

Stream processing on the grid: an Array Stream Transforming Language

Alex Shafarenko

Department of Computer Science, University of Hertfordshire, AL10 9AB, U.K.

<http://homepages.feis.herts.ac.uk/~ctca>

Abstract

Specific requirements of stream processing on the Grid are discussed. We argue that when the stream processing paradigm is used for cluster computing, the processing components can be coded in the form of data-parallel recurrence relations with stream synchronization and filtering at the interfaces. We propose a programming language ASTL in which such components can be written and describe some of its key features. Our approach enables distributed type inference which guarantees correct typing of the whole distributed application. Finally we discuss the stream network architecture and identify issues requiring Grid support.

1. Introduction

Stream computing has been largely confined to the area of signal processing and multimedia content delivery in recent years. Indeed in both areas, there is a need to address static, layered processing schemes whereby data is originated at hardware sources, then put through a processing network, often with real-time constraints, and finally the results are streamed back to the hardware for single- or multimedia output. Thus we hear of video and audio stream processing, signal stream processing, etc. but hardly anything of generic stream processing for distributed applications.

The stream processing paradigm, however, potentially has a broader appeal. Its key defining factor is the temporal regularity of communication, which makes it possible to conceptualize series of data exchanges between procedures as a set of streams of data records flowing between pairs of independent code components. The processing within the components is constrained by this regularity to a fixed computation applied to a current, and a constant number of preceding records every time a new record is received. Such schemes generally both consume records and compute new ones from them, which corresponds to reading from synchronized input streams and wri-

ting to synchronised output streams. It is also a distinguishing feature of stream processing that the relationship between procedures and streams is conceptualized in the form of a persistent communication graph, whose edges are associated with streams and the vertices with stream-transforming procedures, or *stream transformers* for short. The graph *can* change over the runtime of an application, but such changes are infrequent compared to the frequency of stream I/O, so the streams are sufficiently well-defined in the time domain.

In an open, collaborative environment (for example, a Computational Grid) components are deployed on network hosts both statically and dynamically. The advancement of the distributed state has two different time scales: the “fast” time of data streams, and the “slow” time of component evolution. For instance, a pipelined arrangement of search, processing and visualization components could be pumping large amounts of experimental data through its connecting streams and, as a result of computational steering [1], the processing or visualization components could be adjusted or replaced from time to time by a new version or an entirely new algorithm without destroying the data flow. The regularity of stream communication is still present here but it is not as absolute as it is in signal processing schemes.

Thus two almost opposite requirements co-exist in general-purpose stream processing in open systems: the requirement of high efficiency of rigid computational schemes of simple structure, and the requirement of high flexibility of such schemes when they are reconfigured, which happens from time to time.

These conflicting requirements place new demands on the programming method, namely:

1. **separation of communication and computation concerns.** Indeed, as stream transformers participate in a nearly static communication arrangement, the input/output concerns need not be mixed up with the computation that they implement. By keeping the communication separate from the computation, component standardization is facilitated: similarly communicating components that encapsulate different algorithms can be used in a distributed system

interchangeably. The interface between a transformer and a network can be implemented as a universal *wrapper* put around a generic processing program.

2. regularity of computation. Since temporal regularity is a feature of both input and output of a transformer, its computation also tends to be regular. Transformers produce an output functionally dependent on a (small) fixed number of most recent records read from the input. A transformer can have internal streams originating at it, which can be produced alongside the input streams. The former streams encapsulate the transformer state, so temporal locality does not necessarily limit the depth of “history”: thanks to the internal streams, the output records are potentially dependent on *all* input records previously read.

The stream processing paradigm can thus represent iterative numerical methods in computational science as well as signal and image processing algorithms, database and data-mining applications, etc. This is due to the fact that any iterative loop can be conceptualized as a stream transformer that reads the record consisting of the current values of the loop variables and which then produces a new such record corresponding to the new values of these variables after the loop iteration. Such a transformer has part of its output short-circuited to the input. Of course, if *all* the output is fed back, the connection graph does not support much concurrency and hardly any distribution, but the transformer can be replicated and the replicas connected in a larger cycle (which corresponds to loop unrolling in conventional programming). There can be more than one such “loop” so the connection graph could be quite complex. Such schemes naturally appear in data processing tasks, where they simply express the nature and staging of the underlying processing algorithms (see, for instance, the Digital Sky project [10], where figures 10.3 and 10.4 display the structure and connectivity of a large-scale processing network). Despite the lack of tools and reported implementations, there is no reason why a similar approach cannot be used in large-scale, distributed computer modeling and simulation.

