Formal Aspects of Computing (1995) 3:

Parallelization of Divide-and-Conquer in the Bird-Meertens Formalism[1]

Sergei Gorlatch and Christian Lengauer

Fakultät für Mathematik und Informatik, Universität Passau, D–94030 Passau, Germany

Keywords: Bird-Meertens formalism; Divide-and-Conquer; Functional programming; Parallel programming; Transformations

Abstract. An SPMD parallel implementation schema for divide-and-conquer specifications is proposed and derived by formal refinement (transformation) of the specification. The specification is in the form of a mutually recursive functional definition. In a first phase, a parallel functional program schema is constructed which consists of a communication tree and a functional program that is shared by all nodes of the tree. The fact that this phase proceeds by semantics-preserving transformations in the Bird-Meertens formalism of higher-order functions guarantees the correctness of the resulting functional implementation. A second phase yields an imperative distributed message-passing implementation of this schema. The derivation process is illustrated with an example: a two-dimensional numerical integration algorithm.

1. Introduction

One of the main problems in exploiting modern multiprocessor systems is how to develop correct and efficient programs for them. We address this problem using the approach of formal program transformation. We take a class of specifications and construct formally one common parallel implementation schema that applies to every member of this class. The schema exhibits SPMD (Single Program Multiple Data) parallelism, i.e., the same code is executed by each processor in the system.

Correspondence and offprint requests to: Sergei Gorlatch, Fakultät für Mathematik und Informatik, Universität Passau, D–94030 Passau, Germany, E-mail: gorlatch@fmi.uni-passau.de

[1] Parts of this paper were presented at IPPS' 94, April 1994, Mexico and at WTC' 94, September 1994, Italy

Our derivation proceeds within the *Bird-Meertens* Formalism (BMF), a theory of higher-order functions over a variety of data structures [Bir88]. Functional expressions are used for specification and can be transformed to expressions with better properties using BMF's theorems as derivation rules.

The use of higher-order functions results in clear and concise specifications that describe usually a class of problems because the arguments of higher-order functions are functions themselves. Such classes are called *skeletons* [Col88] and are generally viewed as building blocks for composing large application programs. Therefore, people have been trying to identify typical skeletons and to study their parallelism. The importance of divide-and-conquer as one of the widely used skeletons has been noted repeatedly [ACG89, Sto87]. Several methods of its parallelization have been proposed; they are analyzed in Section 8.

Our approach starts by constructing a communication structure inherent for the skeleton specification. Then, using an abstraction function on data types ([Har92]), the specification is transformed within BMF to an implementation on the communication structure, where all parallelism of the specification is preserved. Thus, we combine the practicality of the skeletal approach with the formal soundness of BMF, which guarantees the correctness of the parallelization. These are the main features of the approach:

- The class of admitted specifications includes mutually recursive functional definitions.
- A sequence of transformations that does not depend on the particular specification yields a parallel functional implementation schema. The schema consists of a communication tree and a higher-order functional program that is common to all nodes of the tree.
- The transformations used in the derivation are based on the semantics-preserving rules of the Bird-Meertens formalism and Backus' FP [Bac78].
- The final implementation is an imperative distributed SPMD program schema with message passing; all communications are between neighbours in the tree.
- The implementation of a particular specification is obtained as a specialization of the schema by supplying specific functions as parameters for the higher-order program.
- The target program can be tuned to a given number of processors; it permits also further optimizations.

We transform the schema in general and, in addition, illustrate each phase of the transformation with a specific, realistic example: a two-dimensional numerical integration algorithm. Consequently, most sections of the paper consist of two subsections: one on the general case and one on the example. In Section 2, both the general form of the specification and the example are introduced. Section 3 presents briefly the Bird-Meertens formalism, extended for our purposes, and describes how the initial specification is expressed in this higher-order formalism. In Section 4, the data type of communication trees is defined and an abstraction function for it is constructed. The higher-order specification is transformed systematically into a parallel functional program schema in Section 5. Section 6 is on the generation of a more architecture-related imperative program. Efficiency aspects of this program are discussed in Section 7. Section 8 compares our approach with others. Finally, Section 9 summarizes the results and outlines problems for further study.

2. Specification

In this section, we present the general format of the specifications that we admit and the example that we will come back to throughout the paper.

