to fulfill the following specifications:

3.3.3. Problem 1'. Given a distributed representation of a function \( f \) and given a partitioning of its source, compute a new distributed representation of \( f \) consistent with the given partitioning.

Given:

\[
(f_0 \ldots f_{np-1}), \ f_p :: A(X)
\]
\[
(o_0 \ldots o_{np-1}), \ o_p :: A(Nat)
\]

such that:

i) isOffsets\( o_p \)
ii) size\( (o_p) == np + 1 \)
iii) size\( (f_p) == o_p(np) \)

find:

\[
(f'_0 \ldots f'_{np-1}), \ f'_p :: A(X)
\]

such that:

iv) \( f'_p == \text{concat} (f''_{p,0} \ldots f''_{p,np-1}) \)

where

\[
f''_{p,q} = [f_q(i) \mid i \in [o_q(p) \ldots o_q(p + 1)]]
\]

3.3.4. Problem 2'. Given a distributed representation of a relation \( R \) and given a partitioning of its source, compute a new distributed representation of \( R \) consistent with the given partitioning.

Given:

\[
(R_0 \ldots R_{np-1}), \ R_p :: A(A(Nat))
\]
\[
(o_0 \ldots o_{np-1}), \ o_p :: A(Nat)
\]

such that:

i) isOffsets\( o_p \)
ii) size\( (o_p) == np + 1 \)
iii) source_size\( (R_p) == o_p(np) \)

find:

\[
(R'_0 \ldots R'_{np-1}), \ R'_p :: A(A(Nat))
\]

such that:

iv) \( R'_p == \text{concat} (R''_{p,0} \ldots R''_{p,np-1}) \)
The problem specifications introduced so far allow one to distribute array-based representations of functions and relations to set up efficient SPMD parallel computations of relation-based algorithms. For our triangle centers example, for instance, as in steps 1-6 above.

We now turn the attention to the problem of actually applying relation-based algorithms and to the specification of the third problem introduced at the end of section 2. Again, we consider our triangle centers computation and the simple triangulation of equation (5). For its distributed representation (10) obtained with steps 1)-6,) we want to compute the centers of the (two) triangles in parallel on partition 0 and on partition 1.

Although we have a distributed representation of (5), we cannot directly apply the relation-based algorithm 2 in parallel on the two partitions. This is because of two reasons. The first reason is that, in order to access the vertex coordinates, we have to re-scale the indexes of $vt'_p$ according to the partition number $p$. The second reason is that, on a given partition, we usually need to access vertex coordinates which are stored in other partitions: in our example these coordinates are, on partition 0, $x'(3)$ and, on partition 1, $x'(0)$. Notice that $x'(0)$ is stored on partition 0 as $x'_0(0)$ and $x'(3)$ is stored on partition 1 as $x'_1(1)$.

Obviously, the indexes of $vt'_p$ have to be rescaled according to the offsets associated with the distributed representation $(x'_0, x'_1)$ of $x'$. This means that algorithm 2 has to be modified as follows:

\begin{algorithm}
\begin{algorithmic}
\For{$j$ in $[0 \ldots \text{size}(vt'_p)]$}
\State compute \text{center}($x'_p(vt'_p(j)(0) - ofs(p))$, $x'_p(vt'_p(j)(1) - ofs(p))$, $x'_p(vt'_p(j)(2) - ofs(p))$)
\EndFor
\end{algorithmic}
\end{algorithm}

A few remarks are appropriate here: first, notice that the above SPMD version of the triangle centers algorithm 2 is parameterized on the partition number $p$. This is, in fact, the sense in which the “single program” in the SPMD acronym has to be understood. As the informal descriptions of SPMD parallel computations 1-6 and 1-3 suggest, a SPMD program is not really a “single” program but a family of programs, one for each value of $p$.

Second, it is clear that, before algorithm 8 can actually be applied, data exchange between partitions has to take place to obtain those vertex coordinates which are needed for the local computation of the center and stored on non-local partitions and to compute the offsets $ofs$. Obviously, $ofs$ has to be the same on all partitions. In our example we have $ofs(0) == 0$, $ofs(1) == 2$ and $ofs(2) == 4$. Consider the following specification of problem 3:

3.3.5. Problem 3. Given a distributed representation of a function $f$ and given, on each partition, a subset $d$ of $\text{dom}(f)$, compute, on each partition, the values of $f$ in $d$. 
Given:

\[(f_0 \ldots f_{np-1}), f_p : A(X)\]
\[(d_0 \ldots d_{np-1}), d_p : A(Nat)\]

such that:

i) \(\max_{\text{elem}}(d_p) < ofs(np)\)

where

\(ofs = \text{offsets}(\text{map(size, } (f_0 \ldots f_{np-1})))\)

find:

\[(fd_0 \ldots fd_{np-1}), fd_p : A(X)\]

such that:

ii) size(fd_p) == size(dp)

iii) \(fd_p(i) == f(dp(i))\)

Here, the generic function offsets and the function map fulfill:

\(\text{offsets} :: A(Nat) \leftarrow \text{Nat}^n\)
\(\text{offs} == \text{offsets}(s_0 \ldots s_{n-1})\)
\(\equiv\)
\(\text{size}(\text{offs}) == n + 1 \land \text{offs}(0) == 0 \land \text{offs}(p + 1) == \sum_{k=0}^{p+1} s_k\)

\(\text{map} :: A(X) \leftarrow (X \leftarrow Y)^n\)
\(ax == \text{map}(f, (y_0 \ldots y_{n-1}))\)
\(\equiv\)
\(ax(k) == f(y_k), k = [0 \ldots n]\)

With a \texttt{dom_complete} program implementing the above specification of problem 3 and with an implementation of algorithm 8, it is easy to write a SPMD program to compute the triangle centers. All we have to do is to:

1. Apply \texttt{dom_complete} and complete the data \((x_0' \ldots x_{np-1}')\) on \((\text{breadth}(vt_0') \ldots \text{breadth}(vt_{np-1}'))\). This yields the arrays \((x_0'' \ldots x_{np-1}'')\).

2. Apply algorithm 8 with \(x_p''(3 \ast j + i)\) in place of \(x_p'(vt_p(j)(i) - ofs(p))\).

In our example, step 1 yields \(x_0'' = \left[\begin{array}{c} [1.0, 0.0], [1.0, 1.0], [0.0, 1.0] \end{array}\right]\) and \(x_1'' = \left[\begin{array}{c} [0.0, 0.0], [1.0, 0.0], [0.0, 1.0] \end{array}\right]\). Step 2 provides the centers \([2/3, 2/3], [1/3, 1/3]\) on partitions 0 and 1, respectively. These are indeed the coordinates of the centers of the triangles 0 and 1 of the figure on the left of equation (10).

There are two major problems with the approach outlined above. The first problem is that we are duplicating too much data. Remember that, if we accept to duplicate a lot of data, the problem of structuring SPMD parallel computations becomes trivial:\footnote{If we let apart the problem of ensuring the consistency of the duplicated data.} we simply store the whole \(x'\) on all partitions. Here we increase the memory allocation costs for \(x'\) by a factor \(size(vt')\) instead of \(np \ast size(x')\). On large, plane triangulations, however, the number of triangles is about twice the number of nodes and we store three \(x'\)-values per triangle. Thus, the ratio between \(np \ast size(x')\) and \(size(vt')\) is about \(np/6\). This means that we need at least 6 partitions for our scheme to be competitive with the simple minded, full duplication approach: this is not good.

Notice that, in many applications, the ratio between the memory required to store an \(x'\) element and the memory required to store a Nat can be quite large. In our
example, \( x' \) elements are arrays of 2 doubles. The ratio between \texttt{sizeof(double)} and \texttt{sizeof(Nat)} is, e.g., on my computing architecture, equal to four. This ratio would be six if the triangulation were embedded in a three-dimensional space.

Also notice that, if we have been able to partition the original triangulations “well”, the number of \( x' \)-values which are needed for the local triangle centers computation but which are stored on remote partitions will be much smaller than the number of \( x' \)-values which are stored locally.

The above remarks suggest a more efficient scheme for storing and accessing the \( x' \)-values retrieved from remote partitions. What we want to do is:

1. Extend \( x'_p \) with those values of \( x' \) which are needed for the local triangle centers computation but which are stored on remote partitions.
2. Construct an auxiliary access table \( vt''_p \) such that
   \[
   x'_p(vt''_p(j)(k)) == x'(vt'(j)(k))
   \]

This approach has been originally proposed in the Janus framework and has been adopted in SCDRC. The second problem that affects triangle centers computations based on the specification of problem 3 given above is more subtle. Although we have not discussed how a \texttt{dom_complete} program could be implemented, it is obvious that, in order to compute a tuple \((f_{d_0} \ldots f_{d_{np-1}})\), the following steps have to be done:

a) on each partition \( p \), do: compute the indexes of \( d_p \) (in our example \( d_p = \text{breadth}(vt'_p) \)) which are in \([ofs(q) \ldots ofs(q+1)]\) for partitions \( q \neq p \).

b) On each partition \( p \), do: compute the indexes of \([ofs(p) \ldots ofs(p) + \text{size}(f_p)]\)
   which are in \( d_q \) for partitions \( q \neq p \).

While the first table can be computed without additional data exchange between partitions\(^{11}\), the computation of the second table certainly requires communication between partitions. It is only after each partition \( p \) knows, for any partition \( q \neq p \), which are the indexes whose correspondent \( f \)-values have to be sent that such values can actually be exchanged. The cost of computing such exchange tables can significantly exceed the cost of exchanging the \( f \)-values themselves. The same is true for the cost of computing access tables like the auxiliary relation \( vt''_p \) discussed above.

These exchange and access tables do not depend on the data to be actually exchanged but only on the set of indexes on which \( f \) has to be evaluated on a given partition and, of course, on the partitioning of \( f \) itself.

In many practical cases, the parallel computation of relation-based algorithms is required at each step of some iterative procedure in which the values of \( f \) change from step to step but the exchange and access tables do not. In our triangle centers example, for instance, the vertex coordinates could change from step to step (e.g. because of forces acting on the triangulation) while the vertex triangle relation and the partitioning of the vertex coordinates stay the same. In the iterative solution of implicit problems (e.g. linear systems of equations) relation-based algorithms that represent the action of some discrete operator on a “vector of unknowns” might be evaluated thousands of times. At each time the values of the “unknowns” would change but the relation and the partitioning scheme for such values would not.

Thus, for many practical cases, it would be very inefficient to recompute, at each iteration step, the tables needed to exchange data between partitions and to efficiently

\(^{11}\text{w.r.t. the data exchange required to compute } ofs.\)
access the local data. Therefore it is particularly important to decouple the computation of the exchange and access tables from the actual data exchange. This motivates the specification of the following problem:

### 3.3.6. Problem 3’

Given:

\[(R_0 \ldots R_{np-1}), R_p :: A(Nat)\]

\[(ofs \ldots ofs'), ofs :: A(Nat)\]

such that:

i) is_offsets(ofs) == true

ii) \[\text{max}_v(R_p(j)) < ofs(np)\]

find:

\[(at_0' \ldots at_{np-1}'), at_p' :: A(A(Nat))\]

\[(et_0' \ldots et_{np-1}'), et_p' :: A(A(Nat))\]

such that:

iii) \[\forall (f_0' \ldots f_{np-1}), f_p :: A(X)\]

\[\text{such that:}\]

- offsets(map(size, (f_0' \ldots f_{np-1}))) == ofs

- the tuple \((f_0' \ldots f_{np-1})\) obtained with SPMD algorithm 9 satisfies:

\[f_p'(at_p'(j)(k)) == f(R_p(j)(k))\]

**Algorithm 9**: complete \(f\)

**Require:** \(ofs(p) \leq et_p'(q)(k) \land et_p'(q)(k) < ofs(p + 1)\)

1. \(et_p'' = \text{map}(sf_p', et_p')\)

2. \[\text{where}\]

3. \(sf_p(a) = \text{map}(sf_p, a)\)

4. \(sf_p(i) = f_p(i - ofs(p))\)

5. \(f_p' = \text{concat}(f_p, \text{breadth}(\text{exchange}(et_0'' \ldots et_{np-1}'')(p)))\)

The input data of problem 3’ are, on each partition, a relation \(R_p\) and an array of offsets \(ofs\). We omit the index \(p\) in \(ofs\) to indicate that the array of offsets is the same on all partitions.