3. survival in a changing environment, where streams are reconfigured from time to time. We argue that this demand is best served by employing a type system that can support distributed polymorphic type inference so that, on the one hand, stream genericity can be maintained and, on the other hand, the components themselves can specialize all internal types based on external type information communicated during reconfiguration events. This approach also cuts down on the assortment of stream-manipulation primitives that split, merge and otherwise re-package data streams without computing new values.

The above considerations have motivated us to propose a stream-processing language for computational grids. The language is called ASTL, which stands for “Array Stream-

Transforming Language”, and which also recognizes the fact that numerical data communicated between encapsulated algorithms is often presented in array form. Thus the streams in question are often array-valued and the recurrence relations embodied in stream transformers are data-parallel, array-valued recurrence relations. Data-parallelism of the recurrences comes from the fact that when output arrays are computed from input arrays in a strict order of indices, this corresponds to a hidden recurrence which can be extracted and made explicit using auxiliary streams. Once this is done, the rest of the indices should be free from index order, i.e. the rest of the computation is data-parallel (or scalar). For instance, given a recurrence relation between B (input) and A (output)

$$A_{i,j}^{[0]} = 0; A_{i,j}^{[k+1]} = \begin{cases} 0, & \text{if } i = 0 \\ f(i, A_{i-1,j}^{[k+1]}) + B_{i-1,j}^{[k]}, & \text{otherwise} \end{cases},$$

the computation of the next stage $k+1$ must proceed in the ascending order of the index i . However, the same can be achieved by streaming matrix $B^{[k]}$ row-wise through the following ancillary recurrence

$$V_j^{[0]} = 0; V_j^{[i+1]} = f(i, V_j^{[i]}) + R_j^{[i]},$$

where R_j is the stream of rows of $B^{[k]}$. The latter relation is purely data-parallel. The original relation can therefore be converted to a pipeline $S^{-1} \circ V \circ S$, where S splits matrices into a stream of rows, V is the above recurrence relation and S^{-1} assembles rows back into a matrix. All three components of the pipeline are data-parallel. Note that S must hold its input for N iterations, N being the number of rows in the input matrix, while S^{-1} must not output the result for N iterations, i.e. until the output matrix is fully assembled. These requirements are addressed in ASTL by output and input control, see section 2.4 below. Also note the importance of supporting geometric operations on arrays as well as computational ones.

This example demonstrates the need to address language support for data-parallel recurrence relations as building blocks of a distributed stream-processing system. Popular programming languages make no attempt to provide such support, partly because recurrences are close to declarative programming, where they are a particular case of tail recursion, and partly because they depend on the concept of array-valued streams as first-class objects. A specialized language for programming recurrences seems to be an attractive alternative. We have recently discovered [3] that such a language can be free from type declarations while retaining strong typing. This is due to the fact that an efficient type inference method exists that can recover type information from a distributed set of recurrences in a computationally efficient manner. Since I/O is also decoupled from the computation, the application code looks almost

like a set of mathematical equations, making the language easy to use by computational scientists.

The rest of the paper is organized as follows. In the next section we describe the stream processing mechanism of ASTL in more details. Section 3 introduces the ASTL type system. Section 4 discusses structure and management of stream networks and finally there are some conclusions.

2. Language features

2.1. Arrays and shape

ASTL defines a set of constructs for array manipulation. Unlike most programming languages, it treats all objects as arrays of some rank (i.e. number of dimensions). Objects that have no associated dimensions are called *scalars* and assumed to have the rank 0. The shape of an array is considered to be its dynamic characteristic and is therefore outside the scope of the type system. Conventional binary operations on arrays, such as elementwise addition, multiplication, etc., usually require the operand shapes to be the same. Such a requirement is convenient if conformance can be checked statically in all cases. However, for any comprehensive set of array operations, which include liberal rearrangements of elements and complex selections, array shape quickly becomes untraceable and shape conformance undecidable. ASTL avoids shape-related problems by replacing conformance by the following *intersection rule*: the extent of the result of a binary operation which involves juxtaposition of the operands (such as elementwise addition) is the lesser of the operand extents in each dimension. For example, if A has the shape 100×200 and B 150×70 , the result shape of $A \oplus B$, where \oplus is such an operator, is 100×70 .

2.2. Type and subtyping

ASTL objects have two static attributes: element type t and array rank r . These attributes will be referred to as “types” below. Where this may lead to a confusion, the full term “element type” will be used in contrast with “complete type” which includes both the element type and the rank of the object. The set of element types

$$T = \{bool, char, int, real, complex\} \quad (1)$$

has a subtype relation defined on it:

$$bool \subseteq char \subseteq int \subseteq real \subseteq complex, \quad (2)$$

which is a linear order.