2.1. General Case

We consider a system of n functions $f = (f_1, \cdots, f_n)$ that are mutually recursive, i.e., there is a functional definition with one equation for every function of f:

$$f_i(x) \quad = \quad \textbf{if}\ p_i(x)\ \textbf{then}\ b_i(x)\ \textbf{else}\ E_i(g_i,\ f,\ x)\ \textbf{fi} \qquad\qquad (i=1,\cdots,n) \qquad (1)$$

Here $g = (g_1, \cdots, g_n)$ is a collection of what we call *auxiliary functions*: g_i represents the non-recursive part of the equation for f_i. We suppose that all functions in the systems f, g and b have the same type $\tau \to \sigma$. The domain τ and the range σ are arbitrary sets; they may be structured but we ignore their structural properties. Elements of τ are called *domain parameters*, the p_i *basic predicates* and the b_i *basic functions*. Expression E_i depends on the value of auxiliary function $g_i(x)$ and on the results of (possibly several) recursive calls of functions from f. Within the equation for function f_i, the l-th call of f_j is of the form $f_j(\varphi^l_{ij}(x))$, where functions $\varphi^l_{ij} : \tau \to \tau$ are called *shifts*. Each E_i has a fixed set of shifts.

We view the system of equations (1) as a specification for computing one of the functions f_i, say, f_1. Our goal is to generate a parallel program that, given a particular domain parameter *input*, computes $f_1(\mathit{input})$ and, of course, all values that are necessary for that computation according to the dependences in (1).

The general format (1) includes special cases that have been studied extensively in the literature:

1. Systolic algorithms are often specified in this format, where $\tau = \mathbb{Z}^m$ and the shifts are of the form $\varphi(i) = i + a$, for some fixed $a \in \mathbb{Z}^m$. These and other restrictions enable the use of linear algebra and linear programming for the synthesis of a parallel program [Len93].
2. Conventional divide-and-conquer algorithms correspond to the case of a system with a single function f_1 and two recursive calls of it. This recursion is sometimes called non-linear [Har92]: it reflects the "divide" aspect of an algorithm.

2.2. Example

As a sample specification of the format (1), we consider an algorithm for numerical two-dimensional non-adaptive integration [Zen90]. The value q of the integral in the domain $[a_1, b_1] \times [a_2, b_2]$ for a given function u vanishing on the boundary,

$$q = \int_{a_1}^{b_1} \int_{a_2}^{b_2} u(x1, x2)\ dx1\ dx2$$

is approximated for a given meshwidth $2^{-m}, m \in \mathbb{N}$, by $q^{(m)} = A(a_1, b_1, a_2, b_2, m)$, where A is defined recursively using functions N and HB as follows:

$$\left.\begin{array}{l} A(a_1,b_1,a_2,b_2,m) \;=\; \textbf{if}\,(m\!=\!1)\;\textbf{then}\; HB\,(a_1,b_1,a_2,b_2)\;\textbf{else} \\ A(a_1,\frac{a_1+b_1}{2},a_2,b_2,m\!-\!1) + A(\frac{a_1+b_1}{2},b_1,a_2,b_2,m\!-\!1) + N(a_1,b_1,a_2,b_2,m)\;\textbf{fi} \\[2ex] N(a_1,b_1,a_2,b_2,m) \;=\; \textbf{if}\,(m\!=\!1)\;\textbf{then}\; HB\,(a_1,b_1,a_2,b_2)\;\textbf{else} \\ N(a_1,b_1,a_2,\frac{a_2+b_2}{2},m\!-\!1) + N(a_1,b_1,\frac{a_2+b_2}{2},b_2,m\!-\!1) + HB\,(a_1,b_1,a_2,b_2)\;\textbf{fi} \end{array}\right\} \quad (2)$$

Specification (2) is a special case of (1) with two recursive functions: $f_1 = A$, $f_2 = N$; the domain parameters are from $\tau = \mathbb{R}^4 \times \mathbb{Z}$; some of the shifts are: $\varphi^1_{11}(a_1,b_1,a_2,b_2,m) = (a_1,\frac{a_1+b_1}{2},a_2,b_2,m-1)$, $\varphi^2_{11}(a_1,b_1,a_2,b_2,m) = (\frac{a_1+b_1}{2},b_1,a_2,b_2,m-1)$, $\varphi^1_{12} = id$ (the identity function). There is one basic predicate for both recursive functions; we name it *m.is.1*. It is defined by $(m.is.1)(a_1,b_1,a_2,b_2,m) = (m=1)$. There is no auxiliary non-recursive function in the equation for A, so g_1 does not appear. The auxiliary function for N is HB; HB is also the basic function for both A and N. Rather than defining HB precisely, we capture its dependences in an expression *Expr*:

$$\begin{array}{rl} HB\,(a_1,b_1,a_2,b_2) \;=\; & Expr\,(a_1,b_1,a_2,b_2,u(a_1,a_2),u(a_1,b_2),u(b_1,a_2), \\ & u(b_1,b_2),u(\frac{a_1+b_1}{2},a_2),u(a_1,\frac{a_2+b_2}{2}),u(b_1,\frac{a_2+b_2}{2}),u(\frac{a_1+b_1}{2},\frac{a_2+b_2}{2})) \end{array} \quad (3)$$

Our considerations will be made for the general case (1) and illustrated by the example (2).

