Software components for relation-based distributed computations

N. Botta, C. Ionescu, with contributions from J. Gerlach, R. Klein

1. Two examples
2. Relation-based algorithms
3. Data structures for generic relation-based algorithms
4. Data structures for distributed relation-based algorithms
5. Tentative conclusions

Example 1: sparse matrix / vector multiplication

CRS format: $a[pos[i]+k] = A_{i,col[pos[i]+k]}, \quad 0 \; \leq \; k \; < \; pos[i+1] - pos[i]$

$$
\begin{array}{ccccccc}
 & col[pos[i]] & & col[pos[i]+1] & & col[pos[i+1]-1] & \\
\end{array}
$$

$$
\left(
\begin{array}{ccccccc}
 & \cdot & & \cdot & & \cdot & \\
 & \cdot & & \cdot & & \cdot & \\
 & \cdot & & \cdot & & \cdot & \\
\cdots & a[pos[i]] & \cdots & a[pos[i]+1] & \cdots & a[pos[i+1]-1] & \cdots \\
 & \cdot & & \cdot & & \cdot & \\
 & \cdot & & \cdot & & \cdot & \\
 & \cdot & & \cdot & & \cdot & \\
\end{array}
\right)
$$

Implementation:

```
for(int i = 0; i < m; i++) {
  w[i] = 0.0;
  for(int j = pos[i]; j < pos[i + 1]; j++)
    w[i] += a[j] * v[col[j]];
}
```

Algorithm:

for i in $[0, m)$ do:

$$J = [\; j \mid j \leftarrow [0, n),\; A_{i,j} \neq 0 \;]$$

$$X = [\; A_{i,j}\; v_j \mid j \leftarrow J \;]$$

$$w_i = \mathrm{sum}(X)$$

Example 2: grid cell area computation

Implementation:

```
for(int i = 0; i < n_cells; i++)
  a[i] = area(coord[vt[i][0]],
              coord[vt[i][1]],
              coord[vt[i][2]]);
}


double area(const double a[3],
            const double b[3],
            const double c[3]) {

  double ab[3], ac[3], tmp[3];
  for(int i = 0; i < 3; i++)
    ab[i] = b[i] - a[i], ac[i] = c[i] - a[i];
  tmp[0] = ab[1] * ac[2] - ac[1] * ab[2];
  tmp[1] = ab[2] * ac[0] - ac[2] * ab[0];
  tmp[2] = ab[0] * ac[1] - ac[0] * ab[1];
  return 0.5 * sqrt(tmp[0]*tmp[0] + tmp[1]*tmp[1] + tmp[2]*tmp[2]);
}
```

Algorithm:

for c in C do:

$$V = \text{list of vertexes of } c$$

$$X = [\, \text{coord}(v) \mid v \leftarrow V \,]$$

$$a_c = \text{area}(X)$$

Remarks

- The implementation is overly specified w.r.t the implementation: it cannot be used to compute a solution in every context where the algorithm applies.
- We distinguish two kinds of over-specification:
  - Over-specification of the data structures and . . .

Over-specification of the data structures: example 1

Implementation:

```
for(int i = 0; i < m; i++) {
  w[i] = 0.0;
  for(int j = pos[i]; j < pos[i + 1]; j++)
    w[i] += a[j] * v[col[j]];
}
```

Algorithm:

for i in $[0, m-1]$ do:

$$J = [\, j \mid j \leftarrow [0, n-1],\ A_{i,j} \neq 0\,]$$

$$X = [\, A_{i,j}\, v_j \mid j \leftarrow J\,]$$

$$w_i = \mathrm{sum}(X)$$

Data structures over-specifications:

- A, J and w are represented by means of arrays.
- 0.0 is the neutral element of $+$.