The ranks are taken from a range of integer numbers $R = 0..r_{max}$ with the subtype relation being \leq . Conceptually, rank coercion is achieved by infinite replication of

the object in additional dimensions. For example, integer scalar 3 can be implicitly coerced to rank 1 by forming an infinite one-dimensional array $[3,3,3,\dots]$. In practice, infinite arrays do not remain infinite, since the reason for coercion is to increase the array rank to match it with the rank of another array for a binary operation. In such a case, the other array will determine the shape of the result according to the intersection rule. For instance, if the above array $[3,3,3,\dots]$ is added to an existing one-dimensional array X elementwise, the result is a new array X' conforming to X , its elements being equal to the corresponding elements of X plus 3. For example, if $X = [2, 4, 6]$, $X + 3 = [5, 7, 9]$.

An operator with more than one operand is assumed to act on the operand tuple. The subtyping relation on 2-tuples is defined in the standard way:

$$(a_1, a_2) \subseteq (b_1, b_2) \equiv a_1 \subseteq b_1 \wedge a_2 \subseteq b_2, \quad (3)$$

and similarly for 3-tuples, etc. A similar rule could be introduced for subtyping the complete types: the element type and rank must satisfy their subtyping relations simultaneously. However, since all type signatures of ASTL are formulated for element types and ranks separately, the need for subtyping complete types never arises.

The reason for subtyping to be so widely used in ASTL stems from the fact that the array-manipulation primitives must accept a large number of rank and type combinations that have natural meaning and which hence must be allowed; this would make programs difficult to understand if each such combination required a distinct version of a primitive. On the other hand, some primitives are often not so different semantically and can be generalized as overloaded operator families. For example, elementwise multiplication can be overloaded to represent integer, real and complex versions, and also act on a pair of arrays of any rank even when the ranks do not match. Moreover, some specialist operations become redundant as they merely represent obvious combinations of general operations augmented with rank coercions. For instance, matrix dot product $A.B$ can be regarded as a transposition and rank coercion of $A = a_{ij}$ to $a'_{jim} = a_{ij}$ and the transposition and rank coercion of $B = b_{jk}$ to $b'_{jnk} = b_{jk}$, after which the elementwise product of the a' and b' with subsequent summation in the lowest dimension achieve the desired result.

Operator overloading leads to polymorphism of expressions and ultimately of stream transformers which are polymorphic with respect to ranks and types.

2.3. Expressions

ASTL expressions are formed from variables and operators following a set of syntax rules. The operators in an array manipulation language are traditionally subdivided into the following three categories.

1.Elementwise operators. These include all unary and binary arithmetic operations and elementary functions as well as the ternary choice operator **if** ... **then** ... **else** ... **fi**. A nonunary operation implies juxtaposition of array elements with the same multi-index; then the scalar operator is applied for each value of the multi-index in parallel, hence the name 'elementwise'. These operators assume that the array operands have the same rank; if the ranks are not the same, rank coercion is applied to bring all the operands to the highest operand rank. The shape intersection rule is then applied to determine the shape of the result. For all operations, the element types of the operands and the result must satisfy the offset-homomorphism restriction [3]. Informally, this restriction means that the *value* of the result is essentially the same no matter what combination of input types is used (provided that it is legal). For example, when adding two integer numbers 3 and 5 the result is an integer 8, but if the numbers were real 3.0 and 5.0 the result would be 8.0 which is the value of the integer-to-real coercion of 8 as well. The independence of the result value from where the type coercions are introduced, whether at the operand level or after the operation, is the essence of type homomorphism. Offset-homomorphism is a further restriction, which is required for type inference.

2.Reductions. These are unary operations parametrized by a scalar commutative, associative operator. The result rank is equal to the operand rank less one. The scalar operator is "inserted" between every pair of array elements in the lowest dimension and the resulting expressions are then computed in parallel. For example, $+/A$ is a reduction that sums the first dimension of the array A . If $A = a_{ij}$ is a 2d array, i.e. a matrix, then the result is $+/A = \sum_j a_{ij}$.

3.Rearrangements. These operations copy some or all elements of their operands and place them in a possibly different order within the result. ASTL defines a unary rearrangement, called *recast* and a binary one, called *composition*. In the same group we place constructors, i.e. the primitives for creating arrays by defining the element values (*array constructor*), by adding unit-size dimensions to a scalar (*singleton*) or by replicating an existing array along a new axis a finite number of times. We discuss these next.