3. Higher-order specification

We use the notation of the Bird-Meertens formalism (BMF) and Backus' FP. Function application is denoted by juxtaposition. Sometimes, an argument will be enclosed in parentheses to enforce a precedence or structure a complicated expression. Composition of functions is denoted by $\circ$ and associates to the right.

3.1. Higher-order functions

From the BMF, we take the following *higher-order functions* (also called *functionals*) on lists:

- *map* applies function h to all elements of a list $[a_1,\cdots,a_n]$:

  $map\ h\ [a_1,\cdots,a_n] = [h\,a_1,\cdots,h\,a_n]$
- *red* (*reduction*) computes a value of some type from a list $[a_1,\cdots,a_n]$ of values of that type by applying an associative binary operation $\oplus$:

  $red\,(\oplus)\ [a_1,\cdots,a_n] = a_1 \oplus \cdots \oplus a_n$

In general, we have to deal with more than one function; therefore, we consider lists of functions and lists of lists of arguments. This gives rise to the following functionals:

- The *doublemap*, map^2, *maps* a function h over a list of lists of arguments:

  $map^2\ h \;=\; map\,(map\,h)$

- The *double reduction*, red^2, is defined on a binary associative operation $\oplus$ and a list of lists:

  $$red^2\,(\oplus) = red\,(\oplus)\;\circ\;map\,(red\,(\oplus))$$

- The *distributed map*, *dmap*, is the following functional on a list $[h_1, \cdots, h_n]$ of n functions and a list $[l_1, \cdots, l_n]$ of n lists of arguments:

  $$dmap\,[h_1, \cdots, h_n]\,[l_1, \cdots, l_n] = [map\,h_1\,l_1,\;\cdots, map\,h_n\,l_n]$$

  It can be expressed using functional *fzip* which is a usual *zip* of the BMF with function application as a binary operation:

  $$dmap\,[h_1, \cdots, h_n] \;=\; fzip\,[map\,h_1, \cdots, map\,h_n]$$

  where $fzip\,[h_1, \cdots, h_n]\,[a_1, \cdots, a_n] = [h_1 a_1, \cdots, h_n a_n]$.

It is easy to prove the following equalities, where function *flat* "flattens" a list of lists, i.e., eliminates all inner brackets:

$$red^2\,(\oplus) \;=\; red(\oplus)\;\circ\;flat \tag{4}$$

$$flat\;\circ\;map^2\,h \;=\; map\,h\;\circ\;flat \tag{5}$$

$$dmap\,[h, \cdots, h] \;=\; map^2\,h \tag{6}$$

$$dmap\,[h_1, \cdots, h_n] \circ map^2\,h = dmap\,[h_1 \circ h, \cdots, h_n \circ h] \tag{7}$$

We use FP's *construction* functional for applying a list of functions to one argument; we denote it by $<>$, writing for simplicity $< h_1, \cdots, h_n >$ instead of $< [h_1, \cdots, h_n] >$:

$$< h_1, \cdots, h_n >\; x \;=\; [h_1\,x, \cdots,\; h_n\,x]$$

The following properties hold for the construction and its combinations with the functionals of the Bird-Meertens formalism:

$$< h_1, h_2 > \circ\, h \;= < h_1 \circ h\;,\; h_2 \circ h > \tag{8}$$

$$map\,h_1\;\circ h_2 \;=\; flat \circ map^2\,h_1 \circ < h_2 > \tag{9}$$

$$map\,h \circ\; < h_1, \cdots, h_n > = < h \circ h_1, \cdots, h \circ h_n > \tag{10}$$

$$dmap\,[h_1, \cdots, h_n] \circ < t_1, \cdots, t_n > = \;< map\,h_1 \circ t_1, \cdots, map\,h_n \circ t_n > \tag{11}$$

$$flat \circ << h_{11}, \cdots, h_{1n} >, \cdots\; < h_{m1}, \cdots, h_{mn} >> = \;< h_{11}, \cdots, h_{1n},\; \cdots,\; h_{m1}, \cdots, h_{mn} > \tag{12}$$

