J. Functional Programming **10** (6): 561–605, November 2000. Printed in the United Kingdom

Compilation of a specialized functional language for massively parallel computers

PASCAL FRADET and JULIEN MALLET
IRISA, Campus de Beaulieu, 35042 Rennes, France
(*e-mail:* {fradet,mallet}@irisa.fr)

Abstract

We propose a parallel specialized language that ensures portable and cost-predictable implementations on parallel computers. The language is basically a first-order, recursion-less, strict functional language equipped with a collection of higher-order functions or skeletons. These skeletons apply on (nested) vectors and can be grouped into four classes: computation, reorganization, communication and mask skeletons. The compilation process is described as a series of transformations and analyses leading to SPMD-like functional programs which can be directly translated into real parallel code. The language restrictions enforce a programming discipline whose benefit is to allow a static, symbolic and accurate cost analysis. The parallel cost takes into account both load balancing and communications, and can be statically evaluated even when the actual size of vectors or the number of processors are unknown. It is used to automatically select the best data distribution among a set of standard distributions. Interestingly, this work can be seen as a cross-fertilization between techniques developed within the FORTRAN parallelization, skeleton and functional programming communities.

Capsule Review

The scientific computing world is notoriously (or admirably!) conservative in its adoption of new programming models. This paper describes a system which combines and builds on ideas from a range of approaches beloved of readers of this journal, in order to offer both portability and accurate cost-prediction to parallel scientific programmers. It is noteworthy for its combination of theoretical rigour with concern for practicalities.

1 Introduction

A good parallel programming model must be portable and cost predictable. General purpose languages such as FORTRAN achieve portability, but cost estimations are often very approximate. A precise cost analysis is especially important in this context, since the goal of parallelization is efficiency, and its impact on the overall cost is, at best, a division by a constant. So, orders of magnitude or maximum complexities are insufficient to guide parallel implementation choices.

The approach described in this paper is based on a restricted pure functional language that is portable, and allows us to design an automatic and accurate cost analysis. The language restrictions can be seen as enforcing a programming discipline that ensures a predictable performance on the target parallel computer (there will be no ‘performance bugs’). General recursion and conditionals are replaced by skeletons that encapsulate control and data flow in the sense of Cole (1988) or Darlington *et al.* (1993). The skeletons, which act on (potentially nested) vectors, can be grouped into four classes: the *computation skeletons* (classical data parallel functions), the *reorganization skeletons* (creating and structuring vectors), the *communication skeletons* (data motion over vectors), and the *mask skeletons* (conditional data parallel functions). The skeletons and data structures have been chosen with scientific computing in mind. Matrix computations and nested `for` loops are easy to describe, and many standard numerical algorithms have been expressed easily in our kernel language. Concerning the target parallel architecture, we aimed at MIMD (*Multiple Instructions Multiple Data*) computers with shared or distributed memory, but SIMD (*Single Instruction Multiple Data*) computers could be accommodated as well.

The compilation process is described as a series of program transformations leading to SPMD-like (*Single Program Multiple Data*) functional programs which can be directly translated into true parallel code. Each compilation step transforms a skeleton-based language into another closer to a code with explicit parallelism. The main compilation steps consist of a size analysis, an update-in-place analysis, a transformation making all communications explicit, transformations implementing data distribution, and a symbolic code analysis. Note that using a functional language avoids the dependence analysis needed by imperative languages to determine the computations which can be executed in parallel. A key task in the compilation of data parallelism is the choice of the distribution, since it determines both the load balancing and the communications between processors. In our approach, the restrictions imposed by the source language make it possible to choose automatically the globally best data distribution among a set of standard distributions. This choice relies on the cost analysis which evaluates accurate parallel execution times. One of the main challenges of our work was to tackle this problem without knowing the actual size of vectors or the number of processors. In other words, the analysis ought to be symbolic.

The contributions of this work are both technical and methodological.

- The compilation is efficient and original: it integrates very different analysis techniques in a sequence of program transformations. Maybe the most important contribution lies in the definition of the specialized language. We establish the necessary language restrictions to ensure the accuracy of a symbolic cost analysis. The compilation takes advantage of the analysis to choose the best parallel implementation from among a set of standard implementations. The analyses for FORTRAN nested loops (Gupta and Banerjee, 1992; Feautrier, 1994) do not evaluate precisely the cost of the communications. In particular, collective communications (diffusion, translation etc.) cannot be automatically detected in FORTRAN programs. In our approach, the analysis takes into account both load balancing and communication costs, since the collective communications appear explicitly through communication skeletons. Most existing skeletons implementations (Blelloch *et al.*, 1994; Darlington *et al.*, 1995) are based on fixed implementations for each skeleton. This local view may lead to redistributing data before each skeleton application, and therefore be very inefficient. Finding the best, global distribution is more complex but more efficient, since it takes into account compromises (i.e. trading a local cost increase for a global improvement).

- At the methodology level, this work is a rare example of cross-fertilization between three different research fields: the automatic parallelization of FORTRAN, skeleton based languages and functional programming. The FORTRAN community has worked on symbolic complexity analyses for subsets of FORTRAN. We adapt and extend this work to define the cost analysis in our context. Skeleton-based languages with their collection of data parallel functions provide a framework to define specialized parallel languages. We build upon this approach, and provide two new collections of skeletons to the programmer: the communication skeletons and the mask skeletons. The community of functional programming has produced a large body of work on typing, program transformation and analysis. Our compilation process, made of a collection of program analyses and transformation, relies on this work.

The article is structured as follows. Section 2 is an overview of the compilation process. Section 3 presents the source language and the target parallel language. Sections 4–8 describe each compilation step in turn. We report some experiments done on MIMD distributed memory computers in section 9. Section 10 justifies *a posteriori* the source language restrictions, reviews related work, and suggests directions for further research.

2 Overview

The compilation process consists of a series of program transformations:

$$\mathscr{L}_1 \xrightarrow{\mathscr{G}l} \mathscr{L}_2 \xrightarrow{\mathscr{E}c} \mathscr{L}_3 \xrightarrow{\mathscr{A}bs} \mathscr{L}_4 \xrightarrow{\mathscr{D}ist} \mathscr{L}_5 \xrightarrow{\mathscr{O}pt} \mathscr{L}_6 \xrightarrow{\mathscr{T}ra} \textit{Parallel Code}$$

Each arrow represents a transformation compiling a particular task by mapping skeleton programs from one intermediate language ($\mathscr{L}_i$) into another ($\mathscr{L}_{i+1}$). The source language ($\mathscr{L}_1$) is composed of a collection of higher-order functions (skeletons) acting on vectors (see section 3.1). It is primarily designed for a particular domain where high performance is a crucial issue: numerical algorithms. $\mathscr{L}_1$ is best viewed as a parallel kernel language embedded in a general sequential language (e.g. C). Only parts of programs written in $\mathscr{L}_1$ will be executed in parallel, whereas others parts will be executed sequentially, for example, on the host computer of the parallel machine.

The first compilation step is the *type/size analysis* of $\mathscr{L}_1$ programs. The analysis computes the shape (size) of all the vectors occurring in the program (section 4). It must infer symbolic sizes because the sizes of input vectors may be unknown at compile time. As a byproduct, the analysis infers conditions ensuring that no vector access error occurs at runtime.

The first transformation ($\mathscr{L}_1 \to \mathscr{L}_2$) deals with in-place updating, a standard problem in functional programming with aggregates (section 5). The program is analyzed to check that all vectors can be safely modified in place. If the program does not pass the analysis, it must be transformed by inserting explicit vector copies. This can be done automatically (the analysis indicates the places where to insert copies) or manually (the programmer may want to restructure the program to insert fewer copies). This step is also used to guarantee that vectors are either returned as result or are explicitly deallocated (i.e. a garbage collector is not needed).

The transformation $\mathscr{E}c$ ($\mathscr{L}_2 \to \mathscr{L}_3$) makes all communications explicit (section 6). Intuitively, to execute an expression such as **map** $(\lambda x.x + y)$ in parallel, y must be broadcast to every processor before applying the function. The transformation makes this kind of communication explicit. In the language $\mathscr{L}_3$, all communications are expressed through skeletons.

The transformations from $\mathscr{L}_3$ to $\mathscr{L}_6$ concern automatic data distribution (Section 7). First, λ-abstractions and variables are removed by threading an explicit environment throughout the program (transformation $\mathscr{A}bs$, section 7.1). This transformation, reminiscent of abstraction algorithms, prepares the distribution transformation $\mathscr{D}ist$ ($\mathscr{L}_4 \to \mathscr{L}_5$). We consider a set of standard distributions of the input vectors. A vector can be distributed cyclicly, by contiguous blocks, or allocated to a single processor. For a matrix (vector of vectors), this gives nine possible distributions (cyclic cyclic, block cyclic, row cyclic, etc.). Distribution transforms programs so that they act on a single vector whose elements represent the processors. This implies, in particular, to change all vector accesses according to the distribution (section 7.2). Finally, distributed programs are optimized ($\mathscr{O}pt$: $\mathscr{L}_5 \to \mathscr{L}_6$) using a set of local transformations (section 7.3). After distribution, some vector copies become useless. They are removed in order to improve the sequential execution time. $\mathscr{L}_6$ programs apply on a vector of processors and simulate an SPMD code.

To choose the best distribution, an $\mathscr{L}_4$ program is transformed according to all the possible distributions of its input parameters leading to a set of $\mathscr{L}_6$ programs. The symbolic cost of each version is evaluated and the smallest one chosen (section 8). For most numerical algorithms, the number of input vectors is small and this approach is practical. In other cases, we would have to rely on the programmer to prune the search space.

The transformation $\mathscr{T}ra$ ($\mathscr{L}_6 \to$ *Parallel Code*) is a straightforward translation of the SPMD skeleton program to an imperative program with calls to a standard communication library. We currently use C with the MPI (*Message Passing Interface*) library along with the C compiler of the host machine (section 9 presents experiments on an Intel Paragon XP/S and a CRAY T3E).

All the transformations are automatic. Nevertheless, the user can interact with the compiler, for example to insert explicit copies in the $\mathscr{L}_1 \to \mathscr{L}_2$ step, or to guide the choice of the best distribution in $\mathscr{L}_5$.

For each transformation $\mathscr{T}_i$: $\mathscr{L}_i \to \mathscr{L}_{i+1}$, three correctness properties must be proved. First, it must be shown that the transformation $\mathscr{T}_i$ transforms programs of $\mathscr{L}_i$ into programs of $\mathscr{L}_{i+1}$, formally:

Property 1
$\forall Prog \in \mathscr{L}_i \Rightarrow \mathscr{T}_i[\![Prog]\!] \in \mathscr{L}_{i+1}$.

Secondly, the transformation $\mathscr{T}_i$ must preserve the semantics of programs, formally:

Property 2
$\forall Prog \in \mathscr{L}_i, \mathscr{T}_i[\![Prog]\!] = Prog$

Thirdly, it must be checked that the update-in-place property still holds on transformed programs.

Property 3
$\forall Prog \in \mathscr{L}_i \ (i > 1)\ ,\ \mathscr{U}p(Prog) \Rightarrow \mathscr{U}p(\mathscr{T}_i[\![Prog]\!])$

Since transformations are defined on the structure of expressions, the proofs of these properties usually boil down to a routine inspection of the different cases. Due to their number and length, we do not describe them in this paper. A few examples of proofs are sketched in Appendix B.

The source language comprises 16 skeletons, plus a number of other constructions (pairs, operators, affine expressions, etc.). Further, new skeletons are added as the language gets closer to an SPMD language. Presenting the six compilation steps (analyses and transformations) for the whole language would be lengthy and tiresome. After presenting the complete source and target languages in the next section, we chose to focus on a tiny sublanguage for the rest of the presentation. A simple example (written in the sublanguage) is taken throughout the paper and illustrates the different steps.

The treatment of a more complete language (having at least one skeleton of each type) can be found in Appendix A. The interested reader will find a description of transformations, analyses and proofs for the whole language in Mallet (1988a)[1]. A previous conference paper (Mallet, 1998b) focuses on the cost analysis, and can be seen as a short introduction to this work.

3 Source and target languages

3.1 The source language $\mathscr{L}_1$

The source language $\mathscr{L}_1$ is basically a strict, pure, first-order, recursion-less functional language, extended with a collection of higher-order functions (the skeletons). We have restricted ourselves to a reasonable number of standard skeletons; new ones, especially among the computation and reorganization classes, could be integrated as well. The main data structure is the vector which can be nested to model multidimensional arrays.

The syntax of $\mathscr{L}_1$ is defined in figure 1 (its type system will be described in section 4). A program is a main expression followed by definitions. A definition is either a function definition or the declaration of input variables with their types. The types

[1] Actually, the language presented here is slightly different (it considers a larger class of mask skeletons) than that in Mallet (1998a)

Prog_1	::=	Exp_1 **where** Decl_1
Decl_1	::=	$\text{Decl}_1\ \text{Decl}_1 \mid f = \text{Fun}_1 \mid x :: \text{Type}_1$
Type_1	::=	$(\text{Type}_1, \ldots, \text{Type}_1) \mid \mathit{Vect}\ \text{LinF}_1\ \text{Type}_1 \mid \mathit{Int} \mid \mathit{Float} \mid \mathit{Bool}$
Exp_1	::=	$\text{Fun}_1\ \text{Exp}_1 \mid (\text{Exp}_1, \ldots, \text{Exp}_1) \mid x \mid k$
Fun_1	::=	**iterfor** $\text{LinF}_1\ \text{Fun}_1 \mid \text{Op}_1 \mid \lambda(x_1, \ldots, x_n).\text{Exp}_1 \mid f$
	|	$\text{CompSkel}_1 \mid \text{ReorgSkel}_1 \mid \text{CommSkel}_1 \mid \text{MaskSkel}_1$
Op_1	::=	$+ \mid - \mid *$ | **div** | **exp** | **log** | **cos** | ...
LinF_1	::=	$\text{LinF}_1 + \text{LinF}_1 \mid \text{LinF}_1 - \text{LinF}_1 \mid k{*}\text{LinF}_1 \mid x \mid k$
CompSkel_1	::=	**map** Fun_1 | **fold** $\text{Exp}_1\ \text{Op}_1$ | **scan** $\text{Exp}_1\ \text{Op}_1$
ReorgSkel_1	::=	**zip** | **unzip** | **append** | **makearray** LinF_1
CommSkel_1	::=	**brdcast** LinF_1 | **transfer** $\text{LinF}_1\ \text{LinF}_1$
	|	**rotate** LinF_1 | **scatter** LinF_1
	|	**gather** LinF_1 | **allgather** | **allbrdcast**
MaskSkel_1	::=	$\mathbf{poly}_n\ \lambda(x_1, \ldots, x_n).\text{Ineq}_1\ \text{Fun}_1\ \text{Fun}_1$
Ineq_1	::=	$\text{Ineq}_1 \wedge \text{Ineq}_1 \mid \text{LinF}_1 < \text{LinF}_1 \mid \text{LinF}_1 = \text{LinF}_1$

$x, x_1, \ldots, x_n \in$ VarIdent. $f \in$ FunIdent. VarIdent $\cap$ FunIdent $= \emptyset$. $k \in$ Constant.

Fig. 1. Skeleton language $\mathscr{L}_1$.

of input vectors bear their numerical or symbolic size. An expression (nonterminal *Exp*$_1$) is either an application, a tuple, a variable or a constant. Functions are unary λ-abstractions, unary operators or the predefined iterator **iterfor**. It can be defined in HASKELL (Hudak *et al.*, 1992) as follows:

```
-- iterfor e f x = f(e ...,f(2,f(0,x))...)
iterfor e f x = let until p f x = if p x then x else until p f (f x)
                in (snd.until(\(x,_)->x>e)(\(i,x)->(i+1,f(i,x))))(0,x)
```

iterfor $n\ f\ a$ behaves like a loop; it applies $n+1$ times its function argument f on a. Further, it makes the current loop index (henceforth called the iterator index) accessible to its function argument.

Four classes of skeleton manipulate vectors: computation, reorganization, communication and mask skeletons.

The *computation skeletons* are the classical higher order functions **map**, **fold**, and **scan**. For example, the **fold** skeleton is defined, using the standard array library of HASKELL (Hudak *et al.*, 1999) as:

```
-- fold e f [a0;...;an] = f(...f(f(e,a0)...,an)
fold e f v = iterfor n (\(i,acc)->f(acc,v!i)) e
             where n = sizeRange (bounds v) - 1
```

Many other computation skeletons could have been considered. Note that **fold** or **scan** expressions are considered as parallel constructs only if their operator (Op_1) is associative. Otherwise, they are compiled, analyzed and implemented as sequential constructs.

The four *reorganization skeletons* are **zip**, **unzip**, **makearray** and **append**. They allow the programmer to create and restructure vectors. The skeleton **zip** transforms a pair of vectors into a vector of pairs. The skeleton **unzip** transforms a vector of pairs into a pair of vectors. The skeleton **append** appends its two vector arguments. The skeleton **makearray** a b creates a vector of size a of elements b, for example:

$$\textbf{makearray}\ 5\ 1\ =\ [1;1;1;1;1]$$

There are seven *communication skeletons* which describe families of data motion within vectors. When applied to a vector representing the parallel machine (as in $\mathscr{L}_6$ programs), these motions will denote (and will be implemented as) communications. They have been chosen because of their availability on parallel computers as hard-wired or optimized communication routines. The first three have type $\mathit{Vect}\ n\ \alpha \rightarrow \mathit{Vect}\ n\ \alpha$ (i.e. they map a vector of size n to a vector of the same size):

- **brdcast** e v returns a vector whose elements are equal to the $e+1$th element of v. It can be expressed in Haskell as follows:

```
-- brdcast i [a0;..;ai;..;an] = [ai;...;ai]
brdcast e v = array (0,n) [(i,v!e) |i<-[0..n]]
              where n = sizeRange (bounds v) - 1
```

- **transfer** s d v returns a vector identical to v except for the $d+1$th element that has the value of the $s+1$th element of v.
- **rotate** d v returns a vector whose elements are equal to the elements of v but shifted circularly of d positions to the right.

The next three communication skeletons have type $\mathit{Vect}\ m\ (\mathit{Vect}\ n\ \alpha) \rightarrow \mathit{Vect}\ m\ (\mathit{Vect}\ n\ \alpha)$.

- **gather** i m returns a matrix such that its $i+1$th row is equal to the $i+1$th column of m and other elements are equal to m's,
- **scatter** i m returns a matrix such that its $i+1$th column is equal to the $i+1$th row of m and other elements are equal to m's,
- **allgather** m returns the transpose of m.

Finally, **allbrdcast** v has type $\mathit{Vect}\ n\ \alpha \rightarrow \mathit{Vect}\ n\ (\mathit{Vect}\ n\ \alpha)$ and returns a vector whose elements are equal to v.

The *mask skeletons* are the only form of conditional provided by $\mathscr{L}_1$. They are a family of data-parallel skeletons, written $\textbf{poly}_n$, where n is less or equal to the number of dimensions of its vector argument. For example, $\textbf{poly}_1$ applies on a vector of type $\mathit{Vect}\ n\ \alpha$ and $\textbf{poly}_k$ on a vector of type $\mathit{Vect}\ n_1\ (\ldots(\mathit{Vect}\ n_k\ \alpha)\ldots)$. The skeleton $\textbf{poly}_k\ p\ f_1\ f_2\ v$ applies the function f_1 to the elements of v contained in the k-dimensional polytope[2] described by the predicate p and the function f_2 on the elements outside the polytope. For example, $\textbf{poly}_2$, which takes a matrix (vectors of vectors) as argument, can be defined in Haskell as follows:

[2] An *n-dimensional polytope* is a finite *n-dimensional polyhedron*. An *n-dimensional polyhedron* is a set of points whose integer coordinates $(i_1,\ldots,i_n)$ satisfy a set of inequalities between affine expressions. An *affine expression* has the form $a_1\,x_1+\ldots+a_n\,x_n+a_{n+1}$ where x_i denotes a variable and a_i a constant.

