Towards Parallel Programming by Transformation: The FAN Skeleton Framework*

M. Aldinucci[1], S. Gorlatch[2], C. Lengauer[2] and S. Pelagatti[1]

[1]Dipartimento di Informatica, Università di Pisa, 40 Corso Italia, I-56125 Pisa, Italy

[2] Fakultät für Mathematik und Informatik, Universität Passau, D-94030 Passau, Germany

Abstract

A Functional Abstract Notation (FAN) is proposed for the specification and design of parallel algorithms by means of *skeletons* – high-level patterns with parallel semantics. The main weakness of the current programming systems based on skeletons is that the user is still responsible for finding the most appropriate skeleton composition for a given application and a given parallel architecture.

We describe a transformational framework for the development of skeletal programs which is aimed at filling this gap. The framework makes use of *transformation rules* which are semantic equivalences among skeleton compositions. For a given problem, an initial, possibly inefficient skeleton specification is refined by applying a sequence of transformations. Transformations are guided by a set of performance prediction models which forecast the behavior of each skeleton and the performance benefits of different rules. The design process is supported by a graphical tool which locates applicable transformations and provides performance estimates, thereby helping the programmer in navigating through the program refinement space. We give an overview of the FAN framework and exemplify its use with performance-directed program derivations for simple case studies. Our experience can be viewed as a first feasibility study of methods and tools for transformational, performance-directed parallel programming using skeletons.

Keywords: Parallel programming, program transformations, algorithm design, skeletons, functional programming, performance models, program refinement tools.

Computing Reviews Categories: D.1.1-3, D.2.2, D.2.6, D.2.10, D.3.1-3, F.1.2, F.3.1, F.3.3.

1 Introduction

One main challenge of parallel programming today is the portability between different parallel architectures under preservation of performance. Most application programs are currently being written at the low level of C or Fortran, combined with a communication library like MPI; moreover, they are often tuned towards one specific machine configuration. Since parallel computers are typically replaced within five years, parallel programs which live longer have to be

*Parts of this work were presented in the preliminary version at the HiPS workshop of IPPS'99 (Puerto Rico) and at the PaCT'99 conference (St.Petersburg).

retuned or redesigned. In addition, programming at this low level of abstraction is cumbersome and error-prone.

In sequential programming, coding for a specific machine also prevailed three decades ago. The software engineering solution to overcoming it was to introduce levels of abstraction, effectively yielding a tree of refinements, from the problem specification to alternative target programs [31]. The derivation of a target program then follows a path down this tree. The transition from one node to the next can be described formally by a semantics-preserving program transformation or refinement. Conceptually, porting a program to a different machine configuration means backtracking to a previous node on the path and then following another path to a different target program.

In the parallel setting, high-level programming constructs and a refinement framework for them are necessary due to the inherent difficulties in maintaining the portability of low-level parallelism [10, 13]. In the Nineties, the "skeletons" research community [11] has been working on high-level languages and methods for parallel programming [3, 4, 9, 14, 21, 23]. Skeletons are higher-order functions which can be evaluated efficiently in parallel. They specify abstractly common patterns of parallelism which can be used as program building blocks. Typical skeletons include the pipeline, task farm, reduction and scan.

Current skeleton-based systems typically provide the user with a collection of high-level skeletal constructs and with a compiler for translating skeleton programs into low-level target code [32]. Typically, skeletons carry a large amount of information on program interaction structure, which can be used by the compiler to generate efficient code on different target machines. However, high performance is only reached if a composition of skeletons can be found which matches both the application and the target machine requirements.

What is still lacking is a sufficiently rich collection of transformation rules for skeletons and their compositions, and especially tool support for a navigation through the program refinement tree. One further concept, not as crucial in sequential programming, has to be added: the program refinements must be adorned with a cost model, since efficiency is the main – often the single – reason for using parallelism. Several cost models have been developed [26, 29, 35, 37] and have confirmed that porting a parallel program from one machine configuration to another may dramatically alter its performance [20]. Therefore, program design tools must apply transformations based on performance predictions made in a cost model.

The present paper addresses these two problems – skeleton transformations and cost models – and gives an overview of a feasibility study of the transformation-based, performance-directed approach for parallel programming with skeletons. We present:

- a functional abstract notation, FAN, for the specification of skeleton programs and transformation rules for them, together with its functional semantics given in Haskell,
- examples of semantically sound transformation rules for FAN programs,
- a transformation tool which can identify applicable rules in a FAN program and transform the program correspondingly by applying a rule chosen by the user,
- a cost model for skeletons and transformation rules, and

- case studies of program development by means of transforming FAN programs and estimating their cost.

We concentrate here on the design issues for high-level programs based on skeletons. The translation of skeleton-based programs into executable code, e.g., C+MPI for different parallel machines like the Fujitsu AP1000 and the Cray T3E, and related performance figures are presented elsewhere [5].

2 FAN: Functional Abstract Notation

In this section, we introduce a high-level functional notation, called FAN. This notation is intended for specifying parallelism and for transforming parallel programs in the design process.

The functions in a FAN program are divided into two classes: functions which are skeletons and functions which are not. Skeletons are distinguished because they have a potential for a particularly efficient, customized implementation – often involving parallelism. FAN skeletons are general second-order functions defined on arrays and scalars: they allow the user to concentrate on the program structure, independently of the syntactic details of a particular skeleton environment. E.g., one skeleton system FAN can be embedded into is P3L, developed in Pisa [19].

A FAN program consists of a *header* and a *body*. The header specifies the program name and the list of variables taken as input (in) and produced as output (out), as follows:

prog.name (in *invars*, out *outvars*)

Input/output variables are specified by name and type: for example, $v : T$ states that the variable named v has type T. Types can be scalars or arrays according to the following syntax:

T ::= Scalar | Array idx T

Here, idx defines the array dimensionality and size, and T is the type of the array elements. For instance, Array n Scalar denotes a vector of size n of scalars and Array n (Array m Scalar) denotes a vector of vectors of scalars. The actual type of the scalars (int, real, ...) is not needed during the transformation and can be supplied by the user in later stages of the program development.

A program body is a sequence of equations defining variables via expressions:

$$x_1 = e_1 \;;$$
$$\ldots$$
$$x_k = e_k \;;$$

FAN programs obey the *single-assignment* rule: there is at most one equation defining each variable. All output variables of a program must be assigned in the program's body. Each equation is terminated by a semicolon.

Expressions (e) can be constants (c), variables (x), function definitions, function applications or skeleton applications expressed by E_1, E_2, E_3, such as map, reduce, etc.

$e = c \mid x \mid \; e\; e \mid \; \lambda x.e \mid \; E_1\; e\; e \mid \; E_2\; e \mid \; E_3\; e\; e\; e$
$E_1 =$ map | $\mathsf{map}_{\#}$ | reduce | scanL | copy | split | part | rearrange
$E_2 =$ pair | proj1
$E_3 =$ loopfor | loopwhile | looprepeat

The scope of each variable definition extends across all subsequent definitions in the same body. At the end of the body, we can specify local definitions using a where clause. Names of FAN programs can be used as functions in expressions.

Skeletons offered by the current version of FAN represent common patterns of parallelism. In this paper, we restrict ourselves to the data-parallel skeletons, which comprise data arrangements and computations expressed by map, reduce, and scanL. It turns out that these skeletons form a basis of linear recursion [39]. Similar constructs are included in most skeletal systems in the literature [3, 9, 14]. We define skeletons formally in Section 3 and introduce an execution model for them in Section 6.

Let us now discuss a small example of a FAN program and give an intuitive idea of its possible parallel execution. Figure 1 shows the FAN code for a program, named inner.product, which takes as input two vectors a and b of length n and returns their inner product, scalar c.

```
inner.product (in a, b : Array n Scalar, out c : Scalar)
  t = map (*) (pair (a,b));
  c = reduce (+) t;
```

Figure 1: A FAN program to compute the inner product of two vectors

The program body defines the function to be computed. Expression pair (a,b) returns a vector of length n of pairs $(a[i], b[i])$. The application of the map skeleton yields the vector t of all products $t[i] = a[i] * b[i]$. Then, reduce $(+)$ t yields the sum of all elements in t.

The FAN program in Figure 1 simply defines a function and is not tied to a particular execution model. One possible model of parallelism in FAN programs is that the program statements are evaluated in sequence and only skeleton applications in them are considered for a parallel execution. In the example, supposing $p = n$ processors are available, all pairs $(a[i], b[i])$ and products $a[i] * b[i]$ could be computed in parallel. Then, parallelism can be exploited in the computation of c, exploiting the associativity of operator $+$. If $p < n$, computations can be performed on blocks of data in sequence. The adopted execution model is presented in more detail in Section 6.

Our choice of skeletons studied here is dictated by the current repertoire of the typical skeleton systems and also by the recent results in skeleton transformations.

3 FAN Skeletons

FAN skeletons are partitioned into two classes: skeletons performing actual computation (*computational skeletons*) and skeletons which reorganize data to enable subsequent computation (*arrangement skeletons*). We describe the two classes in the subsequent subsections.

We give every FAN skeleton two different semantics: (1) a *functional* semantics which defines the input/output behavior, and (2) a more detailed *operational* semantics which specifies the opportunities of parallel execution. We provide the functional semantics formally, in Haskell [8], and the operational semantics informally (in a later section). Most Haskell definitions are relegated to the appendix.

3.1 Computational data-parallel skeletons

The simplest computational data-parallel skeleton, map, applies a function f to all elements of an array, and arranges the results in an array with the same shape, for instance:

$$\mathsf{map}\ f\ [x_0, \ldots, x_n] = [f\ x_0, \ldots, f\ x_n]$$

The functional semantics of map is given by the Haskell function `fanmap`:

```
fanmap:: (Ix c) => (a -> b) -> Array c a -> Array c b
fanmap f x = array (bounds x) [ (i,f (x!i)) | i <- (range (bounds x)) ]
```

Function `fanmap` takes as parameters a function `f` and an array of any dimensionality and size, applies `f` to all array elements and returns the array of the same shape with all results. Functions `bounds` and `range` are predefined in Haskell: `bounds` returns the index bounds of an array, and `range` defines the multi-dimensional range corresponding to some bounds.

