Projection of the Array-OL Specification Language onto the Kahn Process Network Computation Model

Abdelkader Amar, Pierre Boulet and Philippe Dumont
Laboratoire d'Informatique Fondamentale de Lille
Université des Sciences et Technologies de Lille, Cité Scientifique
59655 Villeneuve d'Ascq Cedex, France
⟨Firstname⟩.⟨Lastname⟩@lifl.fr

Abstract

The Array-OL specification model has been introduced to model systematic signal processing applications. This model is multidimensional and allows to express the full potential parallelism of an application: both task and data parallelism. The Array-OL language is an expression of data dependences and thus allows many execution orders.

In order to execute Array-OL applications on distributed architectures, we show here how to project such specification onto the Kahn process network model of computation. We show how Array-OL code transformations allow to choose a projection adapted to the target architecture.

1. Introduction

The most computation intensive part of signal processing applications is called systematic signal processing. It is usually the first part of applications such as detection systems (sonar, radar), multimedia (uncompressed video treatment), telecommunications (software radio), etc. These systematic signal processing applications are characterized by a very large amount of data-parallelism and by the handling of multidimensional data arrays. The complexity of these applications comes from the way the data are accessed in the arrays, using multidimensional patterns. An other complication comes from some cyclic dimensions of the arrays, such as frequency dimensions after an FFT or physical tori (as hydrophones around a submarine).

In order to conform to the needs for specification, standardization and efficiency of the multidimensional signal processing, Thomson Marconi Sonar has developed a systematic signal processing oriented language: Array-OL (Array Oriented Language) [8, 7]. As said above, this application domain is characterized by systematic, regular, and massively data-parallel computations. Array-OL relies on a graphical formalism in which the signal processing appears as a graph of dataparallel tasks. Each task reads and writes multidimensional arrays.

Lee et al. have proposed another multidimensional model to deal with such applications in [14, 15, 16]. We compare their model and Array-OL in [10]. Both models have strengths and weaknesses and none includes the other. The Array-OL model is easier to understand and hides most of the complexity of scheduling. This complexity is handled through code transformations that we describe in section 4.2.

In order to allow a distributed execution of Array-OL applications, we study here how this specification model can be projected on a distributed computation model, namely the Kahn Process Network computation model [11, 12]. Distributed executions of systematic signal processing applications are useful because these applications are computation intensive. Thus using the computation power of several computers helps to reduce the execution or simulation time of such applications. Furthermore, they are often embedded and executed on parallel architectures such as Systems-on-Chip or multiprocessors. The projection we propose allows to choose the scheduling of the application and to adapt it to the target architecture.

In section 2 we explain the Array-OL model. Then in section 3, we recall the Kahn process network model of computation and describe the distributed implementation we have used. We then study how to project Array-OL specifications onto process networks and the Array-OL code transformations that allow this projection in section 4. We finally conclude in section 5.

2. Array-OL model of specification

In this section we briefly sketch the Array-OL modeling language. It is important to notice that Array-OL is only a specification language, no rules are specified for executing an application described with Array-OL.

The basic principles underlying the language are:

- Array-OL is a data dependence expression language.
- All the available parallelism in the application should be available in the specification, both task parallelism and data parallelism.
- It is a single assignment formalism.
- The spatial and temporal dimensions are treated equally in the arrays. In particular, time is expanded as a dimension (or several) of the arrays.
- The arrays are seen has tori. Indeed, some spatial dimensions may represent some physical tori (think about some hydrophones around a submarine) and the frequency domains obtained by FFT are toroidal.

The modeling of an application in Array-OL needs two levels of description. The first one is the global model, it defines the task parallelism in the form of dependences between tasks and arrays. The second one is the local model which details the elementary action the tasks realize on array elements. This local model expresses the data parallelism.

2.1. Global model

The global model is a simple directed acyclic graph. Each node represents a task and each edge an array. The number of incoming or outgoing arrays is not limited. The graph is a dependence graph, not a data flow graph.

So it is possible to schedule the execution of the tasks just with the global model. But it's not possible to express the data parallelism of our applications because the details of the computation realized by a task are hidden at this specification level.

2.2. Local model

The local model is a little bit more complicated, it allows to express data parallel repetitions. At this level, we specify how the array elements are consumed and produced in a task. These elements are treated in parallel block by block.

The size and shape of a block associated to an array is the same for each repetition. That's why we call a block of data a *pattern*. In order to allow a hierarchical construction, the patterns are themselves arrays.