For conditional expressions like **if** p **then** b **else** c **fi** we use FP's notation, $p \rightarrow b\,;\,c$, with the following properties:

$$p \rightarrow h\,;\,h \;=\; h \tag{13}$$

$$(p \rightarrow b\,;\,h) \circ t \;=\; p \circ t \rightarrow b \circ t\;;\; h \circ t \tag{14}$$

3.2. General Case

Let us rephrase specification (1) using the introduced notation. Each function E_i in (1) takes a (flat) list of values which are results of applying corresponding auxiliary and recursive functions to the domain parameter. We compose these functions to a list and apply them per construction. We can reformulate (1) as follows:

$$f_i\,(x) = (\,p_i \to b_i\,;\ E_i\,\circ\,\mathit{flat}\circ\,<<g_i>\,,\ \mathit{calls}_i\,>)\,x$$

where function calls_i yields the list of recursive function calls in the equation for f_i.

In order to reason at the functional level, we drop the parameter:

$$f_i \quad = \quad p_i \to b_i\,;\ E_i\,\circ\,\mathit{flat}\circ\,<<g_i>\,,\ \mathit{calls}_i\,> \qquad\qquad (i=1,\cdots,n) \qquad (15)$$

Take any i $(i=1,\cdots,n)$. As mentioned in Subsection 2.1, any f_j $(j=1,\cdots,n)$ may be called by f_i, possibly more than once, with specific "shifted" domain parameters. We combine all corresponding shifts φ^l_{ij} $(l=1,\cdots,d_{ij})$ per construction in the function $\mathit{split}_{ij}\,:\,\tau\to\,\mathit{list}\,\tau$ where d_{ij} is the number of calls of f_j in the equation for f_i. The recursive calls in that equation form the list $[\mathit{map}\,f_1\circ\mathit{split}_{i1},\cdots,\mathit{map}\,f_n\circ\mathit{split}_{in}]$ which, after flattening, yields the result of calls_i:

$$\mathit{calls}_i\ =\ \mathit{flat}\circ\ <\mathit{map}\,f_1\circ\mathit{split}_{i1},\cdots,\mathit{map}\,f_n\circ\mathit{split}_{in}> \qquad (16)$$

Using aggregated functions:

$$\mathit{split}_i\ = <\mathit{split}_{i1},\cdots,\mathit{split}_{in}> \qquad\qquad (i=1\cdots n) \qquad (17)$$

of type $\mathit{split}_i:\tau\to\mathit{list}\ \mathit{list}\,\tau$, we can represent calls_i as follows:

$$\begin{array}{ll}
 & \mathit{calls}_i \\
= & \quad\{\ \text{equality (16)}\ \} \\
 & \mathit{flat}\circ\ <\mathit{map}f_1\circ\mathit{split}_{i1},\cdots,\mathit{map}f_n\circ\mathit{split}_{in}> \\
= & \quad\{\ \text{equality (11)}\ \} \\
 & \mathit{flat}\,\circ\,\mathit{dmap}\,[f_1,\cdots,f_n]\circ<\mathit{split}_{i1},\cdots,\mathit{split}_{in}> \\
= & \quad\{\ \text{equality (17)}\ \} \\
 & \mathit{flat}\,\circ\,\mathit{dmap}\,[f_1,\cdots,f_n]\circ\mathit{split}_i
\end{array}$$

Substituting this expression into (15), we obtain the following higher-order notation:

$$\begin{array}{lll}
f_i\ = & p_i\to b_i\,; & \\
 & E_i\,\circ\,\mathit{flat}\circ<<g_i>\,,\ \mathit{flat}\,\circ\,\mathit{dmap}\,[f_1,\cdots,f_n]\,\circ\mathit{split}_i> & (i=1,\cdots,n)
\end{array}$$

This notation can be simplified if we introduce a binary construction functional, $\prec\succ$, over special arguments $h_1:\tau\to\sigma$ and $h_2:\tau\to\ \mathit{list}\,\sigma$ with the following definition:

$$\prec h_1,h_2\succ x\ =\ h_1x\ \mathit{cons}\,h_2x \qquad (18)$$

$$\prec h_1\succ x\ =<h_1>\ x \qquad (19)$$

$$\prec h_2\succ x\ =\ h_2\,x \qquad (20)$$

where operation *cons* appends an element to a list. Therefore, construction $\prec\succ$ yields always a flat list. The higher-order notation of specification (1) reads now as follows:

$$f_i\ =\ p_i\to b_i\,;\ E_i\circ\ \prec g_i\ ,\ \mathit{flat}\,\circ\,\mathit{dmap}\,[f_1,\cdots,f_n]\,\circ\mathit{split}_i\succ \quad (i=1,\cdots,n) \quad (21)$$