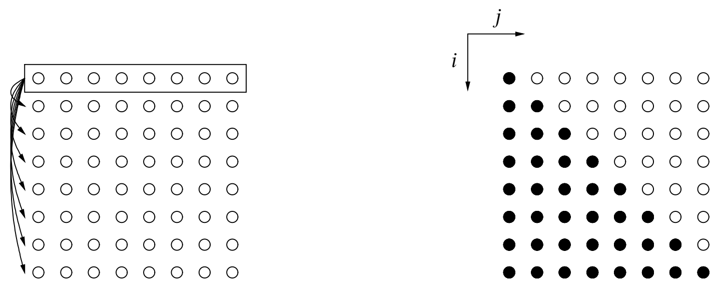

Fig. 2. Data motion (**brdcast** 0) and mask skeleton (**poly**$_2$ ($\lambda(i, j).(i < j)$)).

```
poly2 p f g v
  = array (0,n)
      [(i,array (0,m) [(j,if p(i,j) then f(v!i!j) else g(v!i!j))
                          | j<-[0..m]])| i<-[0..n]]
    where n = sizeRange (bounds v) - 1
          m = sizeRange (bounds (v!0) - 1
```

Note also that **poly**$_1$ $(\lambda i.i = i)$ f Id is equivalent to **map** f.

To enable a precise symbolic cost analysis, additional syntactic restrictions are necessary. The scalar arguments of communication skeletons, of the **iterfor** operator, of mask skeletons, and of **makearray** must be affine expressions of variables denoting iterator indexes or sizes. This restriction is formalized by the nonterminal LinF$_1$, which defines affine expressions of such variables. We rely on the type system with simple subtyping (described in the next section) to ensure that variables in LinF$_1$ expressions are only index or size variables.

Also, if the grammar makes the first order restriction clear, it does not restrict the use of recursion. The constraint that user-defined functions are not recursive must be checked separately (i.e. check that the call graph is acyclic).

Example 1
The following simple program will be used throughout the paper to illustrate the compilation steps:

$$\begin{array}{l} f\ m\ \ \textbf{where} \\ m :: Vect\ n\ (Vect\ n\ (Float, Float)) \\ f = \lambda m.(\textbf{poly}_2\ (\lambda(i,j).(i<j))\ (+)\ (-)\ (\textbf{brdcast}\ 0\ m), m) \end{array}$$

where m is a matrix of pairs of integers.

First, **brdcast** 0 m builds a matrix made of copies of the first row of m (the data movements are depicted by arrows in the left matrix of figure 2). Then, the pairs belonging to the upper triangle of the matrix are summed (represented by white dots in the right matrix of figure 2) and those belonging to the lower triangle (represented by black dots) are subtracted (**poly**$_2$ $(\lambda(i, j).(i < j))$ $(+)$ $(-)$). The result is a pair composed of the computed matrix and the initial one.

The interested reader will find other examples of $\mathscr{L}_1$ programs in Appendix C (namely, LU decomposition and the n-body problem).

3.2 *The language* $\mathscr{L}_6$

The target language $\mathscr{L}_6$ expresses SPMD computations. A program acts on a single vector (which is given the special type *Vectproc*) whose elements represent the processors or, more precisely, their local data spaces. An $\mathscr{L}_6$ program is a composition of parallel computations separated by communications. Typically, a program has the following form:

$$\ldots \textbf{pimap}\ f \circ \mathit{Comm} \circ \textbf{pimap}\ g \ldots$$

where **pimap** is a version of **map** that makes the processor index accessible to its function. The language $\mathscr{L}_6$ introduces new versions of skeletons acting on the vector of processors. In particular, the new versions of communication skeletons express the same data motions as before, but on the vector of processors. They model effective communications and will be implemented as such.

$$\begin{array}{lcl}
\mathrm{Prog}_6 & ::= & \mathrm{FunComm}_6 \\
\mathrm{FunComm}_6 & ::= & \mathrm{FunComm}_6 \circ \mathrm{Comm}_6 \ \mid \mathrm{Comm}_6 \circ \mathrm{FunComm}_6 \\
 & \mid & \textbf{piterfor}\ \mathrm{FunComm}_6 \ \mid \textbf{pimap}\ \mathrm{Fun}_6 \ \mid \mathrm{Comm}_6 \\
\mathrm{Comm}_6 & ::= & \textbf{pbrdcast} \ \mid \textbf{ptransfer} \ \mid \textbf{protate} \\
 & \mid & \textbf{pscatter} \ \mid \textbf{pgather} \ \mid \textbf{pallgather} \ \mid \textbf{pallbrdcast} \\
\mathrm{Exp}_6 & ::= & \mathrm{Fun}_6\ \mathrm{Exp}_6 \ \mid (\mathrm{Exp}_6,\ldots,\mathrm{Exp}_6) \ \mid x \ \mid k \\
\mathrm{Fun}_6 & ::= & \textbf{iterfor}_{\textbf{uc}}\ \mathrm{Fun}_6 \ \mid \mathrm{Op}_6 \ \mid \lambda(x_1,\ldots,x_n).\mathrm{Exp}_6 \\
 & \mid & \mathrm{CompSkel}_6 \ \mid \mathrm{ReorgSkel}_6 \ \mid \mathrm{CommSkel}_6 \ \mid \mathrm{MaskSkel}_6 \\
\mathrm{Op}_6 & ::= & + \ \mid - \ \mid \ldots \ \mid \textbf{dealloc} \ \mid \textbf{copy} \ \mid \textbf{update} \mid \textbf{lookup} \\
\mathrm{CompSkel}_6 & ::= & \textbf{map}\ \mathrm{Fun}_6 \ \mid \textbf{fold}_{\textbf{uc}}\ \mathrm{Op}_6 \ \mid \textbf{scan}_{\textbf{uc}}\ \mathrm{Op}_6 \\
\mathrm{ReorgSkel}_6 & ::= & \textbf{zip} \ \mid \textbf{unzip} \ \mid \textbf{append} \ \mid \textbf{makearray}_{\textbf{uc}} \\
\mathrm{CommSkel}_6 & ::= & \textbf{brdcast}_{\textbf{uc}} \ \mid \textbf{transfer}_{\textbf{uc}} \ \mid \textbf{rotate}_{\textbf{uc}} \ \mid \textbf{scatter}_{\textbf{uc}} \\
 & \mid & \textbf{gather}_{\textbf{uc}} \ \mid \textbf{allgather}_{\textbf{uc}} \ \mid \textbf{allbrdcast}_{\textbf{uc}} \\
\mathrm{MaskSkel}_6 & ::= & \textbf{poly}_n\ \lambda(x_1,\ldots,x_n).\mathrm{Ineq}_6\ \mathrm{Fun}_6\ \mathrm{Fun}_6 \\
\mathrm{Ineq}_6 & ::= & \mathrm{Ineq}_6 \wedge \mathrm{Ineq}_6 \ \mid \mathrm{LinF}_6 < \mathrm{LinF}_6 \ \mid \mathrm{LinF}_6 = \mathrm{LinF}_6 \\
\mathrm{LinF}_6 & ::= & \mathrm{LinF}_6 + \mathrm{LinF}_6 \ \mid \mathrm{LinF}_6 - \mathrm{LinF}_6 \\
 & \mid & k{*}\mathrm{LinF}_6 \ \mid p{*}\mathrm{LinF}_6 \ \mid n{*}\mathrm{Expi}_6 \ \mid x \ \mid k \\
\mathrm{Expi}_6 & ::= & \textbf{div}\ (\mathrm{Expi}_6,\ p) \ \mid \textbf{mod}\ (\mathrm{Expi}_6,\ p) \ \mid ip
\end{array}$$

$x, x_1, \ldots, x_n \in$ VarIdent. $ip \in$ ProcIdent. $p \in$ ProcNb. $k \in$ Constant. $n \in$ SizeIdent.

Fig. 3. Skeleton language $\mathscr{L}_6$.

The syntax of $\mathscr{L}_6$ is defined figure 3. A program is a parallel function FunComm$_6$ applied to a vector of processors. A parallel function is either the composition ($\circ$) of a parallel function with a communication, an iteration **piterfor** LinF$_6$ FunComm$_6$ (which applies LinF$_6 + 1$ times the parallel computation FunComm$_6$ to the vector of processors), or a parallel computation **pimap** Fun$_6$ (which applies Fun$_6$ on each processor). The skeleton **pimap** can be defined as:

```
-- pimap f [a0;...;ap] = [f(0,a0) ;...;f(p,ap)]
pimap f proc = array (0,p) [(i,f(i,(proc!i)))|i<-[0..p]]
            where  p = sizeRange(bounds proc) - 1
```

The nonterminal Comm_6 gathers together the new versions of the communication skeletons (**pbrdcast**, **ptransfer**, **protate**, **pgather**, **pscatter**, **pallgather** and **pallbrdcast**). They describe the same data motion as before but on the vector of processors. The previous communication skeletons remain in $\mathscr{L}_6$ in an uncurried version ($\mathbf{comm_{uc}}$). They are used locally on the processors (they occur only inside **pimap**s) and represent local data motion.

Let us describe the communication skeleton **pbrdcast**. Before the call of this communication, each local processor memory is a 3-tuple of the form (index of the broadcasting processor, value, local memory). The broadcasting processor takes the value (second element of its triple) and sends it to the other processors. The resulting vector of processors has elements of the form (broadcast value, local memory). Its functional semantics in HASKELL is:

```
-- pbrdcast [(e,a0,b0);...;(e,ap,bp)] = [(ae,b0);...;(ae,bp)]
pbrdcast proc = array (0,p) [let (_,_,md) = proc!i
                               in (i,(buf,md))|i<-[0..p]]
                where  e   = first(proc!0)
                       buf = snd(proc!e)
                       p   = sizeRange (bounds proc) - 1
```

The sequential skeletons are similar to $\mathscr{L}_1$. The only differences lie in the arguments of mask skeletons and new versions of communication and reorganization skeletons. The polytope argument of the mask skeleton may now include modulo and integer division (as expressed by the nonterminals LinF_6 and Expi_6 in $\mathscr{L}_6$). Uncurried versions of the communication and reorganizing skeletons are introduced (e.g. $\mathbf{brdcast_{uc}}(e, v) = \mathbf{brdcast}\ e\ v$). Furthermore, Op_6 includes four new functions: explicit vector copy and deallocation (**copy**, **dealloc**), **lookup** which accesses one element of its vector argument ($\mathbf{lookup}\ (e, v) = v!e$) and **update** which modifies one vector element:

```
update(e,x,v)
  = array (0,n) [(i,if i = e then x else v!i) | i<-[0..n]]
  where n = sizeRange (bounds v) - 1
```

The implementation of $\mathscr{L}_6$ programs is relatively straightforward. Functions inside **pimap**s represent the sequential programs to be executed by all processors in parallel. They will be compiled into sequential code using standard techniques. Communication skeletons (such as **pbrdcast**) will be implemented as calls to a standard communication library.

Example 2
If the rows of the matrix m are distributed by blocks on the processors, our simple example is transformed into the following $\mathscr{L}_6$ program:

$$\begin{aligned}&\mathbf{pimap}\ \lambda(ip,(buf,x)).(\mathbf{poly}_2\ (\lambda(i,j).ip * b + i < j)\ (+)\ (-)\\ &\qquad\qquad\qquad\qquad \circ\ \mathbf{makearray_{uc}}(n, buf), x)\\ &\circ\ \mathbf{pbrdcast}\circ\mathbf{pimap}\ \lambda(ip,x).(0,\mathbf{lookup}(0,x),x)\end{aligned}$$

where m is supposed to have n rows and b is the block size (i.e. n divided by the number of processors).

The first parallel computation (**pimap** $\lambda(ip, x).(0, \ldots)$) prepares the communication (**pbrdcast**) which broadcasts the first row of the matrix to all the processors. Then, the vector creation ($\mathbf{makearray_{uc}}$) and the mask skeleton ($\mathbf{poly}_2$) are applied locally in parallel.

4 Type and size inference

Type/size inference has several objectives:

- It checks that programs are well typed.
- It ensures that all vector computations are well defined. For example, **brdcast** is defined only if its index argument is within the bounds of its vector argument.
- It makes sure that some arguments are affine expressions depending only on vector sizes and iterator indexes.
- It computes the symbolic size of each vector expression.

Size inference can be seen both as a static analysis to compute the size of each vector-typed expression, and as a type system to enforce constraints. The static and symbolic evaluation of sizes is made possible by several restrictions:

1. the argument and result of **iterfor** have the same size;
2. indexes and sizes, used by accesses and creations, are affine expressions;
3. vectors are homogeneous, i.e. all the elements of a vector have the same size.

Types are represented as *size types* whose syntax is described in figure 4. Vectors are associated with their size and basic types are either scalar types (integers *Int*, floating point numbers *Float*, or booleans *Bool*), *Size* or *Index* types. Intuitively, an expression has type *Size* if it is a constant, the size of an input vector, or an affine expression of *Size* variables. Similarly, an expression has type *Index* if it is an iterator index, has type *Size* or is an affine expression of *Index* variables. *Size* and *Index* types bear an affine expression denoting their symbolic value.

For example, the size type $Size^n \to Vect\ n\ Int$ indicates that the function takes an integer n and returns a vector of size n.

$$
\begin{array}{lcl}
\mathrm{T} & ::= & \mathrm{T_{Exp}} \to \mathrm{T_{Exp}} \\
\mathrm{T_{Exp}} & ::= & (\mathrm{T_{Exp}}, \mathrm{T_{Exp}}) \mid \mathit{Vect}\ \mathrm{A}\ \mathrm{T_{Exp}} \mid \alpha \mid \mathrm{B} \\
\mathrm{B} & ::= & \mathit{Int} \mid \mathit{Float} \mid \mathit{Bool} \mid \mathit{Index}^{\mathrm{A}} \mid \mathit{Size}^{\mathrm{A}} \\
\mathrm{A} & ::= & \mathrm{A} + \mathrm{A} \mid \mathrm{A} - \mathrm{A} \mid k{*}\mathrm{A} \mid x \mid k
\end{array}
$$

$x \in$ SizeVar, $\alpha \in$ SizeType and $k \in$ Constant.

Fig. 4. Size types.

The type system integrates subtyping based on the following hierarchy:

$$Size^a \subseteq Index^a \subseteq Int \subseteq Float$$

The subtype relation $Size^a \subseteq Index^a$ indicates that loop indexes may depend on sizes but vector sizes cannot be defined in terms of loop indexes. The subtyping rules for functions, pairs, etc. are the usual ones. The inference rules, described in figure 5, are of the form

$$C, \Gamma \vdash e : T, C_1$$

which means that e has size type T with size constraints C_1 in the environment Γ with the subtyping constraints C. If the size constraints C_1 are satisfied by the input parameters then the evaluation of e will not produce any vector access error.

Some rules just express unification in terms of an equality constraint between size types. For example, in the rule [APPL], the constraint $\alpha = \gamma$ unifies the actual and formal parameters. Some others adds constraints on sizes.

- The rule [CONST] gives the smallest (most precise) type according to the subtyping relation (i.e. *Size*) to constants.
- The type of an addition or a subtraction of affine expressions (rule [LINF$_1$]), is the most general type of its two subexpressions. Further, it introduces a constraint denoting the value of the result as an affine expression. For example, the expression $(n + 1)$, where n is a vector size of type $Size^n$, has the size type $Size^s$ along with the constraint $\{s = n + 1\}$. Note that the rule for other operators (e.g. multiplication) must be different since they do not preserve affinity in general.
- The rule [BRD] forces the first argument of **brdcast** to have type *Index*. It also introduces the constraints $(0 \leqslant s_1 \leqslant s)$, which force this index to be within the bounds of the second (vector) argument. For example, the typing of **brdcast** $5\ v$, where v has the size type *Vect n Int*, produces the constraint $\{5 \leqslant n\}$ to ensure that the index of the broadcast element lies within the bounds of v.
- The rule [POLY2] states that the polytope must be defined using arguments of type *Index*. It also ensures the two functions have the same type (including sizes if they have a vector argument) ($\alpha_1 = \alpha_2$ and $\beta_1 = \beta_2$).

As described, the type system is monomorphic. This can be too harsh a limitation, since user-defined functions could be used only for specific vector sizes. However, since user-defined functions are not recursive, all the calls can be replaced by the corresponding function definitions. Such a preliminary unfolding amounts to making the type system polymorphic.

The rules in figure 5 constitute a small representative subset of the inference system. The rules for other skeletons are similar, and the rules for subtyping are the usual ones. The inference rules can be turned into an algorithm using standard techniques of subtype inference (e.g. Mitchell, 1991) and of polyhedric computations (e.g. Wilde, 1993).

Satisfiability of the size constraints is made easy by the fact that constraints are inequalities between affine expressions. The solutions of a set of such constraints can be seen as the points of a convex polyhedron. The dimensions of this polyhedron are the variables denoting sizes and occurring in the inequalities. These variables can be the size variables introduced by the inference (denoted by s_i in the rules), or the symbolic sizes used to express input vector sizes. If $s_1, \ldots, s_p$ are the p size variables and $n_1, \ldots, n_q$ the q symbolic sizes occurring in the constraints, the

$$\frac{C,\Gamma \vdash e : \alpha, C_1 \qquad C,\Gamma \vdash f : \gamma \to \beta, C_2}{C,\Gamma \vdash f\ e : \beta, C_1 \cup C_2 \cup \{\alpha = \gamma\}}\,[\text{APP}] \qquad \frac{}{C,\Gamma \cup \{y : \alpha\} \vdash y : \alpha, \{\}}\,[\text{VAR}]$$

$$\frac{C,\Gamma \vdash e_1 : \alpha_1, C_1 \qquad C,\Gamma \vdash e_2 : \alpha_2, C_2}{C,\Gamma \vdash (e_1, e_2) : (\alpha_1, \alpha_2), C_1 \cup C_2}\,[\text{PAIR}] \qquad \frac{}{C,\Gamma \vdash k : \mathit{Size}^k, \{\}}\,[\text{CONST}]$$

$$\frac{\forall k \in [1,2]\ C,\Gamma \vdash e_k : \alpha^{s_k}, C_k \quad C \vdash \alpha \subseteq \mathit{Index}}{C,\Gamma \vdash e_1\ \mathit{Op}\ e_2 : \alpha^{s_3}, C_1 \cup C_2 \cup \{s_1\, \mathit{Op}\, s_2 = s_3\}}\,[\text{LINF}_1]$$

where $Op \in \{+,-\}$

$$\frac{C,\Gamma \vdash e : \beta, C_1 \qquad C \vdash \beta \subseteq \alpha}{C,\Gamma \vdash e : \alpha, C_1}\,[\text{COER}]$$

$$\frac{C,\Gamma \vdash e : \mathit{Index}^{s_1}, C_1}{C,\Gamma \vdash \textbf{brdcast}\ e : \mathit{Vect}\ s\ \alpha \to \mathit{Vect}\ s\ \alpha, C_1 \cup \{0 \leqslant s_1 < s\}}\,[\text{BRD}]$$

$$\frac{\forall k \in [1,2]\ C,\Gamma \cup \{i : \mathit{Index}, j : \mathit{Index}\} \vdash e_k : \mathit{Index}, \_ \quad C,\Gamma \vdash f_k : \alpha_k \to \beta_k, C_k}{\begin{array}{r} C,\Gamma \vdash \textbf{poly}_2\ \lambda(i,j).e_1 \leqslant e_2\ f_1\ f_2 : \mathit{Vect}\ s_1\,(\mathit{Vect}\ s_2\,\alpha_1) \to \mathit{Vect}\ s_1\,(\mathit{Vect}\ s_2\,\beta_1), \\ C_1 \cup C_2 \cup \{\alpha_1 = \alpha_2, \beta_1 = \beta_2\} \end{array}}\,[\text{POLY2}]$$

Fig. 5. Size inference (extract).

tuple $(s_1, \ldots, s_p, n_1, \ldots, n_q)$ belongs to the convex polyhedron defined by the affine inequalities.