Recast ASTL offers one general operation for element rearrangement of a single array, called *recast*. It has the following syntax:

$$exp_0[exp_1, \dots, exp_m \leftarrow i_1 \mid b_1, \dots, i_n \mid b_n] ,$$

where the 0-th expression is the source array, expressions from 1 to m are indices for the source array and $i_1 \dots i_n$ are the "input" indices. The bounds b_1, \dots, b_n delimit the shape of the resulting array, and can be omitted if the corresponding indices participate only in linear expressions, since under such conditions the bounds can be calculated

automatically from the source array shape. Also, a placeholder $_$ can be used instead of any $i_k \mid b_k$ to denote that the resulting array has an infinite dimension k along which the element values are identical. For example, if A denotes the following 4×4 array:

$$\begin{pmatrix} 11 & 12 & 13 & 14 \\ 21 & 22 & 23 & 24 \\ 31 & 32 & 33 & 34 \\ 41 & 42 & 43 & 44 \end{pmatrix} ,$$

then

$$A[2 - a, 3 - b \leftarrow a \mid 2, b \mid 2] = \begin{pmatrix} 23 & 22 \\ 13 & 12 \end{pmatrix}$$

and

$$A[i, i \leftarrow _, i] = \begin{pmatrix} 11 & 22 & 33 & 44 \\ 11 & 22 & 33 & 44 \\ \dots & \dots & \dots & \dots \end{pmatrix}$$

If the rank ρ of the source array is less than m then the extra $m - \rho$ indices are ignored (this corresponds to the replication of the argument array in the extra dimensions) and the result rank remains at least¹ n . When $\rho > m$, the extra $\rho - m$ indices are appended to the n input indices on the right, giving the result rank of at least $n + \rho - m$.

Concatenation The operator *recast* defined earlier can use only one array as a source. To be able to use more than one source array, an additional operator is required. Since *recast* supports arbitrary rearrangements, the new operator need not be flexible with the placement of the elements in the result: a simple concatenation of the operand arrays is sufficient. Accordingly, ASTL has a concatenation operator, which has the following syntax:

$$exp_1 \sim k \sim exp_2.$$

Concatenation proceeds along the axis indicated by the integer constant $k \in [1, R_{max}]$ placed in the operator symbol, e.g. $a \sim 2 \sim b$ is the concatenation of the arrays a and b along axis 2. If the operands have the rank k or above, the action of concatenation is straightforward. The extent of the result in the k th dimension is the sum of the operands' extents, while the extents in the rest of the dimensions are governed by the intersection rule. If either operand has a lesser rank, it is coerced to the rank $k - 1$ by replication (provided that it is not rank $k - 1$ already) and then on to the rank k by assuming the extent 1 in the k th dimension. Then concatenation proceeds as before.

For instance, $1 \sim 2 \sim 2$ is a 2d array having the following values:

$$\begin{pmatrix} 1 & 1 & 1 & \dots \\ 2 & 2 & 2 & \dots \end{pmatrix}$$

¹We say 'at least' because a higher rank can be expected by the expression context due to the rank subtyping rule.

Constructors The array constructor is the only interesting primitive of this group. It has the following syntax:

$$\text{array}(i_1 | b_1, \dots, i_n | b_n) \text{exp}$$

where $b_1 \dots b_n$ define the extents of the array and i_1, \dots, i_n are the variables to be used as array indices in the expression exp , which defines the content of the corresponding array element. Any $i_k | b_k$ can be replaced by the placeholder $_$ if the index i_k does not occur in the expression.

For instance, the array

$$\text{array}(x|3, y|4) 10 * x + y = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 10 & 11 & 12 & 13 \\ 20 & 21 & 22 & 23 \end{pmatrix}.$$

Note that, by using the array constructor or the recast operator, arrays can be defined to have indices on which they do not depend. This corresponds to replicating the array in extra dimensions ad infinitum. All binary elementwise operations tolerate such arrays without a problem, since the matching dimension of the other operand, if finite, will effectively slice out a finite portion of the first operand. Or, if both dimensions are replicative, then the result is such also, which means that only one application of the scalar operator is required. The same is true of unary elementwise operators, except here an infinite dimension of the operand always produces an infinite dimension of the result. The problem may arise when finite shape is required, which is the case only with the operation concatenation and only in the dimension of concatenation. Obviously if either operand is infinite in this dimension, a runtime error must result.

Infinite dimensions are a useful tool to address translational symmetry of array objects [4]. We have already mentioned the construction of matrix multiplication from two matrices recast to replicative 3d arrays. More examples can be found in various multidimensional array algorithms.