FAN function $\mathsf{map}_{\#}$ is an extended version of map, which expects the argument function to have the element index as an additional parameter:

$$\mathsf{map}_{\#}\ f\ [x_0, \ldots, x_n] = [f\ 0\ x_0, \ldots, f\ n\ x_n]$$

The Haskell definition of the functional semantics of $\mathsf{map}_{\#}$ is

```
fanmapsh:: (Ix c) => (c -> a -> b) -> Array c a -> Array c b
fanmapsh f x = array (bounds x) [ (i,f i (x!i)) | i <- (range (bounds x)) ]
```

The reduce and scan skeletons represent data-parallel computation patterns, whose parallel implementation may involve communication. They are informally defined as follows:

$$\mathsf{reduce}\ (\oplus)\ [x_1, \ldots, x_n] = x_1 \oplus \ldots \oplus x_n$$
$$\mathsf{scanL}\ (\oplus)\ [x_1, \ldots, x_n] = [x_1,\ x_1 \oplus x_2,\ \ldots,\ x_1 \oplus \ldots \oplus x_n]$$

Here, $[x_1, \ldots, x_n]$ denotes an array of length n and $\oplus$ is an associative operator. These definitions can be generalized to arrays of any dimensionality.

The according Haskell definitions are shown in Fig. 2. Note that we have chosen a representation which is parameterized in the index type, i.e., also in the dimensionality of the array. The index set is given by type variable `c`; `elems` returns the list of all the elements of an array, in the lexicographic order of the index set; `listArray` constructs an array from a list of elements and a bound definition; `foldl1` is the Haskell fold operation without neutral element; `scanl1` is the usual scan operation without neutral element. Function `fanscan` is a generic scan function over arrays, which takes a scan function `s` working on lists, an operator `f` and an array `x`, and applies `s` to the list resulting from applying `elems` to `x`. The semantic function for scanL is then obtained by specializing `fanscan` with the appropriate Haskell scan functions on lists (`scanl1`). There is also an analogous rightward version of scan, scanR, whose definition we omit for brevity.

3.2 Data arrangement skeletons

Skeletons for data arrangements take care of data replication, alignment and distribution to arrange data properly for a subsequent application of computational skeletons like map, reduce, etc. Note that the data is viewed globally, since there is no notion of processors at this abstract level.

```
fanred:: (Ix c) => (a-> a -> a) -> (Array c a) -> a
fanred f x = foldl1 f (elems x)

fanscan:: (Ix c) => ((a -> a -> a) -> [a] -> [a])
                   -> (a -> a -> a) -> (Array c a) -> Array c a
fanscan s f x = listArray (bounds x) (s f (elems x))

fanscanl:: (Ix c) => (a -> a -> a) -> (Array c a) -> Array c a
fanscanl = fanscan scanl1
```

Figure 2: Haskell definition of reduce and scanL.

Some of the arrangement functions depend on the array dimensionality, while others can be defined for arrays of any dimensionality. We discuss, first, data arrangements for one-dimensional arrays and, subsequently, some specific two-dimensional arrangements. Similar definitions can be given for higher dimensionalities.

3.2.1 One-dimensional arrays

Here are the arrangement skeletons for one-dimensional arrays:

- pair $(x,y) = [(x_0, y_0), \ldots, (x_n, y_n)]$ pairs two arrays $x = [x_0, \ldots, x_n]$ and $y = [y_0, \ldots, y_n]$ of the same length and returns bottom if the lengths do not match; similar constructors can be defined for making tuples of any other arity.

- proj1 $x = [x_0^1, \ldots, x_n^1]$

  returns the first components of the elements of an array of tuples; similar functions exists for subsequent components; proj1 can be viewed as a notational shorthand for map *fst*, where function *fst* yields the first component of a tuple.

- copy $n\ s = [s, \ldots, s]$

  creates a one-dimensional array of length n, filled with copies of s.

- split $p\ [x_0, \ldots, x_n] = [[x_0, \ldots, x_{k-1}], \ldots, [x_{n-k}, \ldots, x_n]]$

  slices an array into p non-overlapping intervals ($k = (n+1)/p$), and

- part $(r, s)\ [x_0, \ldots, x_{n-1}] = [[x_{(1-r) \bmod n}, \ldots, x_0, \ldots x_s],$
  $\ldots [x_{n-r}, \ldots, x_{n-1}, \ldots x_{(n-1+s) \bmod n}]]$

  returns an array of the same length as the input array, in which each element x_i is surrounded by a *halo* or *stencil* of neighboring elements. The ith element of the returned array corresponds to a segment of x comprising the r elements before and the s elements after x_i.

```
fanpart1:: (Int,Int) -> Array Int a -> Array Int (Array Int a)
fanpart1 (r,s) x =
      let (lb,ub) = bounds x
      in array (lb,ub)
         [ (i, array (0,r+s)
               [ (k, x!((i-r+k)`mod`(ub-lb+1))) | k <- [0..(r+s)] ])
           | i <- [lb..ub]
         ]

fanpair :: Ix a => (Array a b,Array a c) -> Array a (b,c)
fanpair (x,y) =
     if (bounds x) == (bounds y)
        then array (bounds x)
              [ (i,(x!(i),y!(i))) | i <- range (bounds(x)) ]
        else error "Pairing two non-conformant arrays!"

fanproj1:: (Ix a) => Array a (b,c) -> Array a b
fanproj1 x = fanmap fst x
```

Figure 3: Haskell definition of part, pair and proj1.

Figure 3 shows the functional semantics of some of the arrangements. Note that function part in FAN translates to a series of functions `fanpart1`, `fanpart2`, etc. in Haskell. For instance, `fanpart1` returns an array of arrays; each element of the result array is a slice of the input array x. The first and the last element of x are considered neighbors. The Haskell functions corresponding to the other arrangements can be found in the appendix.

3.2.2 Two-dimensional arrays

Some of the arrangement skeletons for two-dimensional arrays are extensions of the skeletons defined for one-dimensional arrays:

- copy (n, m) s creates a two-dimensional array of extent $n \times m$, filled with copies of s.
- split (p, q) x slices a matrix in $p * q$ non-overlapping submatrices.
- part $((r, s), (p, q))$ x returns a matrix t of the dimensionality and size of x; each element $t(i, j)$ is an array of the $(r + s + 1) * (p + q + 1)$ elements surrounding $x(i, j)$.

The corresponding Haskell functions are similar to the one-dimensional case (see the appendix).

A specific arrangement for multi-dimensional arrays is shown in Figure 4. Function rearrange f x takes as input a rearrangement pattern f and permutes the elements of an array x according to f; here, `ixmap` is a standard Haskell function permuting array elements. Function f must be bijective. Using rearrange, it is easy to define the transposition of an array of any dimensionality.

```
fanrearrange :: Ix a => (a -> a) -> Array a b -> Array a b
fanrearrange f x = ixmap (bounds x) f x

fantranspose2:: (Ix a) => Array (a,a) b -> Array (a,a) b
fantranspose2 x = fanrearrange (\(i,j) -> (j,i)) x

fantranspose3:: (Ix a) => Array (a,a,a) b -> Array (a,a,a) b
fantranspose3 x = fanrearrange (\(i,j,k) -> (k,j,i)) x
```

Figure 4: Haskell definition of multi-dimensional array rearrangements.

3.3 Iteration skeletons

FAN offers three different skeletons for iterative computation: forloop iterates in a finite loop, and looprepeat and loopwhile iterate, potentially infinitely, until a given condition is met. The Haskell definition of loopwhile is given in Figure 5. The others can be found in the appendix.

```
fanloopwhile :: (a -> Bool) -> (a -> a) -> a -> a
fanloopwhile c f x =  if c x then fanloopwhile c f (f x) else x
```

Figure 5: Haskell definition of loopwhile.

4 Case Studies: Formulation in FAN

In this section, we illustrate the use of FAN in the specification of some small example problems. The examples will be used later on to demonstrate our approach. In particular, we will discuss how the performance of the programs described in this section can be predicted, and how transformations can be applied to improve their performance.

4.1 Polynomial evaluation

Consider the problem of evaluating a polynomial

$$a_1 * x + a_2 * x^2 + \ldots + a_n * x^n$$

at m points $y_1, \ldots, y_m$. We start the design process by determining how the parallelism inherent in polynomial evaluation can be expressed. To this end, we try to use skeletons with a larger amount of parallelism: for instance, we prefer map and reduction to (sequential) iteration. The

most direct way of computing a polynomial at a coordinate vector $ys = y_1, \ldots, y_m$ is to compute the vectors of powers, $ys^i = [y_1^i, \ldots, y_m^i]$, $i = 1, \ldots, n$, then multiply all values in ys^i by the polynomial coefficient a_i and, finally, sum up all intermediate results.

This simple idea can be expressed by the following FAN algorithm:

```
pol-eval1 (in ys : Array m Scalar, as : Array n Scalar, out zs : Array m Scalar)
  ts = scanL (*) (copy n ys);
  ds = map (*_sa) (pair (as,ts));
  zs = reduce (+) ds;
```

The algorithm takes as input vectors as and ys (represented as arrays) and proceeds as follows:

- copy n ys returns an array of n copies of vector ys;
- scanL $(*)$ computes ts : Array n (Array m Scalar), where $ts[i] = ys^i$ for all i;
- map $(*_{\mathrm{sa}})$ (pair (as,ts)) computes array ds of type Array n (Array m Scalar), such that $ds[i] = [a_i * y_1^i, \ldots, a_i * y_m^i]$; Operator $*_{\mathrm{sa}}$ takes a scalar and an array and multiplies each array element by the scalar:

$$x *_{\mathrm{sa}} [y_1, \ldots, y_k] = [x * y_1, \ldots, x * y_k]$$

- reduce $(+)$ ds sums up all rows of ds elementwise and returns the results in the output array zs, such that $zs = [\sum_{i=1}^{n} a_i * y_1^i, \ldots, \sum_{i=1}^{n} a_i * y_m^i]$.