Satisfiability amounts to normalizing the constraints to get only inequalities of affine expressions, and checking that the obtained polyhedron is not empty. The projection of the polyhedron on $(n_1, \ldots, n_q)$ defines the conditions that the input vector sizes must fulfil to prevent access errors. The projection can detect unavoidable access errors (the projection is empty) or guarantee their absence without further conditions (the projection is the complete q-dimensional space).

The solutions of a size variable s are found by projecting the polyhedron on the dimensions n_i of the symbolic sizes and the dimension corresponding to s. Each size variable is expressed as an affine expression of symbolic sizes, and we use size types to annotate each program expression. Note that testing for emptiness and projection are standard operations of polyhedric libraries (e.g. Wilde, 1993).

Example 3

Let us illustrate the size analysis on our small example:

$$(\textbf{poly}_2\ (\lambda(i,j).i < j)\ (+)\ (-)\ (\textbf{brdcast}\ 0\ m), m)$$

We have

$$\begin{aligned}
&m :: \mathit{Vect}\ n\ (\mathit{Vect}\ n\ (\mathit{Float}, \mathit{Float}))\\
&\mathbf{poly}_2\ (\lambda(i,j).i < j)\ (+)\ (-)\ : \mathit{Vect}\ s_1\ (\mathit{Vect}\ s_2\ (\mathit{Float}, \mathit{Float}))\\
&\qquad\qquad\qquad\qquad\qquad\qquad \to \mathit{Vect}\ s_1\ (\mathit{Vect}\ s_2\ \mathit{Float})\\
&\mathbf{brdcast}\ 0\ m : \mathit{Vect}\ s_3\ (\mathit{Vect}\ s_4\ (\mathit{Float}, \mathit{Float}))
\end{aligned}$$

with the following set of constraints:

$$\{s_1 = s_3, s_3 = n, s_2 = s_4, s_4 = n, 0 \leqslant 0 \leqslant s_2\}$$

By projecting on the n dimension we get $s_1 = n, s_2 = n, s_3 = n, s_4 = n$ and, for example, the mask skeleton has the size type:

$$\begin{aligned}
\mathbf{poly}_2\ (\lambda(i,j).i < j)\ (+)\ (-) : \mathit{Vect}\ n\ (\mathit{Vect}\ n\ (\mathit{Float}, \mathit{Float}))\\
\to \mathit{Vect}\ n\ (\mathit{Vect}\ n\ \mathit{Float})
\end{aligned}$$

5 Update-in-place analysis

As we use a pure functional language and want to manipulate vectors as efficiently as possible, we have to tackle the update-in-place problem. The solution must be precise and fully automatic. However, since this step may have a tremendous impact on runtime costs, it should also yield useful feedback and allow programmers to keep complete control of their programs. As a consequence, the update-in-place analysis is the first step of our compilation chain. The programmer is only able to read and change programs at this stage. Indeed, the next transformations (e.g. the distribution, section 7) considerably change the source program, which may become incomprehensible for the programmer.

Our approach relies on extracting an abstract representation of possible execution traces, which are then used to check whether all vectors can be modified in place or not. The abstract execution trace represents the vector access sequences carried out during the program evaluation. If these access sequences do not satisfy some criteria, two possibilities arise:

1. **copy** operations are automatically inserted in the program so that the criteria hold. Functionally, this new operator is the identity function, but will be implemented by copying its vector argument.
2. The faulty subexpressions and violated criteria are pointed out to the programmer, who may insert copies manually or restructure the program before a new analysis.

The same abstract sequences are used to ensure that that all vectors are either returned as result or explicitly deallocated (using a new operation, **dealloc**). In the following, we present the update-in-place analysis (at a fairly high and intuitive level), and return to the deallocation problem at the end of this section.

To check the updated in place property, we statically compute a runtime trace which corresponds to the sequence of vector accesses during execution of the program. This information is similar to the sequences calculated in Kastens and Schmidt (1986) which, in general, are described by grammars.

Each vector is annotated by labels (e.g. l_1) which serve to represent sharing. The basic vector accesses may be a read (l_1^r denotes a read of the vector with label l_1), a write (${}^{l_2}l_1^w$) which produces a vector annotated with a fresh label (here l_2), a copy (${}^{l_2}l_1^c$ where l_2 is a fresh label), a vector deallocation (l_1^f), or vector display (l_1^p) which indicates that the vector is returned in the final result. A concrete access sequence is either a sequential composition of sequences ($EV_2 \circ EV_1$, where the accesses of EV_1 are made before the accesses of EV_2), a parallel composition ($EV_1 \| EV_2$ which represents any interleaving of EV_1 and EV_2 accesses), or a vector access.

To represent finitely all the possible sequences of accesses, abstract access sequences are described as regular expressions of concrete sequences. An abstract sequence can be a concrete sequence, the union of abstract sequences $EV_1 + EV_2$ (i.e. EV_1 or EV_2), or the repeated composition of abstract sequences $(EV_1 \circ)^* EV_2$ (i.e. EV_2 followed by zero or more occurrences of EV_1). Mallet (1998a) presents computation of the access sequences as an abstraction of an instrumented semantics for the language $\mathscr{L}_1$. This allows us to prove the correctness of the static analysis with respect to the instrumented semantics.

Intuitively, vectors can be safely updated in place if no access of the form ${}^{l_2}l_1^w$, l_1^f, or l_1^p is followed by an access on the form l_1^r, ${}^{l_2}l_1^c$, ${}^{l_2}l_1^w$, l_1^f, or l_1^p in the program access sequence. In other words, after a write to a vector (say, labeled l_1) there should not be accesses to the previous version (i.e. l_1). If this holds, it is clear that the update/write can be done in place.

We say that an access follows another access if they appear, either in a sequential composition ($\circ$) or, since it does not enforce any specific order, in a parallel composition ($\|$). The update-in-place condition amounts to checking for each created, copied or modified vector that no access follows a write on this same vector. Let EV be the access sequence of the program, the condition is expressed formally by

$$\mathscr{U}p(EV) = \forall l. \mathscr{U}p_l(EV)$$

The condition for a specific vector ($\mathscr{U}p_l$) is defined in figure 6. The condition $\mathscr{U}p_l$ for an access sequence $EV_1 \circ EV_2$, and a l-labeled vector, is that either the vector is not written in EV_2 and the condition holds on EV_1, or the vector is written in EV_2 and the vector is not accessed in EV_1, and the condition holds on EV_2. In the parallel composition case, the condition is that either one sequence does not contain an access to l and the condition holds for the other, or no write to l occurs in both sequences. Finally, the condition $\mathscr{U}p_l$ holds on the sequence $EV_1 + EV_2$ if it holds on both sequences EV_1 and EV_2. For a sequence of the form $(EV_1 \circ)^* EV_2$, it holds if either the vector is not accessed in EV_2 and the condition holds in EV_1, either the vector is not written in EV_1 and EV_2.

Programs can be automatically transformed so that they respect the property $\mathscr{U}p$. Any sequence not satisfying the property $\mathscr{U}p$ contains accesses to a vector (annotated by) l following a write to l. Since each access is associated to a unique operation, the faulty writes are easy to find. To ensure the update in place property, it is sufficient to insert an explicit **copy** before each such write operation in the program.

Another possibility is to let the user restructure the program in order to make the program respect the property. This way, the user keeps complete control of the costs

$$\begin{array}{lcl}
\mathscr{U}p_l(EV_1{\circ}EV_2) & = & (\mathscr{U}p_l(EV_2)\wedge\ \nexists\, l^{\mathscr{A}} \in EV_1)\ \vee(\mathscr{U}p_l(EV_1)\wedge\ \nexists\, l^{\mathscr{W}} \in EV_2) \\
\mathscr{U}p_l(EV_1\|EV_2) & = & (\mathscr{U}p_l(EV_2)\wedge\ \nexists\, l^{\mathscr{A}} \in EV_1)\ \vee(\mathscr{U}p_l(EV_1)\wedge\ \nexists\, l^{\mathscr{A}} \in EV_2 \\
 & & \vee(\ \nexists\, l^{\mathscr{W}} \in (EV_1\|EV_2)) \\
\mathscr{U}p_l(EV_1{+}EV_2) & = & \mathscr{U}p_l(EV_1) \wedge \mathscr{U}p_l(EV_2) \\
\mathscr{U}p_l((EV_1\circ)^{*}EV_2) & = & (\ \nexists\, l^{\mathscr{W}} \in EV_1\wedge\ \nexists\, l^{\mathscr{W}} \in EV_2) \vee (\mathscr{U}p_l(EV_2)\wedge\ \nexists\, l^{\mathscr{A}} \in EV_1)) \\
\mathscr{U}p_l(\_) & = & \text{true} \qquad \textbf{otherwise}
\end{array}$$

$$\text{with } \mathscr{A} \in \{p,c,r,w,f\}, \mathscr{W} \in \{w,f,p\}.$$

Fig. 6. Conditions on sequences of accesses.

induced by the program and may potentially minimize the number of necessary copies. In this case, the access sequence computed by the analysis indicates the operation that violates the condition $\mathscr{U}p$.

Example 4
The analysis of our example

$$(\mathbf{poly}_2\ (\lambda(i,j).i<j)\ (+)\ (-)\ (\mathbf{brdcast}\ 0\ m), m)$$

produces the following access information:

$$({l_3}^{p}\,\|\,l_3'^{p}\,\|\,l_1^{p}\,\|\,l_1'^{p})\circ({}^{l_3}l_2^{w}\,\|\,{}^{l_3'}l_2'^{w})\circ({}^{l_2}l_1^{w}\,\|\,{}^{l_2'}l_1'^{w})$$

where the outer vector of m is annotated by the label l_1, and its inner vectors by l_1'.

The condition $\mathscr{U}p$ is violated for the vector annotated l_1, which is also written and returned as a result. So, an explicit copy is inserted just before the write on l_1 (done by the skeleton **brdcast**) leading to:

$$(\mathbf{poly}_2\ (\lambda(i,j).i<j)\ (+)\ (-)\ (\mathbf{brdcast}\ 0\ (\mathbf{copy}\ m)), m)$$

The functional language community has proposed many different update-in-place analyses. They are either syntactic criteria (Schmidt, 1985), semantics-based analyses (Kastens and Schmidt, 1986; Sestoft, 1989) or type systems (Wadler, 1990; Guzmán and Hudak, 1990). Most of them analyze whether function parameters or instances of specific types can be implemented as a global variable. In our context, we want to check that each vector (taken as input or dynamically created) could be updated in place. The type-based approach of Wadler (1990) meets this requirement, though it was not precise enough to prove that the property still held on programs obtained after the distribution transformation. Our main motivation to design a more precise analysis was to be able to prove that the update-in-place property was preserved by the successive transformations.

The same abstract sequences are used to enforce that vectors are explicitly deallocated. This allows us to avoid the need for a garbage collector, which is particularly important to guarantee that our cost analysis evaluates real runtime costs. On abstract sequences, the criteria are that every vector is either written, deallocated or displayed as a result. Any vector which is no longer accessed must be explicitly deallocated. If the criteria are not satisfied, **dealloc** operations are automatically inserted just after the last use of vectors. One remaining problem is that pairs are allocated in the heap. Since the language $\mathscr{L}_1$ is based on vectors, the memory allocated for pairs is quite small in practice, and our implementation does not include a garbage collector. However, it would be more satisfactory either to explicitly deallocate pairs (as we do for vectors), or to implement them in the runtime stack.

6 Making communications explicit

Recall that the target language corresponds to SPMD code, i.e. a program is a sequence made of local computations (a function applied by each processor) followed by communications (a communication skeleton) between processors. All data appearing in computations must be local to the processors. In programming terms, it means that a function to be executed in parallel must be closed. However, there might be expressions in $\mathscr{L}_2$ including free variables that induce communications not expressed by communication skeletons. For example, in the expression

$$\mathbf{poly}_2\ (\lambda(i,j).i < j)\ (\mathbf{fold}\ e\ (+))\ (\mathbf{fold}\ 0\ (+))\ m$$

the variable e occurs free in an expression $(\mathbf{fold}\ e\ (+))$ supposed to be executed locally on each processor.

The transformation $\mathscr{E}c$ produces expressions in the intermediate language $\mathscr{L}_3$, very similar to $\mathscr{L}_2$, but in which no free variable occurs in the functional arguments of **map** or $\mathbf{poly}_n$. It aims at making parallel functions closed and is related to λ-lifting (Johnsson, 1985). The transformation $\mathscr{E}c$ is defined by local transformations applied iteratively to the program until a fixpoint is reached. In figure 7, which describes the rule for $\mathbf{poly}_2$, C denotes a context and $\mathscr{F}v[\![Fun]\!]$ denotes the free variable of *Fun*.

The skeleton $\mathbf{poly}_2$ is transformed so that its free variables are abstracted. Each element of the matrix argument is now associated with the free variables' values. This is done using the size (type) of the matrix argument $m \times n$ (previously inferred) and a composition of the skeletons **makearray**, **map** and **zip**.

When the free variables are vectors, they must be explicitly deallocated after function application to preserve the explicit deallocation property (section 5).

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathscr{E}c[\![C[\mathbf{poly}_2\ \mathrm{P}\ \mathrm{Fun}_1\ \mathrm{Fun}_2]]\!]} \\
= & C[\lambda v.\mathbf{poly}_2\ \ \mathrm{P}\ (\lambda((x_1,...,x_p),x).\mathbf{dealloc}\ (x_1,\ldots\mathbf{dealloc}\ (x_q,\mathrm{Fun}_1\ x)\ldots))\\
 & \qquad\qquad\quad (\lambda((x_1,...,x_p),x).\mathbf{dealloc}\ (x_1,\ldots\mathbf{dealloc}\ (x_q,\mathrm{Fun}_2\ x)\ldots))\\
 & \quad (\mathbf{map\ zip}\ (\mathbf{zip}\ (\mathbf{makearray}\ m\ (\mathbf{makearray}\ n\ (x_1,...,x_p)),v)))]
\end{array}$$

$$\begin{array}{ll}
\text{with} & \mathbf{poly}_2\ \mathrm{P}\ \mathrm{Fun}_1\ \mathrm{Fun}_2 : \mathit{Vect}\ m\ (\mathit{Vect}\ n\ \alpha) \rightarrow \mathit{Vect}\ m\ (\mathit{Vect}\ n\ \beta)\\
 & \text{and } \mathscr{F}v[\![\mathrm{Fun}_1]\!] \cup \mathscr{F}v[\![\mathrm{Fun}_2]\!] = \{x_1,...,x_p\},\ 1 \leqslant p,\\
 & \text{and } \forall\ 1 \leqslant i \leqslant q, x_i : \mathit{Vect}\ n_i\ a, \forall q+1 \leqslant i \leqslant p, x_i : \alpha \neq \mathit{Vect}\ n_i\ a
\end{array}$$

Fig. 7. Transformation $\mathscr{E}c$ (extract).

Example 5
Assuming that the matrix m has size $n \times n$, the above example

$$\mathbf{poly}_2\ (\lambda(i,j).i<j)\ (\mathbf{fold}\ e\ (+))\ (\mathbf{fold}\ 0\ (+))\ m$$

is transformed into

$$\begin{aligned}&(\lambda v.\mathbf{poly}_2\ (\lambda(i,j).i<j)\ (\lambda(e,v).\mathbf{fold}\ e\ (+)\ v)\ (\lambda(e,v).\mathbf{fold}\ 0\ (+)\ v)\\ &\quad(\mathbf{map}\ \mathbf{zip}\ (\mathbf{zip}\ (\mathbf{makearray}\ n\ (\mathbf{makearray}\ n\ e),v))))\ m\end{aligned}$$

A $n \times n$ matrix of integers e is built and zipped with m. The distribution of the value e to local processors will be made explicit by the next compilation step which distributes matrices and introduces communication skeletons.

7 Distribution

This step is decomposed into three transformations. The transformation $\mathscr{A}bs$ replaces program variables by combinators. $\mathscr{D}ist$ transforms the resulting program according to a distribution choice for each input vector. Afterwards, some optimizations become possible and are described as program transformations. This transformation chain produces SPMD programs, and can be described as

$$\mathscr{L}_3 \xrightarrow{\mathscr{A}bs} \mathscr{L}_4 \xrightarrow{\mathscr{D}ist} \mathscr{L}_5 \xrightarrow{\mathscr{O}pt} \mathscr{L}_6$$

7.1 $\mathscr{A}bs$ transformation

The transformation $\mathscr{A}bs$ is comparable to the abstraction algorithms used to compile the β-reduction with combinators (Turner, 1979). For example, the SKI abstraction algorithm suppresses the variable x from an expression E by transforming E into the expression $[x]E$ such that

$$([x]E)x = E$$

In our case, each expression E is transformed into an expression $\mathscr{A}bs[\![E]\!]\overrightarrow{X}$ such that

$$(\mathscr{A}bs[\![E]\!]\overrightarrow{X})\overrightarrow{X} = (E,\overrightarrow{X})$$

where $\overrightarrow{X}$ is made of nested pairs representing the free variables of E. Initially $\overrightarrow{X}$ represents the program input variables. More generally, $\overrightarrow{X}$ represents the environment which is explicitly threaded throughout the program (taken as the argument and returned as the result by each sub-expression). This is the data structure that will be distributed over the vector of processors. The transformation $\mathscr{A}bs$ also unfolds the program. Each function call is replaced by its definition. A $\mathscr{L}_4$ program is a variable-free, call-less expression.

The rules of $\mathscr{A}bs$ for our sublanguage are given in figure 8. In order to express the transformation, new functions are introduced. First, as can be expected, environment management is expressed using combinators. The family of *restructuring combinators* $\mathbf{extract}^{\overrightarrow{X},\overrightarrow{Y}}$ (defined as $\lambda\overrightarrow{X}.\overrightarrow{Y}$) restructures and accesses the environment. For example, the standard combinators **fst** and **snd** can be expressed as $\mathbf{extract}^{(x,y),x}$ and $\mathbf{extract}^{(x,y),y}$, respectively. The function $\mathbf{ftuple}_2$ takes two functions and a pair, and applies the first (resp. second) function to the first (resp. second) pair component.

Composition $\circ$ $(Fun_1 \circ Fun_2 = \lambda x.Fun_1(Fun_2\ x))$ and a curried pair operator $(\mathbf{pair} = \lambda x.\lambda y.(x,y))$ are introduced. Finally, new uncurried versions of functions are needed (e.g. $\mathbf{brdcast_{uc}}(e,v) = \mathbf{brdcast}\ e\ v$).