Over-specification of the data structures: example 2

Implementation:

```
for(int i = 0; i < n_cells; i++)
  a[i] = area(coord[vt[i][0]],
              coord[vt[i][1]],
              coord[vt[i][2]]);
}


double area(const double a[3],
            const double b[3],
            const double c[3]) {

  double ab[3], ac[3], tmp[3];
  for(int i = 0; i < 3; i++)
    ab[i] = b[i] - a[i], ac[i] = c[i] - a[i];
  tmp[0] = ab[1] * ac[2] - ac[1] * ab[2];
  tmp[1] = ab[2] * ac[0] - ac[2] * ab[0];
  tmp[2] = ab[0] * ac[1] - ac[0] * ab[1];
  return 0.5 * sqrt(tmp[0]*tmp[0] + tmp[1]*tmp[1] + tmp[2]*tmp[2]);
}
```

Algorithm:

for c in C do:

$$V = \text{list of vertexes of } c$$

$$X = [\,\text{coord}(v) \mid v \leftarrow V\,]$$

$$a_c = \text{area}(X)$$

Data structures over-specifications:

- V, X and a are represented by means of arrays, X, in particular, cannot be provided by (inlined) functions.

Remarks

- The implementation is overly specified w.r.t the implementation: it cannot be used to compute a solution in every context where the algorithm applies.
- We distinguish two kinds of over-specification:
  - Over-specification of the data structures and ...
  - Over-specification of the context of application.

Over-specification of the context of application: example 2

Implementation:

```
for(int i = 0; i < n_cells; i++)
  a[i] = area(coord[vt[i][0]],
              coord[vt[i][1]],
              coord[vt[i][2]]);
}


double area(const double a[3],
            const double b[3],
            const double c[3]) {

  double ab[3], ac[3], tmp[3];
  for(int i = 0; i < 3; i++)
    ab[i] = b[i] - a[i], ac[i] = c[i] - a[i];
  tmp[0] = ab[1] * ac[2] - ac[1] * ab[2];
  tmp[1] = ab[2] * ac[0] - ac[2] * ab[0];
  tmp[2] = ab[0] * ac[1] - ac[0] * ab[1];
  return 0.5 * sqrt(tmp[0]*tmp[0] + tmp[1]*tmp[1] + tmp[2]*tmp[2]);
}
```

Algorithm:

for c in C do:

$$V = \text{list of vertexes of } c$$

$$X = [\,\text{coord}(v) \mid v \leftarrow V\,]$$

$$a_c = \text{area}(X)$$

Application context over-specifications:

- The cells are *plane* triangles in a *three-dimensional* space.
- The coordinates of the vertexes are given in a *Cartesian* coordinate system.

Remarks

- The implementation is overly specified w.r.t the implementation: it cannot be used to compute a solution in every context where the algorithm applies.
- We distinguish two kinds of over-specification:
  - Over-specification of the data structures and . . .
  - Over-specification of the context of application.
  - STL, generic programming, “type class” techniques can be used to avoid overly specified data structures.
  - Dealing with over-specifications of the context of application has turned out to be more difficult. Possible way out: spell out the context of application of the algorithm and define data structures representing that context, e.g., *grids*.
- Deriving implementations of algorithm 1 and 2 for distributed computations is not trivial.
- In spite of the different implementations, algorithm 1 and 2 share a *common computational pattern*.

Relation-based algorithms

Example 1:

for i in $[0, m-1]$ do:

$$J = [\, j \mid j \leftarrow [0, n-1],\ A_{i,j} \neq 0\,]$$

$$X = [\, A_{i,j}\, v_j \mid j \leftarrow J\,]$$

$$w_i = \mathrm{sum}(X)$$

Example 2:

for c in C do:

$$V = \text{list of vertexes of } c$$

$$X = [\, \mathrm{coord}(v) \mid v \leftarrow V\,]$$

$$a_c = \mathrm{area}(X)$$

Relation-based algorithm:

for y in ys do:

$$xs = [\, x \mid xRy\,]$$

$$fs = [\, f(x, y) \mid x \leftarrow xs\,]$$

$$res_y = h(fs)$$

Example 1:

- $xRy \equiv x$ is the index of a column s.t. $A_{y,x} \neq 0$
- $f(x, y) = A_{y,x}\, v_x$
- $h = \mathrm{sum}$