Note that operator $*$ in the scan skeleton and operator $+$ in the reduction skeleton are overloaded to work both on scalars and on arrays of equal shape; in the latter case, the operator is applied elementwise.

4.2 Maximum segment sum

Our next example is the famous *maximum segment sum* (MSS) problem – a *programming pearl* [6], studied by many authors [7, 12, 34, 36, 38]. Given a one-dimensional array of integers, function *mss* finds a contiguous array segment, whose members have the largest sum among all such segments, and returns this sum. For example, $mss\ [2, -4, 2, -1, 6, -3] = 7$, where the result is contributed by the segment $[2, -1, 6]$.

A first FAN program for computing *mss* could be quite simple:

```
mss-alg1 (in x : Array n Scalar, out r : Scalar)
  s = scanL (op1) x;
  r = reduce (max) s;
```

where operator $op1$ is defined as follows:

$$a_1\ op1\ a_2 \;=\; \max(a_1 + a_2, a_2) \tag{1}$$

Algorithm mss-alg1 takes array x as input, and proceeds intuitively as follows:

- The first stage, scanL $(op1)$, produces array s of type Array n Scalar, whose ith element is the maximum sum of segments of x ending in position i.

- The second stage, reduce (max), computes the maximum element of array s, thus yielding the desired value, r, with $r = mss\, x$.

This first algorithm has several nice features. First, it is intuitively clear, and its correctness with respect to the specification of the MSS problem can be proved. Second, it consists of only two skeletons, scan and reduction, which are both potentially easy to parallelize. However, a closer look reveals also a serious drawback: the scan skeleton cannot be parallelized in this case, since operator $op1$, defined by equation (1), is not associative. Thus, the asymptotic time complexity of this algorithm is linear in the length of the input array.

In order to enable a parallelization, we must construct an associative operator for use in the scan skeleton. A common technique for this purpose is the introduction of auxiliary variables. In our case, we define the new, associative operator on pairs, such that the original operator is modeled by the first element of the pair:

$$(a_1, b_1)\, op2\, (a_2, b_2) \;=\; (\;\max\,(a_1{+}b_2, a_2)\,,\; b_1{+}b_2\;) \tag{2}$$

Note that it is exactly this construction which is the main reason for having data arrangement skeletons like pairing and projection in the FAN language. In particular, the following holds:

$$a_1\, op_1\, a_2 \;=\; \mathsf{proj1}\ ((a_1, a_1)\, op_2\, (a_2, a_2))$$

Consequently, $\mathsf{scanL}\ (op_1)\, x \;=\; \mathsf{proj1}\,(\mathsf{scanL}\,(op_2)\,(\mathsf{pair}(x, x)))$. Using the new operator op_2, we can rewrite the initial program as follows:

```
mss-alg2 (in x : Array n Scalar, out r : Scalar)
  y = scanL (op2) (pair (x,x))
  s = proj1 y;
  r = reduce (max) s;
```

Note that the first two statements of mss-alg2 are semantically equivalent to the first statement of mss-alg1, but the asymptotic complexity has become logarithmic. A transformation of the MSS program with the goal of improving the performance further is described in Subsection 8.2.

4.3 A simple cellular automaton

The *game of life* [16] is a simple cellular automaton that models the world as a two-dimensional grid of cells which can adopt two states: *dead* or *alive*. The grid is represented by a boolean matrix w in which each element corresponds to a cell: $w(i, j)$ is true iff the corresponding cell is alive. Evolution is simulated by iteratively updating the state of the grid at successive instants of time. The state of a cell at instant t_h is determined by the state of its neighbors at the previous instant t_{h-1}: a cell is alive if it has exactly three alive neighbors; if it has exactly two alive neighbors it maintains its state; otherwise, it is dead. The computation terminates when the world has reached a stable state, i.e., when an update does not incur a change of any cell state.

For the sake of illustration, we consider a neighborhood of eight cells, as shown in Figure 6.

An intuitive solution is to update all cells simultaneously, check the termination condition and, if it is not met, repeat. This is expressed by the following FAN algorithm:

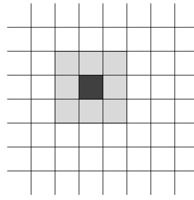

Figure 6: Neighborhood for the game of life cellular automaton.

life.step (in w : Array (n, m) Scalar, d : Scalar, out $w1$: Array (n, m) Scalar, y : Scalar);
  $w1 =$ map $update$ (part $((1,1),(1,1))$ w)
  $c =$ map $(\neq)$ (pair $(w, w1)$)
  $y =$ reduce (or) c

game.life (in x : Array (n, m) Scalar, out w : Array (n, m) Scalar);
  $(w, _) =$ loopwhile is_true life.step $(x,$ True$)$

The algorithm is composed of two FAN functions: life.step updates the cells and computes the termination condition, and game.life iterates the grid update and checks for termination.

Function life.step is computed in three steps as follows:

- First, part returns an array b of arrays in which each element $b(i, j)$ is a 3×3 matrix with the cell (i, j) and the stencil of its neighbors. *update* is a sequential function which takes the stencil of a cell and computes the update as described above, returning the new cell value. map *update* t returns the new grid state applying *update* to all cells.
- Matrices w and $w1$, modeling the new and old state of the grid, are paired and the corresponding elements are compared. This is done by the second map which returns a matrix of boolean values c. $c(i, j)$ is True if and only if the values of $w(i, j)$ and $w1(i, j)$ differ.
- Finally, the reduce skeleton reduces all elements of c using the boolean *or* operator.

Function game.life iterates life.step until function *is_true* returns False. *is_true* returns the second element of the input pair, which is False only when no cell has changed in the last grid update.

5 Transformation Rules

The design of a skeleton program consists of the transformation and the cost estimation steps. The goal is to derive a program with the best performance estimate, which requires very often a reduction of the number of communications. A rich set of transformation rules for various skeletons has been developed recently [2, 18, 20, 39]. These rules transform compositions of two or three skeletons – like copy, reduce, scan, etc. – into more efficient expressions. Some transformations, e.g., the scan-reduction fusion, originated in the functional Bird-Meertens formalism [7] for sequential algorithms, some are new.

In this section, we describe some of the transformations rules available in the repository of our transformation engine. We use these rules to transform our case studies in Section 8. More on these and other rules can be found elsewhere [20].

We present our transformation rules in a format consisting of four boxes, from top to bottom:

1. the FAN program fragment before the transformation (the "left-hand side" of the rule),
2. the fragment after the transformation (the "right-hand side"),
3. a precondition stating when the rule is applicable (optional),
4. local definition(s) of function(s) used in the rule (optional).

The first rule we discuss transforms a computation of scan followed by reduction:

Rule SR-ARA

b = reduce $Op2$ (scanL $Op1$ a)
b = proj1 (reduce $Op3$ (pair (a,a)))
If $Op1$ distributes forward over $Op2$
$(a_1, b_1)\, Op3\, (a_2, b_2) \;=\; (a_1\, Op2\, (b_1\, Op1\, a_2),\, b_1\, Op2\, b_2)$

The name of the rule, SR-ARA, hints on the transformation it specifies: "**S**can;**R**educe → **A**rrange;**R**educe;**A**rrange", where "arrange" stands for any arrangement skeleton. Rule SR-ARA expects two operators as parameters, $Op1$ and $Op2$, and makes use of a local operation, $Op3$, constructed of them. The rule transforms a sequence of two relatively costly skeletons – a scan and a subsequent reduction – into one reduction, with a pre- and a postarrangement of the data. The prearrangement is pairing and the postarrangement is projection. Both are simple operations requiring no communication. Thus, the SR-ARA rule usually saves communication and improves performance (its performance analysis follows in the next section).

Sometimes, when trying to apply rule SR-ARA, we might find that the scan and reduction operation are separated by some skeleton performing data rearrangements. In this case, it may be useful to apply rules which swap a reduce with an arrange skeleton, like the following rule AR-RA for reducing a projection over a list of pairs:

Rule AR-RA

b = reduce $Op1$ (proj1 a)
b = proj1 (reduce $Op2$ a)
$(a_1, b_1)\, Op2\, (a_2, b_2) \;=\; (a_1\, Op1\, a_2,\, b_1\, Op1\, b_2)$

This rule presumes that the input, a, is an array of tuples; we present here the version for pairs. The rule is universally applicable, since it has no If box. The AR-RA rule is applicable in both directions, with completely different consequences. From right to left, the transformation eliminates redundant computations, thereby reducing the overall cost of the composition. From left to right, the rule restructures the program in order to enable further transformations, by pushing the projection behind the reduction, at the price of redundant computations. Since the left-to-right application is useful only if it enables subsequent transformations, such as SR-ARA, it makes sense to combine both the AR-RA and the SR-ARA rule into a new rule, SAR-ARA:

Rule SAR-ARA

$c =$ reduce $Op2$ (proj1 (scanL $Op1$ a))
$c =$ proj1 (proj1 (reduce $Op3$ (pair(a,a))))
If $Op1$ distributes forward over $Op4$
$(a_1, b_1)\, Op3\, (a_2, b_2) \;=\; (a_1\, Op4\, (b_1\, Op1\, a_2)\, , b_1\, Op1\, b_2)$ $(a_1, b_1)\, Op4\, (a_2, b_2) \;=\; (a_1\, Op2\, a_2,\; b_1\, Op2\, b_2)$

Note that, whereas operator $Op2$ works on single elements, operators $Op1$ and $Op4$ are defined for pairs, and $Op3$ works on pairs of pairs.