In order to give all the information needed to create these patterns, we need the following information:

- O: the origin of the reference pattern (for the reference repetition)
- D: the shape (size of all the dimensions) of the pattern
- P: a matrix called the paving matrix that describes how the patterns tile the array
- F: a matrix called the fitting matrix that describes the shape of the tile (how to fill a pattern with array elements)
- M: the shape (size of all the dimensions) of the array

Now with all this information we are able to do all the manipulations around our notion of pattern.

A detailed description of the meaning of these fitting and paving expression can be found in an extended version of this paper [4]. We can summarize all these explanations with two formulas:

- $\forall X_q, 0 \leq X_q < Q, (O + P \times X_q) \mod M$ give all the reference elements of the patterns, Q being the shape of the repetition domain.
- $\forall X_d, 0 \leq X_d < D, (O + P \times X_q + F \times X_d) \mod M$ enumerates all the elements of a pattern for the X_q repetition.

From this formulas, we can deduce that the tiles are not necessarily parallel to the axes nor contiguous, that, as the arrays are not toroidal, points of a tile may cross the borders of the arrays, and finally that an array element may belong to several tiles.

2.3. Hierarchy

According to the specification of the local model, Array-OL applications must respect some obligation on the pattern shape: they are themselves arrays. This allows to build an application in a hierarchical way.

The data dependences visible at a given hierarchical level are approximations of the real data dependences. One needs to look at the whole hierarchical construction to have the precise dependences.

2.4. Summary

Array-OL allows us to describe systematic signal processing applications in a very convenient way. More precisely we describe data dependences. At the global level, the dependences between tasks are given by the input and output arrays. At the local level, the dependences are given in terms of patterns.

Once again, Array-OL is a specification model and does not impose any execution order. We will see below how a distributed execution of Array-OL specifications can be obtained by projection onto the Kahn process network model of computation.

3. Kahn process network model of computation

3.1. Model

The process network model has been proposed by Kahn and MacQueen [11, 12] to easily express concurrent applications. Processes communicate only through unidirectional FIFO queues. A process is blocked when it attempts to read from an empty queue. A process can be seen as a mapping from its input streams to its output streams. The number of tokens produced and their values are completely determined by the definition of the network and do not depend on the scheduling of the processes. Thus the process network model is called determinate.

A more lengthy presentation of this model is available in the extended version of this paper [4].

3.2. Implementation

The implementation we have used in our experiment is a distributed implementation on top of CORBA [3, 2]. One of the goals of this implementation is to hide the complexity of building distributed applications to the programmer, typically a non computer science specialist. The choice of implementation has been directed by the availability to us of a *distributed* implementation.

To reach this goal, the authors have made some restrictions on the processes in their implementation. These restrictions simplify the scheduling of the process network while retaining the expressing power needed for the applications.

These restrictions allow to easily represent complex applications based on an assembly of components. This coarse grain view of the application is better suited to a performant execution on a network of computers than a finer grain view which generates too much communications. This does not forbid the component to be parallel and to execute on a parallel computer.

4. Projection of Array-OL onto Kahn process networks

The Array-OL specification model allows many execution orders. Actually, any execution order compatible with data dependences expressed by the specification is valid. The benefits of using a Kahn process network computation model as a foundation to execute Array-OL specifications are:

- The full parallelism of the specification can be exploited on distributed execution platforms.
- Determinism (main property of Kahn process networks) that gives good debugging and profiling possibilities.
- Simplified synchronization handling by using FIFOs.
- Systematic construction (see below).
- Easy handling of the hierarchy of the specification (see below).

4.1. From arrays to streams

The main question when projecting Array-OL is what kind of data structure is carried by the tokens: arrays or patterns? Indeed, a global model of Array-OL can be seen as a process network with the processes being the data-parallel tasks defined by the local models.

4.1.1 Streams of patterns

The first idea is usually to make a stream of patterns between processes. These processes thus take a set of patterns on each of their input to produce a set of patterns on their output.

The problem we encounter here is that arrays may be produced and consumed in different ways – composed of different pattern sets. Only in special cases are the arrays produced and consumed by the same patterns. One thus generally has to group some producing patterns in a token that is split into a group of consuming patterns by the following process. Determining such groupings is a difficult task that is at the heart of the difficulty to schedule Array-OL specifications. It has been studied as part of the fusion code transformation that will be presented in section 4.2.

Once the tokens have been determined, one still has to choose an execution order for the producing and consuming tasks that allows to pipeline these two tasks. Indeed, the arrays may be large, or even infinite, so pipelining is necessary to ensure a "reasonable" execution. What we mean be "reasonable" is an execution that does not loop infinitely on a given subtask, that does not use unnecessarily large amounts of memory and that does not compute too many times the same intermediate data. This is already difficult when considering two tasks but one needs to pipeline a directed graph of tasks (or process network).