The initial call of $\mathscr{A}bs$ on a program $Prog$ is $\mathscr{A}bs[\![Prog]\!]\overrightarrow{X}$ where $\overrightarrow{X}$ are the $Prog$'s free variables. The transformed program is

$$(\mathbf{extract}^{(x,y),x} \circ \mathscr{A}bs[\![Prog]\!]\overrightarrow{X})\overrightarrow{X}$$

where $(\mathbf{extract}^{(x,y),x}$ deletes the threaded environment, and yields the final result.

The $\mathscr{A}bs$ transformation propagates variable values to the places at which they are used. An access to a variable x now returns a pair made of its value extracted from the environment $\overrightarrow{X}$ and the environment itself. For functions with several arguments (e.g. **brdcast**), the initial function is substituted by its version with one argument. For example, the expression **brdcast** $v\ x$ is transformed into $\mathbf{brdcast_{uc}}(v,x)$.

Note that, due to the previous transformation $\mathscr{E}c$, the rule for $\mathbf{poly}_2$ does not apply recursively $\mathscr{A}bs$ on Fun_1 and Fun_2 because they are closed.

$$\begin{array}{lll}
\mathscr{A}bs[\![\text{Fun Exp}]\!]\overrightarrow{X} & = & \mathscr{A}bs[\![\text{Fun}]\!]\overrightarrow{X} \circ \mathscr{A}bs[\![\text{Exp}]\!]\overrightarrow{X} \quad \textbf{if } \text{Fun} \neq \textbf{brdcast} \\
\mathscr{A}bs[\![\textbf{brdcast } \text{Exp}]\!]\overrightarrow{X} & = & \mathbf{ftuple}_2\ \mathbf{brdcast_{uc}}\ \text{Id} \circ \mathbf{extract}^{(a,(b,c)),((a,b),c)} \\
 & & \circ\ \mathbf{ftuple}_2\ \text{Id}\ (\mathscr{A}bs[\![\text{Exp}]\!]\overrightarrow{X}) \\
\mathscr{A}bs[\![(\text{E}_1,\text{E}_2)]\!]\overrightarrow{X} & = & \mathbf{extract}^{((a_1,b_1),(a_2,b_2)),((a_1,a_2),b_1)} \\
 & & \circ\ \mathbf{ftuple}_2\ (\mathscr{A}bs[\![\text{E}_1]\!]\overrightarrow{X})\ (\mathscr{A}bs[\![\text{E}_2]\!]\overrightarrow{X}) \circ \mathbf{extract}^{x,(x,x)} \\
\mathscr{A}bs[\![x]\!]\overrightarrow{X} & = & \mathbf{extract}^{\overrightarrow{X},(x,\overrightarrow{X})} \qquad (x \in \overrightarrow{X}) \\
\mathscr{A}bs[\![k]\!]\overrightarrow{X} & = & \mathbf{pair}\ k \\
\mathscr{A}bs[\![\mathbf{copy}]\!]\overrightarrow{X} & = & \mathbf{ftuple}_2\ \mathbf{copy}\ \text{Id} \\
\mathscr{A}bs[\![\mathbf{poly}_2\ \text{P}\ \text{Fun}_1\ \text{Fun}_2]\!]\overrightarrow{X} & & \\
 & = & \mathbf{ftuple}_2\ (\mathbf{poly}_2\ \text{P}\ \text{Fun}_1\ \text{Fun}_2)\ \text{Id}
\end{array}$$

Fig. 8. Transformation $\mathscr{A}bs$ (extract).

Example 6
Our simple example

$$(\mathbf{poly}_2\ (\lambda(i,j).i < j)\ (+)\ (-)\ (\mathbf{brdcast}\ 0\ (\mathbf{copy}\ m)), m)$$

is transformed into the variable-free expression

$$(\mathbf{ftuple}_2\ (\mathbf{poly}_2\ (\lambda(i,j).i < j)\ (+)\ (-) \circ \mathbf{brdcast_{uc}} \circ \mathbf{pair}\ 0 \circ \mathbf{copy})\ id \\ \circ\ \mathbf{extract}^{x,(x,x)})(m)$$

It duplicates (combinator $\mathbf{extract}^{x,(x,x)}$) its argument, then applies the composition of skeletons ($\mathbf{poly}_2$ ... $\circ$ ... $\circ$ **copy**) to the first component and the identity to the second one (returned as result).

7.2 $\mathscr{D}$ist transformation

A distribution d can be seen as a function restructuring vectors. Intuitively, transforming the program P according to the distribution d amounts to starting from the equivalent program $P \circ d^{-1} \circ d$ and pushing d^{-1} to the left until the program is of the form $d'^{-1} \circ P' \circ d$. After transformation, the program (P') takes and returns a single vector whose elements can be seen as the local memory of each processor. We give to this data structure the special type *Vectproc*.

Distributions

The data distributions define the allocation of data on the processors. They are functions of the type $\alpha \rightarrow$ *Vectproc* $p\ \alpha$, where p is the number of processors.

We consider a fixed set of standard distributions. For a vector, there are three distributions: **block**, **cyc** and **seq**.

- The distribution **block** p breaks up the vector in p blocks of contiguous elements and allocates each block to a processor (e.g. **block** $2\ [1;2;3;4] = [[1;2];[3;4]]$).
- The distribution **cyc** p distributes cyclicly each vector element on the p processors (e.g. **cyc** $2\ [1;2;3;4] = [[1;3];[2;4]]$).
- **seq** yields a vector with a single processor containing the data vector (e.g. **seq** $[1;2;3;4] = [[1;2;3;4]]$).

They can described using the general distribution **dist** defined as

```
dist b p v
  = array (0,p-1)
          [(i,array (0,m) [(j,v!((div j b)*p*b+i*b+mod j b))
                                         |j<-[0..m]]) |i<-[0..p-1]]
       where m = (div (rangeSize (bounds v)) p) - 1
```

This function distributes cyclicly blocks of size `b` on `p` processors. So, let s be the size of v, we have:

- **block** $p\ v =$ **dist** (**div** $s\ p$) $p\ v$,
- **cyc** $p\ v =$ **dist** $1\ p\ v$,
- **seq** $v =$ **dist** $s\ 1\ v$.

A degenerate case is the distribution of a scalar data. The distribution **const** p allocates its scalar argument to each processor in a vector of processors of size p For example, **const** $5\ 0 = [0;0;0;0;0]$.

These distributions can be combined using higher-order functions (**dp** and **de**) to deal with nested vectors and pairs.

The distribution **dp** $d_1\ d_2$ allocates a pair on the processors. The first pair component is allocated according to the distribution d_1, and the second one according to d_2. The result is a vector of processors containing pairs. For example,

$$\mathbf{dp}\ \mathbf{seq}\ \mathbf{seq}\ ([1;2],[10;20]) = [([1;2],[10;20])]$$

The distribution **de** d_1 d_2 allocates the top-level vector according to d_1 and the inner vectors according to d_2. This distribution takes a vector of vectors and returns a processor vector of vectors of vectors. For matrices (vector of vectors), this entails nine different distributions: sequential (**de seq seq**), row block (**de** (**block** p) **seq**), ...

Example 7
A row cyclic distribution on two processors:

$$\mathbf{de}\ (\mathbf{cyc}\ 2)\ \mathbf{seq}\ [[1;2];[3;4];[5;6];[7;8]] = [[[1;2];[5;6]];[[3;4];[7;8]]]$$

A row cyclic, column block distribution on four processors:

$$\mathbf{de}\ (\mathbf{cyc}\ 2)\ (\mathbf{block}\ 2)\ [[1;2];[3;4];[5;6];[7;8]] = [[[1];[5]];[[2];[6]];[[3];[7]];[[4];[8]]]$$

Each distribution defines a bijection. We write **blocki**, **cyci**, **seqi**, **consti**, **dei** and **dpi** for the corresponding inverse distributions.

Transformation rules

The transformation $\mathscr{D}ist$ assumes that $\mathscr{A}bs$ has been applied and that the program is of the form $Fun\,\overrightarrow{X}$ where $\overrightarrow{X}$ represent the program input parameters. Let d be the distribution of input parameters considered, then Fun is rewritten into the equivalent program $Fun \circ d^{-1} \circ d$. We do not consider different choices of distribution for dynamically allocated vectors. Vectors created by $\mathbf{makearray_{uc}}$ are allocated on a single processor and copied vectors (produced by **copy**) are distributed as their copy. Considering other distributions for these vectors would potentially entail communications. In this respect, it is very similar to the problem of redistribution that we mention in the future work section (section 10.3).

Figure 9 presents the transformation rules required by our simple example. It uses functions (**pimap**, **pbrdcast**, **lookup**, etc.) presented along with the target language in section 3.2. The rules propagate the inverse distribution d^{-1} to the left. Note that since that an inverse distribution d^{-1} has type $Vectproc\ p\ \alpha \rightarrow \alpha$, a function Fun having the type $\tau_1 \rightarrow \tau_2$ is transformed into a function $FunComm$ with type $Vectproc\ p\ \tau_1 \rightarrow Vectproc\ p\ \tau_2$.

The rule for the composition $F_1 \circ F_2 \circ di$ consists in transforming first the function $F_2 \circ di$. This yields an equivalent expression of the form $di' \circ F_2'$. Then, the transformation is recursively called on $F_1 \circ di'$. The final expression is of the form $di'' \circ F_1' \circ F_2'$ where F_1' and F_2' are SPMD functions acting on a unique vector of processors.

The transformation of the restructuring combinator $\mathbf{extract}^{x,(x,x)}$ (which duplicates its argument) consists in duplicating the local memory of each processor. The resulting inverse distribution is an inverse distribution that distributes its pair argument identically (**dpi** di di).

The rules for **copy** and the identity function Id consist in applying it to the local memory of each processor. The inverse distribution is unchanged. In the same way, distributing the **pair** k function (where k is a constant) consists in applying the function to each local memory. The resulting inverse distribution distributes the first

$$
\begin{array}{l}
\mathscr{D}ist[\![\mathrm{F}_1 \circ \mathrm{F}_2 \circ \mathrm{di}]\!] \\
\quad = \text{let } \mathrm{di}' \circ \mathrm{F}_2' = \mathscr{D}ist[\![\mathrm{F}_2 \circ \mathrm{di}]\!] \\
\quad\quad \text{in } \mathscr{D}ist[\![\mathrm{F}_1 \circ \mathrm{di}']\!] \circ \mathrm{F}_2' \\
\mathscr{D}ist[\![\mathbf{extract}^{x,(x,x)} \circ \mathrm{di}]\!] \\
\quad = \mathbf{dpi}\ \mathrm{di}\ \mathrm{di} \circ \mathbf{pimap}\ (\mathbf{extract}^{(ip,x),(x,x)}) \\
\mathscr{D}ist[\![\mathrm{Fun} \circ \mathrm{di}]\!] \\
\quad = \mathrm{di} \circ \mathbf{pimap}\ (\mathrm{Fun} \circ \mathbf{extract}^{(ip,x),x}) \qquad \text{where Fun} \in \{\mathbf{copy}, \mathrm{Id}\} \\
\mathscr{D}ist[\![\mathbf{pair}\ k \circ \mathrm{di}]\!] \\
\quad = \mathbf{dpi}\ (\mathbf{consti}\ p)\ \mathrm{di} \circ \mathbf{pimap}\ (\mathbf{pair}\ k \circ \mathbf{extract}^{(ip,x),x}) \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{with di: } \mathit{Vectproc}\ p\ \alpha \rightarrow \alpha \\
\mathscr{D}ist[\![\mathbf{poly}_2\ \mathrm{P}\ \mathrm{F}_1\ \mathrm{F}_2\ \circ \mathbf{dei}\ (\mathbf{blocki}\ p)\ \mathbf{seqi}\ ]\!] \\
\quad = \mathbf{dei}\ (\mathbf{blocki}\ p)\ \mathbf{seqi} \\
\quad\quad \circ\ \mathbf{pimap}\ (\lambda(ip,v).\mathbf{poly}_2\ (\mathrm{P} \circ \lambda(i_1,i_2).(ip * b + i_1, i_2))\ \mathrm{F}_1\ \mathrm{F}_2\ v) \\
\text{with } \mathbf{blocki}\ \ p : \mathit{Vectproc}\ p\ (\mathit{Vect}\ b\ \alpha) \rightarrow \mathit{Vect}\ n\ \alpha \\
\mathscr{D}ist[\![\mathbf{brdcast_{uc}}\ \circ \mathbf{dpi}\ (\mathbf{consti}\ p)\ (\mathbf{dei}\ (\mathbf{blocki}\ p)\ \mathbf{seqi})]\!] \\
\quad = \mathbf{dei}\ (\mathbf{blocki}\ p)\ \mathbf{seqi} \\
\quad\quad \circ\ \mathbf{pimap}\ (\lambda(ip,(buf,(e,v))).\mathbf{brdcast_{uc}}(0,\ (\mathbf{update}\ (0,buf,v)))) \\
\quad\quad \circ\ \mathbf{pbrdcast}\ \circ \mathbf{pimap}\ (\lambda(ip,(e,v)).(\mathbf{div}\ (e,\ b),\ \mathbf{lookup}\ (\mathbf{mod}\ (e,\ b),v),(e,v))) \\
\text{with } \mathbf{blocki}\ p : \mathit{Vectproc}\ p\ (\mathit{Vect}\ b\ \alpha) \rightarrow \mathit{Vect}\ n\ \alpha \\
\mathscr{D}ist[\![\mathbf{ftuple}_2\ \mathrm{F}_1\ \mathrm{F}_2 \circ \mathbf{dpi}\ \mathrm{di}_1\ \mathrm{di}_2]\!] \\
\quad = \text{let } \mathrm{di}'_1 \circ \mathrm{F}'_1 = \mathscr{D}ist[\![\mathrm{F}_1 \circ \mathrm{di}_1]\!] \\
\qquad\quad\ \ \mathrm{di}'_2 \circ \mathrm{F}'_2 = \mathscr{D}ist[\![\mathrm{F}_2 \circ \mathrm{di}_2]\!] \\
\quad\quad \text{in } \mathbf{dpi}\ \mathrm{di}'_1\ \mathrm{di}'_2 \circ \mathscr{F}us[\![(\mathrm{F}'_1,\ \mathrm{F}'_2)]\!]
\end{array}
$$

Fig. 9. Transformation $\mathscr{D}ist$ (extract).

component of its pair argument according to **consti** p (where p is the number of processors) and the second according to the original distribution (di).

The rule for the mask skeleton $\mathbf{poly}_2$ modifies the index in the inequalities describing the polytope (P) according to the distribution. After transformation, each processor applies the mask skeleton to its local vector. The inequalities still define a polytope thanks to the restricted set of distributions. This key property would not hold for more general forms of distribution. Figure 9 presents the transformation rule for a row block distribution only (a more complicated generic rule exists also). Note that the symbolic size computed by the size inference (section 4) is used by this rule. The rule is based on the fact that an element whose indexes are (i_1, i_2) in the distributed matrix has indexes $(ip * b + i_1, i_2)$ in the initial matrix (where ip denotes the processor index and b the size of blocks, that is the number of rows n divided by the number of processors p). For example, the expression

$$\mathbf{poly}_2\ (\lambda(i,j).i < j)\ f\ g$$

applied to an 8×8 matrix returns the matrix given in figure 10 where the function f has been applied on white dots and g on black ones. After a row block distribution on four processors, each processor must apply $\mathbf{poly}_2$ on the elements contained in their local memory The local function f must be applied to the elements (i, j) such that $2 * ip + i < j$.

To give the idea of the transformation for the communication skeleton $\mathbf{brdcast_{uc}}$,

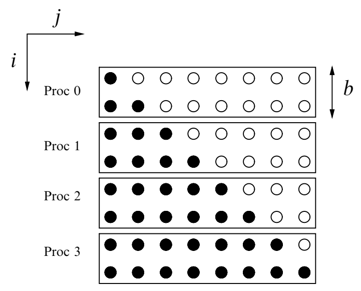

Fig. 10. Impact of a row block distribution on $\mathbf{poly}_2\ (\lambda(i,j).i<j)\ f\ g$

let us consider a specific distribution: **dei** (**blocki** 2) **seqi** i.e. the inverse of a row block distribution on two processors. The transformed expression consists of a local access to the matrix (preparing the communication), a communication **pbrdcast** and a $\mathbf{brdcast_{uc}}$ executed by each processor. For example, let b be the size of blocks, the expression $\mathbf{brdcast_{uc}} \circ \mathbf{dei}\ (\mathbf{blocki}\ 2)\ \mathbf{seqi}$ is transformed into:

$$\begin{aligned}&\mathbf{dei}\ (\mathbf{blocki}\ 2)\ \mathbf{seqi}\\&\circ\ \mathbf{pimap}\ \lambda(ip,(buf,(e,v))).(\mathbf{brdcast_{uc}}(0,\mathbf{update}(0,buf,v)))\\&\circ\ \mathbf{pbrdcast}\ \ \circ\ \mathbf{pimap}\ \lambda(ip,(e,v)).(\mathbf{div}\ (e,b),\mathbf{lookup}\ (\mathbf{mod}\ (e,b),v,(e,v))\end{aligned}$$

The first function builds a triple containing the number of the broadcasting processor, a value **lookup** (**mod** $(e,b),x$) (on the broadcasting processor it will be the value to broadcast), and the local memory. Then, the value is broadcast (**pbrdcast**), bound to *buf*, integrated in the local matrix (**update**) to be copied locally to all the vector elements ($\mathbf{brdcast_{uc}}$).

$$\begin{aligned}&\mathscr{F}us[\![(\mathbf{pimap}\ \mathrm{F}_1,\mathbf{pimap}\ \mathrm{F}_2)]\!]\\&\qquad=\quad \mathbf{pimap}\ (\mathbf{ftuple}_2\ \mathrm{F}_1\ \mathrm{F}_2\circ(\lambda(ip,(x,y)).(\mathrm{F}_1(ip,x),\mathrm{F}_2(ip,y))))\\&\mathscr{F}us[\![(\mathrm{F}_1\circ \mathrm{F}_2,\mathbf{pimap}\ (\lambda(ip,x).x))]\!]\\&\qquad=\quad \mathscr{F}us[\![(\mathrm{F}_1,\ \mathbf{pimap}\ (\lambda(ip,x).x))]\!]\circ\mathscr{F}us[\![(\mathrm{F}_2,\ \mathbf{pimap}\ (\lambda(ip,x).x))]\!]\\&\mathscr{F}us[\![(\mathbf{pbrdcast},\mathbf{pimap}\ (\lambda(ip,x).x))]\!]\\&\qquad=\quad \mathbf{pimap}\ (\lambda(ip,(a,(c,d))).((a,c),d))\circ\mathbf{pbrdcast}\\&\qquad\qquad\circ\ \mathbf{pimap}\ (\lambda(ip,((a,b,c),d)).(a,b,(c,d)))\\&\ldots\\&\mathscr{F}us[\![(\mathrm{F}_1,\mathrm{F}_2)]\!]\\&\qquad=\quad \mathscr{F}us[\![(\mathrm{F}_1,\mathbf{pimap}\ (\lambda(ip,x).x))]\!]\ \circ\mathscr{F}us[\![(\mathbf{pimap}\ (\lambda(ip,x).x),\mathrm{F}_2)]\!]\end{aligned}$$

Fig. 11. Transformation $\mathscr{F}us$ (extract).

The distribution of the function pair $\mathbf{ftuple}_2$ consists in propagating the distributions associated with each function, and merging the resulting SPMD functions into a single parallel function.

Merging is performed by the auxiliary transformation $\mathscr{F}us$ (figure 11). When each function of the pair is a **pimap** then the pair is rewritten into a single **pimap** function. Otherwise, we have to sequentialize the two functions (last rule of figure 11).