Another transformation, rule CS-CM, states that a copy operation followed by a scanL with operator $Op1$ can always be transformed into a copy followed by some local computation:

Rule CS-CM

$b =$ scanL Op (copy n a)
$b =$ map$_\#$ f (copy n a)
$f\ i\ x\ = fst(repeat\ i\ (x, x))$ $repeat\ \ k\ x =$ if $k = 0$ then x else $repeat$ (k div 2) (if (k mod $2 = 0$) then $e\ x$ else $o\ x$) $e(t, u) = (t, u\ Op\ u)$, $o(t, u) = (t\ Op\ u, u\ Op\ u)$

Here, function *repeat* performs the local computation of g^k using a logarithmic-time algorithm. It traverses the binary digits of number k from the least significant to the most significant. If the digit is 0, *repeat* applies function e, if the digit is 1 it applies function o. When the computation is finished, we select the first element of the result pair (*fst*). More details can be found in [20].

Finally, we consider the map-fusion law [7] which suggests the transformation of a composition of two maps into one map. In FAN, we extend the rule for both versions of the map skeleton, map and map$_\#$; moreover, they may be interspersed with an arrangement skeleton, e.g., pair:

Rule M$_\#$M-M$_\#$

$b =$ map$_\#$ f a $c =$ map g b
$c =$ map$_\#$ h a
$h\ i\ x = g\ (f\ i\ x)$

Rule M$_\#$AM-M$_\#$

$b =$ map$_\#$ f a $c =$ map g (pair (d,b))
$c =$ map$_\#$ h (pair (d,a))
$h\ i\ (x,y) = g\ (x,\ f\ i\ y)$

6 An Execution Model, and Performance Prediction

The functional semantics of FAN provided in Section 3 does not prescribe any particular execution model. This allows for several different parallel implementations of one FAN program, depending on the machine architecture or the system software used. On the other hand, this freedom of choice prevents a practical estimation of the program performance and is, thus, a hindrance for performance-guided design by transformation. One of the advantages of FAN is that a small number of execution models can cover the main classes of parallel and distributed systems.

In this section, we describe one such execution model, oriented towards parallel machines with distributed memory, programmed using communication libraries like MPI and PVM. We use this model to estimate the execution time of the main FAN skeletons, depending on the main machine parameters like the processor and channel speed and the number of processors.

We assume a processor network with bidirectional communication links: any two processors can send messages of size m to each other simultaneously in time $t_s + m * t_w$, where t_s is the start-up time and t_w is the per-word transfer time. A processor is allowed to send/receive messages on only one of its links at a time. We ignore the computation time it takes to split or concatenate vectors within a processor.

6.1 One-dimensional arrays

Let us first consider the cost (i.e., time required) for executing each skeleton on an array of n elements on p processors, where $p \leq n$. We assume that all input arrays are distributed blockwise on processors, while input scalars are replicated. Skeletons in the body of a FAN program are executed in sequence on all available processors, and the total cost of a program is the sum of the costs of all skeletons in the body. The results of our analysis of the execution schemata are listed in Table 1 and explained subsequently.

FAN Operation	Time required
map f x	$m * t_f$
proj1 x	0
pair (x,y)	$2 * m * t_{copy}$
copy n x	$p * t_s + m * (p-1) * t_w / p$
part (r, s) x	$2 * t_s + (r+s) * t_w$
reduce $(\oplus)$ x	$m * t_\oplus + \log p * (t_s + t_w + t_\oplus)$
scanL $(\oplus)$ x	$2 * m * t_\oplus + \log p * (t_s + t_w + 2 * t_\oplus)$

Table 1: Costs of skeletons on one-dimensional arrays.

Since $p \leq n$, $m = \lceil n/p \rceil$ contiguous elements are allocated on each processor. When executing map f, each processor applies f sequentially to all local elements. Since this is done in parallel on all processors, the total cost of map is $m * t_f$, where t_f is the cost of applying f to a single element.

The behavior and cost of arrangement skeletons depends on the amount of data being moved. Projection proj1 does not incur any cost, since it only selects one local value and ignores the rest. Consider now pair (x,y). Since x and y have the same length, our allocation strategy places $x[i]$ and $y[i]$ on the same processor. Thus, processors can compute the local block of the result array t using only local data. Each $t[i]$ requires two copies of the corresponding elements of x and y. The total cost is $2 * m * t_{\mathsf{copy}}$, since all processors can compute the local part of result in parallel. The cost of skeleton copy n x depends on whether x is a scalar or an array. Scalars are replicated on all processors, which requires $m = \lceil n/p \rceil$ local copies to construct the local portion of the results.

The cost is 0 if we choose to make the implementation code a bit more complex and share a single result copy on each processor. When x is an array, we gather all the blocks distributed on different processors, which costs $p * t_s + m * (p-1) * t_w / p$. Finally, part (r, s) x requires that each processor reads $r + s$ elements from its two neighbors, which costs $2 * t_s + (r + s) * t_w$.

Skeleton reduce $(\oplus)$ x is executed in two steps. The first step reduces data locally to each processor according to the operator $\oplus$; this costs $m * t_\oplus$. Then, a global reduce step is computed in $\log p$ phases using a butterfly-like communication pattern [17]. Note that we do not assume any particular communication topology: in our model, the communication time is independent of the two processors which communicate. In each phase, two elements of the vector are exchanged pairwise between processes. Since $t_s + m * t_w$ is the cost for exchanging m-sized messages between two processors, the cost estimate in the global step is $\log p * (t_s + t_w + t_\oplus)$. Adding up the cost of the two steps, we obtain $m * t_\oplus + \log p * (t_s + t_w + t_\oplus)$ as the cost of reduce $\oplus$.

The execution model for scanL $\oplus$ x differs from reduction in that we need three steps. The first step computes scanL locally in parallel on the blocks of data resident on each processor (cost: $m * t_\oplus$). The second step executes a global scanL combining the rightmost element of the vector resulting from the local scan on each processor. This global scan of p values can be computed using a schema similar to the one used for reduction, except that reduction requires one operation per element received whereas scan requires two. The second step costs $\log p * (t_s + t_w + 2 * t_\oplus)$; its result is a vector of length p distributed one element per processor. In the third step, each processor combines the local element of the vector from step two with the result of the local scan computed in step one, which costs $m * t_\oplus$. Thus, the total cost is $2 * m * t_\oplus + \log p * (t_s + t_w + 2 * t_\oplus)$.

The costs in Table 1 can be generalized straight-forwardly to the two-dimensional arrangement skeletons. Note that, if t_f and $t_\oplus$ are not uniform, different scheduling strategies must be adopted in the execution model in order to keep the load of the processors balanced and allow for similarly simple prediction models.

6.2 Costing the rules

The costs of our transformation rules are listed in Table 2. We assume the cost of copying and local operations to be uniformly constant, and normalize t_s and t_w with respect to this cost.

The cost of the left-hand side of rule SR-ARA is obtained by adding the costs of reduce and scanL (vectors a and b have length n). On the right-hand side, initial pairing doubles the size of the elements in vector b with respect to vector a. Thus, reduce works on elements of length 2, and the cost of sending/computing is $2 * (t_w + 1)$.

The cost of rule SAR-ARA is derived assuming that, before the application, a and b are vectors of pairs of length n and, after the application, a is a vector of pairs of length n and b is a vector of quadruples of length n. Thus, before the application, scanL and reduce work on elements of length 2 while, after the application, the composed reduce works on elements of length 4.

We can use these costs to state the conditions under which a particular rule is worth applying; see the rightmost column of the table. Rule SR-ARA always reduces the cost, whereas rule AR-RA, applied alone, increases the cost (as expected). The combined rule SAR-ARA reduces the cost if a particular condition relating the machine characteristics and the problem size holds:

$$(t_s + t_w + 2) * \log p \;>\; 2m$$

Rule	Time left hand side	Time right hand side	Improves if
SR-ARA	$3*m+\log p*(2*(t_s+t_w)+3))$	$2*m+\log p*(t_s+2*t_w+2)$	always
AR-RA	$m+\log p*(t_s+t_w+1)$	$2*m+\log p*(t_s+2*(t_w+1))$	never
SAR-ARA	$3*m+\log p*(2*(t_s+t_w)+3)$	$5*m+\log p*(t_s+t_w+1)$	$(t_s+t_w+2)*\log p>2m$
CS-CM	$p*t_s+m*(t_w+\frac{t_w}{p}+1)$ $+\log p*(t_s+t_w+2)$	$p*t_s+m*(t_w+\frac{t_w}{p}+1)$	always
$M_{\#}$M-$M_{\#}$	$2*m$	m	always
$M_{\#}$AM-$M_{\#}$	$4*m$	m	always

Table 2: Performance estimates for optimization rules

The remaining three rules always reduce the cost.

7 The FAN Transformation Tool

In this section, we describe a transformation tool which allows the user to write, evaluate and transform FAN programs while preserving their functional semantics and possibly improving their performance. The tool has an interactive behavior. Given an initial FAN algorithm, it suggests a set of transformation rules along with their expected performance impact. The programmer chooses a rule to be applied and, after the application, the tool looks for new matches. This process is repeated until the programmer deems the resulting program satisfactory or there are no more applicable rules.

The strategy of program transformation is in the hands of the programmer since, in general, the rewriting calculus of FAN is not confluent: applying the same rules in a different order may lead to programs with different performance. The best transformation sequence may require a (potentially exponential) exhaustive search.

In the following subsections, we define an abstract representation of FAN programs and transformation rules, describe the algorithm used for rule matching, and sketch the structure of the tool.

7.1 Representing programs and rules

The FAN transformation system is basically a term-rewriting system. Both FAN programs and transformation rules are represented by means of labeled trees (so-called *dependence trees*). Thus, the search for applicable rules reduces to the well established theory of subtree matching. The tool attempts to annotate as many nodes of the tree as possible with a *matching rule instance*, i.e., a structure describing which rule can be used to transform the subtree rooted at the node, together with the information describing how the rule has been instantiated, the performance improvement expected and the applicability conditions to be checked (e.g., the distributivity of one operator over another).