4.1.2 Streams of arrays

The other alternative is to make streams of arrays. One has to look at an Array-OL specification at another level: The main level of the hierarchical expression is now the local model (the data-parallel repetition). If the repetitive task is hierarchical, this task is described by a global model itself that can be seen as a process network.

The repetition (local model at depth l of the hierarchical specification) generates a number n of data-parallel repetitions of computations of the process network (global model at depth $l + 1$). The order of execution of these repetitions is not specified and can be chosen at will. Instead of having n instances of each task at level $l + 1$ exchanging 1 array, one can have 1 instance of each of those tasks working on a stream of arrays. The only thing that has to be done carefully is filling the input array streams and storing the output array streams in a consistent way. This is easy to do by choosing an enumeration order of the repetition space described by the paving at level l. Thus an array at level l is transformed into a stream of arrays at level $l+1$ (or patterns of level l).

This projection of Array-OL onto process networks can be summarized as follows:

- "Array" ↦ "token".
- "Elementary task" ↦ "process".
- "Local model data-parallel repetition" ↦ "stream".

Figure 1 shows an example of such a projection.

Such a translation has the advantage that it is direct and systematic. The only choice is the order of the patterns of level l in the streams. The difficulty comes from the fact that the time and space dimension are uniformized. So, to choose an order that is coherent with the availability of data at the input of the application, one has to ensure an order compatible with the flow of time.

From there comes the main drawback of this projection: it is not always possible to find such an order! Indeed, if the top level of the application is a global level manipulating infinite arrays, there is no way to produce a stream because there is no surrounding repetition. If we apply the projection, the first task reads a token that is an infinite array and no computation is ever completed.

This problem has already been observed when trying to execute Array-OL applications on multithreaded workstations [18, 17]. The solution that has been proposed is to transform the application in order to create a new hierarchical level where the infinite arrays only appear at the top level and only one task reads from and writes into such infinite arrays. This task is itself repetitive and can thus be transformed into a stream of finite arrays. Such a stream is the usual implicit repetition of data-flow formalisms.

In the following section, we will describe the available Array-OL transformations that can be used to allow the projection we have described here.

4.2. Array-OL transformations

As shown in section 2, the Array-OL model has some particularities that prevent the use of general loop transformations like loop fusion, distribution or unimodular loop transformations [6, 1]. These particularities are the systematic presence of the modulo operator in the paving and fitting expression and the possibly infinite size of one of the dimensions of the arrays. On the other hand, the fact that the Array-OL formalism respects the single assignment form [5] and that the repetitions are completely parallel simplifies some transformations.

In [18, 17], the authors have introduced an ad-hoc formalism (the ODT, array distribution operators – *Opérateurs de Distribution de Tableaux* in French) to transform Array-OL specifications in order to make the application executable with a naive execution scheme. In [9], the *fusion* transformation has been completely proved and generalized to more complicated cases. This fusion is the building block of nearly all the other transformations that are needed for the projection of Array-OL onto Kahn process networks. We will first explain this transformation and its interest for this study and then show the other useful transformations.

4.2.1 Fusion of two repetitive tasks

The fusion aims at reducing two tasks in a single one. This new task is hierarchical and calls the two original tasks as sub tasks.

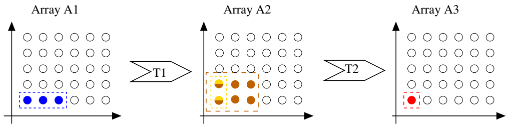

Figure 2. Before fusion

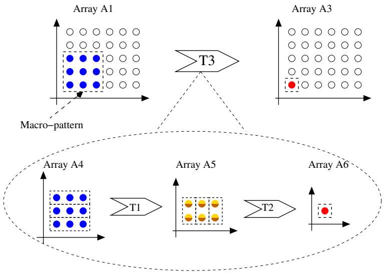

Figure 3. After fusion

Figure 2 shows an example of two tasks to be fused. Array $A2$ is produced by task $T1$ by the way of patterns of two elements and consumed by task $T2$ by patterns of 6 elements. Figure 3 shows the result of the fusion. The new task $T3$ realizes in one step what was computed by tasks $T1$ and $T2$ .The inputs and outputs are kept identical. Thus the fusion is a local transformation that does not disturb the rest of the application. The new subtasks $T1'$ and $T2'$ compute the same operations as $T1$ and $T2$ but on different inputs and produce different outputs. These new inputs and outputs