2.4. Transformer constructs

Having discussed the data-parallel toolkit of ASTL, we are now ready to present the core functionality of the language, i.e. stream transformer constructs. The syntax of the stream transformer is as follows:

$$\text{name} \langle \text{par}_1, \dots, \text{par}_n \rangle \text{out in body},$$

where the *name* is the transformer identifier, the variables in the angular brackets are non-stream formal parameters, *out* is the output interface and *in* the input interface; the *body* contains the recurrent relations associated with the transformer. The input interface is omitted if the transformer has no input streams. The output interface has the

following syntax:

$$(\text{exp}_1, \dots, \text{exp}_n) \text{check exp},$$

with a rank-0 Boolean exp which may depend on stream variables (see below) and which filters tuples $(\text{exp}_1, \dots, \text{exp}_n)$ for the output stream. The output expressions can name any variables including the parameters.

The input interface is similar:

$$(\text{var}_1, \dots, \text{var}_n) \text{hold exp}.$$

Here the expression is also a rank-0 boolean, which controls whether the input tuple $(\text{var}_1, \dots, \text{var}_n)$ is held or updated from the input streams. The **hold** expression can name stream variables as well.

ASTL transformers can be either fixed-shape or variable-shape.

Variable-shape transformer (VST) A variable-shape transformer body contains any number $m \geq 0$ of stream-variable definitions in the following syntax: $\text{var}_i \leftarrow \text{exp}_{0i}, \text{exp}_i$, where $1 \leq i \leq m$, exp_{0i} is the initial value of the stream variable var_i and exp_i is the recurrence step, i.e. an expression that defines the new value of var_i given the current values of all stream variables. The stream variables are introduced by naming them on the left of \leftarrow . They must not clash with any of the variables of the input interface and may occur in the output interface. The transformer produces a stream one stage at a time, by simultaneously evaluating the recurrence steps and associating the results with the corresponding stream variables. For instance, the stream of natural numbers can be programmed as $n \leftarrow 1, n + 1$, the stream of Fibonacci numbers as $m \leftarrow 1, k; k \leftarrow 1, m + k$, with the actual Fibonacci sequence associated with m , etc. Note that both the initial-values and the step expressions can name formal parameters par_j and variables from the input interface. It should be noted that all expressions involved in stream transformer definitions are arbitrary array expressions.

Fixed-shape transformer (FST) A fixed-shape transformer, as the name suggests, defines output streams of fixed-shape array values. Since the shape is fixed, the recurrence step no longer defines the shape of the stream which is now fully determined by the initial values. ASTL exploits this restriction by allowing part of the step value to be undefined, taking the undefined part from the corresponding portion of the previous step. Another new feature of fixed-shape transformers is left-expressions in the stream definition. The left-expression is a well-formed expression containing exclusively concatenation, recast, singleton and choice operators, where the condition part and the recast index expressions are “normal”, right-expressions. The semantics of the left-expression is one

of a *target* for the step. The target can be thought of as a reference object which re-directs the values coming from the right-hand side to the receiving array-elements of the stream variables named in it; the rest of the elements of the stream variables take their default values. For instance, the step definition $a[i, j \leftarrow i|N - 1, j] \sim 2 \sim b := b \sim 2 \sim a$ (assuming that the initial values of a and b are shaped as $N \times N$ matrices) causes the values of the b elements to be assigned to the corresponding elements of the matrix a except the last row which remains the same as the last step; and at the same time the old values of a are assigned to the corresponding values of the matrix b . The shapes of the target and the right-hand side are subject to the same intersection rule as the operands of a binary elementwise operation.

The syntax of the fixed-shape transformer includes two sections, *initial* and *update*. The *initial* section contains initializations of stream variables in the form $var = exp$. The *exp* can only name stream parameters par_j . The *update* section consists of the step declarations in the form $lexp := exp$ explained above. The left-expression $lexp$ cannot name variables of the input interface or any of the stream parameters, except in conditions and index formulae where they are allowed. The need for these restrictions is straightforward from the semantics of the target.

3. Type issues

As mentioned above, ASTL requires no explicit type declarations since all types can be inferred. A detailed description of the ASTL type-inference process is presented in [3] where the inference of local types is proven to be equivalent to a graph-theoretical problem of complexity $O(n^3)$, n being the number of local variables. For every type variable t_i associated with an internal variable v_i the exact least type is given by the following formula:

$$t_i = \max_{0 \leq j < m} t_j + w_{ij},$$

where the range of j includes all external type variables, the matrix w is the solution of the all-to-all shortest path problem for a weighted type-dependency graph derived from the program tree, and the meaning of $t + k$ is the k th supertype of type t , e.g. $integer + 1 = real$, $rank_3 - 2 = rank_1$, etc.