We require \(ofs\) to be an offsets array i.e. to satisfy equation (2). Moreover, the largest index appearing in \(R_p(j)\) shall not exceed \(ofs(np)\). This means that the indexes of \(R_p(j)\) are in the source of functions which have been distributed according to \(ofs\).

What is sought in problem 3’ are, on each partition, an access table \(at_p'\) and an exchange table \(et_p'\). For any functions \(f\) distributed according to \(ofs\), \(at_p'(j)\) is required to provide, on partition \(p\), access to the values of \(f\) at the indexes of \(R_p(j)\) through a suitable extension \(f_p'\) of \(f_p\).

Algorithm 9 describes how such an extension will be constructed from \(f_p\) and from the exchange table \(et_p''\). This represents the table mentioned in step b) on page 22: the array \(et_p'(q)\) contains those indexes of \(R_q\) which are in \([ofs(p) \ldots ofs(p + 1)]\). Since the values of \(f\) corresponding to these indexes are stored in partition \(p\) (in \(f_p\)), we say that \(et_p'(q)\) is the “table of requests” issued from partition \(q\).

In the first step of algorithm 9 we compute, for each index of \(et_p''\), the corresponding \(f\)-value in \(et_p''\). The pre-condition guarantees that this can be done. In the second
step we exchange the table of \( f \) values \( et_p' \) between partitions. The function exchange plays an outstanding role in SCDRC. Although SCDRC users will almost never call exchange directly, most SCDRC algorithms that require data communication between partitions are designed around this communication primitive. We will come back to the implementation of exchange in section 4. Here we provide its specification:

\[
\text{exchange} :: A(A(X))^{np} \leftarrow A(A(X))^{np} \\
(et_0' \ldots et_{np-1}') == \text{exchange}(et_0 \ldots et_{np-1}) \\
\equiv \quad \text{is\_in}(x, et_p'(q)) == \text{is\_in}(x, et_q(p))
\]

We read the specification in the following way: \( x \) is an element that partition \( p \) receives from partition \( q \) iff \( x \) is an element that partition \( q \) sends to partition \( p \).

After having exchanged \( et_p' \) between partitions, we obtain, in line 4 of algorithm 9 a tuple of arrays of arrays of elements of the same type of the elements of \( f \). On partition \( p \), the \( p \)-th array of the tuple is flattened and concatenated with \( f_p \).

If we have a \textit{exchange} program that implements the specification of exchange, it is easy to write a \textit{complete} program that implements algorithm 9. Also implementing a program \textit{access\_exch\_table} that fulfills 3’ is not difficult if \textit{exchange} is available: tables of requests \( et_p \) can be computed locally by sorting the elements of \( R_p \) whose \( f \)-values are stored on remote partitions according to their corresponding partition number. A call to \textit{exchange} yields then the exchange tables \( et_p' \). The computation of the access tables \( at_p' \) is a little bit less straightforward.

Equipped with \textit{access\_exch\_table} and with \textit{complete}, it is now easy to write a SPMD program for the parallel computation of our triangle centers. In fact, we are now ready to make a further abstraction step and outline a program that, given distributed representations \( (f_0 \ldots f_{np-1}), (R_0 \ldots R_{np-1}) \) of matching \( f \) and \( R \) and given some “reduction” operator \( h \), implements a SPMD parallel version of our relation-based algorithm 3.

1. Compute the offsets \( ofs \) of \( (f_0 \ldots f_{np-1}) \).
2. Compute \( at_p', et_p' \) with \textit{access\_exch\_table} and \( (R_0 \ldots R_{np-1}), (ofs \ldots ofs) \).
3. Compute \( f_p' \) with \textit{complete} and \( f_p, ofs \) and \( et_p' \).

\begin{algorithm}
for \( j \) in \([0 \ldots \text{source\_size}(R_p)]\) do
\hspace{1cm} compute \( h([f_p'(i) \mid i \in at_p'(j)]) \)
end for
\end{algorithm}

Here we have assumed that the RBA is to be evaluated on the whole source of \( R \). Of course, this assumption can be easily weakened. Notice that steps 2–3 could now be embedded in some iteration in which the values of \( f \) change from step to step, e.g., as a result of the iteration itself. At each step, only \textit{complete} would be called to extend and synchronize the local partial representations \( f_p \). The computationally expensive and communication intensive computation of the access and exchange tables could be done only once before entering the iteration.
The algorithm outlined above shows how parallel SPMD computations of relation based algorithms can be structured using software components provided by SC-DRC. As mentioned in the introduction, these components – e.g. access, complete and components representing RBAs themselves – are designed to support structured user control over communication steps. They allow to distinguish between the kind of communication that takes place in steps 1 and 2 from the communication needed to exchange $f$-values. However, developers of SC-DRC do not need to care about message passing level communication and related synchronization, mutual exclusion, deadlock or race condition problems.

Of course, developers of SC-DRC-based, application dependent software are free to hide some of these communication steps and aggregate functionalities in more specific components. For instance, a software component representing a vertex-centered Laplace operator on triangulations could be defined in terms of RBAs in which $R$ and $h$ are fixed. Users could be enabled to construct concrete instances of such Laplace operator by simply passing a distributed function $f$ of the vertexes of the triangulation to suitable constructors. These, in turn, could automatically call RBA constructors and access functionalities to set up a parallel evaluation of the Laplace operator without further user intervention.

In section 2, we have mentioned the problem of combining grid relations for computing neighborhood relationships and motivated the implementation of simple basic relational operations. We close this section with the specifications of the problems of parallel conversing a distributed relation and of composing two distributed relations.

3.3.7. Problem 4. Given a distributed representation of a relation $R$ and of a partitioning of its target, compute a distributed representation of $R^\circ$ consistent with the given partitioning.

Given:

$$(R_0 \ldots R_{np-1}), \quad R_p :: \text{A(A(Nat))}$$

$$(ofs \ldots ofs), \quad ofs :: \text{A(Nat)}$$

such that:

i) $\text{is.offsets}(ofs) == \text{true}$

ii) $\max_{elem}(R_p(j)) < ofs(np)$

find:

$$(R_0^\circ \ldots R_{np-1}^\circ), \quad R_p^\circ :: \text{A(A(Nat))}$$

such that:

iii) $\text{offsets}(\text{size}(R_0^\circ) \ldots \text{size}(R_{np-1}^\circ))$

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

3.3.8. Problem 5. Given consistent, distributed representations of relations $S$ and $T$, compute a consistent, distributed representation of $S \cdot T$.

Given:

$$(S_0 \ldots S_{np-1}), \quad S_p :: \text{A(A(Nat))}$$

$$(T_0 \ldots T_{np-1}), \quad T_p :: \text{A(A(Nat))}$$

such that:

i) $\max_{elem}(T_p(j)) < \text{offsets(map(source.size, (S_0 \ldots S_{np-1}))))(np)$

find:
4. Implementation dependent elements

In this section we present and discuss implementation dependent aspects of SC-
DRC. In the first part we outline the architecture of SCDRC. In particular, we explain
the approach used to represent the two computational environments of SCDRC, we
describe the file system structure and the most important SCDRC components and
we explain how SCDRC is documented.

In the second part we discuss a small number of data structures and functions in
some detail. These functionalities are going to be used in the next and last section
to set up a simple SPMD parallel application. As you might have guessed this is
the relation-based algorithm example we have been using throughout this report: the
computation of the centers of the triangles of a distributed triangulation.

4.1. Computational environments and namespaces. In the previous section we
have discussed formal specifications for the problems introduced at the end of section
2. The specifications are based on the conceptual model of distributed functions and
relations discussed in section 3.2: in this model, distributed functions and relations are
represented by tuples of arrays. Accordingly, functions acting on distributed functions
and relations take arguments which are tuples of arrays. In general, functions acting
of distributed data take tuple arguments.

This is clearly visible in the signature of exchange and has been implicitly assumed
for functions like offsets, complete etc. Let us have a closer look at offsets. A specification
for this function can be expressed as follows:

\[
\text{offsets} :: A(Nat)^{np} \leftarrow Nat^{np}
\]
\[
(ofs_0 \ldots ofs_{np-1}) == \text{offsets}(s_0 \ldots s_{np-1})
\]
\[
\equiv
\]
\[
\text{is}\cdot\text{offsets}(ofs_p) == \text{true}
\]
\[
\wedge
\]
\[
ofs_p == ofs_q
\]
\[
\wedge
\]
\[
ofs(p + 1) == \sum_{k=0}^{k<p+1} s_k, \ p \in [0 \ldots np)
\]

With a function exchange fulfilling the specification given in section 4, it is easy to
write a SPMD algorithm that implements the above specification:

As usual, the algorithm is parameterized on the partition number \( p \). At line 1 we
construct an array \( et_p \) of \( np \) elements. Each element of \( et_p \) is itself an array of Nats
of size one and contains the single value \( s_p \). Therefore \( et_p(q)(0) == s_p \) independently
of \( q \). That is, each partition sends its local size \( s_p \) to all other partitions. After
exchange, \( et'_p \) contains \( np \) arrays of Nats of size one. According to the specification
of exchange, \( et'_p(q)(0) == s_q \) independently of \( p \). This means that \( et'_p \) is the same table
on all partitions. Thus, in the loop at lines 4-6, the same offset array is computed on
all partitions.
Algorithm 11: offsets

1: \( \text{et}_p = \text{make\_array}(np, \text{make\_array}(1, s_p)) \)
2: \( \text{et}'_p = \text{exchange}(\text{et}_0 \ldots \text{et}_{np-1})(p) \)
3: \( \text{ofs}_p = \text{make\_array}(np + 1, 0) \)
4: for \( k \) in \( [0 \ldots np) \) do
5: \( \text{ofs}(k + 1) = \text{ofs}(k) + \text{et}'_p(k)(0) \)
6: end for

Ensure: is\_offsets(\text{ofs}_p) == true
Ensure: \( \text{ofs}_p == \text{ofs}_q \)
Ensure: \( \text{ofs}(p + 1) == \sum_{k=0}^{k<p+1} s_k, p \in [0 \ldots np) \)

Notice that steps 2 and the testing of the last two post-conditions requires communication between remote partitions. This is formally indicated by the presence of non-local variables. These are variables with a “partition index” which is, in general, different from \( p \). Let us consider a SCIRC function \text{exchange} that implements exchange. As mentioned in the introduction, SCIRC components are C++ programs. Therefore \text{exchange} is a C++ function.

In the second part of this section we will discuss how exchange is actually implemented. For the moment, it is important to understand that the signature of \text{exchange} is quite different from the signature of the exchange function it implements:

\[
\text{template<typename X>}
\text{void exchange(CRS<X>& t, const CRS<X>& s) /*}{
\text{using local::size;}
\text{using local::pos;}
\text{REQUIRE(size(pos(s)) == n_p() + 1);}
\text{...}
\text{*/};
\]

As exchange, the \text{exchange} function is generic w.r.t. the type of the data to be exchanged \( X \). The SCIRC type \text{CRS<X>} is a compact raw storage representation of arrays of arrays of type \( X \). For the purpose of this discussion you can think of an object of type \text{CRS<X>} as of an object of type \text{Array<Array<X>>}\(^{12}\). In contrast to exchange, however, \text{exchange} does not take as arguments tuples of arrays of arrays and does not “return” a tuple objects. Instead, \text{exchange} simply takes a “local” array of arrays \( s \) and returns a “local” array of arrays \( t \).