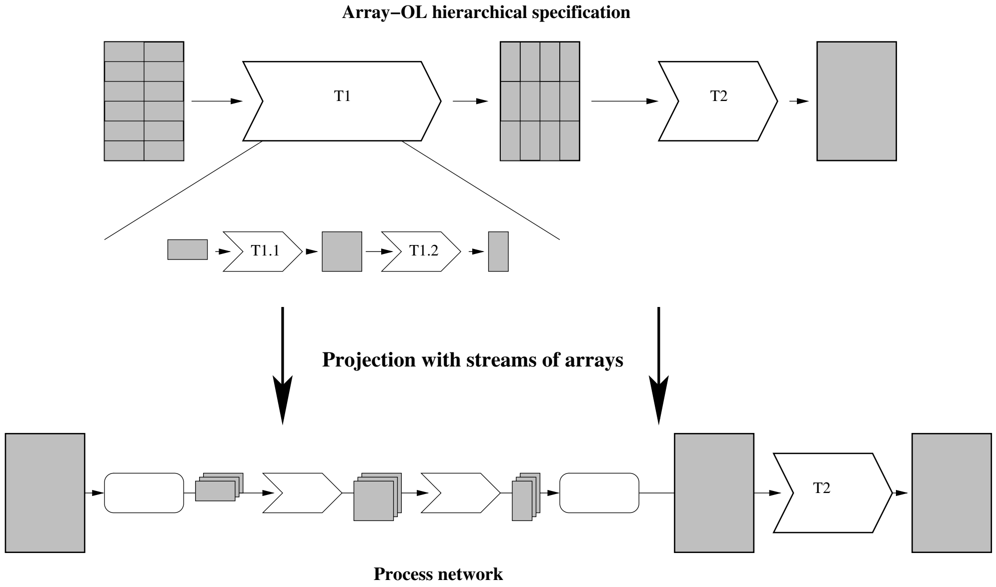
Array−OL hierarchical specification
T1
T2
T1.1
T1.2
Projection with streams of arrays
T2
Process network

Figure 1. Example of the projection of a hierarchical Array-OL specification onto a process network using streams of arrays.

are sub-arrays of the arrays which $T1$ and $T2$ were connecting. They are called macro-patterns and are constituted of an whole number of patterns of the original repetitions in $T1$ and $T2$.

The computation of the fusion in the general case (when the fittings and pavings are non parallel to the axes, present some shifting, tile some arrays in a cyclic way or are non compact) is very challenging (comparable to the scheduling of GMDSDF [16]). The complete description of this process is beyond the scope of this paper (it takes several tens of pages) and is available in [9]. A complete implementation of this transformation has been realized in the Gaspard project [13]. This transformation has been extended to several useful cases such as when the tasks have several inputs or outputs or even when they are connected by several intermediate arrays.

Fusing two tasks this way allows to build a hierarchy level that can then be used to make a stream of arrays and then allow the successful projection of the transformed application onto a process network.

4.2.2 Other Array-OL transformations

The fusion alone is not sufficient. Indeed, this transformation has a number of drawbacks: It may introduce redundant computations; The chaining of transformation produces deep hierarchies; And other minor problems may appear in border cases.

To alleviate the first two problems, two other transformations have been proposed in [17]: the "change paving" and the "one level" transformations. The change paving increases the size of the macro-pattern in a way to reduce the repeated computations. And the one level allows to realize the fusion of several tasks based on a chain of fusions and change pavings. That transformation can be used to deal with the problem of an application whose top level is a global model whose arrays have an infinite dimension. In that case, the transformed application exhibits a top level repetitive task that infinitely repeats a finite global model. That infinite repetition can then be projected as a stream and the global model as a process network.

A final useful transformation is the "nesting" where the repetition domain is split in two nested repetitions. It consists in the creation of a hierarchical level whose global model is a single repetition whose patterns (called macro-patterns) are unions of the original patterns. These macro-patterns are then consumed by a second nested repetition that works with the original patterns. This transformation allows to adapt the granularity of the application.

4.3. Projecting Array-OL specifications onto distributed process networks

The above transformations can be used together to transform the specification into another one that expresses the same dependences between the array elements but that is more suited to be executed on a given platform. The kind of platforms we focus on in this article is an heterogeneous distributed architecture (as a network of workstations or a System-on-Chip).

There are two ways one could benefit from the regular parallelism exhibited by a data-parallel repetitions:

1. A SPMD execution on several execution units (or using several threads).
2. The transformation of that repetition in a stream as expressed in section 4.1.2.

Using a third possibility, namely a sequential execution, one can propose a large family of schedules for a given specification by tagging each data-parallel repetition by the execution strategy: SPMD, Process Network or Sequential.