Consequently every local type is a non-decreasing function of external types. The global type-inference procedure uses two global types as external, namely the bottom of the element-type hierarchy, $\perp_t = boolean$ and the bottom of the rank hierarchy, $\perp_r = rank_0$. Owing to the above formula, each local type is offset from the bottom by a certain distance both directly and due to the offsets from the input types to the transformer, see an example in

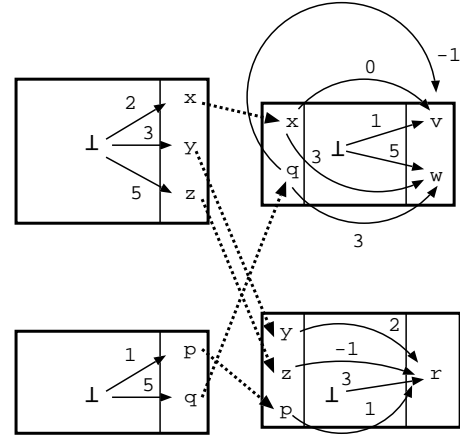


Figure 1. Distributed type inference.

fig 1, where four transformers are shown, two of which are stream sources. In the figure streams are depicted by dotted arrows and the solid arrows designate type² dependencies, with integer labels being used for offsets. For example the bottom left transformer and the top right one share a stream q . The former transformer constrains the rank of objects communicated over q to be at least 5 above the bottom rank (which is rank 0). At the same time the top-right transformer constrains its two output-stream variables, v and w to be at least (-1) and 3 ranks higher than the rank of q respectively. Focusing on variable v , although its direct offset from the bottom type is 1 v must be offset by at least 0 from x , the actual minimum rank of v depends on the rank of x as well. Since the constraint from q is tighter, the least acceptable type of v is still $\perp + 4$.

One would envisage the following reconciliation procedure. Each transformer approximates its input type variables by the bottom type and calculates the type of the output variables using the above equation. These output types are passed along the data links to the receiving transformer in a special message. When a transformer receives such a message, it treats the received type values as a new approximation of the input type variables. It then generates a new approximation of the output types to be sent to the corresponding inputs, etc. It can be proven that after a finite number of messages have passed between the transformers, each transformer either reaches a termination point or receives types which violate the maximum type constraints. In either case the inference process terminates.

The transformers in a transformer network need to know that they all have reached a termination point and

²For the sake of certainty, we assume that the diagram relates to ranks; a similar diagram could be drawn for element types

whether or not errors have been encountered. The former task is an instance of a well-known distributed termination-detection problem (see survey [6] for existing algorithms). The latter task can either be incorporated in the termination detection infrastructure or it can be completed separately by back-propagation of failure to the neighboring nodes.

The problem with this inference procedure is that its complexity depends on the range of types as well as the size of the longest path in the network graph. A long path and a large type range could cause reconciliation to be very slow. To remedy that, we have adopted the synchronous Bellman-Ford algorithm (see [5] for details) for type inference. This is known to deliver all distances from a given vertex (in our case it is the bottom type) to all vertices of the graph (in our case all variables defined within stream transformers) in linear time. Every step of the algorithm is performed by the whole network at once and is synchronised by a global barrier. It is known that after a maximum of N steps, N being the graph diameter, the process converges to an exact solution (or an error if the solution does not exist).

4. Streaming Network Support

A language for stream processing must include tools for creating and managing streaming networks. In a recent paper [7], a way of structuring such networks was proposed, whereby stream transformers (called, confusingly, “streams” therein) are connected into pipelines, optionally put between split-join (de-)jinterleavers and optionally augmented by feedback loops. The authors of [7] argue that such constrained structuring imposes good discipline on the process of streaming-network design and that it is a discipline not dissimilar in nature to the avoidance of GOTO in conventional programming. While not disputing their argument, we remark that in an open, collaborative environment such as the Grid, the structure of the stream-processing network is not under the control of a single agent and hence local mutations can in principle violate any pre-set discipline. Network structuring is arguably less important for the Grid than it is for embedded systems (which the authors of [7] concern themselves with) where the behaviour of the whole network is statically known and can be optimised.

What certainly is absolutely necessary for the viability of stream computing on the Grid in the form promoted by ASTL, is support for vital stream-processing primitives that cannot be described by flat, deterministic recurrence relations. We shall consider them next.

Merger A transformer may need to accept input from several sources by responding to objects received in the order of their arrival. Since all members of the input tuple are synchronized, this requires an external merger that im-

plements nondeterministic stream confluence. The merger output can then be fed to the input of a stream transformer.

The nondeterministic merger cannot be implemented as a recurrence relation since all recurrences are deterministic by construction, although a deterministic round-robin merger of two streams is of course easily achieved by holding the input for two iterations. Another difficulty arises when the streams to be merged are of different types. This requires a union type for the output and gives rise to the issue of record types which we have so far avoided.