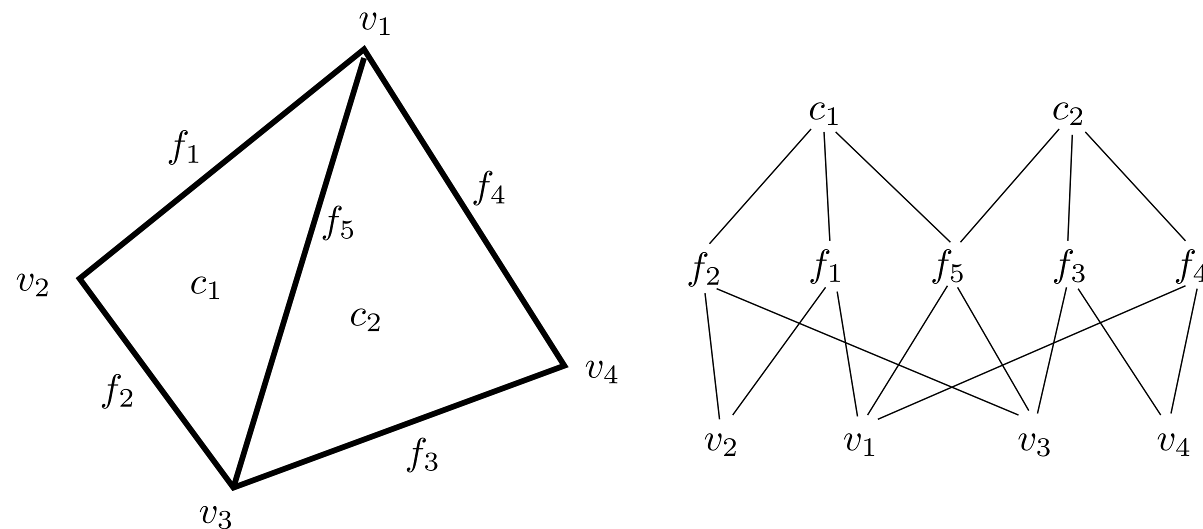
Example 2:

- $xRy \equiv x$ is a vertex of y
- $f(x, y) = \mathrm{coord}(x)$
- $h = \mathrm{area}$

Remarks

- We can express example 1 and example 2 as relation-based algorithms.
- Relation based algorithms can be used to express *grid algorithms* commonly used, among others, in numerical methods for PDE, e.g., FE and FV methods.
- In these methods, grids represent both *geometrical* and *combinatorial* properties of discrete coverings of computational domains.
- Combinatorial properties are accounted for through relations between grid elements.

Grid relations



Grid relations commonly used in numerical methods for PDE:

- $\leq$, where $e \leq e' \;\equiv\; e \in \overline{e'}$ (e is on the boundary of e')
- $\gtreqless$, where $e \gtreqless e' \;\equiv\; e \leq e' \;\vee\; e' \leq e$ (incidence)
- (d, d'), where $e(d, d')e' \;\equiv \dim(e) = d \;\wedge\; e \gtreqless e' \wedge \dim(e') = d'$
- (d, d', d), where $e(d, d', d)e' \;\equiv \exists e'' : e(d, d')e'' \;\wedge\; e''(d', d)e'$ (adjacency)

Relation-based algorithm:

for y in ys do:

$$xs = [\, x \mid xRy \,]$$

$$fs = [\, f(x, y) \mid x \leftarrow xs \,]$$

$$res_y = h(fs)$$

Which data structures do we need to implement *generic*, *distributed* relation-based algorithms ?

Data structures for x and y

- The types of x and y depend on the application. We can account for this by means of:
  - Parameterization (of the data structures representing relation-based algorithms)
  - Abstraction
- In many applications, f is going to be efficiently represented by *arrays*. In such cases, x and y are integers. Therefore parameterization is not adequate !
- In all applications we are interested in, sets, relations and functions are *finite*. Therefore *zero-based integer intervals* can be used as abstraction for representing x and y.