The apparent contradiction between the signature of exchange and the signature of the correspondent implementation \text{exchange} is typical in the SPMD distributed computational model. In this model, \( np \) copies of a program are executed in parallel. Each copy has an associated local memory space and local data. The program is parameterized on a program id \( p \) and there is a \text{computational environment} that associates to each copy of the program a different value of \( p \) in \( [0 \ldots np) \).

In the SPMD computational model, only local data – data in the local memory space – appear in function signatures. This makes the signature of SPMD functions

\(^{12}\)In C++, expressions involving text like \text{Array<Array<X>>} yield syntax errors because the double closing angular brackets are interpreted as the “right-shift” operator. In this report we do not care about this problem and freely write \text{Array<Array<X>>} to denote the type of arrays of arrays of some type \( X \).
similar to their SPSD (single program single data) sequential counterpart. In the
SPMD computational model, the visibility of local data is controlled by the same
scoping rules as in the SPSD model. SPMD functions, however, can also access non-
local data via communication between remote partitions. This can be done because
the computational environment provides, beside the local program id $p$, the total
number of program copies $n_p$ and suitable communication primitives.

The absence of explicit representation of non-local argument data in the signa-
ture of SPMD functions is often a source of ambiguity and confusion. It makes it
challenging to design software which is easily understandable.

Consider a simple size function that computes the size of an array. In a SPSD
computational environment size has the following signature:

(11) $size :: \text{Nat} \leftarrow \text{A}(\text{X})$

In a SPMD computational environment one can think of two size functions. One with
signature as in 11 and one with signature

(12) $size :: \text{Nat} \leftarrow \text{A}(\text{X})^{n_p}$

The latter function is understood to take a distributed array argument and to compute
its size. This is the sum of the sizes of the tuple elements. SPMD implementations of
12 obviously require data exchange between partitions and are different from imple-
mentations of 11. As explained above, however, SPMD implementations of 12 would
have exactly the same signature as SPMD implementations of 11, namely:

```cpp
template<typename X>
Nat
size(Array<X>& a);
```

Thus, one of the first problems a design has to face is that of developing an unam-
biguous naming scheme for functions that appear both in the SPSD and, possibly
in two flavors, in the SPMD computational environment. A related problem is that
of representing the computational environment itself and of avoiding the erroneous
usage of SPMD features in a SPSD program and vice-versa.

To understand the possible consequences of mixing up SPSD and SPMD func-
tions in the same program consider the role of pre- and post-conditions. In SC-
DRC, pre- and post-conditions play an important role in code-level documentation.
A pre-condition example can be seen in the signature of `exchange` given above:

`REQUIRE(size(pos(s)) == n_p() + 1);` Here $s$ is an object of type $\text{CRS<X>}$.
For such objects, `size(pos(s))` returns a natural number equal to the size of the array
of arrays $s$ represents. In SCRD and in the SPMD computational environment,
`REQUIRE(expr)` checks, on partition 0, that the value of $expr$ is true on all parti-
tions\(^{13}\). A necessary condition for `REQUIRE` to work is that it is called on all parti-
tions. Obviously it would be a mistake to call a SPMD `REQUIRE` in a SPSD program. It also
would be a mistake to use `REQUIRE`, in a SPMD program, to check a pre-condition of
a function that is not called on all partitions!

Because of the fact that even simple functions like size and pre- and post con-
ditions have implementations that depend on the computational environment (and,
in the SPMD environment possibly come in two flavors), all or almost all SCRD
components depend on the computational environment.

\(^{13}\)What happens on other partitions and why `REQUIRE` has “tuple-semantics” on partition 0 is not
relevant for the present discussion.
Thus, if the computational environment were represented by some type, possibly a singleton, all SCDCR types and functions would be parameterized on such type or take an extra “environment” argument. Beside being very cumbersome, this approach would be of little help in preventing inconsistent mixtures of SPSD and SPMD features.

In SCDCR, we represent computational environments by means of C++ namespaces. In contrast to (static) classes, C++ namespaces can be defined and extended in files which can be compiled separately. Thus, namespaces are very suitable for accounting for dependencies that affect all software components. This can be done through embedding: a given component — for instance our size function — can be made behave in two different ways — for instance according to two different definitions of pre- and post-conditions — by embedding it into two different namespaces.

The SPSD and the SPMD distributed computational environment are represented by the namespaces SPSD and SPMD\_{Distr}\textsuperscript{14}. A namespace SPMD\_{Shared} might be added as a further development to represent a SPMD shared computational environment. Programs that use SCDCR usually include either SPSD or SPMD\_{Distr} components. These are accessed via standard using declarations like in the following example:

```cpp
#include <numeric_types/src_cc/Nat.h>
#include <spmd_distr/src_cc/SPMD_Distr.h>
#include <spmd_distr_ops/src_cc/SPMD_Distr_ops.h>
#include <spmd_distr_array/src_cc/SPMD_Distr_Array.h>
#include <spmd_distr_array/src_cc/SPMD_Distr_Array_ops.h>
#include <spmd_distr_iter/src_cc/SPMD_Distr_Interval_Iter.h>
#include <spmd_distr_iter/src_cc/SPMD_Distr_Interval_Iter_ops.h>

using namespace SPMD_Distr;

int main(int argc, char **argv) {
    initialize(argc, argv);
    const Nat sz = p() + 1;
    Array<Nat> ofs;
    offsets(ofs, sz);
    VERIFY(local::is_offsets(ofs));
    VERIFY(ofs[n_p()] == local::sum(local::interval_iter(n_p())));
    finalize();
    return 0;
}
```

At lines 1-7 we include standard SCDCR components. The file Nat.h provides a type for positive integer numbers. SPMD\_{Distr}.h, SPMD\_{Distr.ops}.h, SPMD\_{Distr Array}.h and SPMD\_{Distr Array ops}.h provide the SPMD distributed computational environment, a set of related operations — in this case offsets — arrays and array operations. The last two header files provide iterators over zero-based integer intervals and related operations.

\textsuperscript{14}In order to avoid ugly line breaking in code listings, in the remaining of this report we use Distr, distr, Tri, Rect, Coord, Sys, iter instead of Distributed, distributed, Triangulation, Rectangular, Coordinate, System, iterator respectively.
With the using declaration at line 9 we bring the names of SPMD::Distr in the local scope. These are, among others: initialize, p, Array, offsets, n, p and finalize and local. The latter is the name of a namespace embedded in SPMD::Distr. Most of the names in SPMD::Distr::local are function names. The correspondent functions do not have “tuple semantics” and only act on the local arguments. Thus, in the SPMD::Distr namespace, there is a function size implementing (12) and a function local::size that implements (11). If we were in the SPSD computational environment, there would be only a size function.

The functions initialize, p, n, p and finalize are all declared in SPMD::Distr.h. The call to initialize at line 13 initializes the computational environment. From line 13 and up to the end of finalize at line 19, np copies of the program are running in parallel on np partitions. Program copies and partitions have id numbers from zero to np − 1. The local id number is provided by the function p. The value of np depends on how the program has been started and is provided, by n, p.

At line 14 we store p() + 1 in sz. Thus, sz is equal to 1 on partition 0, 2 on partition 1 and so on. An empty array of Nats ofs is initialized at line 15. The SCDRC type Array is declared in SPMD::Distr::Array.h. Array is essentially an STL (Standard Template Library) vector with overridden element access operators for bounds checking. Of course, bounds checking is implemented with pre-conditions and, as seen above, the implementation of pre-conditions depends on the computational environment. Therefore we have, in SCDRC, SPSD and SPMD::Distr arrays.

At line 16, ofs and sz are passed as arguments to a function offsets. This is declared in SPMD::Distr::ops.h and is a C++ implementation of algorithm 11. At line 17 we verify that the local result ofs is actually an offsets array. The function local::is_offsets is defined in SPMD::Distr::Array::ops.h and is particularly simple:

```cpp
template<typename A>
inline
bool
is_offsets(const Array<Nat, A>& a) {
    using local::size;
    using local::is_non_decreasing;
    return (size(a) > 0 && a[0] == 0 && is_non_decreasing(a));
}
```

Notice that, if offsets is an implementation of algorithm 11, the assertion at line 17 should always evaluate to true since it corresponds to the first post-condition of the algorithm. At line 18 we check that the value of the last component of ofs is in fact the sum of the sizes which have been passed to the offsets function. We compute this sum by first constructing an object of type Interval::Iter. This type is provided by SPMD::Distr::Interval::Iter.h. The correspondent “ops” header file provides the factory function interval::iter.

Factory functions are “global” functions that return an object of a given type X by simply calling a constructor of X. They can be effectively used to avoid complicated type declaration where lightweight objects can be constructed “on the fly” and passed by value to functions like sum. In SCDRC this is often the case when working with iterators. To appreciate the gains in readability that can be achieved by systematic usage of factory functions, compare the program fragments:

```cpp
return sum(filter_iter(e8, map_iter(f, array_iter(a))));
Array_Iter<int> ia(a);
```
Map_Iter<F, Array_Iter<Nat> > ifa(f, ia);
Filter_Iter<Eq, Map_Iter<F, Array_Iter<int> > > > iffa(e8, ifa);
return sum(iffa);

return sum(Filter_Iter<Eq, Map_Iter<F, Array_Iter<int> > > (e8, Map_Iter<F, Array_Iter<int> > (f, Array_Iter<int>(a))));

While the first fragment is both terse and readable the same can hardly be said of the other two\textsuperscript{15}. The \texttt{interval\_iter} factory function takes a natural number \(n\) and constructs an object of type \texttt{Interval\_Iter}. Iterator objects can be advanced with a \texttt{next} command, queried with \texttt{is\_end} and dereferenced with the standard star operator. \texttt{sum} is a generic function that adds the values visited by its iterator argument to the value of an initial default argument. \texttt{sum} is implemented in \texttt{SPMD\_Distr\_Iter\_ops.h}:

\begin{verbatim}
template<class Iter>
inline
    typename Iter::value_type
    sum(Iter iter, typename Iter::value_type initial = 0) {
        using local::begin;
        using local::is_end;
        using local::next;
        typename Iter::value_type result = initial;
        begin(iter);
        while(!is_end(iter)) {
            result += *iter;
            next(iter);
        }
        return result;
    }
\end{verbatim}

4.2. Components, files, directories. While the two computational environments supported by SCDRC are implemented through C++ namespaces, most software components are implemented as classes or collections of functions.

In SCDRC names for variables, functions, namespaces, classes, files and directories, single words are separated by underscores. Variable, function and directory names are written in lower case like in \texttt{is\_non\_decreasing}. In names of namespaces and classes, the first character of single words starts in uppercase like in \texttt{Nat}. The other characters are usually written in lowercase. Exceptions are class and namespace names which contain acronyms. They are written as in \texttt{CRS\_Rel} where CRS means “compact row storage”.

As explained above, most SCDRC names are embedded either in the SPSD or in the SPMD\_Distr namespace. As you can see from the code examples, the namespace name appears as a prefix of both file and of the directory names. Thus, a relation based on a compact row storage representation is implemented, in the SPSD computational environment, in the class SPSD::CRS\_Rel. This is declared in the file SPSD\_CRS\_Rel.h which is found in the spsd\_relation/src/ directory.