The dependence tree is essentially an abstract syntax tree in which each non-leaf node represents a skeleton, with children representing the skeleton parameters that may in turn be skeletons or sequential functions. The leaves are either sequential functions, constants or special nodes,

Arg( ), for program arguments. The dependence tree of a program is defined constructively, combining information held in the parse tree (PT) and in the data flow graph (DFG) of the program. A compact representation of the DFG is obtained by adding edges to the parse tree. Both structures – and the algorithms for their construction – are part of standard compiler technology. The parse tree, the data flow graph and the data dependence tree for the polynomial evaluation algorithm pol-eval1, introduced in Subsection 4.1, are shown in Figure 7. The node labeled with *DPblock* represents the root of a data-parallel FAN program; nodes *Arg(as)* and *Arg(ys)* represent the input data of the program. Each edge in the dependence tree represents the dependence of the head node on the data produced by the tail node.

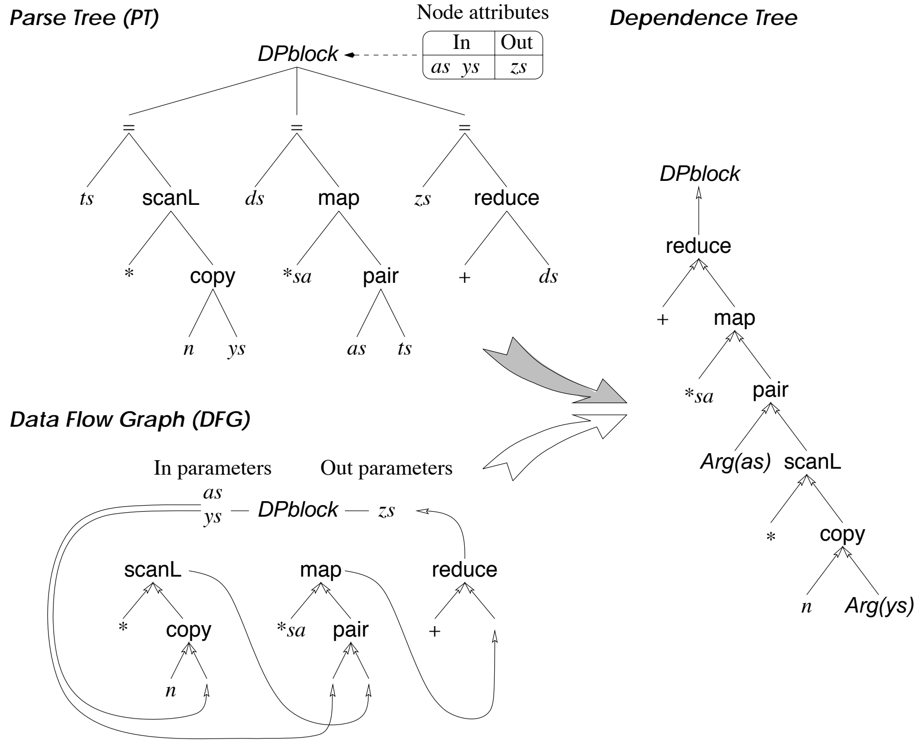

Figure 7: The parse tree, the data flow graph and the dependence tree of polynomial evaluation. FAN skeletons are in serif font. Special nodes are in *slanted serif* font.

The dependences shown in Figure 7 are rather simple. In general, as shown in Figure 8, a data structure produced by one FAN statement may be used by more than one statement in the rest of the program. We have two choices: (1) to keep a shared reference to the expression (tree), or (2) to replicate it. In option (1), the data flow can no longer be fully described with a tree. Moreover, sharing the subtrees rules out the possibility of applying different transformations at the shared expression (tree) for different contexts. The FAN transformational engine adopts the second option, allowing us to map the data flow graph to a tree-shaped dependence structure. The drawback of replicating expressions is a possible explosion of the code size when we rebuild a FAN program from the internal representation. To avoid this, the engine keeps track of all the

replications made. This ensures a single copy of all replicated subtrees that have not been subject to an independent transformation.

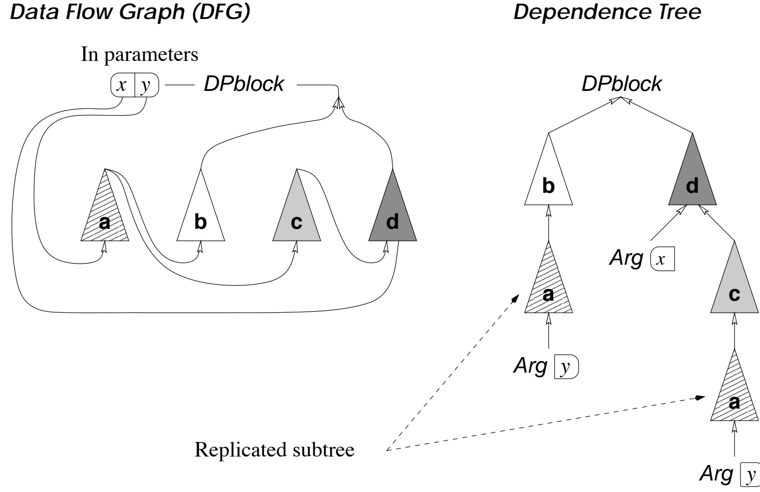

Figure 8: Replicating shared trees. Each triangle stands for a tree representing a FAN expression.

Let us consider how a FAN rule $L = R$ is represented, where L and R are pieces of FAN programs with variables ranging over FAN types. We require that every variable occurring in L must occur also in R, that L is not a variable, and that every variable in L occurs at most once. Moreover, a variable may be constrained to assume a specified type or satisfy a specific property (e.g., we may require an operator to distribute over another operator). The left-hand side L of a rule is called a *pattern*.

As an example, Figure 9 depicts the internal representation of rule SR-ARA from Section 5. We represent the two sides of the rule as dependence trees, some leaves of which are variables represented by circled numbers. During the rule application, the instantiations of the left-hand side variables are substituted for their counterparts on the right-hand side. We call the set of circled figures a *rule interface*. Since, in all of our rules, the left-hand and the right-hand sides have the same variables, occurring the same number of times, the interfaces of both rule sides are the same. Figure 9 demonstrates also how the conditions of applicability and the performance of the two sides of a rule are reported to the programmer.

7.2 Rule matching

Since programs and rules are represented by trees, we can state the problem of finding a candidate rule for transforming an expression as the well-known *subtree matching problem* [25, 27, 30]. In the most general case, given a pattern tree P and a subject tree T, all occurrences of P as a subtree of T can be determined in time $\mathcal{O}(|P| + |T|)$ by applying a fast string matching algorithm to a proper string representation [30]. Our problem is more complicated: patterns are matched against a subject which may be modified incrementally by the sequence of rule applications. Therefore, we distinguish the *preprocessing phase* for a given set of skeletons and their transformation rules, and the *matching phase* for a particular subject tree.

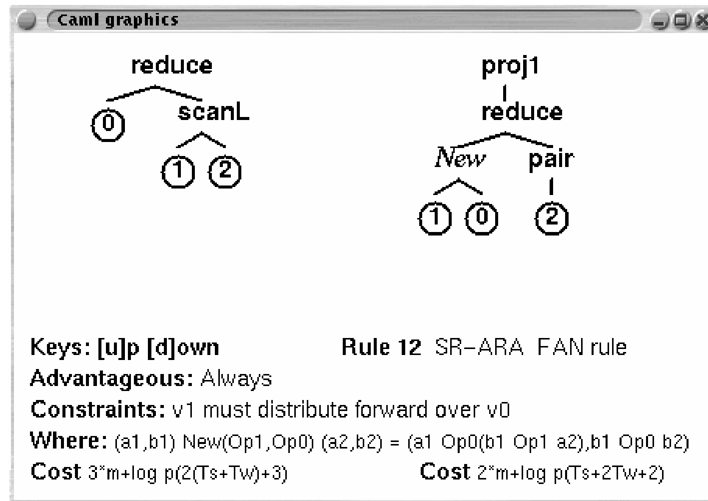

Figure 9: Internal representation of rule SR-ARA, conditions of its applicability and performance of the two sides of the rule.

Minimizing the matching time is our first priority. The Hoffmann-O'Donnell bottom-up algorithm [25] fits our problem better than the string matching algorithm mentioned before. With it, we can find all occurrences of a forest of patterns, F, as subtrees of T in time $\mathcal{O}(|T|)$, after appropriate preprocessing of the pattern set. Moreover, the algorithm is efficient in practice: after the preprocessing, all the occurrences of elements in F can be found with a single traversal of T. The algorithm works in two steps: it constructs a *driving table*, which contains the patterns and their interrelations; the table is then used to drive the matching algorithm.

The complexity of the generation of the driving table (the preprocessing phase) is, in the worst case, exponential in the cardinality of the pattern set. Nevertheless, there is a broad class of pattern sets which can be preprocessed in polynomial time/space in the size of the set: all sets yielding *simple pattern forests*. For a formal definition, we refer to Hoffmann-O'Donnell [25] and provide only a brief explanation here.

Let $F = \{P_1, P_2, \ldots\}$ be a pattern set in which each pattern P_i is a tree. The set of all subtrees of trees in F is called a *pattern forest*. Now, let P and P' be pattern trees. P *subsumes* P' if, for all subject trees T, P has a match in T implies that P' has a match in T. Then P is *inconsistent* with P' if there is no subject tree T matched by both P and P'. P and P' are *independent* if there exist T_1, T_2, and T_3 such that T_1 is matched by P but not by P', T_2 is matched by P' but not by P, and T_3 is matched by both P and P'. Given distinct patterns P and P', exactly one of the three previous relations must hold. A pattern forest is called *simple* if it contains no independent subtrees. For instance, the pattern forest including the pattern trees $P = a(b, \nu)$ and $P' = a(\nu, c)$ is not simple, since P and P' are independent w.r.t. $T_1 = a(b, b), T_2 = a(c, c), T_3 = a(b, c)$; here, $P = a(b, \nu)$ denotes a tree with root a and subtrees b and ν.