Each function is associated with the identity function for the vector of processors (i.e. **pimap** $(\lambda(ip,x).x)$) and composed together. The fusion of a communication function and the identity requires to restructure the local data of the processors before and after the communication operation (to thread the unused element of the pair argument). For example, in the case of a **pbrdcast**, the number of the broadcasting processor and the value to broadcast must be in the first and second components of the local processor memory. The reorganization is carried out by **pimap** $(\lambda(ip,(a,(c,d))).((a,c),d))$ and **pimap** $(\lambda(ip,((a,b,c),d)).(a,b,(c,d)))$.

Example 8
Given a row block distribution **de** (**block** p) **seq**, our example is transformed into the SPMD function:

$$\begin{aligned}&\textbf{pimap}\ \lambda(ip,(x,y)).(\textbf{poly}_2\ (\lambda(i,j).ip*b+i<j)\ (+)\ (-)\ x,y)\\ &\circ\ \textbf{pimap}\ \lambda(ip,(buf,(x,y))).\textbf{brdcast}_{\textbf{uc}}(0,\textbf{update}\ (0,buf,x),y)\\ &\circ\ \textbf{pbrdcast}\ \circ\ \textbf{pimap}\ \lambda(ip,(x,y)).(0,\textbf{lookup}\ (0,x),(x,y))\\ &\circ\ \textbf{pimap}\ \lambda(ip,x).(\textbf{copy}\ x,x)\end{aligned}$$

Given a column block distribution (**de** **seq** (**block** p)), we would get the following function:

$$\begin{aligned}&\textbf{pimap}\ \lambda(ip,(x,y)).(\textbf{poly}_2\ (\lambda(i,j).i<ip*b+j)\ (+)\ (-)\ x,y)\\ &\circ\ \textbf{pimap}\ \lambda(ip,x).(\textbf{brdcast}_{\textbf{uc}}(0,(\textbf{copy}\ x)),x)\end{aligned}$$

7.3 Optimizing transformations

The transformations described in this section aim at simplifying and optimizing local (sequential) and parallel computations. This step is described as a set of local program transformations. For example, the following rules are applied:

- merging of **pimaps**

$$\textbf{pimap}\ F\ \circ\ \textbf{pimap}\ G = \textbf{pimap}\ (\lambda(ip,v).F(ip,G(ip,v)))$$

- merging of **rotates**

$$\textbf{rotate}_{\textbf{uc}}\ \circ\ \textbf{ftuple}_2\ f\ \textbf{rotate}_{\textbf{uc}} = \textbf{rotate}_{\textbf{uc}}\ \circ\ \lambda(e_1,(e_2,v)).(f\ e_1+e_2,v)$$

In the rest of this section, we concentrate on an optimization that removes vector allocations (**copy** or **makearray**$_{\textbf{uc}}$) that have been made useless by the distribution. The impact of this optimization on performances can be considerable.

The local optimization rules are described as rewrite rules in figure 12. In order to preserve the update-in-place property (section 5), these local transformations are applied only if they do not violate a criterion on access sequences. This condition remains implicit in the rules.

There are two classes of rules: transformations propagating the vector allocation to the left of the expression (P.n rules); and transformations performing the elimination (E.n rules). Elimination becomes possible when an allocation has been shifted next to its deallocation.

Propagation transformations amount to delaying vector allocations. The rule [P.1] transforms function applications whose argument contains an allocation.

$$\begin{array}{lll}
\text{Fun}_1(\text{Exp}_1,\ldots,\text{Exp}_{i-1},\text{Fun}_2\ \text{Exp}_i,\ldots,\text{Exp}_n) & & \\
\quad \rightleftharpoons\ \text{Fun}_1 \circ \mathbf{ftuple}_n\ \text{Id} \ldots \text{Id}\ \text{Fun}_2 \ldots \text{Id} & & \\
\quad\quad (\text{Exp}_1,\ldots,\text{Exp}_{i-1},\text{Exp}_i,\ldots,\text{Exp}_n) & & \\
\quad\quad \mathbf{if\ copy},\ \mathbf{makearray_{uc}} \in \text{Fun}_2 & & [\text{P.1}] \\
\mathbf{pimap}\ (\lambda(ip,(x_1,\ldots,x_n)).\text{Exp}) \circ \mathbf{pimap}\ (\mathbf{ftuple}_n\ \text{Fun}_1 \ldots \text{Fun}_n\ \circ \text{Fun}) & & \\
\quad \rightleftharpoons\ \mathbf{pimap}\ (\lambda(ip,(x_1,\ldots,x_n)).\text{Exp}[x_i \mapsto \text{Fun}_i\ x_i]) & & \\
\quad\quad \circ\ \mathbf{pimap}\ \mathbf{ftuple}_n\ \text{Fun}_1 \ldots \text{Id} \ldots \text{Fun}_n\ \circ \text{Fun} & & \\
\quad\quad \mathbf{if}\ \text{Fun}_i \text{ is closed } \mathbf{and\ copy},\ \mathbf{makearray_{uc}} \in \text{Fun}_i & & [\text{P.2}] \\
\mathbf{brdcast_{uc}} \circ \mathbf{ftuple}_2\ \text{Fun}\ \mathbf{copy} & & \\
\quad \rightleftharpoons\ \lambda(e,v).\mathbf{makearray_{uc}}(n,\ \mathbf{lookup}\ (\text{Fun}\ e,v)) & & \\
\quad\quad \text{where } \mathbf{copy} : \mathit{Vect}\ n\ \alpha \rightarrow \mathit{Vect}\ n\ \alpha & & [\text{P.3}] \\
\mathbf{dealloc}\ \circ \mathbf{ftuple}_2\ \mathbf{copy}\ \text{Fun} & & \\
\quad \rightleftharpoons\ \text{Fun} \circ \mathbf{extract}^{(x,y),y} & & [\text{E.1}]
\end{array}$$

Fig. 12. Copy elimination (extract).

For example, the expression

$$\mathbf{dealloc}\ (\mathbf{brdcast_{uc}}(1,x), \mathbf{copy}\ y)$$

is transformed into

$$(\mathbf{dealloc} \circ \mathbf{ftuple}_2\ Id\ \mathbf{copy})(\mathbf{brdcast_{uc}}(1,x), y)$$

The rule [P.2] applies to a composition of **pimap**s and propagates the vector allocation occurring in the first **pimap** into the second one.

The rule [P.3] indicates that copying a vector and broadcasting its ith value is similar to fetching and duplicating the ith value. The **copy** is transformed into a $\mathbf{makearray_{uc}}$ and the allocation is shifted to the left.

The rule [E.1] applies to the case where an allocated vector is immediately freed. It eliminates both the vector allocation and its deallocation.

Example 9

The previous distributed version of our example with a column block distribution which is

$$\begin{array}{l}\mathbf{pimap}\ \lambda(ip,(x,y)).(\mathbf{poly}_2\ (\lambda(i,j).i < ip * b + j)\ (+)\ (-)\ x, y) \\ \quad \circ\ \mathbf{pimap}\ \lambda(ip,x).(\mathbf{brdcast_{uc}}(0, \mathbf{copy}\ x), x)\end{array}$$

is transformed into

$$\begin{array}{l}\mathbf{pimap}\ \lambda(ip,(x,y)).(\mathbf{poly}_2\ (\lambda(i,j).i < ip * b + j)\ (+)\ (-)\ x, y) \\ \quad \circ\ \mathbf{pimap}\ \lambda(ip,x).(\mathbf{makearray_{uc}}(n, \mathbf{lookup}\ (0,x), x))\end{array}$$

using rules [P.1] and [P.3]. The copy of the complete matrix is avoided and replaced by the allocation of a single row.

8 Cost analysis

This step aims at automatically evaluating the cost of $\mathscr{L}_6$ programs in order to select the most efficient distribution. The complexity analysis is based on polytope volume computations (Tawbi, 1994; Pugh, 1994; Clauss, 1996), and yields accurate symbolic costs. This approach is made possible by the restrictions of $\mathscr{L}_1$ and the fixed set of data distributions considered. Together, they guarantee that the cost of all transformed source programs can be expressed as a sum of polytope volumes.

The goal is to find the most efficient $\mathscr{L}_6$ program from among the different distribution choices considered. First, an abstraction function $\mathscr{C}a$ transforms programs into their symbolic parallel cost. Costs are expressed using inequalities, sums and maxima. Then, standard methods to compute the volume of polytopes are applied to get a symbolic cost in polynomial form. Finally, symbolic costs are simplified and compared using a symbolic math package.

8.1 Cost abstraction

The abstraction function $\mathscr{C}a$ extracts cost information from parallel programs. The main rules are shown in figure 13. We use a non-standard notation for indexed sums: we write $\sum_i \left\{\begin{smallmatrix} 0 \leqslant i \\ i \leqslant n \end{smallmatrix}\right\}$ instead of $\sum_{i=0}^{n}$. This notation is needed because polytopes (introduced by the $\mathbf{poly}_n$ skeletons) may be defined by more than two inequalities.

The abstraction relies heavily on the size information present in types (see section 4). The cost of (**pimap** Fun) is the maximum of the costs of Fun on each processor. Communication costs are expressed as polynomials whose constants depend on the target computer. For example, the cost of **pbrdcast** involves the parameters α_{transf} and α_{init} which denote respectively the time of one-word transfer between two processors and the message startup time on the parallel computer considered. Further, we use the function size which returns a polynomial representing the symbolic size of the argument type. Basic arithmetic operators are also given a machine dependent cost (α_{Op}). One has to set those constants to adapt the analysis for a specific parallel machine. The cost of the mask skeleton $\mathbf{poly}_2$ is the sum of the cost of the first function for elements belonging to the polytope $\lambda(i,j).Ineq$, and of the cost of the second one for elements belonging to the complementary (written $\overline{Ineq}$). The complementary polytope is expressed as the difference between the whole (i,j)-domain and the polytope $\lambda(i,j).Ineq$. This is a standard operation of polyhedric libraries that yields a union of polytopes.

The cost expression obtained is then applied to symbolic arguments and reduced to remove all the λ-abstractions. Since we deal with terminating programs, any abstracted expression will have a normal form. Moreover, since costs do not depend upon scalar values the normal form boils down to generalized sums (G-sum) and generalized maxima (G-max). A G-sum is of the form $\sum_{i_1,\ldots,i_n} \{Ineq\}\ Poly$ where $Ineq$ are inequalities made of affine expressions of loop indexes and vector sizes and $Poly$ is a polynomial whose variables are vector sizes or the processor number. A G-max is of the form $\max\limits_{i=0}^{k} E_i$ and denotes the maximum among the expressions $E_0, \ldots, E_k$.

Let us emphasize that writing the source program in $\mathscr{L}_1$ is crucial to get an accurate symbolic cost. First, without a severe limitation of the use of recursion, no precise cost could be evaluated in general. Further, the restrictions imposed by $\mathscr{L}_1$ ensure that the mask skeletons (which limit conditional application to polytope domains), the communication and the computation skeletons all have a complexity

$$
\begin{array}{lcl}
\mathscr{C}a[\![\mathrm{Fun}_1 \circ \mathrm{Fun}_2]\!] & = & (\lambda x.\ \mathscr{C}a[\![\mathrm{Fun}_1]\!](\mathrm{Fun}_2\ x) + \mathscr{C}a[\![\mathrm{Fun}_2]\!]\ x) \\
\mathscr{C}a[\![\mathbf{pimap}\ \mathrm{Fun}]\!] & = & (\lambda x. \max_{i_p=0}^{p-1} \mathscr{C}a[\![\mathrm{Fun}]\!](i_p, x\,!i_p)) \\
\multicolumn{3}{l}{\ \text{where } \mathbf{pimap}\ \mathrm{Fun} : \mathit{Vectproc}\ p\ \alpha \rightarrow \mathit{Vectproc}\ p\ \beta} \\
\mathscr{C}a[\![\mathbf{pbrdcast}]\!] & = & (\lambda x.(\alpha_{transf} * \mathsf{size}\,\beta + \alpha_{init}) * p) \\
\multicolumn{3}{l}{\ \text{where } \mathbf{pbrdcast} : \mathit{Vectproc}\ p\ (\alpha, \beta, \gamma) \rightarrow \mathit{Vectproc}\ p\ (\beta, \gamma)} \\
\mathscr{C}a[\![\mathrm{Op}]\!] & = & (\lambda x.\alpha_{\mathrm{Op}}) \\
\multicolumn{3}{l}{\mathscr{C}a[\![\mathbf{poly}_2\ \lambda(i,j).\mathit{Ineq}\ \mathrm{Fun}_1\ \mathrm{Fun}_2]\!]} \\
 & = & (\lambda x.\sum_{i,j} \left\{ \begin{smallmatrix} 0 \leqslant i < n_1 \\ 0 \leqslant j < n_2 \\ \mathit{Ineq} \end{smallmatrix} \right\} \mathscr{C}a[\![\mathrm{Fun}_1]\!](x\,!i) \\
 & & \qquad + \sum_{\mathit{Ineq}' \in \overline{\mathit{Ineq}}} \sum_i \left\{ \begin{smallmatrix} 0 \leqslant i < n_1 \\ 0 \leqslant j < n_2 \\ \mathit{Ineq}' \end{smallmatrix} \right\} \mathscr{C}a[\![\mathrm{Fun}_2]\!](x\,!i)) \\
\multicolumn{3}{l}{\ \text{where } \mathbf{poly}_2\ \lambda(i,j).\mathit{Ineq}\ \mathrm{Fun}_1\ \mathrm{Fun}_2 : \mathit{Vect}\ n_1\ (\mathit{Vect}\ n_2\ \alpha) \rightarrow \mathit{Vect}\ n_1\ (\mathit{Vect}\ n_2\ \beta)}
\end{array}
$$

Fig. 13. Cost abstraction function $\mathscr{C}a$ (extract).

which depends polynomially upon vector sizes. Their costs can be described as nested sums. Another important restriction is that expressions involving iteration indexes (mask skeletons, communication skeletons and iterfor bounds) are affine. This restriction, expressed by the nonterminal $LinF_1$ in the definition of $\mathscr{L}_1$, along with the standard distributions considered which keep vector accesses affine, ensures that the inequalities of G-sums are affine.

Example 10

The following expression is extracted from our example after a column block distribution.

$$\mathbf{pimap}\ \lambda(ip,(x,y)).(\mathbf{poly}_2\ (\lambda(i,j).i < ip * b + j)\ (+)\ (-)\ x, y)$$

Its cost is expressed as

$$\mathscr{C}_{col} \equiv \max_{i_p=0}^{p-1} \left(\sum_{i,j} \left\{ \begin{smallmatrix} 0 \leqslant j < b \wedge 0 \leqslant i < n \\ i < i_p * b + j \end{smallmatrix} \right\} \alpha_+ + \sum_{i,j} \left\{ \begin{smallmatrix} 0 \leqslant j < b \wedge 0 \leqslant i < n \\ i_p * b + j \leqslant i \end{smallmatrix} \right\} \alpha_- \right)$$

8.2 Symbolic cost analysis

If the vector sizes and the number of processors are known at compile time, the cost comparison is done directly by evaluating the value of the abstracted cost expression. An enumeration of the polytope points makes it possible to obtain the accurate numerical value of the execution time on the target machine. The best distribution is that whose cost value is the smallest.

For unknown vector sizes, the cost computation consists of a symbolic evaluation which yields a polynomial expression. This computation is decomposed into two transformations: the first one evaluates polytope volumes symbolically by reusing existing techniques; and the second one removes G-maxs by calculating the maximum value of the polynomial.

Parametrized polytope volume computation

The method of Clauss (1996) makes it possible to obtain an accurate computation of the polytopes volumes. The computation principle is based on the fact that such a volume is equal to an extension of polynomials (Ehrhart's pseudo-polynomials), such that the coefficients of its monomials vary according to the remainder of integer division of the symbolic sizes by a constant.

However, this method cannot be simply extended when the processor number is unknown at compile time. In this case, the coefficients of affine expressions in the inequalities can contain an unknown. The volume of this kind of polytope cannot be represented by a pseudo-polynomial.

When we want to keep the number of processors as a parameter, the technique described in Tawbi (1994) can be used. It consists of cutting out the polytope by breaking up the inequalities into several subsets such that each of them contains only two inequalities for each variable and that the lower limit is lower than the upper limit (to rule out null polytopes). After this step, traditional formulae of symbolic summation can be applied and polytope volumes are expressed by polynomials whose variables are symbolic sizes. Integer divisions are approximated by their real division minus a constant representing the average difference between the two operations (e.g. $\lfloor \frac{i}{2} \rfloor$ is approximated by $\frac{i}{2} - \frac{1}{4}$). The evaluated cost is no longer absolutely accurate but the introduced approximations appear to be negligible in practice.

Example 11
Using Tawbi's technique, $\mathscr{C}_{col}$ is simplified into

$$\max_{i_p=0}^{p-1} \left(b^2 (\alpha_+ - \alpha_-)\, i_p + b^2 \left(\frac{\alpha_+}{2} + \frac{\alpha_+}{2} + (p-1)\, \alpha_- \right) \right)$$

G-max removal

Before comparing symbolic costs, G-maxs occurring in cost expressions must be removed. This can be done by computing the maximum value taken by the polynomial on the symbolic interval of the G-max. This calculation is based on the fact that a polynomial defined over a finite interval $[0..n]$ reaches its maximum value at 0, n, or one of the zeros of its derivative. There can be several maximums, depending on conditions on symbolic sizes. A G-max can be rewritten as a collection of polynomials, each polynomial being defined on an interval (i.e. the conditions on sizes) where a zero of the derivative maximizes the original expression. This technique removes G-maxs without any approximations. It may however creates complex symbolic expressions for high-degree polynomials. In such cases, simpler, approximated solutions exist (e.g. considering only the bounds of the interval).

Example 12
With the hypothesis $\alpha_+ \equiv \alpha_-$, the former expression of $\mathscr{C}_{col}$ is simplified into

$$\max_{i_p=0}^{p-1} \left(b^2 (\alpha_+ - \alpha_-)\, i_p + b^2 \left(\frac{\alpha_+}{2} + \frac{\alpha_+}{2} + (p-1)\, \alpha_- \right) \right) = b^2 p \alpha_+$$

8.3 Symbolic cost comparison

The last task is to compare the symbolic costs obtained for different distribution choices. It amounts to computing the symbolic intervals where the difference of two costs (i.e. polynomials) is positive or negative. Symbolic math packages such as Maple (Char *et al.*, 1992) can be used for solving this problem. It may be the case that a distribution is definitely better than another and Maple will determine it. But in general, it will depend upon sizes and the number of processors. In such cases, Maple can be seen as a simplifier that will produce symbolic conditions (e.g. $Cost_1 > Cost_2$ iff $n > p$). The programmer may have to indicate if these conditions are satisfied or not. Another (automatic) solution is to use these conditions as run-time tests which choose between several versions of the program.

For our example, the column block distribution is always better than the row block one because local computations are identical in both cases but the column distribution does not entail any communication between processors.

9 Translation and experiments

9.1 Translation

The SPMD programs produced by the compilation chain are translated into a sequential language with calls to a communication library. In the current implementation, we use the C language and the MPI library (Clarke *et al.*, 1994). These are *de facto* standards which guarantee portability across many different parallel machines.

Since functions are strict, first-order, and vectors are single-threaded (property guaranteed by the update-in-place transformation, section 5), the translation of $\mathscr{L}_6$ programs into C is straightforward. The single program executed by all the processors is made of the local functions (argument of **pimaps**), the loops (**piterfor**) and calls to MPI. The composition ($\circ$) is simply translated, locally, by the sequence (`;`); it does not represent a synchronization barrier. The only synchronization between processors is introduced through parallel communications (e.g. **pbrdcast**).