\textsuperscript{15}We are thankful to Dr. Andreas Priesnitz for suggesting the usage of factory functions in SCDRC.
Header files usually contain the declaration of a single class or of a set of functions. In the second case, such functions are often operations involving a particular type. Then, the corresponding file is named after the involved type and carries the `ops` extension as in `SPSD_CRS_Rel_ops.h`. Directories usually contain more than one file. In `spsd_relation/src/`, for instance, you will also find `SPSD_Reg_Rel.h`, `SPSD_CRS_Rel_ops.h` and `SPSD_Reg_Rel_ops.h`. The following scheme shows the most important components and the file structure of SCDRC:
4.3. Interfaces, class operations, contracts and documentation. The SCDRC components listed above have been conceived on the basis of the formal specifications presented in section 3. Those specification are part of the documentation of SCDRC. In the process of designing and implementing such components, we had to make choices that cannot be motivated on the basis of the specifications alone. In section 4.1 we have discussed one such choices: how to represent computational environments in implementations. From that discussion it is obvious that the motivation for selecting namespace-based representations instead of singleton parameterization was motivated by language specific considerations.

4.3.1. Interfaces. Another choice that has a deep impact on how SCDRC components look like is that of how interfaces are designed. There are, in C++, essentially two possibilities.

One possibility is to represent interfaces explicitly by means of abstract classes. Concrete classes that implement that interface – in Haskell, types that are instances of that type class, see [17] – inherit from the abstract (interface) class. This approach is often called, in C++, dynamic polymorphism, see [15]. In this context, dynamic polymorphism means that objects of (the types of) concrete classes can be manipulated through pointers or references of (the type of) the interface class. Using dynamic polymorphism, a single program, e.g. to converse relations, can be used to compute the converse of a number of concrete relation types. This approach allows one to write generic programs and to avoid code duplication. This, in turn, improves code correctness, documentation and maintainability. In this approach, a function \( f \) that takes as argument an object of type (castable to a) reference to an abstract class \( \text{Rel} \) can be read as a generic rule of the kind: “for all types which are instances of \( \text{Rel}, f \) does ...”.

Dynamic polymorphism has two main drawbacks. The first one is efficiency. The second one is fragility. The possibility to manipulate concrete objects via references to abstract classes implies some extra storage requirements. Moreover, the access
to concrete objects necessarily takes place via a (virtual) pointer (table). This fact
together with limitation in compiler implementation effectively prevent inlining of
function calls. Extra storage and potentially slower access are negligible if the data
structures and the function call they affect are not atomic. Otherwise they are unac-
ceptable. In our examples it would be unacceptable if, in a concrete implementation
of a relation, the access to the indexes which are in relation with a given source index
would take place via virtual pointer table.

The second drawback of dynamic polymorphism is that it leads to deep and possibly
stiff class hierarchies. It also tends to lead to “fat” interfaces if multiple inheritance
is not systematically used. Systematic use of multiple inheritance is not, in turn,
without problems, see [15].

Another possibility of representing interfaces in C++ is implicitly by means of
parameterization. This approach is called static (or compile-time) polymorphism. In
this approach, any parameterized data structure implicitly defines a set of require-
ments. For instance, the template function sum listed at page 31 implicitly defines
the following set of requirements for the type Iter of argument object:

1. Iter must export the type value_type.
2. A void function local::begin must be defined for objects of type Iter.
3. A function local::is_end of type castable to bool must be defined for objects
   of type Iter.
4. A void function local::next must be defined for objects of type Iter.
5. A (dereferencing) operator* of type Iter::value_type must be defined for
   objects of type Iter.
6. operator+= must be defined for objects of type Iter::value_type.

The above set of requirement could be interpreted as a refinement of an iterator
concept in the STL sense. In general, however, sets of requirements implicitly defined
by parameterized data structures cannot be easily re-conducted to useful concepts.
Implicit interfaces tend to reflect more the way types are actually used than the
concepts they represent.

Implicit interfaces are more efficient and more flexible than explicit ones but they
also have serious drawbacks. Some drawbacks are of practical nature and reflect the
state of the art in C++ compiler technology: error messages after failures in instanti-
ations of heavily parameterized data structures are known for their poor readability.
Other drawbacks are typical of the C++ implementation of templates: constrained
genericity – expressing conditions like “for all types Iter such that . . . ” – is not
directly supported in C++. Complex schemes for mimicking “type-class instance re-
lationships” with static polymorphism in C++ have been proposed, among other, by
Barton-Nackman [2] and Kothari and Sulzmann [13]. A more systematic approach is
proposed in [16].

In SCDRC, all interfaces are represented statically and are therefore implicit. We
have tried to document the intended usage of parameterized data structures with
careful choices of template argument (type) names. The above definition of sum, for
instance, suggests that the first argument is expected to be an iterator. Of course,
every type fulfilling the requirements 1-6 can be used as an argument of sum. We have
not been able to systematically apply Barton-Nackman or “enable_if”-like approaches
to constrain generic template parameters in SCDRC.
4.3.2. Class operations. For a given SCDRC type, a certain number of operations are declared in the corresponding ops files. Some operations are implemented for almost all SCDRC types. These are those operations that correspond to the type’s member functions and output operations. Examples of operations corresponding to a type’s member functions are the already seen factory functions. They correspond to calls of a type’s constructor. The map\_iter function used above, for instance, is defined as follows:

```cpp
template<class Fun, class Forward_Iter>
inline
Map_Iter<Fun, Forward_Iter>
map_iter(const Fun& f, const Forward_Iter iter) {
    return Map_Iter<Fun, Forward_Iter>(f, iter);
}
```

Other standard functions corresponding to a type’s member functions are size, is\_empty, begin, end for arrays and lists and is\_end, next for iterators. A version of operator<< to output an object’s value is implemented for almost all SCDRC types. Notice that the names standard functions corresponding to some type’s member functions are necessarily heavily overloaded. Of course, heavily overloaded names negatively affect compile time. On the other hand, they allow to achieve a higher uniformity of notation than the more conventional mixture of member and global functions. A comparison of the following code fragments shows that uniformity of notation can, in fact, significantly improve readability:

```cpp
REQUIRE(target_size(t) <= back(offsets(source_size(s))));
REQUIRE(t.target_size() <= offsets(s.source_size()).back());
```

4.3.3. pre- and post-conditions and documentation. An important aspect of the design of SCDRC is the systematic usage of contracts as a documentation element. Consider, for instance, the following code fragment:

```cpp
template<typename A>
inline
Array<Nat>
invert_permutation(const Array<Nat, A>& p) {
    using local::require;
    using local::ran_size;
    using local::source_size;
    using local::is\_in\_normal\_form;
    using local::compose;
    using local::ensure;
    REQUIRE(ran_size(p) == source_size(p));
    REQUIRE(is\_in\_normal\_form(p));
    const Nat sz = source_size(p);
    Array<Nat> result(sz);
    for(Nat i = 0; i < sz; i++)
        result[p[i]] = i;
    ENSURE(compose(result, p) == id(sz));
    ENSURE(compose(p, result) == id(sz));
    return result;
}
```
The function `invert_permutation` takes an array of natural numbers \( p \) and returns an array of natural number \( \text{result} \). \( p \) is understood to represent a permutation of \([0 \ldots \text{size}(p)]\). Therefore the first pre-condition of `invert_permutation` requires the size of the range of \( p \) to coincide with the size of \( p \). The second pre-condition requires \( p \) to be in normal form. The query function `is_in_normal_form` is declared as follows:

```cpp
template<class A>
inline
bool
is_in_normal_form(const Array<Nat, A>& a) /*{
    using local::is_empty;
    using local::min_max_elem;
    using local::ran_size;
    using local::target_size;
    using local::ensure;
    ...
    ENSURE(result == (ran_size(a) == target_size(a)));
    return result;
}*/
```

The post-condition of `is_in_normal_form` shows that the pre-conditions of `invert_permutation` actually require:

\[
\text{target\_size}(p) = \text{ran\_size}(p) = \text{source\_size}(p)
\]

For non empty arrays of natural numbers, the size of the target is defined to be equal to one plus the maximal element of the array. Thus, the first equality implies that \( p \) is surjective on \([0 \ldots \max\_elem(p)]\). The second equality guarantees that \( p \) is bijective: if there were distinct indexes \( i \) and \( j \) in \([0 \ldots \text{source\_size}(p)]\) with \( p(i) = p(j) \), the size of the range of \( p \) could be, at most, equal to \( \text{source\_size}(p) - 1 \). Thus, the pre-conditions of `invert_permutation` require \( p \) to be a permutation. The post-conditions require the result of `invert_permutation` to be the inverse permutation of \( p \): composition of result with \( p \) (and of \( p \) with result) shall yield the identity. The SCDRC function `id` is another example of a factory function.

The discussion above shows that consistent usage of contracts and sensible naming schemes can significantly improve code understandability and documentation. Of course, contracts cannot, in general, express full problem specifications and for many SCDRC we have not been able to derive suitably implementable post-conditions. Contracts and pre-conditions in particular, however, can be extremely useful in documenting the intended usage of functions. As an example we list the pre-conditions on the “in” arguments of `partition_recursive`, a subset of the the SCDRC interface to Metis for partitioning the source of symmetric, anti-reflexive relations with minimal edge cut:

```cpp
template<class Rel>
void
partition_recursive(Array<Nat>& part,
    Nat& n_cuts,
```

\( ^{16} \)In SCDRC, \text{size} and \text{source\_size} are, for arrays, the same function. Both return the length (the number of elements) of the array. We use \text{size} and \text{source\_size} for container and function arguments, respectively.
const Rel& r,
const Nat n_partitions,
const Array<Nat>& elem_weights = Array<Nat>(0),
const Array<Nat>& pair_weights = Array<Nat>(0),
const Array<Nat>& options = Array<Nat>(0)) {
...
REQUIRE(is_symmetric(r));
REQUIRE(is_anti_reflexive(r));
REQUIRE(n_partitions <= source_size(r));
REQUIRE(size(elem_weights) == 0 ||
    size(elem_weights) % source_size(r) == 0);
REQUIRE(size(pair_weights) == 0 ||
    size(pair_weights) == size(r));
REQUIRE(size(pair_weights) == 0 ||
    is_symmetric_weights(pair_weights, r));
REQUIRE(size(options) == 0 || size(options) == 5);
REQUIRE(size(options) == 0 || options[0] == 0 || options[0] == 1);
REQUIRE(size(options) == 0 ||
    (options[0] == 1 &&
REQUIRE(size(options) == 0 ||
    (options[0] == 1 && options[2] == 1));
REQUIRE(size(options) == 0 ||
    (options[0] == 1 && options[3] == 1));
REQUIRE(size(options) == 0 ||
    (options[0] == 1 && options[4] == 0));
...
}

At the present state of development, the SCDRC approach towards source-level documentation is based on contracts and on a set of naming rules. In addition to the rules discussed in section 4.2, we have used the following conventions:

- **Function names:**
  - Boolean queries start with *is* or *are* followed by the queried feature as in *is_empty*, *is_initialized*, *are_subarrays_nubbed*.
  - Feature queries are named after the queried feature, usually a noun as in *size*, *breadth*, *back*.
  - Commands use imperative forms as in *invert_permutation*, *compose*, *converse*.

- **Variable names:**
  - Cardinalities (the number of dimensions, the number of partitions) are prefixed with *n* as in *n_dims*, *n_partitions*, *n_triangles*.
  - Function arguments, in particular constructor arguments, are spelled in length as in CR3(const Array<T>& breadth, const Array<Nat>& pos).

4.4. **Iterators.** Iterators play an important role in SCDRC. As we will see in the next section, all SCDRC relations implement a common interface. This interface
essentially consists of five iterators. Examples of SCDRC functions acting on iterators or producing iterators are the factory functions discussed in section 4.1.

As in the standard template library, SCDRC iterators act as a link between generic algorithms and a variety of traversable data structures. However, SCDRC iterators are different from STL iterators in many ways, being more similar to Boost ranges. In SCDRC, a type `Iter` represents an iterator if:

- `Iter` exports the type `value_type`. This means that `Iter` contains a public typedef declaration where `value_type` appears as the second argument of typedef as in `typedef Nat value_type;` in the SCDRC class `Interval_Iter`.
- `Iter`s can be dereferenced with `operator*`.
- `Iter`s can be queried with `is_end`.
- `Iter`s can be incremented with `next`.