Relation-based algorithm:

for j in $[0, n)$ do:

$is = [\, i \mid iRj \,]$

$fs = [\, f(i, j) \mid i \leftarrow is \,]$

$res_y = h(fs)$

Data structures for relations

- Relations can be represented as:
  - Characteristic functions: computation of is in $O(|\operatorname{target}(R)|)$.
  - Sets / lists / arrays of pairs: computation of is in $O(|R|)$, inefficient storage.
  - Set- / list- / array-valued functions, possibly represented by arrays: computation of is in $O(1)$.
- Data structures for relations provide a method R to deliver a representation $R(j)$ of the image of a source element j in $O(1)$.
- $R(j)$ contains no duplicates.

Relation-based algorithm:

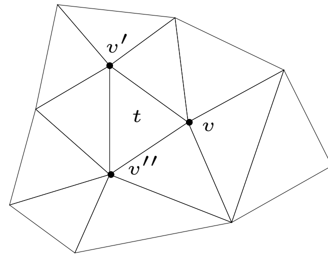
for j in $[0, n)$ do:

$is = R(j)$

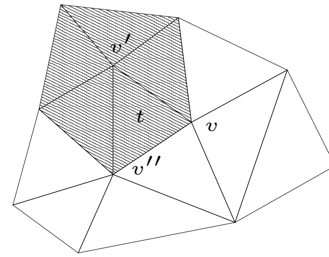
$fs = [\, f(i,j) \mid i \leftarrow is \,]$

$res_y = h(fs)$

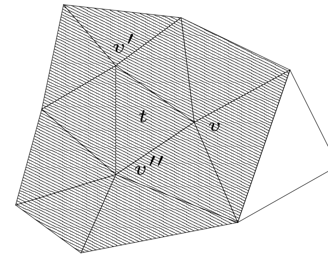
Data structures for relations



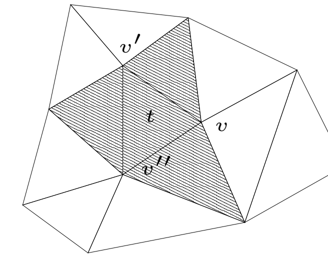
$(0,2)(t)$



$(0,2)^\circ(v')$



$(2,0,2)(t)$



$(2,1,2)(t)$

$$
\begin{aligned}
(2,0,2) &= (2,0) \cdot (0,2) \\
&= (0,2)^\circ \cdot (0,2)
\end{aligned}
$$

$$(2,1,2)(t) = [\, t' \mid |(0,2)(t) \cap (0,2)(t')| = 2 \,]$$

- Data structures for relations have to support, beside relation-based algorithms, algorithms for *composing*, *conversing*, *intersecting* relations.

Data structures for relations

Relation-based algorithm:

for j in $[0, n)$ do:

$is = R(j)$

$fs = [\ f(i, j) \mid i \leftarrow is\ ]$

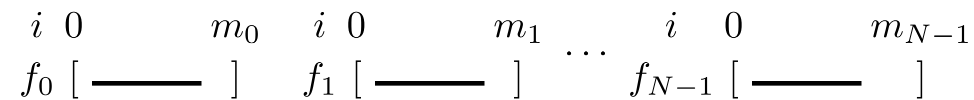
$res_y = h(fs)$

- $is = R(j)$ is only used by the “draw” operator $\leftarrow$. The only operation to be done on $R(j)$ is to access, one by one, its elements. This suggests that R should provide *forward iterator* functionalities .
- Efficient implementations of relation-based algorithms have to avoid temporary data:
  for j in $[0, n)$ do: $res_y = h(\ \mathrm{map}\ f\ [\ (i, j) \mid i \leftarrow R(j)\ ]\ )$
- h is a user-defined function and we have again two ways to account for its type variability:
  - Parameterization
  - Abstraction
- Parameterization is not viable (no support for “structural” genericity, in particular, genericity w.r.t. the number of arguments taken by h) and we are left with abstraction. This suggests that the type of $\mathrm{map}\ f\ [\ (i, j) \mid i \leftarrow R(j)\ ]$ should be itself an iterator.