The transformation $\mathscr{T}ra$ of local functions is a straightforward translation into C. For example, the expression $e1 + e2$ will be translated in

```
{ int tmp1;
  <evaluation of e1>; tmp1 = res;
  <evaluation of e2>;
  res = tmp1+res
}
```

Since functions are strict and, in essence, first order, they are implemented by C functions. Sequential skeletons such as $\mathbf{brdcast_{uc}}$, **copy**, etc. are translated into nested loops. Memory management is implemented using the C library functions `malloc` and `free`. The translation of communications amounts to calling the corresponding communication function of MPI (recall that we have restricted ourselves to communication skeletons having a direct counterpart in the functions implemented in standard libraries such as MPI).

In all cases, symbolic sizes present in types are needed by the code generation. For example, the vector size is required by the translation of the sequential communication skeleton $\mathbf{brdcast}_{\mathbf{uc}}$ (to set the corresponding loop bound), and by the translation of the parallel communication **pbrdcast** (to set the size of the message).

9.2 Prototype

The compiler has nine stages. A *parser* produces an abstract syntactic tree starting from the initial program. The *size analysis* is similar to a type inference. It uses the library of polyhedric computations PolyLib (Wilde, 1993) to check the coherence of constraints and to determine the overall size constraints. The *update in place transformation* first approximates sharing and the access sequence of the execution. It then checks the property $\mathscr{U}p$ for the abstract access sequences and may insert automatically vector copies and deallocations. The transformations *making communications explicit* and *distribution* are simple syntactic transformations. The *symbolic cost analysis* is relatively complex to implement. Indeed, the cost computation must inter-operate with PolyLib and Maple. The *optimization* and the *translation* towards the target language (C+MPI) are also syntactic transformations which can be implemented directly. Lastly, the *production of object code* is carried out by the C compiler available on the target machine and by linking with the MPI library.

The compilation stages made up of syntactic transformations are easily written in HASKELL. The static analyses are more complex to implement. For our experiments, some compilation steps, such as the destructive update step and most of the symbolic cost computation, were done manually.

9.3 Experiments

We have performed experiments on an Intel Paragon XP/S and a CRAY T3E with a handful of standard linear algebra programs (LU, Cholesky factorization, Jacobi iteration, etc.). The experiments have two objectives. First, we study the adequacy between the theoretical execution times produced by the cost analysis and those measured in practice. Then, we compare the performances of $\mathscr{L}_1$ programs with other implementations (standard sequential C code, HPF, the skeleton language NESL, and the linear algebra library SCALAPACK).

Figure 14 gives the execution times, both measured and theoretical (i.e. statically evaluated), for LU decomposition on a CRAY T3E with a row cyclic distribution and a row block distribution. The number of processors varies between 1 and 16 and we give times for two matrix sizes (1024×1024 and 2048×2048). The differences between the theoretical cost and the measured cost are less than 6%. Such differences are very hard to avoid. Our cost model does not take into account low level software and hardware mechanisms like routing protocols or local cache policies. Even if we can adapt the communication cost to the topology, we cannot model every low-level functionality available on the target computer. The execution times on the Paragon with the same example show the same differences with the theoretical cost. Our experiments suggest that the cost analysis is portable. It is sufficient to change the

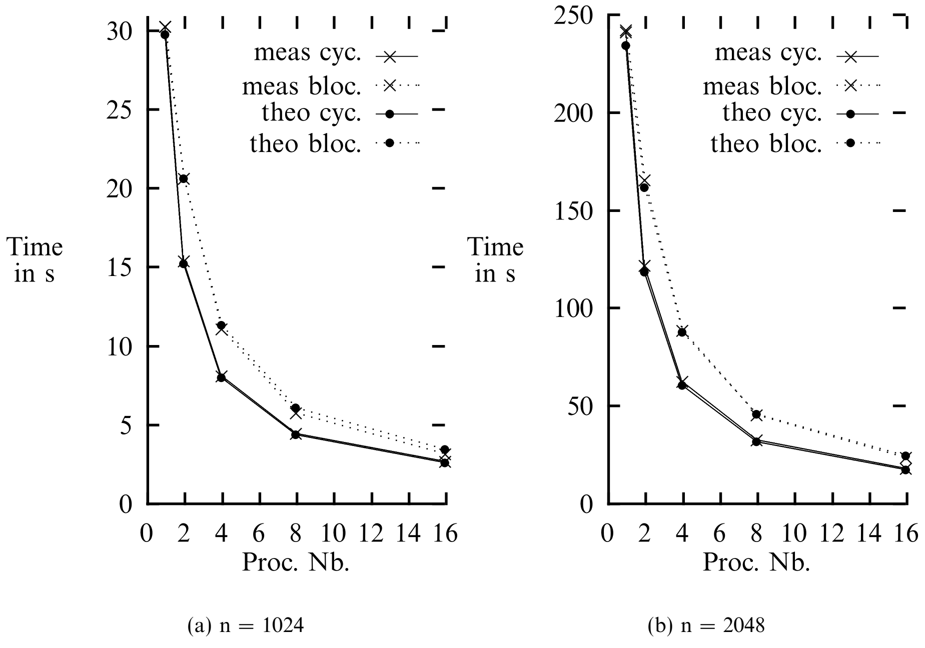

Fig. 14. Theoretic (theo) and Measured (meas) Times for LU Decomposition on a Cray T3E.

constants representing the basic operations costs to obtain a fairly faithful estimation of the actual execution times.

Figure 15 gathers the execution times obtained for LU decomposition, Cholesky factorization, the Householder method, the Jacobi iteration, and the n-body problem on the Intel Paragon. For all programs, the distribution chosen by the cost analysis proved to be the best one in practice.

We compared the sequential execution of $\mathscr{L}_1$ programs with standard (and portable) C versions taken from Press *et al.* (1986). We also compared our parallel implementation with High Performance Fortran (a manual distribution approach). No significant sequential or parallel runtime penalty seems to result from programming using skeletons, at least for such regular algorithms. We believe that the important differences for Jacobi and Nbody are due to the inability of the HPF compiler to recognize collective communications in general.

We compared our code with the parallel implementation of Nesl, a skeleton-based language (Blelloch *et al.*, 1994). The work on the implementation of Nesl has mostly been directed towards SIMD machines. On the Paragon, the Nesl compiler distributes vectors uniformly on processors and communications are not optimized. Not surprisingly, the parallel code is very inefficient (at least fifty times slower than our code).

Finally, we considered ScaLapack, an optimized library of linear algebra programs designed for distributed memory MIMD parallel computers (Choi and Dongarra,

	LU (n=256)			LU (n=512)			Cholesky (n=1024)		
Proc.	Skel.	Seq.	HPF	Skel.	Seq.	HPF	Skel.	Seq.	HPF
1	2.14	1.73	2.16	14.77	13.61	15.36	67.45	53.17	65.40
2	1.34	×	1.38	8.43	×	8.67	35.54	×	35.97
4	0.93	×	0.95	5.25	×	5.41	21.35	×	20.65
8	0.76	×	0.77	3.23	×	3.38	14.81	×	13.10
16	0.66	×	0.67	2.97	×	3.06	11.55	×	9.53
32	0.62	×	0.61	2.57	×	2.67	9.91	×	7.83

	Householder (n=1024)			Jacobi (n=512)			N body (n=2048)		
Proc.	Skel.	Seq.	HPF	Skel.	Seq.	HPF	Skel.	Seq.	HPF
1	318.16	308.17	325.44	63.42	55.10	56.20	125.15	122.76	127.86
2	159.04	×	164.13	32.98	×	29.81	62.99	×	91.77
4	82.12	×	84.62	17.19	×	16.83	31.53	×	49.04
8	44.59	×	46.32	7.66	×	10.19	15.51	×	27.39
16	26.68	×	27.03	3.90	×	6.93	7.59	×	17.35
32	17.98	×	18.23	1.98	×	5.19	3.64	×	12.68

Fig. 15. Times (in s) for Skeletons and HPF on Intel Paragon XP/S.

	LU (n=512)		Householder (n=1024)		Cholesky (n=1024)	
Proc.	Skel.	ScaLAPACK	Skel.	ScaLAPACK	Skel.	ScaLAPACK
1	14.77	3.78	318.16	56.35	67.45	55.80
2	8.43	2.4	159.04	35.23	35.54	34.95
4	5.25	1.84	82.12	22.57	21.35	21.80
8	3.23	1.66	44.59	16.27	14.81	15.56
16	2.97	1.50	26.68	12.82	11.55	12.56
32	2.57	1.41	17.98	10.83	9.91	11.32

Fig. 16. Times (in s) for Skeletons and ScaLAPACK on Intel Paragon XP/S.

1995). In ScaLAPACK, the user may explicitly indicate the data distribution. So, we indicated the best distribution found by the cost analysis in each ScaLAPACK program considered. If our code on one processor is much slower than its ScaLAPACK equivalent (between 3–6 times slower), the difference decreases as the number of processors increases (typically, 1.8 times slower on 32 processors). We believe that much of this difference comes from the machine specific routines used by ScaLAPACK for performing matrix operations (the BLAS library). This suggests a possible interesting extension of our source language. The idea would be to extend $\mathscr{L}_1$ with new skeletons corresponding to the BLAS operations in order to benefit from these machine specific routines.

Note that ScaLAPACK allows block cyclic distributions with a variable size of blocks which are a more general form of distribution than ours. This enables the programmer sometimes to find a better compromise between communication costs and load balancing by guessing the right block size. This is the case for the Cholesky factorization where the optimal distribution is block cyclic with a size of blocks between one and the vector size divided by the number of processors. Integrating block cyclic distributions (with a variable size of blocks) within our framework is not obvious. First, the combination of such distributions and the $\mathbf{poly}_n$ skeletons gives rise to expressions whose cost cannot be expressed as polytopes. Secondly, it is clear that an exhaustive analysis of all possible distributions would become unrealistic in this case. Extending the cost analysis to this kind of distributions would presumably require approximations and interactions with the user.

10 Conclusions

We have presented the compilation of a skeleton-based language for parallel computers. Our compilation process makes use of a variety of techniques: typing, static analyses, program transformations, polytope volume computation. Working by program transformations in a unified framework simplifies the correctness proofs of the implementation. One can show independently for each step that the transformation preserves the semantics, and that the transformed program respects the restrictions enforced by the target language. We could have described the complete compilation process in terms of program transformations as in Douence and Fradet (1998). However, the SPMD-like skeleton programs of $\mathscr{L}_6$ (strict, first order functions and single-threaded arrays) are so close to C code that it was more pragmatic to reuse the C compiler. The most important characteristic of our approach is the source language restrictions. We now review and justify them.

10.1 Source language restrictions

Most $\mathscr{L}_1$ restrictions were guided by our need for an accurate, symbolic cost analysis. Relaxing any of the following restrictions would not be possible without changing the approach drastically.

- *Restricted recursion.* Disallowing general recursion is requisite for an accurate cost analysis. The existence of a fixpoint operator in the language makes complexity analysis undecidable. Skeletons are a way to tame recursion, since the data and control flow are known *a priori.*
- *Restricted vector manipulations.* The symbolic cost analysis requires the symbolic size of all vectors. This information is inferred by the size analysis. In order to do so, the size of a vector should not depend upon scalar values but only on constants or parameter sizes. A program observing this condition

is called 'shapely' (Jay and Steckler, 1998). For example, a skeleton *filter p* producing a vector made only of the elements of its vector argument satisfying the predicate p cannot be allowed. It would produce vectors whose size is unpredictable.

- *Restricted conditionals.* One cannot associate an accurate cost to conditionals whose test depends on scalar values. Only the maximum, the minimum, or, with probabilistic information, the average complexity could be evaluated in this case. Moreover, to reuse the powerful tools based on polytopes, costs must be defined by affine (in)equations depending on symbolic sizes and iterator indexes. To ensure the affinity of cost expressions, conditionals are constrained to be mask skeletons whose condition characterizes a convex polytope. Typing enforces that the condition is an affine expression of symbolic sizes and iterator indexes only. If this condition were expressed as a polynomial, its cost could not be expressed as the volume of a polytope.
- *Restricted communication skeletons.* The communications involved by $\mathscr{L}_1$ programs must be statically predictable. This is ensured by enforcing the arguments of the communication skeletons to be of type *Index* or *Size* in order to infer their symbolic values at compile time.

The following restrictions were chosen because they entailed simplifications or a more efficient implementation. They could be relaxed to a certain extent.

- The communication skeletons considered are standard collective communication primitives (broadcast, translation, etc.) which are either hard-wired or optimized on many parallel machines. However, new communication skeletons could be taken into account. Similarly, the collections of computation and reorganization skeletons could be extended.
- We have considered only nested vectors, not multi-dimensional arrays. Nested vectors are general, but make a distinction between dimensions. For example, some operations on columns are less easily expressed than the same operations on rows. It would be possible to include new families of skeletons acting on 2D or 3D matrices.
- User-defined functions are first order. We feel that higher order functions (and the use of closures) would make the analyses and the implementation much more complex. However, a simple solution to relax this restriction would be to use a preliminary transformation removing higher order functions (e.g. Chin, 1990).

10.2 Related work

The existing specialized languages for data parallelism are generally based on restricted forms of recursion. These restrictions are either syntactic constraints on the form of recursive calls or the fixpoint operator is replaced by a collection of skeletons.

The ALPHA language (Wilde, 1994) is a first order, strongly typed, functional language. An ALPHA program is a system of recursive functions where the arguments within recursive calls are restricted to be affine expressions of the function parameters. These restrictions make it possible to define precise static analyses and lead to efficient implementations on MIMD machines (Quinton *et al.*, 1995). ALPHA was initially introduced to express systolic algorithms, and may seem too restricted to express less regular programs. A possibility is to relax the affinity restriction, as in the systolic language CRYSTAL (Chen *et al.*, 1991), but the static analyses lose their precision.

The FORTRAN community has studied automatic data distribution through parallel cost estimation (Gupta and Banerjee, 1992; Chatterjee *et al.*, 1993). If the complete FORTRAN language (unrestricted conditional, indexing with runtime value, etc.) is to be taken into account, communication and computation costs cannot be accurately estimated. In practice, the approximated cost may be far from the real execution time leading to a bad distribution choice. Tawbi (1994), Pugh (1994) and Clauss (1996) focus on a subset of FORTRAN: loop bound and array indexes are affine expressions of the loop variables. This restriction allows them to compute a precise symbolic computation cost based on polytopes. Unfortunately, using this approach to estimate communication costs is not realistic. Indeed, the cost would be expressed in terms of point-to-point communications without taking into account hard-wired communication primitives (Feautrier, 1994). These approaches estimate real costs too roughly to ensure that a good distribution is chosen.

The skeletons of Cole (1988) and Darlington *et al.* (1993) can be seen as hard-wired parallel schemes (e.g. divide-and-conquer, pipe, etc.). Each skeleton comes with a fixed, optimal, distribution and implementation; this may entail a (costly) redistribution before each skeleton. This approach cannot exploit the nested parallelism expressed by the combination of skeletons (e.g. only the parallelism of the top-level **map** would be taken into account in the expression **map** (**map** $(+1)$)). The skeletons of Darlington *et al.* (1995) include classic higher-order functions (**map**, **fold**, **scan**) and coordination and distribution skeletons. In the same vein, Südholt (1997) introduces high-level distribution skeletons much more general than HPF distributions. In these approaches, data distribution is explicit and chosen by the programmer. The skeletons of Blelloch *et al.* (1994) and Cai and Skillicorn (1995) can be nested and facilitate the expression of algorithms having several levels of parallelism. Finally, Shafarenko (1995) introduces skeletons to describe sophisticated data motions into multi-dimensional arrays. A type inference with subtyping is used to detect particular data motions (e.g. collective communications such as translations or broadcasts). Shafarenko's skeletons are much more general than the communication skeletons of $\mathscr{L}_1$ and may describe complex communication schemes whose cost can only be approximated.

The skeleton community have defined several static analyses (size, cost) for specialized languages. Jay and Steckler (1998) defines a shape analysis for a polymorphic imperative language with arrays (the FISh language). Programs are restricted to be 'shapely', i.e. the shape (size) of arrays does not depend upon scalar values. This restriction makes it possible to evaluate statically the size of arrays. However, the numerical size of input arrays must be known at compile time. The analysis is not symbolic and may have the same complexity as the source program. Nitsche (2000) aims at detecting 'shapely' expressions in a standard functional language. The 'shapeliness' property is undecidable in general. His analysis only finds a subset of 'shapely' expressions. Herrmann and Lengauer (1998) defines a size analysis for functional language with nested lists. The analysis is symbolic but incomplete (recursive functions makes the problem undecidable).

The implementation of skeleton languages are based on cost analyses. Gorlatch *et al*, (1999) define precise communication costs for combination of scan and fold skeletons on several parallel topologies (hypercube, mesh, etc.). Cai and Skillicorn (1995), Rangaswami (1996) and Jay (2000) define cost analyses for skeleton-based languages. Their skeletons are less restricted than ours leading to an approximate parallel cost (communication or/and computation). Furthermore, the costs are not symbolic: the size of input matrices and the number of processors are supposed to be known. Bratvold (1993) and Michaelson *et al.* (1998) use cost estimations based on profiling to choose the distribution for each skeleton. Such experimental approaches do not ensure good and portable parallel performances for different machines, number of processors, or sizes of inputs.

Most implementations use cost information to apply local, cost-reducing, transformations (Darlington *et al.*, 1993) or to choose the best distribution for each skeleton (Bacci *et al.*, 1999). In both cases, implementation decisions are local and no arbitration of trade-offs is possible.

In our approach, we start from a high level language ($\mathscr{L}_1$) where skeletons can be freely nested and obtain a skeleton language ($\mathscr{L}_6$) with explicit distribution, communication and allocation similar to the source language of Darlington *et al.* (1995). Contrary to local optimization approaches, we consider a global distribution and cost analysis, and we are able to select the best implementation (among a restricted set of choices).

10.3 Future work

The preliminary results obtained by our prototype are promising but more experiments are necessary to assess both the expressiveness of the language and the efficiency of the compilation. The expressiveness may be evaluated by encoding a significant set of examples requiring high performances. The algorithms introduced in the Cowinchan set (Wilson, 1994) seem a good starting point, since they were conceived to test the expressiveness and the elegance of parallel programming languages. It would be possible and useful to introduce new computation skeletons or to let the user call (possibly recursive) sequential C functions from $\mathscr{L}_1$ programs. For the latter option, the user should also provide the symbolic cost of the C functions and indicate whether they update some vector arguments (in such case, copies must be performed before the call).

Experiments are also necessary to evaluate more thoroughly the precision of the cost analysis, in particular, the costs associated with the communication primitives. Another field of experimentation is the evaluation of our compilation process for SIMD machines. For such parallel machines, the cost of synchronization is negligible, but the load balancing of computations is of primary importance to obtain good performances. Our cost analysis seems reusable in this context. Experiments are needed to support this claim, and may also suggest new optimizations specific to SIMD machines.

Another research direction is to study dynamic redistributions chosen at compile-time. Some parallel algorithms (e.g. Alternative Direction Implicit Integration (Golub and Ortega, 1993)) are much more efficient in the context of dynamic data redistribution. A completely automatic and precise approach to this problem would be possible in our framework. However, this would lead to a search space of exponential size. A possible solution to this problem is to consider (high-level) interactions with the user.