Because the state of SCDRC iterators can be queried with `is_end`, traversable data structures can be visited with a single iterator and there is no need to operate with “begin” and “end” iterator pairs as in STL. A second important difference between STL and SCDRC iterators is that SCDRC iterators can be used to represent lazily evaluated computational rules. Consider, for instance, the program fragment:

```c++
Map_Iter<F, Array_Iter<Nat>> ifa(f, ia);
```

Here `ifa` is an object of type `Map_Iter`. It is constructed with a function `f` of type `F` and with an iterator over arrays of natural numbers. We can step through and dereference `ifa` with the iterator interface, for instance as in:

```c++
while(!is_end(ifa)) {
    if(is_odd(*ia))
        cout << *ifa << endl;
    next(ia);
    next(ifa);
}
```

The program fragment prints the values of `f` at the odd elements of the array iterated by `ia`. The evaluation of `f` takes place where `ifa` is dereferenced. This means that `ifa` represents the mapping of `f` on `ia` lazily: if the natural numbers visited by `ia` are all even, `f` is never evaluated.

A small set of basic iterator classes is implemented, in SCDRC, in `spsd_iter` and `spmd_distr_iter`: `Interval_Iter`, `Array_Iter`, `Slist_Iter`, `Map_Iter` and `Filter_Iter`. These operators extend the above interface with the command `begin`, the query `size` and with the random access operator `operator[]`.

4.5. Relations. As explained in 4.3.1, SCDRC interfaces are implicit. As for iterators, there is no abstract class from which concrete relation classes are derived and we use the expression relation interface to denote a set of functionalities that any concrete SCDRC relation `Rel` has to implement. These are:

- Public definitions of five nested iterator classes:
  - `Lambdas_Iter`
  - `Lambda_Sizes_Iter`
  - `Lambda_Offsets_Iter`
  - `Graph_Ran_Iter`
  - `Graph_Dom_Iter`

Each class implements the basic iterator functionalities of the previous section.
- Public definition of five void constant `Rel` member functions: `lambdas`, `lambda_sizes`, `lambda_offsets`, `graph_ran` and `graph_dom`. The return types of these functions correspond with the types of the nested classes.
- Definition of five factory functions `lambdas`, `lambda_sizes`, `lambda_offsets`, `graph_ran` and `graph_dom`. These functions take (constant references) `Rel` arguments and return types correspondent to those of the nested classes. They simply call the correspondent member function on the argument. For instance, `lambdas` is defined as follows:

```cpp
inline
Rel::Lambdas_Iter
lambdas(const Rel& r) {
    return r.lambdas();
}
```

In `Lambdas_Iter`, the value of `value_type` is `Array_Iter<const Nat>`: this is the type of the objects which are obtained by dereferencing the output of `lambdas`. If `r` is a SCDRC object representing some relation `R`, `lambdas(r)[j]` returns an iterator over the indexes of `R[j]`.

In the other nested classes, the value of `value_type` is `Nat`. `lambda_sizes(r)[j]` and `lambda_offsets(r)[j]` provide access to the size and to the offset associated to `R[j]`. For `j` in `[0...source_size(r)]`, they fulfill:

```cpp
size(lambda_sizes(r)) == source_size(r)
∧
size(lambda_offsets(r)) == source_size(r) + 1
∧
lambda_offsets(r)[0] == 0
∧
lambda_offsets(r)[j + 1] == lambda_offsets(r)[j] + lambda_sizes(r)[j]
```

The iterators returned by `graph_ran` and by `graph_dom` allow traversing a relation as a set of pairs. This is particularly useful in the implementation of relational operations like `converse` and `compose`.

At the present stage, SCDRC provides two data structures for relations: `CRS_Rel` and `Reg_Rel<n>`. The first class is useful for `irregular` relations i.e. for relations for which the size of `R[j]` is not constant. The second class is useful for `regular` relations such as the vertex-triangle relation of our example. The template `Nat` value `n` represents the size of the `R(j)`.

Both `CRS_Rel` and `Reg_Rel<n>` use, internally, arrays of `Nats` to store the elements of `R(j)` and guarantee access to these elements in asymptotically constant time. As mentioned in section 2, efficient access to `R(j)` is crucial for the implementation of relation-based algorithms.

In many important applications, however, implementations of `R(j)` in terms of arrays are sub-optimal. For neighborhood relations on structured grids, for instance, functions like `R(j)` can be written in terms of simple analytical expression. In this case, it would be computationally inefficient to represent such relations with `Reg_Rel<n>s` or with `CRS_Rel`s. Using the examples of `CRS_Rel` and `Reg_Rel<n>`, it is easy to extend SCDRC with lightweight types which are optimized for particular classes of relations.
As far as the new types implement the interface described above, they can be used in relation-based computations in the same way as \texttt{CRS	extunderscore Rel} and \texttt{Reg	extunderscore Rel\textless n\textgreater}. We plan to provide such extensions upon user’s demand.

4.6. Relation-based algorithms. In section 2 we have introduced relation-based algorithms as generic computational rules defined in terms of a relation $R$, of a functions $f$ taking values in the target of $R$ and of a “reduction” operator $h$. In section 4 we have seen that, in order to apply parallel SPMD relation-based algorithms, auxiliary data structures and functionalities are in general needed. These can be described in terms of suitable exchange and access tables and in terms of functionalities to complete local array-based partial representations of $f$.

In SCDRC, relation-based algorithms are implemented as specializations of a generic type RBA:

$$\text{template<class H, class F, class R, \\
Nat h\_arity = H::arity, \\
Nat f\_arity = F::arity> \\
class RBA;}$$

Objects of type \texttt{RBA} can be easily constructed by passing objects of type \texttt{H}, \texttt{F} and \texttt{R} to the \texttt{RBA} constructor as in

\begin{verbatim}
RBA<H, F, R> tca(triangle_center, 
vertex_coordinates(tri), 
vertex_triangle(tri));
\end{verbatim}

\texttt{H} is, in general, a user-defined type. It is required to export the type \texttt{return\_type}, the constant \texttt{Nat} value \texttt{arity} and to implement a generic function call operator \texttt{operator()}. For the case $\texttt{arity} == 1$, \texttt{operator()} is required to have the following signature:

$$\text{template<class Random\_Access\_Data> 
return\_type 
operator() (Random\_Access\_Data\& x) const; \;}$$

In most practical cases, \texttt{H} is a user defined wrapper of some specific, problem dependent computational rule. In the example given above, for instance, \texttt{H} represents a rule for computing the triangle centers. As mentioned in section 2, this rule depends on application specific aspects like the kind of triangles (plane, spherical), the coordinate system associated with the triangle vertexes etc. In our triangle centers example, for instance, \texttt{H} is defined as follows:

$$\text{typedef Triangle\_Area<Coordinate\_System> H; \;}$$

$$\text{template<class Coordinate\_System> 
class Triangle\_Center \{} \;}$$

$$\text{typedef Vector<Real, Coordinate\_System::n\_crds> return\_type; 
static const Nat arity = 1; \;}$$

$$\text{template<class Random\_Access\_Data> 
Vector<Real, Coordinate\_System::n\_crds> 
operator() (Random\_Access\_Data\& x) const \{} \;}$$
In many practical cases, \( F \) is an array of a type matching the type of the arguments of the specific rule wrapped by \( H \), in our example \texttt{triangle\_center}. However, \( F \) could be an RBA itself. This allows relation-based algorithms to be composed to express complex computational patterns. The type \( R \) is an SCDRC relation. As discussed above, SCDRC provides, at the present stage, two relation types: \texttt{CRS\_Rel} and \texttt{Reg\_Rel\langle n\rangle}.

In the SPSD computational environment, relation-based algorithms can be evaluated right after construction. The triangle center algorithm object \texttt{tca} of our example, for instance, could be evaluated on the triangles of \texttt{tri} with the simple loop:

\begin{verbatim}
for(Nat i = 0; i < source_size(vertex_triangle(tri)); i++)
    cout << tca(i) << endl;
\end{verbatim}

In the SPMD distributed computational environment, and for the case in which \( F \) is an array, the auxiliary access and exchange tables have to be computed and the local arrays \( f \) have to be completed before relation based algorithms can be evaluated. The SPMD distributed computational environment supports these computations with two functions. They can be called with relation-based algorithm arguments as follows:

\begin{verbatim}
init_access_exch_tables(tca);
complete_f(tca);
\end{verbatim}

As discussed in section 3.3, these functions allow users to set up parallel computations of distributed relation-based algorithms without having to care about message passing level communication and related synchronization, mutual exclusion, deadlock or race condition problems. At the same time, \texttt{init\_access\_exch\_tables} and \texttt{complete\_f} support the optimization of computational procedures in which \texttt{complete\_f} has to be called at each step of some iteration, \texttt{init\_access\_exch\_tables}, however, only once at the beginning of the iteration. In scientific computing, such iterative procedures are found, e.g. in the numerical integration of (partial) differential equations, in linear algebra and in the solution of optimization problems.

### 4.7. Communication primitives, exchange and MPI interface

In the next and last section of this report, we show how to setup a parallel computation of the centers of the triangles of a distributed triangulation. We are going to use some of the SCDRC data structures discusses in this section, functionalities like \texttt{init\_access\_exch\_tables} and \texttt{complete\_f} and primitives for redistributing distributed arrays and relations.

As mentioned in section 3, all SCDRC functionalities that require data communication between partitions have been designed on the top of a single communication primitive called exchange. An implementation of exchange is provided, in the SPMD distributed computational environment by the function \texttt{SPMD\_Distr::exch}. This function is implemented, internally, in terms of the MPI primitive \texttt{MPI\_Alltoallv}. This is one of the few MPI functionalities used by SCDRC. In SCDRC, most MPI entities are accessed via a small interface. This is implemented as a MPI namespace embedded in \texttt{SPMD\_Distr}. \texttt{SPMD\_Distr::MPI} contains:

- The type \texttt{Comm (MPI\_Comm)}.
- The constant values:
  - \texttt{INT (MPI\_INT)}.  


STRUCTURING DISTRIBUTED RELATION-BASED COMPUTATIONS WITH SCDRC

- CHAR (MPI_CHAR).
- communicator (MPI_COMM_WORLD).

- The functions:
  - initialize, implemented in terms of MPI_Init.
  - is_initialized, implemented in terms of MPI_Initialized.
  - finalize, implemented in terms of MPI_Finalize.
  - is_finalized, implemented in terms of MPI_Finalized.
  - get_rank, implemented in terms of MPI_Comm_rank.
  - get_size, implemented in terms of MPI_Comm_size.

Other MPI entities used in SCDRC are the already mentioned MPI_Alltoallv and MPI_Abort and MPI_Allgather. MPI_Alltoallv is used only in the implementation of exch. The other two MPI primitives are used only in the implementation of contracts in the SPMD distributed environment. Thus, the dependency of SCDRC on MPI is limited to a few primitives and confined to very few files.

5. Preliminary results, outlook

In this section we show how SCDRC can be used to set up a parallel computation of the center of the areas of a triangulation and present some preliminary results. We close this report with an outlook on future activities.

5.1. Center of area computations. In this example we set up a parallel computation of the center of the areas of a triangulation. The computation is embedded in a time stepping procedure. On each partition and at each iteration step, the coordinates of the vertexes of the triangulation are displaced according to a simple rule and a new center is computed. The computation of the center is based on two relation-based algorithms, one for computing the centers and one for computing the areas of the triangles. The coordinates of the center of the triangulation and of its total area are accumulated in local variables c and a. At the end of the iteration, the average center coordinates and the average areas are collected, together with simple computation metrics, on partition zero. Here the average center is computed and the results are sent to the standard output.