Let us give an example in the FAN framework. Labels are names of either skeletons or sequential functions. Since, in pattern matching, the names of variables are not significant, we replace all variables $\nu_1, \nu_2, \ldots$ with ν (the distinction of names become important in the application of rules). The

pattern of rule SR-ARA (the left tree in Figure 9) is represented by $P = \mathsf{reduce}(\nu, \mathsf{scanL}(\nu, \nu))$. The pattern of rule CS-CM is represented by $P' = \mathsf{scanL}(\nu, \mathsf{copy}(\nu, \nu))$. The pattern forest associated with $F = \{P, P'\}$ is $PF = \{\nu, \mathsf{scanL}(\nu, \nu), \mathsf{copy}(\nu, \nu), \mathsf{scanL}(\nu, \mathsf{copy}(\nu, \nu)), \mathsf{reduce}(\nu, \mathsf{scanL}(\nu, \nu))\}$. Since there is no pair of independent subtrees in PF, the pattern forest PF is simple.

Our current set of FAN rules can be described fully by a simple pattern forest. Simple pattern forests suffice even for the implementation of much more complex languages like LISP and the combinator calculus [24]. In addition, since the driving table depends only on the language and on the list of rules, it can be generated once and for all for a given set of rules and stored permanently for several subsequent match searches.

7.3 Tool architecture and implementation

The transformation engine applies the matching algorithm in an interactive cycle as follows:

1. Use the matching algorithm to annotate the dependence tree with a matching rule.
2. Check whether the found rules satisfy the type constraints and whether the side conditions hold (possibly interacting with the user).
3. Apply the performance estimates to establish the effect of each rule.
4. Request the programmer to select one rule for application. In case no rule is applied, terminate; otherwise start again with Step 1.

We envision the matching engine as a part of a general tool implementing the FAN transformation framework. The global tool structure is depicted in Figure 10 (the part already implemented is highlighted with a dotted box). The whole system has two main capabilities: the conversion of FAN programs into dependence trees and the transformation engine working on dependence trees.

The system architecture is divided into five basic blocks:

1. The *Front End* converts a FAN program into a parse tree (PT) and a data flow graph (DFG).
2. The *Normalization* uses the PT and DFG to build the dependence tree both for the FAN program and for the set of transformation rules.
3. The *Rule Manager* implements the matching phase; it delivers a matching table to drive the transformation engine. This table may be stored in a file.
4. The *Transformation Engine* interacts with the user and governs the transformation cycle.
5. The *Back End* generates a new FAN program from the internal representation.

A prototype of the system kernel (highlighted in Figure 10 with a dotted box) has been implemented in Objective Caml 2.02. Our implementation is based on an abstract data type (ADT) which describes the internal representation (dependence tree) and the functions working on it.

The implementation is very general and can handle, via instantiation of the ADT, different languages with the requirement that rules and programs are written in the same language. Moreover, since several execution models and many cost calculi may be associated with the same language, any compositional way of describing program performance may be embedded in the tool by just

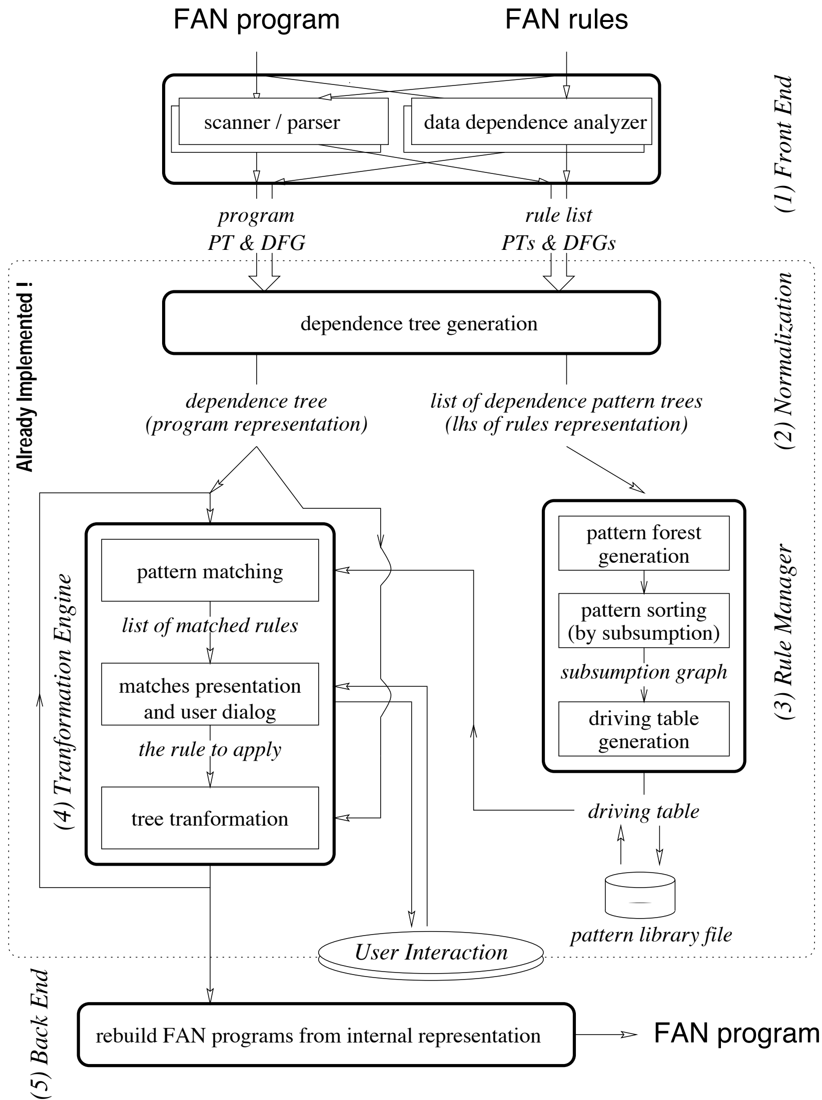

Figure 10: Global structure of the FAN transformation system.

instantiating the performance formulae of every construct. We call a cost calculus *compositional* if the performance of a language expression is either described by a function of its components or by a constant.

The prototype of the FAN transformation tool is currently running under both Linux and Microsoft Windows. A graphical interface is implemented using the embedded OCaml graphics library.

8 Case Studies: Design by Transformation

In this section, we discuss how our transformation tool is used in the program design process for algorithms mss-alg2 and pol-eval1, introduced in Section 4. The transformation for the game of life is omitted due to its simplicity.

8.1 Polynomial evaluation

Algorithm pol-eval1 is rather communication-intensive: it broadcasts all copies of ys, and then it applies two skeletons, scanL and reduce, both communicating m-sized vectors. We can improve the program using the FAN transformation tool. The transformation process is depicted in Figure 11. The left part shows the program structure during the two transformation steps. The right part shows the user dialogues which propose the applicable rules to the programmer; windows (c) and (f) show how two transformation rules have been applied.

First, the tool displays the internal representation of the program (Figure 11(a)) and proposes to the user rule CS-CM (Figure 11(b)). According to our cost estimates, every application of rule CS-CM results in a performance improvement. Let us suppose the user chooses to apply rule CS-CM (Section 5). The system will transform the program by instantiating the operator $Op1$ with $*$ (Figure 11(c)). In this case, functions e and o are of the following type:

$$e, o : (\mathsf{Array}\ m\ \mathsf{Scalar}, \mathsf{Array}\ m\ \mathsf{Scalar}) \to (\mathsf{Array}\ m\ \mathsf{Scalar}, \mathsf{Array}\ m\ \mathsf{Scalar})$$

and are instantiated by the system as follows: $e\ (t, u) = (t, u * u)$, and $o\ (t, u) = (t * u, u * u)$.

The resulting version of polynomial evaluation is (Figure 11(d)):

```
pol-eval2 (in ys : Array m Scalar, as : Array n Scalar, out zs :  Array m Scalar)
  ts = map# (f) (copy n ys);
  ds = map (*sa) (as,ts);
  zs = reduce (+) ds;
  where
 f i x =  fst (repeat i (x, x))
```

Next, the transformation engine will propose to transform our program using rule $\mathrm{M}_{\#}$AM-$\mathrm{M}_{\#}$, to take advantage of data locality (Figure 11(e)–(f)). In particular, the block distribution of data implies that, for all i, $as[i]$ and the ith rows of ds and ts are located on the same processor, so that the computation in $\mathsf{map}_{\#}$ and map and the data arrangement (as, ts) can be performed without communication.

The final, transformed program is (Figure 11(g)):

```
pol-eval3 (in ys : Array m Scalar, as : Array n Scalar, out zs :  Array m Scalar)
  ds = map# (h) (as, copy n ys);
  zs = reduce (+) ds;
  where
 h i (x,y) = (*sa) (x, f i y)
```

Experiments have shown that program pol-eval3 runs indeed faster than pol-eval1 on different target platforms [5].