The language $\mathscr{L}_1$ introduces only data parallelism. However, certain algorithms are more easily expressed in the form of control parallelism such as 'divide and conquer' algorithms. The extension of the language $\mathscr{L}_1$ with such skeletons would be useful to increase the expressiveness of the language. Herrmann and Lengauer (1999) describe the compilation of 'divide and conquer' skeletons into nested sequential and parallel loops. This can be seen as transforming control parallelism into data parallelism. So, a solution to accommodate control parallelism in our approach would be to characterize a class of control parallelism skeletons that can be transformed into $\mathscr{L}_1$.

More generally, skeletons appear to be an interesting technique to the design of Domain Specific Languages (DSLs). They make it possible to describe high-level languages enjoying important properties without preventing further extensions (by adding new skeletons). We believe that this approach to DSLs deserves more consideration.

Acknowledgements

Thanks are due to Rémi Douence, Mario Südholt and the anonymous referees for their useful comments.

References

Bacci, B., Gorlatch, S., Lengauer, C. and Pelagatti, S. (1999) Skeletons and transformations in an integrated parallel programming environment. *Parallel Computing Technologies: LNCS 1662*, pp. 13–27.

Blelloch, G. E., Hardwick, J. C., Sipelstein, J., Zagha, M. and Chatterjee, S. (1994) Implementation of a portable nested data-parallel language. *J. Parallel & Distributed Comput.*, **21**(1), 4–14.

Bratvold, T. (1993) A Skeleton-Based Parallelising Compiler for ML. *Proc. International Workshop on Parallel Implementation of Functional Languages*, pp. 23–33.

Cai, W., & Skillicorn, D. (1995) Calculating recurrences using the Bird-Meertens formalism. *Parallel Processing Lett.*, **5**(2), 179–190.

Char, B. W., Geddes, K. O., Gonnet, G. H., Leong, B. L., Monagan, M. B. and Watt, S. M. (1992) *Maple V Language Reference Manual.* Springer-Verlag.

Chatterjee, S., Gilbert, J. R., Schreiber, R. and Teng, S. (1993) Automatic array alignment in data-parallel program. *Proc. ACM Symposium on Principles of Programming Languages*, pp. 16–28.

Chen, M., Choo, Y. and Li, J. (1991) Crystal: Theory and Pragmatics of Generating Efficient Parallel Code. In: Szymanski, B. K. (ed.), *Parallel Functional Languages and Compilers*, pp. 255–308. ACM Press.

Chin, W. N. (1990) *Automatic methods for program transformation.* PhD thesis, Imperial College.

Choi, J. and Dongarra, J. J. (1995) Scalable linear algebra software libraries for distributed memory concurrent computers. *Proc. IEEE Workshop on Future Trends of Distributed Computing Systems*, pp. 170–177.

Clarke, L., Glendinning, I. and Hempel, R. (1994) The MPI Message Passing Interface Standard. *Programming Environments for Massively Parallel Distributed Systems: Working conference of the IFIP*, pp. 213–218.

Clauss, P. (1996) Counting solutions to linear and nonlinear constraints through Ehrhart polynomials: Applications to analyze and transform scientific programs. *Proc. ACM International Conference on Supercomputing*, pp. 278–285.

Cole, M. (1988) A skeletal approach to the exploitation of parallelism. *Proc. CONPAR*, pp. 667–675. Cambridge University Press.

Darlington, J., Field, A. J., Harrison, P. G., Kelly, P. H. J., Sharp, D. W. N., Wu, Q. and While, R. L. (1993) Parallel programming using skeleton functions. *Proc. of the PARLE: LNCS 694*, pp. 146–160.

Darlington, J., Guo, Y. K, To, H. W. and Jing, Y. (1995) Skeletons for structured parallel composition. *Proc. ACM Symposium on Principle and Practice of Parallel Programming*, pp. 19–28.

Douence, R. and Fradet, P. (1998) A systematic study of functional language implementations. *ACM Trans. Programming Languages and Systems*, **20**(2), 344–387.

Feautrier, P. (1994) Toward automatic distribution. *Parallel Processing Letters*, **4**(3), 233–244.

Fox, G., Johnson, M., Lyzenga, G., Otto, S., Salmon, J. and Walker, D. (1988) *Solving Problems on Concurrent Processors.* Prentice-Hall.

Golub, G. H. and Ortega, J. M. (1993) *Scientific Computing: An introduction with parallel computing.* Academic Press.

Gorlatch, S., Wedler, C. and Lengauer, C. (1999) Optimization rules for programming with collective operations. *Proc. Symp. on Parallel and Distributed Processing*, pp. 492–499.

Gupta, M. and Banerjee, P. (1992) Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Trans. Parallel & Distributed Syst.*, **3**(2), 179–193.

Guzmán, J. C. and Hudak, P. (1990) Single-threaded polymorphic lambda calculus. *Proc. Symposium on Logic in Computer Science*, pp. 333–343. IEEE Press.

Herrmann, C. and Lengauer, C. (1998) Size inference of nested lists in functional programs. *Proc. International Workshop on Implementation of Functional Languages*, pp. 347–364. University College, London.

Herrmann, C. and Lengauer, C. (1999) Parallelization of divide-and-conquer by translation to nested loops. *J. Functional Programming*, **9**(3), 279–310.

Hudak, P., Peyton Jones, S., Walder, P., Boutel, B., Fairbairn, J., Fasel, J., Guzman, M. M., Hammond, K., Hughes, J., Johnsson, T., Kieburtz, D., Nikhil, R., Partain, W. and Peterson, J. (1992) Report on the programming language haskell. *Sigplan Notices*, **27**(5).

Hudak, P., Peyton Jones, S., Wadler, P., Hughes, John, Augustsson, L., Barton, D., Boutel, B., Burton, W., Fasel, J., Hammond, K., Hinze, R., Johnsson, T., Jones, M., Launchbury, J., Meijer, E., Peterson, J., Reid, A. and Runciman, C. (1999) *Standard Libraries for the Haskell 98 Programming Language*. Research report, Yale university.

Jay, B. (2000) Costing parallel programs as a function of shapes. *Science of Computer Programming*, **37**(1), 207–224.

Jay, B. and Steckler, P. (1998) The functional imperative: shape! *European Symposium on Programming: LNCS 1381*, pp. 139–153.

Johnsson, T. (1985) Lambda lifting: Transforming programs to recursive equations. *Functional Programming Languages and Computer Architecture: LNCS 201*, pp. 190–203.

Kastens, U. and Schmidt, M. (1986) Lifetime analysis for procedure parameters. *European Symposium on Programming: LNCS 213*, pp. 53–69.

Mallet, J. (1998a) *Compilation d'un langage spécialisé pour machine massivement parallèle*. Doctorat d'université, Rennes I.

Mallet, J. (1998b) Symbolic cost analysis and automatic data distribution for a skeleton-based language. *Euro-par'98 parallel processing: LNCS 1470*, pp. 688–697.

Michaelson, G., Scaife, N., Bristow, P. and King, P. (1998) Engineering a parallel compiler for SML. *Proc. of the International Workshop on Implementation of Functional Languages*, pp. 213–226. University College, London.

Mitchell, J. C. (1991) Type inference with simple sub-types. *J. Functional Programming*, **1**(3), 245–286.

Nitsche, T. (2000) Shapeliness analysis of functional programs with algebraic data types. *Science of Computer Programming*, **37**(1), 225–252.

Press, W. H., Teukolsky, S. A., Vetterling, W. T. and Flannery, Brian P. (1986) *Numerical Recipes in FORTRAN: The art of scientific computing*. Cambridge University Press.

Pugh, W. (1994) Counting solutions to presburger formulas: How and why. *Proc. Conference on Programming Language Design and Implementation*, pp. 121–134.

Quinton, P., Rajopadhye, S. and Wilde, D. (1995) On deriving data parallel code from a functional program. *Proc. of the International IEEE Symposium on Parallel Processing*, pp. 766–773.

Rangaswami, R. (1996) *A cost analysis for a higher-order parallel programming model*. PhD thesis, Edinburgh University.

Schmidt, D. A. (1985) Detecting global variables in denotational definitions. *ACM TOPLAS*, **7**(2), 299–310.

Sestoft, P. (1989) Replacing functional parameters by global variables. *Proc. of the Conference on Functional Programming Languages and Computer Architecture*, pp. 39–53.

Shafarenko, A. (1995) Symmetries in data parallelism. *The Computer Journal*, **38**(5), 365–380.

Südholt, M. (1997) *The transformational derivation of parallel programs using data-distribution algebras and skeletons*. PhD thesis, Technische Universität Berlin.

Tawbi, N. (1994) Estimation of nested loops execution time by integer arithmetic in convex polyhedra. *Proc. of International Symposium on Parallel Processing*, pp. 217–223.

Turner, D. A. (1979) A new implementation technique for applicative languages. *Software–Practice and Experience*, **9**, 31–49.

Wadler, P. (1990) Linear types can change the world! *Programming Concepts and Methods*, pp. 561–581. North Holland.

Wilde, D. (1993) *A library for polyhedral operations*. Publication Interne 785. IRISA.

Wilde, D. (1994) *The ALPHA language*. Technical Report 2295, INRIA, France.

Wilson, G. (1994) Assessing the usability of parallel programming systems: The cowichan problems. *Proc. IFIP Working Conference on Programming Environments for Massively Parallel Distributed Systems*.

A Transformations and analyses (Addendum)

We describe here additional rules for the size inference and the transformations. Along with the rules shown in the main text, there is at least one rule for each skeleton class. The reader is referred to Mallet (1998a) for complete descriptions.

A.1 Size inference

$$\frac{C \vdash \mathit{Int} \sqsubseteq \alpha \qquad \mathit{Op} \in \{+,-\}}{C,\Gamma \vdash \mathit{Op} : (\alpha,\alpha) \to \alpha, \{\}}\,[\mathrm{OP}] \qquad \frac{C,\Gamma \vdash e : \alpha^{s_1}, C_1 \qquad C \vdash \alpha \sqsubseteq \mathit{Float}}{C,\Gamma \vdash k * e : \alpha^{s_2}, C_1 \cup \{s_2 = k * s_1\}}\,[\mathrm{LINF_2}]$$

$$\frac{C,\Gamma \vdash e : \mathit{Index}^{s_1}, C_1 \qquad C,\Gamma \vdash f : (\mathit{Index}^{s_2},\beta) \to \gamma, C_2 \qquad C_1 \cup C_2 \Rightarrow \{1 \leqslant s_2 \leqslant s_1\}}{C,\Gamma \vdash \mathbf{iterfor}\ e\ f : \alpha \to \alpha, C_1 \cup C_2 \cup \{\alpha = \beta = \gamma\}}\,[\mathrm{ITE}]$$

$$\frac{C,\Gamma \cup \{x_1 : \alpha_1, ..., x_n : \alpha_n\} \vdash e : \beta, C_1}{C,\Gamma \vdash \lambda(x_1,...,x_n).e : (\alpha_1,...,\alpha_n) \to \beta, C_1}\,[\mathrm{ABS}]$$

$$\frac{}{C,\Gamma \vdash \mathbf{zip} : (\mathit{Vect}\ s_1\ \alpha, \mathit{Vect}\ s_2\ \beta) \to \mathit{Vect}\ s_1\ (\alpha,\beta), \{s_1 = s_2\}}\,[\mathrm{ZIP}]$$

$$\frac{C,\Gamma \vdash e : \mathit{Size}^{s}, C_1}{C,\Gamma \vdash \mathbf{makearray}\ e : \alpha \to \mathit{Vect}\ s\ \alpha, C_1}\,[\mathrm{MAK}]$$

$$\frac{C,\Gamma \vdash e : \beta_1, C_1 \qquad C,\Gamma \vdash f : (\beta_2,\alpha) \to \beta_3, C_2}{C,\Gamma \vdash \mathbf{fold}\ e\ f : \mathit{Vect}\ s\ \alpha \to \beta_1, C_1 \cup C_2 \cup \{\beta_1 = \beta_2 = \beta_3\}}\,[\mathrm{FOLD}]$$

A.2 Distribution

Abstraction transformation

$$\begin{aligned}
\mathcal{A}bs[\![\lambda\overrightarrow{X}.\mathrm{Exp}]\!]\overrightarrow{Y} &= \mathcal{A}bs[\![\mathbf{extract}^{(a,b),a} \circ \mathcal{A}bs[\![\mathrm{Exp}]\!]\overrightarrow{X}]\!]\overrightarrow{Y}\\
\mathcal{A}bs[\![\mathbf{iterfor}\ \mathrm{Exp}\ \mathrm{Fun}]\!]\overrightarrow{X} &= \mathbf{iterfor_{uc}}\ (\mathcal{A}bs[\![\mathrm{Fun}]\!]\overrightarrow{X})\\
&\quad \circ \mathbf{extract}^{(a,(b,c)),(b,(c,a))} \circ \mathbf{ftuple}_2\ \mathrm{Id}\ (\mathcal{A}bs[\![\mathrm{Exp}]\!]\overrightarrow{X})\\
\mathcal{A}bs[\![\mathrm{Op}]\!]\overrightarrow{X} &= \mathbf{ftuple}_2\ \mathrm{Op}\ \mathrm{Id}\\
\mathcal{A}bs[\![\mathbf{zip}]\!]\overrightarrow{X} &= \mathbf{ftuple}_2\ \mathbf{zip}\ \mathrm{Id}\\
\mathcal{A}bs[\![\mathbf{makearray}\ \mathrm{Exp}]\!]\overrightarrow{X} &= \mathbf{ftuple}_2\ \mathbf{makearray_{uc}}\ \mathrm{Id}\\
&\quad \circ \mathbf{extract}^{(a,(b,c)),((a,b),c)} \circ \mathbf{ftuple}_2\ \mathrm{Id}\ (\mathcal{A}bs[\![\mathrm{Exp}]\!]\overrightarrow{X})\\
\mathcal{A}bs[\![\mathbf{fold}\ \mathrm{Exp}\ \mathrm{Op}]\!]\overrightarrow{X} &= \mathbf{ftuple}_2\ (\mathbf{fold_{uc}}\ \mathrm{Op})\ \mathrm{Id}\\
&\quad \circ \mathbf{extract}^{(a,(b,c)),((b,a),c)} \circ \mathbf{ftuple}_2\ \mathrm{Id}\ (\mathcal{A}bs[\![\mathrm{Exp}]\!]\overrightarrow{X})
\end{aligned}$$

Distribution transformation

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathscr{D}ist[\![\mathbf{iterfor}_{\mathbf{uc}}\ \mathrm{F} \circ \mathbf{dpi}\ (\mathbf{loci}\ p)\ \mathrm{di}]\!]} \\
= & \mathrm{let} & \mathrm{di} \circ \mathrm{F'} = \mathscr{D}ist[\![\mathrm{F} \circ \mathbf{dpi}\ (\mathbf{loci}\ p)\ \mathrm{di}]\!] \\
 & \mathrm{in} & \mathrm{di} \circ \mathbf{piterfor}\ \mathrm{F'} \qquad \text{with di} : Vect\ p\ \alpha \to \alpha \\
\multicolumn{3}{l}{\mathscr{D}ist[\![\mathbf{dealloc}\ \circ \mathbf{dpi}\ \mathrm{di}_1\ \mathrm{di}_2]\!]} \\
= & \multicolumn{2}{l}{\mathrm{di}_2 \circ \mathbf{pimap}\ (\lambda(ip, x).\mathbf{dealloc}\ x)} \\
\multicolumn{3}{l}{\mathscr{D}ist[\![\mathrm{Op} \circ \mathbf{consti}\ k\ p]\!]} \\
= & \multicolumn{2}{l}{\mathbf{consti}\ k\ p\ \circ \mathbf{pimap}\ (\lambda(ip, x).\mathrm{if}\ ip = k\ \mathrm{then\ Op}\ x\ \mathrm{else}\ x)} \\
\multicolumn{3}{l}{\mathscr{D}ist[\![\mathbf{zip}\ \circ \mathbf{dpi}\ \mathrm{fdi\ fdi}]\!]} \\
= & \multicolumn{2}{l}{\mathrm{fdi} \circ \mathbf{pimap}\ (\lambda(ip, x).\mathbf{zip}\ x)} \\
\multicolumn{3}{l}{\mathscr{D}ist[\![\mathbf{zip}\ \circ \mathbf{dpi}\ (\mathbf{dei}\ \mathrm{fdi\ di}_1)\ (\mathbf{dei}\ \mathrm{fdi\ di}_2)]\!]} \\
= & \multicolumn{2}{l}{\mathbf{dei}\ \mathrm{fdi}\ (\mathbf{dpi}\ \mathrm{di}_1\ \mathrm{di}_2) \circ \mathbf{pimap}\ (\lambda(ip, x).\mathbf{zip}\ x)} \\
\multicolumn{3}{l}{\mathscr{D}ist[\![\mathbf{fold}_{\mathbf{uc}}\ \mathrm{Op} \circ \mathbf{dpi}\ \mathbf{seqi}\ (\mathbf{consti}\ k\ p)]\!]} \\
= & \multicolumn{2}{l}{\mathbf{consti}\ 0\ p \circ \mathbf{pimap}\ (\lambda(ip, (e, v)).\mathrm{if}\ ip = 0\ \mathrm{then}\ \mathbf{fold}_{\mathbf{uc}}\ \mathrm{Op}(e, v)\ \mathrm{else}\ e)} \\
 & \multicolumn{2}{l}{\ \circ \mathbf{ptransfer}\ \ \circ \mathbf{pimap}\ (\lambda(ip, (v, e)).(k, 0, e, v))} \\
\multicolumn{3}{l}{\mathscr{D}ist[\![\mathbf{fold}_{\mathbf{uc}}\ \mathrm{Op} \circ \mathbf{dpi}\ (\mathbf{blocki}\ p)\ (\mathbf{consti}\ k\ p)]\!]} \\
= & \multicolumn{2}{l}{\mathbf{consti}\ k\ p} \\
 & \multicolumn{2}{l}{\ \circ \mathbf{pimap}\ (\lambda(ip, (v_1, v_2)).\mathrm{if}\ ip = k\ \mathrm{then}\ \mathbf{fold}_{\mathbf{uc}}\ \mathrm{Op}\ (v_1, v_2)\ \mathrm{else}\ 1_{\mathrm{Op}})} \\
 & \multicolumn{2}{l}{\ \circ \mathbf{pgather}\ \ \circ \mathbf{pimap}\ (\lambda(ip, (v_1, v_2)).(k, \mathbf{fold}_{\mathbf{uc}}\ \mathrm{Op}\ (v_1,\ 1_{\mathrm{Op}}),\ v_2))} \\
 & \multicolumn{2}{l}{\qquad \text{if Op is commutative}} \\
\multicolumn{3}{l}{\mathscr{D}ist[\![\mathbf{fold}_{\mathbf{uc}}\ \mathrm{Op} \circ \mathbf{dpi}\ (\mathbf{cyci}\ p)\ (\mathbf{consti}\ k\ p)]\!]} \\
= & \multicolumn{2}{l}{\mathbf{consti}\ k\ p} \\
 & \multicolumn{2}{l}{\ \circ \mathbf{pimap}\ (\lambda(ip, (v_1, v_2)).\mathrm{if}\ ip = k\ \mathrm{then}\ \mathbf{fold}_{\mathbf{uc}}\ \mathrm{Op}\ (v_1, v_2)\ \mathrm{else}\ 1_{\mathrm{Op}})} \\
 & \multicolumn{2}{l}{\ \circ \mathbf{pgather}\ \ \circ \mathbf{pimap}\ (\lambda(ip, (v_1, v_2)).(k, \mathbf{fold}_{\mathbf{uc}}\ \mathrm{Op}\ (v_1,\ 1_{\mathrm{Op}}),\ v_2))} \\
 & \multicolumn{2}{l}{\qquad \text{if Op is commutative}} \\
\multicolumn{3}{l}{\mathscr{D}ist[\![\mathbf{fold}_{\mathbf{uc}}\ \mathrm{Op} \circ \mathbf{dpi}\ (\mathbf{cyci}\ p)\ (\mathbf{consti}\ k\ p)]\!]} \\
= & \multicolumn{2}{l}{\mathbf{consti}\ k\ p} \\
 & \multicolumn{2}{l}{\ \circ \mathbf{pimap}\ (\lambda(ip, (v_1, v_2)).\mathrm{if}\ ip = k\ \mathrm{then}\ \mathbf{fold}_{\mathbf{uc}}\ \mathrm{Op}\ (\mathbf{cyci}\ p\ v_1, v_2)\ \mathrm{else}\ 1_{\mathrm{Op}})} \\
 & \multicolumn{2}{l}{\ \circ \mathbf{pgather}\ \ \circ \mathbf{pimap}\ (\lambda(ip, x).(k, x))} \\
\multicolumn{3}{l}{\mathscr{D}ist[\![\mathbf{makearray}_{\mathbf{uc}}\ \circ \mathbf{dpi}\ (\mathbf{consti}\ k\ p)\ (\mathbf{consti}\ k\ p)]\!]} \\
= & \multicolumn{2}{l}{\mathbf{consti}\ k\ p\ \circ \mathbf{pimap}\ (\lambda(ip, (e, x)).\mathbf{makearray}_{\mathbf{uc}}(e, x))}
\end{array}
$$