Beside the SCDRC functionalities discussed in section 4, we use simple components for representing surface triangulations and basic rules for geometrical computations in a three-dimensional Cartesian coordinate system. These components are not part of SCDRC. The type Tri, in particular, just wraps a vertex-triangle relation and a vertex coordinates array in a data structure. For this data structure, basic input-output functionalities are provided in different data formats, e.g. for visualization.

In setting up the computation, we assume, as discussed at the end of the specification of problem 2 in section 3, that the triangulation, together with correspondent offsets for the vertex coordinates and for the vertex-triangle relation, are stored in files which are given as arguments on the command line. This is consistent with steps 1-3 of page 18. This means that, in a pre-processing step, we have already computed a partitioning of the triangles and of the vertexes of the triangulation and we have renumbered the original vertex-triangle relation and the vertex coordinates array accordingly. The output of these computations, a “new” triangulation and the correspondent offsets arrays, is the input of our example. We interleave the listing of the program with remarks.

1 #include <numeric_types/src_cc/Nat.h>
At lines 1-2, we include the header files for natural numbers and contracts. These do not depend on the computational environment. Subsequently, we include the SPMD distributed computational environment, its operations, distributed arrays, relations, iterators and relation-based algorithms with the respective operations.

Next, we include non-SCDRC components for geometrical rules, triangulations and IO in ASCII format. This is the format in which the triangulation and the offset arrays are stored in the input files. At line 24, we bring all the names of the SPMD distributed computational environment in the global scope. At line 26, we inject local::operator<< in the global scope. As mentioned earlier, most SCDRC data structures can be processed by operator<<. At the present, we have not implemented any version of operator<< with “tuple-semantics”. The implemented operator<< act on local data and are therefore embedded, in the SPMD distributed computational environment, in the local namespace. Since there is no ambiguity in the usage of these operators, we make them available in the global scope.
At line 38, after the number of command line arguments check, we issue the first SPMD function call: \texttt{initialize} starts the SPMD distributed computational environment. From line 38 and up to the call to \texttt{finalize} at line 135, \texttt{n\_p()} copies of the program are running in parallel on \texttt{n\_p()} partitions.

First we initialize, on each partition, an empty vertex-triangle regular relation \texttt{vt}, an empty array of vertex coordinates \texttt{vx} and trivial partitioning offsets \texttt{vtofs} and \texttt{vxofs} for the source of \texttt{vt} and for the source of \texttt{vx}, respectively. These variables correspond to \texttt{vt\_p}, \texttt{x\_p}, \texttt{ovt\_p} and \texttt{ox\_p} of steps 1-3 of page 18.

Then, on partition 0, we initialize an empty triangulation \texttt{tri} in a three-dimensional rectangular (Cartesian) coordinate system (lines 45-47), we read \texttt{tri} from the file given on the command line (lines 48-49) and we initialize \texttt{vt} and \texttt{vx} with the vertex-triangle relation and with the vertex coordinates array of \texttt{tri}.

At lines 52-61, we read the offsets arrays and we verify the consistency of the input data. In particular, we check that \texttt{vtofs} and \texttt{vxofs} are indeed offsets arrays of proper size and that they represent non-decreasing partitioning functions for \texttt{vt} and \texttt{vx}, respectively. Notice the local qualifier in front of the \texttt{VERIFY} macros and in their argument expressions.
Array<Nat> ofs;
offsets(ofs, vxp.size());
typedef Rect_Coord_Sys<3> Coord_Sys;
typedef Triangle_Area<Coord_Sys> TA;
typedef Triangle_Center<Coord_Sys> TC;
typedef Array<Vector<Real, 3> > F;
typedef Reg_Rel<3> R;
TA triangle_area;
TC triangle_center;
RBA<TA, F, R> taa(triangle_area, vxp, vtp, ofs);
RBA<TC, F, R> tca(triangle_center, vxp, vtp, ofs);

At this point, we are ready to set up a parallel computation of the center of the areas of tri. We first redistribute vt and vx according to vtofs, vxofs. This is done at lines 64-67 and yields, on each partition, a new distributed vertex-triangle relation vtp and a new vertex coordinates array vxp.

Notice the usage of redistribute after offsets: this is an overloaded function (it is used, at lines 65 to redistribute a Reg_Rel<3> relation and, at line 67, to redistribute an array) which implements the specifications of problem 2' and 1' of section 3.3. Therefore, redistribute after offsets has “tuple-semantics” although, as explained in section 4.1, only local data formally appear as “function” arguments.

redistribute after offsets is an example of how SCDRC supports structuring parallel computations by providing communication primitives that relieve the user from the burden of low-level data transferring. As explained in the introduction, however, SCDRC does not attempt to hide the underlying distributed data model to the user. This model is visible in the offsets array arguments that “high-level” SCDRC primitives like redistribute after offsets require.

The offsets arrays vtofs and vxofs describe, on each partition, which chunks of vt and vx have to be sent to which partition. In order to efficiently initialize the relation-based algorithms, it is useful to compute, on each partition, the offsets that describe how the new distributed vertex coordinates array vxp is actually distributed. This is done at lines 67 and 68.

Lines 69 to 77 is where the relation-based algorithms taa and tca for triangle areas and centers are actually constructed. The typedefs instructions just introduce convenient type synonyms; the RBA constructors are called in the last two lines. Notice that both taa and tca use the same “f” function vxp and the same “R” relation vtp. The “h” functions of taa and tca are triangle_area and triangle_center, respectively. These rules are provided by non-SCDRC components. As the example shows, RBAs are quite flexible generic types: users can define their own “h” functions to construct ad hoc RBA objects.

init_access_exch_tables(taa);
init_access_exch_tables(tca);
const Nat n_iter = 10000;
const Real alpha = 0.1;
const Real omega = 1.0;
Array<Vector<Real, 3> > vx0(vxp);
Real a = 0.0;
Vector<Real, 3> c = Vector<Real, 3>(0.0);
for(Nat iter = 0; iter < n_iter; iter++) {
const Real t = 2.0 * Real_PI * Real(iter) / Real(n_iter - 1);
for(Nat i = 0; i < vx0.size(); i++)
    vxp[i] = vx0[i] * (1.0 + alpha * sin(omega * t));
complete_f(taa);
complete_f(tca);
for(Nat j = 0; j < local::source_size(vtp); j++) {
    const Real area = taa(j);
    const Vector<Real, 3> center = tca(j);
    a += area;
    c += area * center;
}
a /= Real(n_iter);
c /= Real(n_iter);

Since we are in the SPMD computational environment and the “f” function of taa and tca is represented by a distributed array, the auxiliary access and exchange tables have to be computed and the local arrays f have to be completed before taa and tca can actually be evaluated.

The computation of the access and exchange tables is done at lines 78 and 79. This is outside the main iteration (lines 86 to 98) because the tables only depend on the vertex-triangle relation and on the offsets of the vertex coordinate arrays. In contrast to the vertex coordinates, the relation and the offsets do not change during the iteration. As explained in section 3.3, the computation of the access and exchange table is computationally demanding and it would be very inefficient to recompute such tables every time we need to complete the local vertex coordinates arrays.

At lines 80-82 we initialize the number of iterations n_iter, the amplitude alpha and the frequency omega. Amplitude and the frequency are used at line 89 to impose a simple periodic motion on the vertexes of the triangulation. At lines 83-85 we initialize an auxiliary array of vertex coordinates vx0 and the variable a and c. We use a and c to accumulate the sums of the areas and of the centers of the triangulation.

Inside the iteration we first compute an “iteration time” t and move the vertexes of the triangulation. Then, at lines 90-91, we complete the local vertex coordinate array with the “new” values. As for the access and exchange table computations discussed above, SCDRC users are supposed to know that, at each iteration step, vertex coordinates have to be exchanged between partitions. However, SCDRC users do not need to know the details of which data have to be exchanged and of how this can be done. They simply call the SCDRC primitives init_access_exch_tables and complete_f. After these calls, all partitions have enough data to compute the areas and the centers of the triangles independently of each other: the loop at lines 92-97 can be done in parallel. This loop is where the computational work is actually done: for each triangle, we compute the triangle area with taa and the triangle center with tca. These results are stored in the loop variables area and center. These are used to increment a and c at lines 95 and 96. At lines 99-100, a and c are divided by the number of iteration thus yielding the average area and triangulation center.

finalize();
return 0;
5.2. Preliminary results. We have used the center of areas program discussed above to set up a test parallel computation. In spite of its simplicity, the parallel computation of the center of areas is representative in terms of computational complexity for an important set of applications, for example matrix-vector multiplications and approximation of integrals of fluxes in FVMs.

As source data, we used the earth surface triangulation shown in figure 1. This triangulation has been computed by 8 adaptive refinement steps starting from a simple icosahedron. We have used a version of the red-green local mesh adaptation algorithm developed by D. Hempel and described in [11]. At each step, we have marked for refinement those triangles which are cut by the level set $z = 0$. Here $z$ represents the altitude above or below the mean sea-level. All triangles are plane triangles. All vertexes lie in a piecewise bilinear approximation to the earth surface. This is computed with the etopo20 earth surface dataset\(^{17}\).

The triangulations have been partitioned into 2, 4, 8, 16, 32 and 64 subtriangulations using the SCDRC interface to Metis described in section 2.2.

\(^{17}\text{http://ferret.wrc.noaa.gov/cgi-bin/dods/nph-dods/data/slh/etopo20.cdf.html}\)
Figure 2 shows the result of this partitioning into 32 subpartitions and some important metrics associated with different partitionings are summarized in table 1.

In particular, the communication volume for a given partition (table 1) is defined as the number of elements (of the type of the elements of \( \text{vx, Vector<Real, 3>} \)) that that partition has to send in a single \texttt{complete_f} call.

After \texttt{init_access_exch_tables} has been called on a relation-based algorithm, this number and the number of partitions the data have to be sent to (the number of communication peers) can be queried with a SCDRC function.

The computations have been run on an IBM p655 cluster consisting of 30 nodes with 8 Power4 1.1GHz CPUs per node. For computations involving 8 partitions and above, we have used a blocking factor of 8. The blocking factor specifies the way in which processes will be assigned to a node. For a blocking factor of 8, entire nodes

<table>
<thead>
<tr>
<th></th>
<th>2 part.</th>
<th>4 part.</th>
<th>8 part.</th>
<th>16 part.</th>
<th>32 part.</th>
<th>64 part.</th>
</tr>
</thead>
<tbody>
<tr>
<td>total num. triangles</td>
<td>86750</td>
<td>86750</td>
<td>86750</td>
<td>86750</td>
<td>86750</td>
<td>86750</td>
</tr>
<tr>
<td>min. triangles per part.</td>
<td>43311</td>
<td>21344</td>
<td>10527</td>
<td>5226</td>
<td>2629</td>
<td>1307</td>
</tr>
<tr>
<td>max. triangles per part.</td>
<td>43439</td>
<td>22020</td>
<td>11153</td>
<td>5583</td>
<td>2794</td>
<td>1398</td>
</tr>
<tr>
<td>tot. comm. volume</td>
<td>187</td>
<td>406</td>
<td>638</td>
<td>1138</td>
<td>1745</td>
<td>2538</td>
</tr>
<tr>
<td>min. comm. volume</td>
<td>77</td>
<td>77</td>
<td>56</td>
<td>53</td>
<td>34</td>
<td>23</td>
</tr>
<tr>
<td>max. comm. volume</td>
<td>110</td>
<td>112</td>
<td>92</td>
<td>128</td>
<td>84</td>
<td>65</td>
</tr>
</tbody>
</table>

Table 1. Parallel triangulation center computation metrics.
Table 2. Parallel triangulation center computation times.