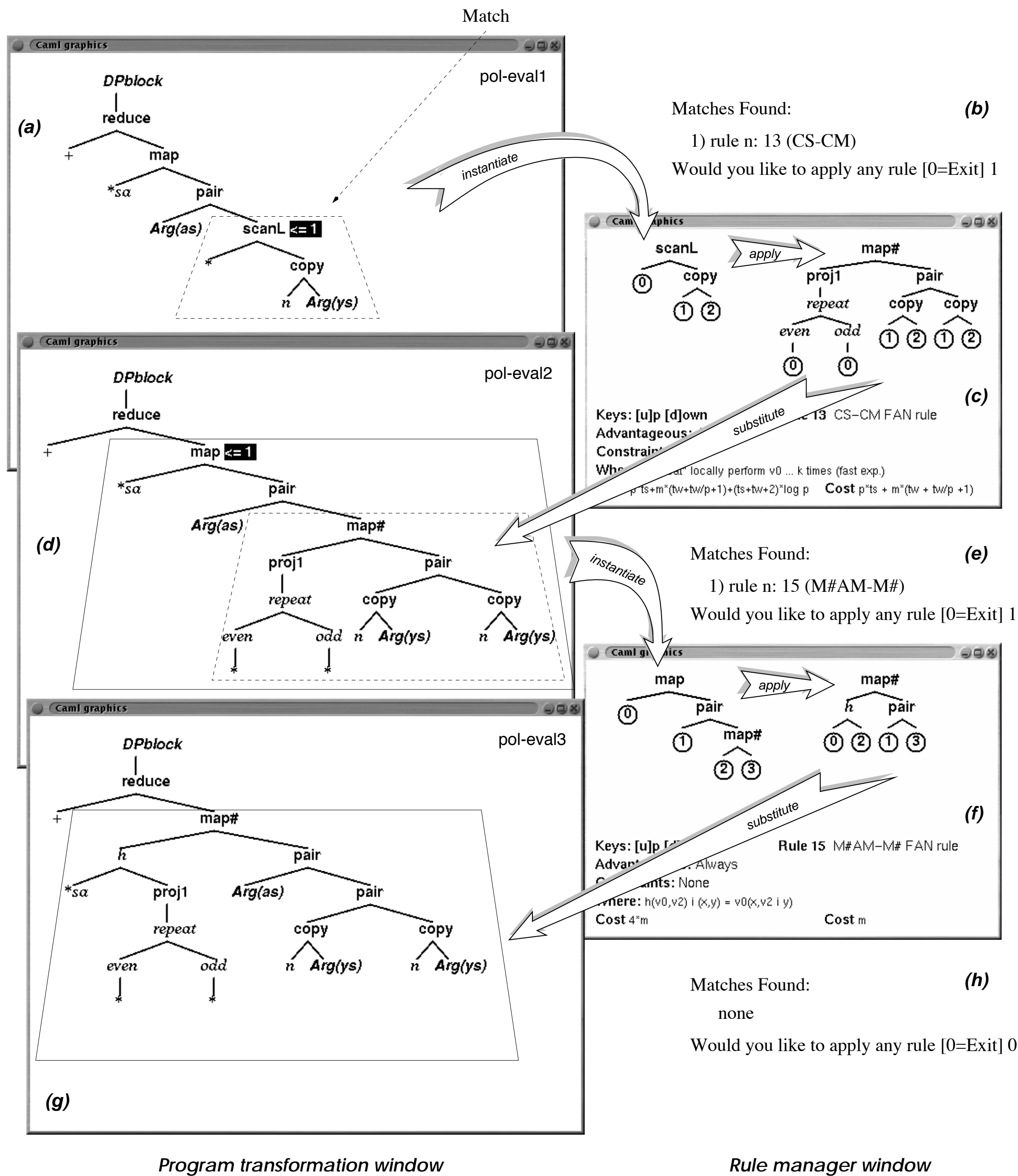

Figure 11: Transformation of polynomial evaluation using the FAN tool.

8.2 Maximum segment sum

Algorithm mss-alg2 has logarithmic complexity in the size of the input array. This follows from the logarithmic complexity of both scan and reduction with associative operators, and from the constant parallel complexity of map. Thus, the algorithm is asymptotically optimal. However, its performance in practice can suffer from the fact that it exploits two collective operations, each of which involves a considerable amount of interprocessor communication.

Note that the algorithm presented in a popular textbook [1] may have an even higher cost than mss-alg2, since it exploits three collective operations: two scans and one reduce. The user's responsibility is not only to produce an algorithm but also to understand whether it is really usable in practice and, even more importantly, how it can be transformed and how the possibly different versions can be compared.

The goal of our transformation is to reduce the number of collective operations in mss-alg2. The scan-reduce fusion (SR-ARA) looks like a candidate rule for mss-alg2. However, it is not directly applicable since there is an extra arrangement skeleton between scan and reduction. The rule which fits is SAR-ARA: it combines the swap of reduce and pair and SR-ARA.

Rule SAR-ARA can be applied with the following instantiations of the parameter operators $Op1$ and $Op2$ with MSS operators:

$$Op1 \;=\; op2,\; Op2 \;=\; \max$$

The applicability condition is the forward distributivity of $Op1$ over $Op4$ which, in the case of mss-alg2, are the following operations:

$$(a_1, b_1)\, Op1\, (a_2, b_2) \;=\; (\, \max\left((a_1{+}b_2), a_2\right),\, b_1{+}b_2\,) \tag{3}$$

$$(a_1, b_1)\, Op4\, (a_2, b_2) \;=\; (\, \max(a_1, a_2)\,,\, \max(b_1, b_2)\,) \tag{4}$$

The transformation engine will ask the programmer to check the distributivity required by the rule. This can be done straight-forwardly, under the obviously correct assumption that operator max is commutative.

After applying rule SAR-ARA to program mss-alg2, we obtain the following result program for the maximum segment sum problem:

```
mss-alg3 (in x : Array n Scalar, out r : Scalar)
  s = reduce (Op3) pair (pair (x,x), pair (x,x)) ;
  r = proj1 (proj1 x);
```

The target program mss-alg3 exploits only one computational skeleton, reduce, with an associative base operator. Such a reduction is easily parallelized. Thus, we arrive at a better solution than both the intuitive and the first parallelizable version. Note that both the data arrangements and the operator used by the reduction are quite complicated. It is rather unlikely that the programmer will discover this program without support.

9 Conclusions

We have discussed the FAN framework for the transformation of high-level specifications to efficient parallel programs. The main novelty of our framework, in comparison with other skeleton systems available [9, 14, 32], is the intensive use of program transformations in the early stages of the design process, supported by corresponding cost models and programming tools. The framework is language-independent and should be easy to integrate with existing high-level parallel programming environments, as our experience with P3L demonstrates [19]. The paper presents a functional semantics of FAN in Haskell. A consequence of this is that FAN programs can be type checked and prototyped using the available Haskell implementations.

Our work was inspired by research on combining skeletons and transformations [14, 35] and by the work on BMF [7]. However, our approach allows a more natural expression of parallel algorithms with respect to BMF, since intermediate results can be named and reused throughout the program. FAN differs also from SCL [14], since it abstracts from the actual data distribution which is not programmed explicitly in the notation. This allows for more freedom in the definition of the model and a more extensive reuse of the early stages of the refinement tree.

Compared to the earlier work on the transformational design of functional programs using higher-order combinators [22], our approach has two main features: (1) transformation rules are equipped with a cost model which enables an estimation of the impact of the transformations on program performance, and (2) the transformation process is supported by a transformation engine with a user interface.

The design of our transformation engine, especially the choice of its data structures, was influenced by the PARAMAT system [15, 28]. However, our approach differs in many aspects. First, our goal is the optimization of high-level parallelism, rather than the parallelization of low-level sequential codes. Second, we do not define an *a priori* parallel structure as PARAMAT does. Rather, we search for a sequence of transformations toward the best parallel structure, depending on the particular instance of the given problem.

Our first case studies and machine experiments have confirmed the feasibility of the proposed high-level, transformation-based, performance-directed approach to the design of parallel programs. We plan to assess the framework by applying it in the parallelization of larger applications and to extend it with task-parallel skeletons which model common patterns of interaction among data-parallel modules [3, 4, 14, 33].

Acknowledgements

We are very grateful to Bruno Bacci from QSW Ltd. for many fruitful discussions. The three anonymous referees were very helpful in improving the quality of presentation. This work has been supported by a travel grant from the German-Italian exchange programme VIGONI.

References

[1] S. G. Akl. *The Design and Analysis of Parallel Algorithms.* Prentice-Hall, 1989.

[2] M. Aldinucci, M. Coppola, and M. Danelutto. Rewriting skeleton programs: How to evaluate the data-parallel stream-parallel tradeoff. In S. Gorlatch, editor,*Proc. 1st Int. Workshop on Constructive Methods for Parallel Programming (CMPP'98)*, pages 48–58. Fakultät für Mathematik und Informatik, Universität Passau, May 1998. Technical Report MIP-9805.

[3] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^3L: A structured high level programming language and its structured support. *Concurrency: Practice and Experience*, 7(3):225–255, May 1995.

[4] B. Bacci, M. Danelutto, S. Pelagatti, and M. Vanneschi. SkIE: an heterogeneous HPC environment. *Parallel Computing*, 25(13–14):1827–1852, December 1999.

[5] B. Bacci, S. Gorlatch, C. Lengauer, and S. Pelagatti. Skeletons and transformations in an integrated parallel programming environment. In *Parallel Computing Technologies (PaCT-99)*, LNCS 1662, pages 13–27. Springer-Verlag, 1999.

[6] J. J. Bentley. Programming pearls: Algorithm design techniques. *Comm. ACM*, 27(9):865–871, Sept. 1984.

[7] R. Bird. Lectures on constructive functional programming. In M. Broy, editor, *Constructive Methods in Computing Science*, NATO ASI Series F: Computer and Systems Sciences, Vol. 55, pages 151–216. Springer-Verlag, 1988.

[8] R. Bird. *Introduction to Functional Programming using Haskell*, 2nd edition. Series in Computer Science. Prentice Hall Europe, 1998.

[9] G. H. Botorog and H. Kuchen. Skil: An imperative language with algorithmic skeletons for efficient distributed programming. In *Proc. Fifth Int. Symp. on High Performance Distributed Computing (HPDC-5)*, pages 243–252. IEEE Computer Society Press, 1996.

[10] M. Cole, S. Gorlatch, J. Prins, and D. Skillicorn, editors. *High Level Parallel Programming: Applicability, Analysis and Performance*. Dagstuhl-Seminar Report 238, Schloß Dagstuhl. 1999.

[11] M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

[12] M. I. Cole. Parallel programming with list homomorphisms. *Parallel Processing Letters*, 5(2):191–204, June 1995.

[13] M. I. Cole, S. Gorlatch, C. Lengauer, and D. B. Skillicorn, editors. *Theory and Practice of Higher-Order Parallel Programming*. Schloß Dagstuhl, Feb. 1997. Report 169.

[14] J. Darlington, Y. Guo, H. W. To, and Y. Jing. Skeletons for structured parallel composition. In *Proc. of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1995.

[15] B. di Martino and C. W. Kessler. Program comprehension engines for automatic parallelization: A comparative study. In I. Jelly, I. Gorton, and P. Croll, editors, *Software Engineering for Parallel and Distributed Systems*, pages 146–157. Chapman & Hall, 1996.