Data structures for distributed relation-based algorithms

- In many applications, R and f are represented as arrays.
- Arrays can be distributed easily.

is split into



We say that $f[i]$ is ”stored on partition p” iff

$$m_0 + m_1 + \cdots + m_{p-1} \leq i \quad \wedge$$

$$i < m_0 + m_1 + \cdots + m_p \quad \wedge$$

$$f_p[i - (m_0 + m_1 + \cdots + m_{p-1})] = f[i]$$

A simplified relation-based algorithm:

for j in $[0, n)$ do:

$$is = R[j]$$

$$fs = [\ f[i] \mid i \leftarrow is\ ]$$

$$res[j] = h(fs)$$

$$\begin{array}{ll} & 0 \qquad\qquad m_0 \qquad\qquad m_0 + m_1 \\ f & [\ \text{———} \mid \text{———}\ ] \end{array}$$

$$\begin{array}{ll} & 0 \qquad\qquad n_0 \qquad\qquad n_0 + n_1 \\ R & [\ \text{———} \mid \text{———}\ ] \end{array}$$

$$\begin{array}{ll} & 0 \qquad\qquad n_0 \qquad\qquad n_0 + n_1 \\ res & [\ \text{———} \mid \text{———}\ ] \end{array}$$

A simplified relation-based algorithm:

for j in $[0, n)$ do:

$$is = R[j]$$

$$fs = [\ f[i] \mid i \leftarrow is\ ]$$

$$res[j] = h(fs)$$

On partition 0:

for j in $[0, n_0)$ do:

$$is_0 = R_0[j]$$

$$fs_0 = [\ f_0[i] \mid i \leftarrow is_0\ ]$$

$$res_0[j] = h(fs_0)$$

On partition 1:

for j in $[0, n_1)$ do:

$$is_1 = R_1[j]$$

$$fs_1 = [\ f_1[i] \mid i \leftarrow is_1\ ]$$

$$res_1[j] = h(fs_1)$$

A simplified relation-based algorithm:

for j in $[0, n)$ do:

$is = R[j]$

$fs = [\ f[i] \mid i \leftarrow is\ ]$

$res[j] = h(fs)$

On partition p:

for j in $[0, n_p)$ do:

$is_p = R_p[j]$

$fs_p = [\ f_p[i] \mid i \leftarrow is_p\ ]$

$res_p[j] = h(fs_p)$

Remarks on notation and semantics

On partition p:
for j in $[0, n_p)$ do:

$$is_p = R_p[j]$$
$$fs_p = [\ f_p[i] \mid i \leftarrow is_p\ ]$$
$$res_p[j] = h(fs_p)$$

- We index the distributed variables (but not the counters or the global parameters) with the partition number. A variable x_p is going to have a ”counterpart” on every partition, therefore its semantics is going to be that of a tuple $(x_0, x_1, \cdots, x_{N-1})$
- If the above code fragment is thought of as a function ”parameterized” on p, i.e.

  $$prog_p : A \leftarrow B$$

  then the distributed computations represents the function

  $$prog : A^N \leftarrow B^N$$

  which is the product of the ”local computations”

  $$prog = prog_0 \times prog_1 \times \cdots \times prog_{N-1}$$

Problems in distributed relation-based computations

$$fs_p = [\ f_p[i] \mid i \leftarrow is_p\ ]$$

- i is in $[0, m)$ but f_p is defined on $[0, m_p)$! Therefore, we must
- if $m_0 + m_1 + \cdots + m_{p-1} \leq i < m_0 + m_1 + \cdots + m_p$ re-scale i to $i - (m_0 + m_1 + \cdots + m_{p-1})$
- otherwise retrieve $f_{p'}[i'] = f[i]$ from remote partition p' where

$$m_0 + m_1 + \cdots + m_{p'-1} \leq i < m_0 + m_1 + \cdots + m_{p'}$$
$$i' = i - (m_0 + m_1 + \cdots + m_{p'-1})$$

$\Longrightarrow$ there must be a *send* from p', synchronizations, etc.