<table>
<thead>
<tr>
<th></th>
<th>2 proc.</th>
<th>4 proc.</th>
<th>8 proc.</th>
<th>16 proc.</th>
<th>32 proc.</th>
<th>64 proc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>time (n = 1)</td>
<td>1866</td>
<td>944</td>
<td>485</td>
<td>321</td>
<td>385</td>
<td>784</td>
</tr>
<tr>
<td>speedup</td>
<td>1</td>
<td>1.98</td>
<td>3.84</td>
<td>5.81</td>
<td>4.84</td>
<td>2.38</td>
</tr>
<tr>
<td>time (n = 10)</td>
<td>1854</td>
<td>919</td>
<td>466</td>
<td>247</td>
<td>144</td>
<td>139</td>
</tr>
<tr>
<td>speedup</td>
<td>1</td>
<td>2.01</td>
<td>3.98</td>
<td>7.51</td>
<td>12.88</td>
<td>13.34</td>
</tr>
<tr>
<td>time (n = 100)</td>
<td>1890</td>
<td>976</td>
<td>480</td>
<td>241</td>
<td>129</td>
<td>67</td>
</tr>
<tr>
<td>speedup</td>
<td>1</td>
<td>1.93</td>
<td>3.94</td>
<td>7.84</td>
<td>14.65</td>
<td>28.21</td>
</tr>
<tr>
<td>time (no comm.)</td>
<td>1878</td>
<td>926</td>
<td>458</td>
<td>229</td>
<td>114</td>
<td>57</td>
</tr>
<tr>
<td>speedup</td>
<td>1</td>
<td>2.02</td>
<td>4.1</td>
<td>8.2</td>
<td>16.47</td>
<td>32.95</td>
</tr>
</tbody>
</table>

are reserved for the application. For 2 or 4 tasks and a blocking factor of 2 or 4 respectively, it is guaranteed that the tasks will run on the same node.

Execution times were recorded for several sets of runs and speedup values calculated accordingly. Each run had a fixed number of iterations (\(N = 100000\)). Speedup is defined here as the ratio between the time taken to run on \(np\) processors relative to the time taken with 2 processors. Times have been calculated by calling the MPIWtime() function - which returns current wall-clock time - before and after the main execution loop. All processes are synchronized before measurements are taken by calling the MPIBarrier() function.

We also modified the code to investigate the effect of changing the ratio between computation and communication costs. This was done by changing how often the communication was carried out during a run while keeping the number of computation steps constant. Specifically, this was carried out by changing how often complete_f was called inside the main execution loop. Table 2 shows the execution times (in seconds) and speedup values for communication at every step (\(n = 1\)) and at every 10th (\(n = 10\)) and 100th (\(n = 100\)) step and for the case of no communication (\(n = 0\)).

With the partitioning scheme described above and with no communication, we expect the total run time to halve with each doubling of the number of processors.

Table 2 shows that this is indeed the case. As the frequency of communication increases, and with a greater number of processors, it can be seen that we get much less than this ideal speedup. In the extreme case of 64 processors and communication at every iteration (\(n = 1\)), the performance is only marginally better than when using just 2 processors.

We will now try to explain the results in table 2 by means of a simple computational cost model. The time taken for a run is the sum of two components, computation time and communication time: \(t = t_{\text{comp}} + t_{\text{comm}}\). As can be seen from table 2 computation time is inversely proportional to the number of processors being used in the no-communication case. We expect this to hold for all communication frequency cases: \(t_{\text{comp}} = k_1/np\).

The communication time for N iterations is proportional to the number of communication steps, i.e. \(N/n\).

\[
t_{\text{comm}} = t_1 \cdot \frac{N}{n}
\]
The time for one communication step $t_1$ depends of course on the amount of data to be communicated. As can be seen from Table 1, this increases with an increasing number of partitions. The relationship between $t_1$ and $np$ can be estimated as follows:

$$ t_1 = c_1 \cdot np \cdot pbs_{np} $$

where $pbs_{np}$ is the average partition boundary size for the case of $np$ partitions. Because the perimeter is proportional to the square root of the surface area:

$$ pbs_{np} = c_2 \cdot \sqrt{ps_{np}} $$

As seen from Table 1 we have subdivided the triangulation into roughly equally-sized subtriangulations and therefore

$$ ps_{np} = \frac{c_3}{np} $$

From 13, 14 and 15 it follows that

$$ t_1 = k_2 \cdot \sqrt{np} $$

with $k_2 = c_1 \cdot c_2 \cdot \sqrt{c_3}$. Thus

$$ t = \frac{k_1}{np} + k_2 \cdot \frac{N}{n} \cdot \sqrt{np} $$

This formula can be seen to be similar to the BSP cost model presented in (see [6], section 1.2).

The least-squares method was used determine the parameters $k_1$ and $k_2$ from the entire data set in table 2. It can be seen from figure 4 that equation 16 fits well with the observed results for moderate communication frequencies (bottom).

In the case of communication occuring at every step, the fit is rather poorer (figure 4, top left).
The constants $k_1$ and $k_2$ were also determined for each frequency case independently, but there was no improvement in the fit of the curve. This suggests that the relationship between $k_2$ and $np$ is perhaps not as simple as hypothesized above or that the times taken for computation and communication are not independent of each other. It may be that the high frequency communication has passed some threshold, for example a network buffer is more quickly reaching its maximum capacity when the number of processors is higher.

The experiments showed that it is not necessarily beneficial to increase the number of processors in a parallel computation and therefore it is important, when setting up a parallel computation for a given practical application, to run some preliminary tests to ascertain what a sensible value for $np$ might be. As our experiments show, this will depend dramatically on the computing architecture and the ratio between computation and communication costs.

Despite the limitation of this simple model at high communication frequencies, it can be applied to this preliminary analysis and can guide the design of a parallel application in terms of the number of processors which can be effectively applied to a problem while still achieving a reasonable speedup.

Of course, practical applications might require, at each step of some iterative procedure (e.g. for iteratively solving systems of equations) more or less operations and greater or lesser communication volume than our example. These factors will also play a role in designing the overall structure of a parallel application.

Our model fits well for low numbers of processors at low communication frequencies. Further work would need to be carried out to determine if a greater number of processors at low frequency also display the poor speedup figures we have observed at high frequencies.
5.3. Outlook. We have presented a prototype version of SCDRC, a small set of software components for distributed relation-based computations. As explained in the introduction, SCDRC is a thin software layer above message passing libraries and is not meant to be directly used by applications. Applications are expected to use SCDRC indirectly via application dependent components.

The triangulation and geometry components used to write the parallel program for computing the center of a triangulation are simple examples of an application dependent software layer built upon SCDRC. In fact, SCDRC has been written in the framework of a project for developing software components for distributed adaptive finite volume computations\textsuperscript{18}. In finite volume computations, relation-based algorithms and relation-based computations play an outstanding role.

Relation-based algorithms and relation-based computations, however, are found in different research areas at the core of many computationally demanding procedures: adaptive stochastic sequential decision processes and Bayesian network inference are two examples.

We plan to use SCDRC to develop prototype application dependent software components for one or more such applications domains. This will provide a natural testbed for our prototype and lead to a stable version of SCDRC, possibly extended with new functionalities for basic relational operations.

Acknowledgements

The authors thank the reviewers, whose comments have triggered various nontrivial improvements of the paper. It is our pleasure to thank J. Gerlach and A. Priesnitz for contributing to the development of SCDRC and for the many interesting discussions. J. Gerlach (author of Janus, [10]) has participated to the early development phase of SCDRC and has introduced us to the architecture of Janus. This work presented in this report heavily relies on free software, among others on the libraries:


In developing, testing and documenting SCDRC we have taken advantage of, among others, the GCC compiler, Emacs, \LaTeX, gnuplot, MeshTV, and the Debian / GNU Linux operating system. 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


\textsuperscript{18}Finite volumes is a technique for computing approximate solutions of partial differential equations. It is particularly suited for approximating discontinuous solutions of certain classes of partial differential equations called conservation laws.


PIK Report-Reference:

Tagungsband der Vorträge und Poster (April 1994)

No. 2 Extremer Nordsommer ‘92
Meteorologische Ausprägung, Wirkungen auf naturnahe und vom Menschen beeinflußte Ökosysteme, gesellschaftliche Perzeption und situationsbezogene politisch-administrative bzw. individuelle Maßnahmen (Vol. 1 - Vol. 4)

No. 3 Using Plant Functional Types in a Global Vegetation Model
W. Cramer (September 1994)

No. 4 Interannual variability of Central European climate parameters and their relation to the large-scale circulation
P. C. Werner (Oktober 1994)

No. 5 Coupling Global Models of Vegetation Structure and Ecosystem Processes - An Example from Arctic and Boreal Ecosystems
M. Pöchli, W. Cramer (Oktober 1994)

No. 6 The use of a European forest model in North America: A study of ecosystem response to climate gradients
H. Bugmann, A. Solomon (Mai 1995)

No. 7 A comparison of forest gap models: Model structure and behaviour
H. Bugmann, Y. Xiaodong, M. T. Sykes, Ph. Martin, M. Lindner, P. V. Desanker, S. G. Cumming (Mai 1995)

No. 8 Simulating forest dynamics in complex topography using gridded climatic data
H. Bugmann, A. Fischlin (Mai 1995)

No. 9 Application of two forest succession models at sites in Northeast Germany
P. Lasch, M. Lindner (Juni 1995)

No. 10 Application of a forest succession model to a continentality gradient through Central Europe
M. Lindner, P. Lasch, W. Cramer (Juni 1995)

No. 11 Possible Impacts of global warming on tundra and boreal forest ecosystems - Comparison of some biogeochemical models
M. Pöchli, W. Cramer (Juni 1995)

No. 12 Wirkung von Klimaveränderungen auf Waldökossysteme
P. Lasch, M. Lindner (August 1995)

No. 13 MOSES - Modellierung und Simulation ökologischer Systeme - Eine Sprachbeschreibung mit Anwendungsbeispielen
V. Wenzel, M. Küchen, M. Flechsig (Dezember 1995)

No. 14 TOYS - Materials to the Brandenburg biosphere model / GAIA
Part 1 - Simple models of the "Climate + Biosphere" system

No. 15 Änderung von Hochwassercharakteristiken im Zusammenhang mit Klimaänderungen - Stand der Forschung
A. Bronstert (April 1996)

No. 16 Entwicklung eines Instruments zur Unterstützung der klimapolitischen Entscheidungsfindung
M. Leimbach (Maif 1996)

No. 17 Hochwasser in Deutschland unter Aspekten globaler Veränderungen - Bericht über das DFG-Rundgespräch am 9. Oktober 1995 in Potsdam
A. Bronstert (ed.) (Juni 1996)

No. 18 Integrated modelling of hydrology and water quality in mesoscale watersheds
V. Krysanova, D.-I. Müller-Wohlfeil, A. Becker (Juli 1996)

No. 19 Identification of vulnerable subregions in the Elbe drainage basin under global change impact

No. 20 Simulation of soil moisture patterns using a topography-based model at different scales