Selecting the "best" solution, both in terms of specification transformation and schedule choice is way beyond the focus of this paper and is a research interest we will pursue in future research.

Due to space constraints, we invite the reader to refer to the extended version of this paper [4] for an experiment on mapping a sonar application specified in Array-OL onto a distributed process network.

5. Conclusion

We have presented in this paper the Array-OL specification model. This multidimensional model allows to fully express the potential parallelism of systematic signal processing applications. This specification is completely indepentent on the execution architecture. We have shown how several code transformations have been implemented to allow to derive a form of the application that can be projected directly onto a process network.

This projection allows a distributed execution of Array-OL applications onto heterogeneous distributed architectures such as Systems-on-Chip or networks of workstations.

In future work, we will study heuristics to select the chain of transformations that leads to the "best" schedule of the application on a target architecture, taking into account real time constraints, power consumption, cost, etc.

References

[1] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers, Oct. 2001. `http://www.mkp.com/books_catalog/catalog.asp?ISBN=1-55860-286-0`.

[2] A. Amar. *Support d'exécution pour le metacomputing à l'aide de CORBA*. PhD thesis, Université des sciences et technologies de Lille, Laboratoire d'informatique fondamentale de Lille, Dec. 2003. (In French).

[3] A. Amar, P. Boulet, J.-L. Dekeyser, and F. Theeuwen. Distributed process networks using half FIFO queues in CORBA. In *ParCo'2003, Parallel Computing*, Dresden, Germany, Sept. 2003.

[4] A. Amar, P. Boulet, and P. Dumont. Projection of the Array-OL specification language onto the Kahn process network computation model. Research Report RR-5515, INRIA, Mar. 2005. `http://www.inria.fr/rrrt/rr-5515.html`.

[5] J.-F. Collard. *Reasoning about program transformations: imperative programming and flow of data*. Springer-Verlag, 2003. ISBN 0-387-95391-4.

[6] A. Darte, Y. Robert, and F. Vivien. *Scheduling and Automatic Parallelization*. Birkhauser Boston, 2000. `http://www.birkhauser.com/detail.tpl?isbn=0817641491`.

[7] A. Demeure and Y. Del Gallo. An array approach for signal processing design. In *Sophia-Antipolis conference on Micro-Electronics (SAME 98)*, France, Oct. 1998.

[8] A. Demeure, A. Lafage, E. Boutillon, D. Rozzonelli, J.-C. Dufourd, and J.-L. Marro. Array-OL : Proposition d'un formalisme tableau pour le traitement de signal multi-dimensionnel. In *Gretsi*, Juan-Les-Pins, France, Sept. 1995.

[9] P. Dumont and P. Boulet. Transformations de code Array-OL : implémentation de la fusion de deux tâches. Technical report, Laboratoire d'Informatique fondamentale de Lille et Thales Communications, Oct. 2003.

[10] P. Dumont and P. Boulet. Another multidimensional synchronous dataflow: Simulating Array-OL in ptolemy II. Research Report RR-5516, INRIA, Mar. 2005. `http://www.inria.fr/rrrt/rr-5516.html`.

[11] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing 74: Proceedings of the IFIP Congress 74*, pages 471–475. IFIP, North-Holland, Aug. 1974.

[12] G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. In B. Gilchrist, editor, *Information Processing 77: Proceedings of the IFIP Congress 77*, pages 993–998. North-Holland, 1977.

[13] Laboratoire d'informatique fondamentale de Lille, Université des sciences et technologies de Lille. Gaspard home page. `http://www.lifl.fr/west/gaspard/`, 2005.

[14] E. A. Lee. Multidimensional streams rooted in dataflow. In *Proceedings of the IFIP Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, Orlando, Florida, Jan. 1993. North-Holland.

[15] P. K. Murthy. *Scheduling Techniques for Synchronous and Multidimensional Synchronous Dataflow*. PhD thesis, University of California, Berkeley, CA, 1996.

[16] P. K. Murthy and E. A. Lee. Multidimensional synchronous dataflow. *IEEE Transactions on Signal Processing*, July 2002.

[17] J. Soula. *Principe de Compilation d'un Langage de Traitement de Signal*. Thèse de doctorat (PhD Thesis), Laboratoire d'informatique fondamentale de Lille, Université des sciences et technologies de Lille, Dec. 2001. (In French).

[18] J. Soula, P. Marquet, J.-L. Dekeyser, and A. Demeure. Compilation principle of a specification language dedicated to signal processing. In *Sixth International Conference on Parallel Computing Technologies, PaCT 2001*, pages 358–370, Novosibirsk, Russia, Sept. 2001. Lecture Notes in Computer Science vol. 2127.