Record types are useful as they represent a synchronous state of several field variables, and in this sense a stream of records is not identical to a group of named streams of array objects. On the other hand recursive record types, such as lists or trees are not generally streamable in their original form as they lack the regularity that gives stream processing a high degree of temporal locality. Indeed, not only would the streaming of recursive objects necessitate their linearization and de-linearization, the nature of processing for such types is fundamentally recursive rather than iterative which makes recurrence relations an inadequate programming tool.

We conclude that an array stream processing language requires only non-recursive record types. Interestingly, these need very little language support. An ASTL record consists of named fields that represent either co-existing entities or alternatives. The former is analogous to the C struct object, while the latter corresponds to the C union. Each of the fields can itself be either a union or a struct, etc. Accordingly, an atomic component of a record is fully identified by the sequence of tags that leads to it from the root of the record type tree. ASTL union tags are lexically distinct (they start with a capital letter) which makes the dot-separated list construct unambiguous. An input variable can now occur in expressions in the form *var.tags* where *var* is the stream name and *tags* are a dot-separated list of tags defining the path down the struct/union type tree of the record type *var*.

Stream transformers must also allow *var.tags* variables to occur on the left-hand side of a recurrence relation. Such occurrences correspond to a variant declaration for the output stream when a union tag is used in the tag list. Consequently several recurrences may be present to yield different variants of the output record. In the presence of variant records on the input, the recurrences inside the transformer are partitioned into groups that produce one variant of the output each. These groups are given a common tag, which becomes the tag of the output record variant.

Note that record types only result in the variable name becoming structured, and not in the emergence of any new operators or constructs within stream transformers. This means that the type inference strategy explained in the previous section remains largely intact. The only difference

to it in the presence of record types is that for each input variable a collection of paths used inside a transformer is determined and communicated back to the originating transformer. If the collection matches the type tree for that variable at the originator, then the record structures are *reconciled*, otherwise this results in a *structural* error. Structure reconciliation allows the originator to have extra struct fields which are ignored by the recipient, and also allows the recipient to define extra union fields that the originator may not use. However, since no structural features are carried over along the streams, reconciliation is always a strictly local affair involving the two communicating transformers.

Splitter The splitter is a primitive that allows one stream to be split into several identical streams directed to different destinations. Each of these streams progresses at its own pace without holding the others, which is why the splitter cannot be implemented as a recurrence relation inside a transformer. A degenerate splitter, which only has one output stream has the meaning of a buffer and is also essential for smoothing over the output rate of a transformer when the amount of work at each recurrent step may vary significantly.

Streaming network architecture is managed by running a set of ASTL servers on the Grid. Each server provides a set of services, including the *deployment* service. This allows individual transformers to be deployed on the Grid at a server location and register in relevant directories. The deployment could be under the user's control or it could be made automatic by adding monitoring facilities to the streams and mobility support to the servers. Under automatic deployment, the transformers that communicate frequently are pulled towards each other to reduce communication cost, while those that do not communicate intensively are pushed out onto the Grid to increase concurrency.

Next, a set of transformers intended for a single job is identified by communicating the task connection graph to the group of servers, which then establish connections between their transformers according to the graph. At this stage the parameters of each transformer (if any) are set and the recurrences are initialized. The connection graph can include any number of mergers and splitters, whose parameters are supplied with the graph definition, such parameters being the tags for merger inputs, the buffering capacity of individual splitters, etc.

When the connections are made, the servers initiate the distributed type inference procedure and finally the transformers are allowed to run (i.e. to receive data from streams) and the job begins. It can be terminated by propagating termination signals along the streams as variant records, or by signalling to the ASTL servers involved in the job. Similarly, any alterations of the streaming network for a job at its run-time can be made by deploying new or

modifying existing transformers and re-running the type inference for relevant parts.

The task of deployment and monitoring/steering a stream network is facilitated by an ASTL client, which is a program that communicates with the server network and provides the user with an adequate interface for performing the above functions.

5. Related Work

Coordinated networks have been considered in several publications prior to us. The most significant work reported to date is the system K2 [11], which bears strong superficial similarity to the proposal herein. K2 is based on the concept of a coloured Petri net, which is a network of parallel processes connected by streams. In K2, the streams carry untyped, wrapped tokens which may contain any self-contained data, possibly including high-order functions. K2 is process-, rather than data-centric in that it concerns itself with the safety and liveness issues of process coordination. It does, however support non-determinism by similarly introducing mergers as special processes, which was the essence of the original proposals from Turner [12], Schepers[13] and possibly others in the early 90s. K2 is purely a coordination infrastructure, so it implements stream transformers as wrappers around foreign code. By contrast we provide a data-centric model which is inherently type-safe and type-flexible, and we also suggest that where stream processing plays a significant role, the nodes can be made stateless, represented as data-parallel recurrence relations. Another important distinction is that, similarly to the Stream-It project [7], we support the idea of simple structured coordination based on stream data splitting/merging rather than unstructured general coordination using Petri Nets.