No. 21 International relations and global climate change

No. 22 Modelling the possible impact of climate change on broad-scale vegetation structure - examples from Northern Europe
W. Cramer (August 1996)
<table>
<thead>
<tr>
<th>No.</th>
<th>Title</th>
<th>Authors</th>
<th>Publication Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>A method to estimate the statistical security for cluster separation</td>
<td>F.-W. Gerstengarbe, P.C. Werner</td>
<td>Oktober 1996</td>
</tr>
<tr>
<td>24</td>
<td>Improving the behaviour of forest gap models along drought gradients</td>
<td>H. Bugmann, W. Cramer</td>
<td>Januar 1997</td>
</tr>
<tr>
<td>25</td>
<td>The development of climate scenarios</td>
<td>P.C. Werner, F.-W. Gerstengarbe</td>
<td>Januar 1997</td>
</tr>
<tr>
<td>28</td>
<td>Implementing carbon mitigation measures in the forestry sector - A review</td>
<td>M. Lindner</td>
<td>Mai 1997</td>
</tr>
<tr>
<td>31</td>
<td>Comparing global models of terrestrial net primary productivity (NPP): Analysis of the seasonal behaviour of NPP, LAI, FPAR along climatic gradients across ecotones</td>
<td>A. Bonneau, J. Kaduk, D. W. Kicklighter, participants of &quot;Potsdam '95&quot;</td>
<td>Oktober 1997</td>
</tr>
<tr>
<td>32</td>
<td>Evaluation of the physiologically-based forest growth model FORSANA</td>
<td>R. Grote, M. Erhard, F. Suckow</td>
<td>November 1997</td>
</tr>
<tr>
<td>34</td>
<td>Simulation of the global bio-geophysical interactions during the Last Glacial Maximum</td>
<td>C. Kubatzki, M. Claussen</td>
<td>Januar 1998</td>
</tr>
<tr>
<td>36</td>
<td>Geocybernetics: Controlling a rather complex dynamical system under uncertainty</td>
<td>H.-J. Schellnhuber, J. Kropp</td>
<td>Februar 1998</td>
</tr>
<tr>
<td>38</td>
<td>Die Berücksichtigung natürlicher Störungen in der Vegetationsdynamik verschiedener Klimagebiete</td>
<td>K. Thonicke</td>
<td>Februar 1998</td>
</tr>
<tr>
<td>39</td>
<td>Decadal Variability of the Thermohaline Ocean Circulation</td>
<td>S. Rahmstorf</td>
<td>März 1998</td>
</tr>
<tr>
<td>42</td>
<td>Reversing Course: Germany’s Response to the Challenge of Transboundary Air Pollution</td>
<td>D. F. Sprinz, A. Wahl</td>
<td>März 1998</td>
</tr>
<tr>
<td>44</td>
<td>Capabilities and Limitations of Physically Based Hydrological Modelling on the Hillslope Scale</td>
<td>A. Bronstert</td>
<td>April 1998</td>
</tr>
<tr>
<td>46</td>
<td>Wirkung von Klimaveränderungen in mitteleuropäischen Wirtschaftswäldern</td>
<td>M. Lindner</td>
<td>Juli 1998</td>
</tr>
<tr>
<td>47</td>
<td>SPRINT-S: A Parallelization Tool for Experiments with Simulation Models</td>
<td>M. Flechsig</td>
<td>Juli 1998</td>
</tr>
</tbody>
</table>
A. Bronstert, A. Ghazi, J. Hladny, Z. Kundzewicz, L. Menzel (eds.) (September 1998)

No. 49 Struktur, Aufbau und statistische Programmbibliothek der meteorologischen Datenbank am Potsdam-Institut für Klimafolgenforschung
H. Österle, J. Glauser, M. Denhard (Januar 1999)

No. 50 The complete non-hierarchical cluster analysis
F.-W. Gerstengarbe, P. C. Werner (Januar 1999)

No. 51 Struktur der Amplitudengleichung des Klimas
A. Hauschild (April 1999)

No. 52 Measuring the Effectiveness of International Environmental Regimes
C. Helm, D. F. Sprinz (Mai 1999)

No. 53 Untersuchung der Auswirkungen erhöhter atmosphärischer CO₂-Konzentrationen innerhalb des Free-Air Carbon Dioxide Enrichment-Experimentes: Ableitung allgemeiner Modelllösungen

No. 54 Flächenhafte Modellierung der Evapotranspiration mit TRAIN
L. Menzel (August 1999)

No. 55 Dry atmosphere asymptotics
N. Botta, R. Klein, A. Almgren (September 1999)

No. 56 Wachstum von Kiefern-Ökosystemen in Abhängigkeit von Klima und Stoffeintrag - Eine regionale Fallstudie auf Landschaftsebene
M. Erhard (Dezember 1999)

No. 57 Response of a River Catchment to Climatic Change: Application of Expanded Downscaling to Northern Germany
D.-I. Müller-Wohlfeil, G. Bürger, W. Lahmer (Januar 2000)

No. 58 Der "Index of Sustainable Economic Welfare" und die Neuen Bundesländer in der Übergangsphase
V. Wenzel, N. Herrmann (Februar 2000)

No. 59 Weather Impacts on Natural, Social and Economic Systems (WISE, ENV4-CT97-0448) German report

No. 60 The Need for De-Aliasing in a Chebyshev Pseudo-Spectral Method
M. Uhlmann (Juni 2000)

No. 61 National and Regional Climate Change Impact Assessments in the Forestry Sector - Workshop Summary and Abstracts of Oral and Poster Presentations
M. Lindner (ed.) (Juli 2000)

No. 62 Bewertung ausgewählter Waldfunktionen unter Klimaänderung in Brandenburg
A. Wenzel (August 2000)

No. 63 Eine Methode zur Validierung von Klimamodellen für die Klimawirkungsforschung hinsichtlich der Wiedergabe extremen Ereignisse
U. Böhm (September 2000)

No. 64 Die Wirkung von erhöhten atmosphärischen CO₂-Konzentrationen auf die Transpiration eines Weizenbestandes unter Berücksichtigung von Wasser- und Stickstofflimitierung
S. Grossman-Clarke (September 2000)

No. 65 European Conference on Advances in Flood Research, Proceedings, (Vol. 1 - Vol. 2)
A. Bronstert, Ch. Bismuth, L. Menzel (eds.) (November 2000)

No. 66 The Rising Tide of Green Unilateralism in World Trade Law - Options for Reconciling the Emerging North-South Conflict
F. Biermann (Dezember 2000)

No. 67 Coupling Distributed Fortran Applications Using C++ Wrappers and the CORBA Sequence Type
T. Sławig (Dezember 2000)

No. 68 A Parallel Algorithm for the Discrete Orthogonal Wavelet Transform
M. Uhlmann (Dezember 2000)

No. 69 SWIM (Soil and Water Integrated Model), User Manual
V. Krysanova, F. Wechsung, J. Arnold, R. Srinivasan, J. Williams (Dezember 2000)

No. 70 Stakeholder Successes in Global Environmental Management, Report of Workshop, Potsdam, 8 December 2000
M. Welp (ed.) (April 2001)
No. 71 GIS-gestützte Analyse globaler Muster anthropogener Waldschädigung - Eine sektorale Anwendung des Syndromkonzepts
M. Cassel-Gintz (Juni 2001)

No. 72 Wavelets Based on Legendre Polynomials
J. Fröhlich, M. Uhlmann (Juli 2001)

No. 73 Der Einfluß der Landnutzung auf Verdunstung und Grundwasserneubildung - Modellierungen und Folgerungen für das Einzugsgebiet des Glan
D. Reichert (Juli 2001)

No. 74 Weltumweltpolitik - Global Change als Herausforderung für die deutsche Politikwissenschaft
F. Biermann, K. Dingwerth (Dezember 2001)

No. 75 Angewandte Statistik - PIK-Weiterbildungsseminar 2000/2001
F.-W. Gerstengarbe (Hrsg.) (März 2002)

No. 76 Zur Klimatologie der Station Jena
B. Orlowsky (September 2002)

No. 77 Large-Scale Hydrological Modelling in the Semi-Arid North-East of Brazil
A. Güntner (September 2002)

No. 78 Phenology in Germany in the 20th Century: Methods, Analyses and Models
J. Schaber (November 2002)

No. 79 Modelling of Global Vegetation Diversity Pattern
I. Venevskaja, S. Venevsky (Dezember 2002)

No. 80 Proceedings of the 2001 Berlin Conference on the Human Dimensions of Global Environmental Change “Global Environmental Change and the Nation State”
F. Biermann, R. Brohm, K. Dingwerth (eds.) (Dezember 2002)

No. 81 POTSDAM - A Set of Atmosphere Statistical-Dynamical Models: Theoretical Background
V. Petoukhov, A. Ganopolski, M. Claussen (März 2003)

No. 82 Simulation der Siedlungsflächenentwicklung als Teil des Globalen Wandels und ihr Einfluß auf den Wasserhaushalt im Großraum Berlin
B. Ströbl, V. Wenzel, B. Pfützner (April 2003)

No. 83 Studie zur klimatischen Entwicklung im Land Brandenburg bis 2055 und deren Auswirkungen auf den Wasserhaushalt, die Forst- und Landwirtschaft sowie die Ableitung erster Perspektiven

No. 84 Well Balanced Finite Volume Methods for Nearly Hydrostatic Flows
N. Botta, R. Klein, S. Langenberg, S. Lützenkirchen (August 2003)

No. 85 Orts- und zeitdiskrete Ermittlung der Sickerwassermenge im Land Brandenburg auf der Basis flächendeckender Wasserhaushaltsberechnungen
W. Lahmer, B. Pfützner (September 2003)

No. 86 A Note on Domains of Discourse - Logical Know-How for Integrated Environmental Modelling, Version of October 15, 2003
C. C. Jaeger (Oktober 2003)

No. 87 Hochwasserrisiko im mittleren Neckarraum - Charakterisierung unter Berücksichtigung regionaler Klimaszenarien sowie dessen Wahrnehmung durch befragte Anwohner
M. Wolff (Dezember 2003)

No. 88 Abflußentwicklung in Teileinzugsgebieten des Rheins - Simulationen für den Ist-Zustand und für Klimaszenarien
D. Schwand (April 2004)

No. 89 Regionale Integrierte Modellierung der Auswirkungen von Klimaänderungen am Beispiel des semi-ariden Nordostens von Brasilien
A. Jaeger (April 2004)

No. 90 Lebensstile und globaler Energieverbrauch - Analyse und Strategieansätze zu einer nachhaltigen Energiestruktur
F. Reusswig, K. Gerlinger, O. Edenhofer (Juli 2004)

No. 91 Conceptual Frameworks of Adaptation to Climate Change and their Applicability to Human Health

No. 92 Double Impact - The Climate Blockbuster ‘The Day After Tomorrow’ and its Impact on the German Cinema Public
F. Reusswig, J. Schwarzkopf, P. Polenz (Oktober 2004)

No. 93 How Much Warming are we Committed to and How Much Can be Avoided?
B. Hare, M. Meinshausen (Oktober 2004)
No. 94 Urbanised Territories as a Specific Component of the Global Carbon Cycle
A. Svirejeva-Hopkins, H.-J. Schellnhuber (Januar 2005)

No. 95 GLOWA-Elbe I - Integrierte Analyse der Auswirkungen des globalen Wandels auf Wasser, Umwelt und Gesellschaft im Elbegebiet
F. Wechsung, A. Becker, P. Gräfe (Hrsg.) (April 2005)

No. 96 The Time Scales of the Climate-Economy Feedback and the Climatic Cost of Growth
S. Hallegatte (April 2005)

No. 97 A New Projection Method for the Zero Froude Number Shallow Water Equations
S. Vater (Juni 2005)

No. 98 Table of EMICs - Earth System Models of Intermediate Complexity
M. Claussen (ed.) (Juli 2005)

No. 99 KLARA - Klimawandel - Auswirkungen, Risiken, Anpassung
M. Stock (Hrsg.) (Juli 2005)

No. 100 Katalog der Großwetterlagen Europas (1881-2004) nach Paul Hess und Helmut Brezowsky
6., verbesserte und ergänzte Auflage
F.-W. Gerstengarbe, P. C. Werner (September 2005)

No. 101 An Asymptotic, Nonlinear Model for Anisotropic, Large-Scale Flows in the Tropics
S. Dolaptchiev (September 2005)

No. 102 A Long-Term Model of the German Economy: lagom\textsubscript{d, sim}
C. C. Jaeger (Oktober 2005)

No. 103 Structuring Distributed Relation-Based Computations with SCDRC
N. Botta, C. Ionescu, C. Linstead, R. Klein (Oktober 2006)