[16] M. Gardner. The fantastic combinations of John Conway's new game of life. *Scientific American*, pages 100–123, October 1970.

[17] S. Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96: Parallel Processing, Vol. II*, Lecture Notes in Computer Science 1124, pages 401–408. Springer-Verlag, 1996.

[18] S. Gorlatch and C. Lengauer. (De)Compositions for parallel scan and reduction. In *Proc. 3rd Working Conf. on Massively Parallel Programming Models (MPPM'97)*, pages 23–32. IEEE Computer Society Press, 1998.

[19] S. Gorlatch and S. Pelagatti. A transformational framework for skeletal programs: Overview and case study. In J. Rohlim et al., editors, *Parallel and Distributed Processing. IPPS/SPDP'99 Workshops Proceedings*, Lecture Notes in Computer Science 1586, pages 123–137. Springer-Verlag, 1999.

[20] S. Gorlatch, C. Wedler, and C. Lengauer. Optimization rules for programming with collective operations. In M. Atallah, editor, *13th Int. Parallel Processing Symp. & 10th Symp. on Parallel and Distributed Processing (IPPS/SPDP'99)*, pages 492–499. IEEE Computer Society Press, 1999.

[21] M. Hamdan, G. Michaelson, and P. King. A scheme for nesting algorithmic skeletons. In K. Hammond, T. Davie, and C. Clack, editors, *Proc. 10th Int. Workshop on the Implementation of Functional Languages (IFL'98)*, pages 195–211. Department of Computer Science, University College London, 1998.

[22] P. G. Harrison. A higher-order approach to parallel algorithms. *The Computer Journal*, 35(6):555–566, 1992.

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

[24] C. M. Hoffmann and M. O'Donnell. Interpreter generation using tree pattern matching. In *Proc. 6th ACM Symp. on Principles of Programming Languages (POPL'79)*, pages 169–179. ACM Press, 1979.

[25] C. M. Hoffmann and M. O'Donnell. Pattern matching in trees. *J. ACM*, 29(1):68–95, Jan. 1982.

[26] B. Jay, M. Cole, M. Sekanina, and P. Steckler. A monadic calculus for parallel costing of a functional language of arrays. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *Parallel Processing. Euro-Par'97*, Lecture Notes in Computer Science 1300, pages 650–661. Springer-Verlag, 1997.

[27] S. R. Kasaraju. Efficient tree pattern matching. In *Proc. 30th IEEE Ann. Symp. on Foundations of Computer Science (FOCS'89)*, pages 178–183. IEEE Computer Society Press, 1989.

[28] C. W. Kessler. Pattern-driven automatic program transformation and parallelization. In *Proc. 3rd EUROMICRO Workshop on Parallel and Distributed Processing (PDP'95)*, pages 76–83. IEEE Computer Society Press, 1995.

[29] W. F. McColl. Scalable computing. In J. van Leeuwen, editor, *Computer Science Today*, Lecture Notes in Computer Science 1000, pages 46–61. Springer-Verlag, 1995.

[30] E. Mäkinen. On the subtree isomorphism problem for ordered trees. *Parallel Processing Letters*, 32(5):271–273, Sept. 1989.

[31] D. L. Parnas. On the design and development of program families. *IEEE Trans. on Software Engineering*, SE-2(1):1–9, Mar. 1976.

[32] P. J. Parsons and F. A. Rabhi. Generating parallel programs from paradigm based specifications. *Journal of Systems Architecture*, 45(4):261–283, 1998.

[33] T. Rauber and G. Rünger. A coordination language for mixed task and data parallel programs. In *Proc. of 3rd Annual ACM Symposium on Applied Computing (SAC'99)*, pages 146–155. ACM Press, 1999.

[34] D. B. Skillicorn. *Foundations of Parallel Programming.* Cambridge International Series on Parallel Computation. Cambridge University Press, 1994.

[35] D. B. Skillicorn and W. Cai. A cost calculus for parallel functional programming. *J. Parallel and Distributed Computing*, 28(1):65–83, July 1995.

[36] D. Smith. Applications of a strategy for designing divide-and-conquer algorithms. *Science of Computer Programming*, 8(3):213–229, 1987.

[37] M. Südholt. Data distribution algebras — a formal basis for programming using skeletons. In E.-R. Olderog, editor, *Programming Concepts, Methods and Calculi (PROCOMET'94)*, pages 19–38. Elsevier, 1994.

[38] D. Swierstra and O. de Moor. Virtual data structures. In B. Möller, H. Partsch, and S. Schuman, editors, *Formal Program Development*, LNCS 755, pages 355–371. Springer-Verlag, 1993.

[39] C. Wedler and C. Lengauer. On linear list recursion in parallel. *Acta Informatica*, 35(10):875–909, 1998.

Appendix : Haskell Definitions of FAN Skeletons

Here is complete Haskell definition of the FAN skeletons used in the paper, for one- and two-dimensional arrays.

```
fanpart1:: (Int,Int) -> Array Int a -> Array Int (Array Int a)
fanpart1 (r,s) x = let (lb,ub) = bounds x in array (lb,ub)
  [ (i, array (0,r+s) [ (k, x!((i-r+k)`mod`(ub-lb+1))) | k <- [0..(r+s)] ])
  | i <- [lb..ub] ]

fanpart2:: ((Int,Int),(Int,Int)) -> Array (Int,Int) a
               -> Array (Int,Int) (Array (Int,Int) a)
fanpart2 ((r,s),(p,q)) x =
  let ((lb0,lb1),(ub0,ub1)) = bounds x in
   array ((lb0,lb1),(ub0,ub1)) [ ((i,j), array ((0,0),(r+s,p+q))
     [ ((k,l), x!((i-r+k)`mod`(ub0-lb0+1),(j-p+l)`mod`(ub1-lb1+1)))
     | k <- [0..(r+s)], l <- [0..(p+q)] ])
   | i <- [lb0..ub0], j <- [lb1..ub1] ]

fansplit1:: Int -> Array Int a -> Array Int (Array Int a)
fansplit1 p x = let (lb,ub) = bounds x
  chsize = if (nelems(x) `mod` p) == 0 then (nelems(x) `div` p)
                                        else (nelems(x) `div` p) +1
  in array (lb,lb+(p-1))
    [ (i, array (0,chsize-1) [ (k, x!(((i*chsize)+k)`mod`(ub-lb+1)))
                             | k <- [0..(chsize-1)] ])
    | i <- [lb..(lb+(p-1))] ]

fansplit2 :: (Integral a, Ix a) => (a,a) -> Array (a,a) b
               -> Array (a,a) (Array (a,a) b)
fansplit2 (p,q) x = let
  ((lb0,lb1),(ub0,ub1)) = bounds x
  nrow = ub0-lb0 +1  ncol = ub1-lb1 +1
  rowsize = if (nrow `mod` p) == 0 then (nrow `div` p) else (nrow `div` p) +1
  colsize = if (ncol `mod` q) == 0 then (ncol `div` q) else (ncol `div` q) +1
  in array ((lb0,lb1),(lb0+(p-1),lb1+(q-1)))
     [ ((i,j), array ((0,0),(rowsize-1,colsize-1))
        [ ((k,l), x!(((i*rowsize)+k)`mod`nrow,((j*colsize)+l)`mod`ncol))
         | k <- [0..(rowsize-1)], l <- [0..(colsize-1)] ])
      | i <- [lb0..(lb0+(p-1))], j <- [lb1..(lb1+(q-1))] ]

fanrearrange :: Ix a => (a -> a) -> Array a b -> Array a b
fanrearrange f x = ixmap (bounds x) f x
```

```
fantranspose2:: (Ix a) => Array (a,a) b -> Array (a,a) b
fantranspose2 x = fanrearrange (\(i,j) -> (j,i)) x

fancopy1:: Int  -> a -> Array Int a
fancopy1 n x = array (0,n-1) [ (i , x)   | i <- [0..(n-1)] ]

fancopy2:: (Int,Int) -> a -> Array (Int,Int) a
fancopy2 (n,m) x = array ((0,0),(n-1,m-1))
                      [ ((i,j), x) | i <- [0..(n-1)], j <- [0..(m-1)] ]

fanpair :: Ix a => (Array a b,Array a c) -> Array a (b,c)
fanpair(x,y) = if (bounds x) == (bounds y)
  then array (bounds x) [ (i,(x!(i),y!(i))) | i <- range (bounds(x)) ]
  else error "Pairing two non-conformant arrays!"

fanproj1:: (Ix a) => Array a (b,c) -> Array a b
fanproj1 x =  fanmap fst x

fanmap:: (Ix c) => (a -> b) -> Array c a -> Array c b
fanmap f x = array (bounds x) [ (i,f (x!i)) | i <- (range (bounds x)) ]

fanmapsh:: (Ix c) => (c -> a -> b) -> Array c a -> Array c b
fanmapsh f x = array (bounds x) [ (i,f i (x!i)) | i <- (range (bounds x)) ]

fanred:: (Ix c) => (a-> a -> a) -> (Array c a) -> a
fanred f x = foldl1 f (elems x)

fanscan:: (Ix c) => ((a -> a -> a) -> [a] -> [a]) -> (a -> a -> a)
                         -> (Array c a) -> Array c a
fanscan s f x = listArray (bounds x) (s f (elems x))

fanscanl:: (Ix c) => (a -> a -> a) -> (Array c a) -> Array c a
fanscanl = fanscan scanl1

fanlooprepeat :: (a -> Bool) -> (a -> a) -> a -> a
fanlooprepeat c f x = let t = f x in if c t then t else fanlooprepeat c f t

fanloopwhile :: (a -> Bool) -> (a -> a) -> a -> a
fanloopwhile c f x = if c x then fanloopwhile c f (f x) else x

fanloopfor :: (Num a, Ord a) => (a,a,a) -> (b -> b) -> b -> b
fanloopfor (n1,n2,st) f x =
                if n1 <= n2 then fanloopfor (n1+st) n2 st f (f x) else x
```