It is also appropriate to cite related work in the area of stateless data-parallel languages, since our solution is based on one such language, ASTL. The most fully developed and implemented language of this category is currently the language SAC [14], a referentially-transparent version of C. SAC is more general than we feel is required for stream processing, which makes the code potentially more dynamic with fewer properties available to the compiler. However, the inclusion of SAC, under a wrapper, into the context of streaming networks would be very advantageous thanks to its similarity to C (usually appreciated by the users) and the highly efficient compilation techniques for it reported recently in [15].

Conclusions and future work

We have presented a view on stream processing on the Grid. The cornerstone of our approach is the concept of

communicating array-valued recurrence relations embodied in stream transformers. We argue that the availability of effective type inference results in convenient notation and increased flexibility of the array-stream processing paradigm. We have proposed a stream-processing language ASTL, some features of which have been presented here, and for which a compiler into C is being developed.

We have delineated a network architecture for stream processing where transformers are deployed on the Grid. The server functionality and protocols will be the focus of our next step. We will rely on emerging Grid standards for stream programming [9] and develop the server and client software for stream network management. The most interesting issue that remains to be explored is automatic deployment. Thanks to temporal regularity of data streams, a predictive, automatic re-deployment of the kind we explored earlier [8] is likely to be effective here too.

References

[1] J. Mulder, J. van Wijk, and R. van Liere. A Survey of Computational Steering Environments. *Future Generation Computer Systems*, 13(6), 1998.

[2] Parallel computing with the Sisal applicative language Chinhyun Kim, Jean-Luc Gaudiot, Wlodek Proskowski *Software Practice and Experience* September 1996, Volume 26 Issue 9

[3] A Shafarenko. Coercion as Homomorphism: Type Inference in a System with Overloading and Subtyping. *Proceedings of PPDP'2002*, Pittsburgh NJ, 6-8 October 2002, ACM Press pp. 14-25.

[4] A. Shafarenko. Symmetry-based formalism for array subtyping. *APL Quote Quad* March, 2001, pp41-52

[5] N.A. Lynch, *Distributed Algorithms*, Morgan Kaufmann, 1996

[6] J. Matocha and T. Camp, A Taxonomy of Distributed Termination Detection Algorithms, *The Journal of Systems and Software*, vol. 43, no. 3, pp 207-221, 1998.

[7] Michael I. Gordon *et al.* A Stream Compiler for Communication-Exposed Architectures, *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, October, 2002.

[8] A Shafarenko and V Vasekin. An adaptive, reconfigurable interconnect for computational clusters. *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*. IEEE Comput. Soc, Los Alamitos, CA, USA; 2001; pp. 229-36

[9] Craig Lee. *Stream Programming: In Toto and Core Behavior*

<http://www.eece.unm.edu/~apm/WhitePapers/stream.pdf>

[10] The Sloan Digital Sky Project Book. The SDSS Data System and Data Products, online ver-

sion available from <http://www.astro.princeton.edu/PBOOK/datasys/datasys.htm>

[11] C. Assmann. Coordinating Functional Processes using Petri Nets. In: *Proceedings of IFL'96*, LNCS 1268, pp.162-183, Springer, 1997.

[12] D.A. Turner, 1990. An Approach to Functional Operating Systems. In Turner, D.A. ed., *Research topics in Functional Programming*, Addison-Wesley Publishing Company, pp. 199-217.

[13] J. Schepers. Using Functional Languages for Process Specifications. In *Proc. Int. Workshop on the Parallel Implementation of Functional Languages*, Southampton U.K., 1991. Computer Science Technical Report CSTR 91-07.

[14] S.B. Scholz. On Programming Scientific Applications in SaC – A Functional Language Extended by a Subsystem for High-Level Array Operations. In W. Kluge: *Proceedings of IFL'96*, LNCS 1268, p. 85-104, Springer-Verlag 1997

[15] C. Grelck, D.Kreye, and S.B. Scholz. On Code Generation for Multi-Generator With-Loops in SaC. In: P.Koopman and C. Clack: *Proceedings of IFL'99* (selected Papers), LNCS 1868, p. 77-95, Springer-Verlag 2000.