A.3 Cost analysis

$$
\begin{array}{lcl}
\mathscr{C}a[\![\lambda(x_1, ..., x_n).e]\!] & = & \lambda(x_1, ..., x_n).\mathscr{C}a[\![e]\!] \\
\mathscr{C}a[\![\mathrm{Fun\ Exp}]\!] & = & \mathscr{C}a[\![\mathrm{Fun}]\!](\mathrm{Exp}) + \mathscr{C}a[\![\mathrm{Exp}]\!] \\
\mathscr{C}a[\![(\mathrm{Exp}_1,\ \mathrm{Exp}_2)]\!] & = & \mathscr{C}a[\![\mathrm{Exp}_1]\!] + \mathscr{C}a[\![\mathrm{Exp}_2]\!] \\
\mathscr{C}a[\![x]\!] & = & 0 \\
\mathscr{C}a[\![k]\!] & = & 0 \\
\mathscr{C}a[\![\mathbf{iterfor}_{\mathbf{uc}}\ \mathrm{Fun}]\!] & = & (\lambda x.\sum_i \left\{ {}^{0 \leqslant i}_{i \leqslant s} \right\} \mathscr{C}a[\![\mathrm{Fun}]\!](i, x)) \\
 & & \text{where } \mathbf{iterfor}_{\mathbf{uc}}\ \mathrm{Fun} : (Index^s, \alpha) \to \alpha \\
\mathscr{C}a[\![\mathbf{copy}]\!] & = & (\lambda x.\alpha_{copy} * \mathsf{size}\ \alpha) \qquad \text{where } \mathbf{copy} : \alpha \to \beta \\
\mathscr{C}a[\![\mathbf{dealloc}]\!] & = & (\lambda x.\alpha_{dea} * \mathsf{size}\ \alpha) \qquad \text{where } \mathbf{dealloc} : (\alpha, \beta) \to \beta \\
\mathscr{C}a[\![\mathbf{zip}]\!] & = & (\lambda x.\alpha_{zip} * n) \qquad \text{where } \mathbf{zip} : (Vect\ n\ \alpha,\ Vect\ n\ \beta) \to \gamma \\
\mathscr{C}a[\![\mathbf{makearray}_{\mathbf{uc}}]\!] & = & (\lambda x.\alpha_{make} * e * \mathsf{size}\ \alpha) \qquad \text{where } \mathbf{makearray}_{\mathbf{uc}} : (\beta, \alpha) \to Vect\ e\ \alpha \\
\mathscr{C}a[\![\mathbf{fold}_{\mathbf{uc}}\ \mathrm{Op}]\!] & = & (\lambda x.\sum_i \left\{ {}^{0 \leqslant i}_{i < n} \right\} \left(\mathscr{C}a[\![\mathrm{Op}]\!](x!i) + \alpha_{fol}\right)) \\
 & & \quad \text{where } \mathbf{fold}_{\mathbf{uc}}\ \mathrm{Op} : Vect\ n\ \alpha, \beta) \to \gamma \\
\mathscr{C}a[\![\mathbf{brdcast}_{\mathbf{uc}}]\!] & = & (\lambda x.\alpha_{\mathtt{br}} * n * \mathsf{size}\ \alpha) \qquad \text{where } \mathbf{brdcast}_{\mathbf{uc}} : (\alpha,\ Vect\ n\ \alpha) \to Vect\ n\ \alpha
\end{array}
$$

B Examples of proofs

The compilation process is proven to be correct by showing for each transformation step three properties (cf. section 2). Proofs are mainly simple structural inductions. We sketch here the proof for Property 1 for the abstraction $\mathscr{A}bs$ and Property 2 for the distribution transformation $\mathscr{D}ist$.

B.1 Proof of Property 1

The source language of the transformation $\mathscr{A}bs$ is $\mathscr{L}_3$ which is the language $\mathscr{L}_1$ with two additional functions **copy** and **dealloc** , and such that the argument functions of the skeleton $\mathbf{poly}_n$ are closed.

The target language of $\mathscr{A}bs$ is $\mathscr{L}_4$ whose syntax is:

Prog_4	$::=$	$\text{Fun}_4\ \text{VarTuple}_4$
Fun_4	$::=$	$\text{Fun}_4 \circ \text{Fun}_4 \mid \mathbf{iterfor_{uc}}\ \text{Fun}_4 \mid \text{Op}_4 \mid \mathbf{extract}^{(\text{VarTuple}_4,\text{VarTuple}_4)}$
	$\mid$	$\mathbf{ftuple}_n\ \text{Fun}_4\ ...\text{Fun}_4$
	$\mid$	$\text{CompSkel}_4 \mid \text{ReorgSkel}_4 \mid \text{CommSkel}_4 \mid \text{MaskSkel}_4$
Op_4	$::=$	$+ \mid - \mid * \mid \mathbf{div} \mid \mathbf{exp} \mid \mathbf{log} \mid \mathbf{cos} \mid \dots \mid \mathbf{copy} \mid \mathbf{dealloc}$
CompSkel_4	$::=$	$\mathbf{map}\ \text{Fun}_4 \mid \mathbf{fold_{uc}}\ \text{Op}_4 \mid \mathbf{scan_{uc}}\ \text{Op}_4$
ReorgSkel_4	$::=$	$\mathbf{zip} \mid \mathbf{unzip} \mid \mathbf{append} \mid \mathbf{makearray_{uc}}$
CommSkel_4	$::=$	$\mathbf{brdcast_{uc}} \mid \mathbf{transfer_{uc}} \mid \mathbf{rotate_{uc}} \mid \mathbf{scatter_{uc}}$
	$\mid$	$\mathbf{gather_{uc}} \mid \mathbf{allgather_{uc}} \mid \mathbf{allbrdcast_{uc}}$
MaskSkel_4	$::=$	$\mathbf{poly}_n\ \lambda(x_1,\dots,x_n).\text{Ineq}_4\ \text{Fun}_4\ \text{Fun}_4$
Ineq_4	$::=$	$\text{Ineq}_4 \wedge \text{Ineq}_4 \mid \text{LinF}_4 < \text{LinF}_4 \mid \text{LinF}_4 = \text{LinF}_4$
LinF_4	$::=$	$\text{LinF}_4 + \text{LinF}_4 \mid \text{LinF}_4 - \text{LinF}_4 \mid k{*}\text{LinF}_4 \mid x \mid k$
VarTuple_4	$::=$	$(\text{VarTuple}_4,...,\text{VarTuple}_4) \mid x$

Property 1 is expressed as:
$\forall P \in \text{Prog}_3, (\mathscr{A}bs[\![P]\!]\overrightarrow{X})\overrightarrow{X} \in \text{Prog}_4$ where $\overrightarrow{X}$ are the free variables of P.

The proof of this property boils down to the proof of the corresponding properties on (recursive) non-terminals of $\mathscr{L}_3$. That is:

- $\forall E \in \text{Exp}_3, \mathscr{A}bs[\![E]\!]\overrightarrow{X} \in \text{Fun}_4$ where $\overrightarrow{X}$ contains the free variables of E
- $\forall F \in \text{Fun}_3, \mathscr{A}bs[\![F]\!]\overrightarrow{X} \in \text{Fun}_4$ where $\overrightarrow{X}$ contains the free variables of F

This proof is done by structural induction. We show only the case of pairs.
Let E_1, $\text{E}_2 \in \text{Exp}_3$, we have

$$\mathscr{A}bs[\![(\text{E}_1,\text{E}_2)]\!]\overrightarrow{X} = \mathbf{extract}^{((a_1,b_1),(a_2,b_2)),((a_1,a_2),b_1)} \circ \mathbf{ftuple}_2\ (\mathscr{A}bs[\![\text{E}_1]\!]\overrightarrow{X})\ (\mathscr{A}bs[\![\text{E}_2]\!]\overrightarrow{X})\ \circ \mathbf{extract}^{x,(x,x)}$$

By the induction hypothesis, we have $\mathscr{A}bs[\![\text{E}_1]\!]\overrightarrow{X} \in \text{Fun}_4$ and $\mathscr{A}bs[\![\text{E}_2]\!]\overrightarrow{X} \in \text{Fun}_4$. So $\mathbf{ftuple}_2\ (\mathscr{A}bs[\![\text{E}_1]\!]\overrightarrow{X})\ (\mathscr{A}bs[\![\text{E}_2]\!]\overrightarrow{X}) \in \text{Fun}_4$, and the right-hand side expression belongs to Fun_4.

B.2 Proof of Property 2

To prove that the distribution transformation preserves the semantics, we show, for each transformation rule, that the left-hand side expression is semantically equal to the right-hand side one.

For example, to prove the $\mathbf{poly}_2$ rule, we use the Haskell definition of the inverse distribution **dei** (**blocki** *p*) **seqi** :

```
dei (blocki p) seqi proc
  = array (0,p*b-1)
      [(ip*b+i, array (0,n-1) [(j,proc!ip!i!j)| j<-[0..n-1]])
        | i<-[0..b-1], ip<-[0..p-1]]
    where p = sizeRange(bounds proc)
          b = sizeRange(bounds proc!0)
          n = sizeRange(bounds proc!0!0)
```

Let **dei** (**blocki** *p*) **seqi** : *Vectproc p* (*Vect b* (*Vect n* α)) $\rightarrow$ *Vect m* (*Vect n* α),
Then, for all *proc* : *Vectproc p* (*Vect b* (*Vect n* α))

$(\mathbf{poly}_2\ \mathrm{P}\ \mathrm{F}_1\ \mathrm{F}_2 \circ \mathbf{dei}\ (\mathbf{blocki}\ p)\ \mathbf{seqi}\ )\ proc$

$= \mathbf{poly}_2\ \mathrm{P}\ \mathrm{F}_1\ \mathrm{F}_2\ (\mathbf{dei}\ (\mathbf{blocki}\ p)\ \mathbf{seqi}\ proc)$

$\circ$ *definition*

$= \mathbf{poly}_2\ \mathrm{P}\ \mathrm{F}_1\ \mathrm{F}_2$
$\quad(\mathtt{array\ (0,}p{*}b-1\mathtt{)\ [(ip{*}}b\mathtt{+i,\ array\ (0,}n-1\mathtt{)\ [(j,}proc\mathtt{!ip!i!j)}$
$\qquad\mathtt{|\ j{<}{-}[0..}n-1\mathtt{]])\ |\ i{<}{-}[0..}b-1\mathtt{],\ ip{<}{-}[0..}p-1\mathtt{]])}$

dei *definition*

$= \mathtt{array\ (0,}p*b-1\mathtt{)}$
$\quad\mathtt{[(ip{*}}b\mathtt{+i,\ array\ (0,}n-1\mathtt{)}$
$\qquad\mathtt{[(j,if\ P(ip{*}}b\mathtt{+i,j)\ then\ F_1(}proc\mathtt{!ip!i!j)\ else\ F_2(}proc\mathtt{!ip!i!j))}$
$\qquad\quad\mathtt{|\ j{<}{-}[0..}n-1\mathtt{]])\ |\ i{<}{-}[0..}b-1\mathtt{],\ ip{<}{-}[0..}p-1\mathtt{]]}$

$\mathbf{poly}_2$ *definition* + *! definition*

$= \mathbf{dei}\ (\mathbf{blocki}\ p)\ \mathbf{seqi}$
$\quad(\mathbf{pimap}\ \lambda(ip,v).\mathtt{(array\ (0,}b-1\mathtt{)\ [(i,\ array\ (0,}n-1\mathtt{)}$
$\qquad\mathtt{[(j,if\ P(}ip{*}b\mathtt{+i,j)\ then\ F_1(}v\mathtt{!i!j)\ else\ F_2(}v\mathtt{!i!j))}$
$\qquad\quad\mathtt{|\ j{<}{-}[0..}n-1\mathtt{]])|\ i{<}{-}[0..}b-1\mathtt{]])}\ proc)$

dei *definition* + **pimap** *definition*

$= \mathbf{dei}\ (\mathbf{blocki}\ \mathrm{p})\ \mathbf{seqi}$
$\quad(\mathbf{pimap}\ (\lambda(ip,v).\mathbf{poly}_2\ \mathrm{P} \circ \lambda(i,j).(ip*b+i,j)\ \mathrm{F}_1\ \mathrm{F}_2\ v)\ proc)$

$\mathbf{poly}_2$ *definition*

$= (\mathbf{dei}\ (\mathbf{blocki}\ p)\ \mathbf{seqi}$
$\quad\circ\ \mathbf{pimap}\ (\lambda(ip,v).\mathbf{poly}_2\ \mathrm{P} \circ \lambda(i,j).(ip*b+i,j)\ \mathrm{F}_1\ \mathrm{F}_2\ v))\ proc$

$\circ$ *definition*

Thus,

$\mathbf{poly}_2\ \mathrm{P}\ \mathrm{F}_1\ \mathrm{F}_2 \circ \mathbf{dei}\ (\mathbf{blocki}\ p)\ \mathbf{seqi}$
$\quad = \mathbf{dei}\ (\mathbf{blocki}\ p)\ \mathbf{seqi}\ \circ \mathbf{pimap}\ (\lambda(ip,v).\mathbf{poly}_2\ \mathrm{P} \circ \lambda(i,j).(ip*b+i,j)\ \mathrm{F}_1\ \mathrm{F}_2\ v$
$\quad = \mathscr{D}ist[\![\mathbf{poly}_2\ \mathrm{P}\ \mathrm{F}_1\ \mathrm{F}_2\ \circ \mathbf{dei}\ (\mathbf{blocki}\ p)\ \mathbf{seqi}\ ]\!]$

C Examples of programs

C.1 LU decomposition

The LU decomposition computes the decomposition of a square matrix M into two matrices L (triangular lower) and U (triangular higher), such as $M = L * U$.

The $\mathscr{L}_1$ program implementing this algorithm is given figure C 1. This program takes a matrix M of float of size $n \times n$, and returns a matrix containing the matrices L and U. The algorithm consists of an iteration (main function LU). At the kth step, the loop body determines the kth row and the kth column of the matrix result. This calculation is carried out by elimination by taking the element of the row $k-1$ and the column $k-1$ as pivot. The function $calc$ applies the function $fcalc$ to all the elements having indexes of row and column greater than k. This computation is iterated until the calculation of the nth row and the nth column. A Fortran version of this program can be found in (Press *et al.*, 1986).

```
LU(M) where
    M :: Vect n (Vect n Float)
    LU = iterfor (n − 1) loop
    loop = λ(k, a).calc(fac(apivot(colrow(k, a))))
    calc = λ(k, a, row, piv).poly₂ (λ(i, j).k ⩽ i ∧ k ⩽ j) fcalc first
                                  (map zip3(zip3(a, row, piv)))
    fcalc = λ(a, row, piv).a − row * piv
    fac = λ(k, a, row, col, piv).(k, a, row, map (map /) (map zip (zip (col, piv))))
    apivot = λ(k, a, row, col).(k, a, row, col, map (brdcast (k − 1)) row)
    colrow = λ(k, a).(k, a, brdcast (k − 1) a, map (brdcast (k − 1)) a)
    zip3 = λ(x, y, z).map p2t (zip (x, zip (y, z)))
    p2t = λ(x, (y, z)).(x, y, z)
    first = λ(x, y, x).x
```

Fig. C 1. LU decomposition in $\mathscr{L}_1$.

C.2 The n-body problem

The n-body problem concerns the simulation of N particles interacting via a long-range force such as gravity.

The $\mathscr{L}_1$ program given in figure C 2 evaluates the new configuration of N particles after one unit of time. It takes a vector V of size N of a 7-tuple of floats. Each tuple element represents parameters of one particle (coordinates, speed, weight, etc.) and returns a vector containing the new parameters of particles. The main iteration computes the interaction between each pair in turn (function $loop$). This function takes two copies of the initial vector, shifts the second copy to the right in order to compute (function $floop$) the interaction between each pair of particles. After $(N-1)/2$ iterations, the interactions of all pairs have been computed. The resulting vector contains the new configurations induced by the interactions of the N particles. A Fortran version of this program can be found in Fox *et al.* (1988).

```
NBODY(V) where
   V :: Vect N (Float, Float, Float, Float, Float, Float, Float)
   NBODY = λv.end(iterfor (div (N − 1) 2) loop (v, v))
   loop = λ(i, (p, q)).floop (p, rotate 1 q)
   floop = λ(p, q).unzip (map assign(zip3(p, q, (calc(zip (p, q))))))
   calc = λv.map (txyz.fac.sq.dxyz) v
   dxyz = λ((pm, px, py, pz, a, b, c), (qm, qx, qy, qz, a, b, c)).
               (pm ∗ qm, px − qx, py − qy, pz − qz, 0)
   sq = λ(a, b, c, d, e).(a, b, c, d, ((b ∗ b) + c ∗ c) + (d ∗ d))
   fac = λ(a, b, c, d, e).(a/(e ∗ (sqrt e)), b, c, d, e)
   txyz = λ(a, b, c, d, e).(a ∗ b, a ∗ c, a ∗ d)
   assign = λ((pm, px, py, pz, pfx, pfy, pfz), (qm, qx, qy, qz, qfx, qfy, qfz), (tx, ty, tz)).
                  ((pm, px, py, pz, pfx − tx, pfy − ty, pfz − tz),
                  (qm, qx, qy, qz, qfx + tx, qfy + ty, qfz + tz))
   end = λ(p, q).map plus(zip (p, rotate (−(÷N 2)) q))
   plus = λ((a1, b1, c1, d1, pfx, pfy, pfz), (a2, b2, c2, d2, qfx, qfy, qfz)).
                  (pfx + qfx, pfy + qfy, pfz + qfz)
   zip3 = λ(x, y, z).(map p2t(zip (zip (x, y), z)))
   p2t = λ((x, y), z).(x, y, z)
```

Fig. C 2. N-bodies in $\mathscr{L}_1$.