In order to send and receive the needed values of f, we have to compute the following lists:

$$\begin{aligned} outs_p &= [\,(i, p') \mid i \leftarrow \mathrm{ran}(R_p) \;\wedge\; f[i] \text{ stored on } p' \neq p\,] \\ ins_p &= [\,(i, p') \mid i \leftarrow \mathrm{ran}(R_{p'}) \;\wedge\; f[i] \text{ stored on } p \neq p'\,] \end{aligned}$$

- $outs_p$ contains the indexes of values of f needed to apply the algorithm on partition p but stored on other partitions.
- ins_p contains the indexes of values that are stored on partition p, but which are needed to apply the algorithm on other partitions.
- Only $outs_p$ can be computed on partition p on the basis of available information, and this only if $m_0, \cdots, m_{N-1}$ are *locally* known.
- Since the partitioning of f depends only on the partitioning of the interval $[0, m)$ (and not on, e.g., values taken by f on this interval), it follows that $outs_p$ and ins_p can be used for *any* array representing a function defined on this interval.

We can compute $outs_p$ locally, and we can obtain ins_p if we have a communication primitive such as the following:

$$\mathrm{exch} : (List(A \times [0, N)))^N \leftarrow (List(A \times [0, N)))^N$$

$$\mathrm{exch}(xs_0, xs_1, \cdots, xs_{N-1}) = (ys_0, ys_1, \cdots, ys_{N-1})$$

where $ys_p = [\ (a, p') \mid p' \leftarrow [0, N),\ (a, p)$ is in $xs_{p'}\ ]$

Therefore

$$ins_p = \mathrm{exch}(outs_p)$$

Here, we have used the convention that variables indexed with a partition number represent tuples.

Having obtained ins_p, we can now compute

$$fins_p = [\ (f[i], p') \mid (i, p') \leftarrow ins_p\ ]$$

In order to apply the algorithm, however, we need

$$fouts_p = [\ (f[i], p') \mid (i, p') \leftarrow outs_p\ ]$$

It can now be shown that

$$fouts_p = \mathrm{exch}(fins_p)$$

The proof makes use of the following property of exch:

$$\mathrm{exch} \cdot \mathrm{exch} = \mathrm{id}$$

Remark: we need $outs_p$ in order to retrieve the values $f[i]$ stored in $fouts_p$.

Data structures for communication

- ins_p and $outs_p$ have, in many applications, a longer ”expected lifetime” than f.
- It is useful to encapsulate this type of *communication structures* in special data structures. These are the main interface to message-passing libraries, therefore are constrained by the requirements of these libraries.
- Such data structures also represent relations, associating to a partition p none, one or several elements of the form (x, p'). It is natural to want them to reflect this fact by implementing a relation interface. This will also facilitate their usage in other algorithms.

Data structures for efficient execution of the algorithms

- Looking up the indexes of $outs_p$ every-time we need to access values of f stored in $fouts_p$ is very inefficient. At the very least, we need a data structure that allows an efficient usage of the information retrieved, e.g. a hash table.
- Even so, checking whether an index i corresponds to a value of f stored locally or globally is time-consuming. In most cases, we want to avoid this by operating on an array (or array-like) structure f'_p derived from f_p and $fouts_p$. This could be achieved by a (relatively complex) scaling function, i.e.

$$fs_p = [\ f'_p[\text{scale}(i)] \mid i \leftarrow is\ ]$$

- Alternatively, we could work with a modified representation of R_p:

$$i\ R_p\ j \equiv\ \text{scale}(i)\ R'_p\ j$$

  Therefore, the elements in

$$is_p = R'_p[j]$$

  would already be scaled. Again, the modified R'_p is independent of the actual values of f. This alternative is, in our applications, the more efficient one.

- Software components for distributed relation-based computations (in progress)
  - communication-oriented
    * exch, split
  - relation operators
    * $\mathrm{compose}$, $\mathrm{converse}$, $\mathrm{intersect}$
  - distribution-related
    * interface to graph partitioning libraries (e.g. Metis)
    * distributing a relation
    * complete a local array
  - relation-based algorithms
- Software components for distributed adaptive FV methods (to do)
  - grids as sets of distributed relations (combinatorial properties) and arrays (geometry)
    * grid algorithms as relation-based algorithms
    * refinement / coarsening algorithms
  - FV methods
    * recovery algorithms
    * quadrature rules