3.3. Example

Our example specification (2) contains the following splitting functions:

$$\begin{aligned} split_1(a1,b1,a2,b2,m) = [\,[\,(a1,\tfrac{a1+b1}{2},a2,b2,m-1),(\tfrac{a1+b1}{2},b1,a2,b2,m-1)\,]\,,\\ [\,(a1,b1,a2,b2,m)\,]\,] \end{aligned}$$

$$split_2(a1,b1,a2,b2,m) = [\,[\,]\,,\,[(a1,b1,a2,\tfrac{a2+b2}{2},m-1),(a1,b1,\tfrac{a2+b2}{2},b2,m-1)]\,]$$

The representation of specification (2) in the higher-order notation follows schema (21), specialized by concrete expressions for predicates, basic functions and splitting functions. Since addition is associative, *red* and *red*^{2} can be applied to it in E_i. We specialize , again, by formal transformations in order to be able to prove correctness. The following equational transformations are based on the rules for higher-order functions and on the fact that function $split_{21}$ yields the empty list for any argument (we denote this function by ε).

$$\begin{array}{ll}
 & A \\
= & \{ \text{ specialization of (21) with the empty auxiliary function } \} \\
 & m.is.1 \to HB;\ red\,(+) \circ \prec flat \circ dmap\,[A,N] \circ split_1 \succ \\
= & \{ \text{ equality (20) } \} \\
 & m.is.1 \to HB;\ red\,(+) \circ flat \circ dmap\,[A,N] \circ split_1 \\
= & \{ \text{ equality (4) } \} \\
 & m.is.1 \to HB;\ red^2\,(+) \circ dmap\,[A,N] \circ split_1
\end{array}$$

$$\begin{array}{ll}
 & N \\
= & \{ \text{ specialization of (21) } \} \\
 & m.is.1 \to HB;\ red\,(+) \circ \prec HB,\ flat \circ dmap\,[A,N] \circ split_2 \succ \\
= & \{ \text{ definition of } split_2 \ \} \\
 & m.is.1 \to HB;\ red\,(+) \circ \prec HB,\ flat \circ dmap\,[A,N] \circ < split_{21}, split_{22} > \succ \\
= & \{ \ split_{21} = \varepsilon \ \} \\
 & m.is.1 \to HB\,;\ red\,(+) \circ \prec HB,\ flat \circ dmap\,[N] \circ < split_{22} > \succ \\
= & \{ \text{ equality (6) } \} \\
 & m.is.1 \to HB\,;\ red\,(+) \circ \prec HB\,,\ flat \circ map^2\,N \circ < split_{22} > \succ \\
= & \{ \text{ equality (9) } \} \\
 & m.is.1 \to HB\,;\ red\,(+) \circ \prec HB,\ map\,N \circ split_{22} \succ
\end{array}$$

The higher-order representation of the example (2) is thus:

$$\begin{aligned} A &= m.is.1 \to HB;\ red^2\,(+) \circ dmap\,[A,N] \circ split_1 \\ N &= m.is.1 \to HB;\ red\,(+) \circ \prec HB,\ map\,N \circ split_{22} \succ \end{aligned} \tag{22}$$

4. Evaluation Tree and Abstraction Function

The presence of higher-order functions $<>$, $\prec\succ$, *map*, *dmap* and *red* in the general case (21) and in the example (22) points already to divide-and-conquer parallelism in the specification: elements of the corresponding lists can be evaluated simultaneously. Some of these elements are, again, recursive functions. Unfolding the recursion creates an evaluation tree whose nodes represent values of functions in f; the nodes at one level can be evaluated in parallel.

In this section, we define a new data type that represents possible evaluation trees for a given specification; these trees are further used as communication structures for parallel computation, with processors mapped to the nodes. We establish a correspondence between this type and the original domain type τ in the form of an abstraction function and use it in the next section for the derivation of a parallel program schema that implements the initial specification.

4.1. General Case

We introduce n types of trees, $TREE_i$ $(i=1,\cdots,n)$, one for each function f_i in (1). The nodes are taken from the set $\{NODE_i \mid i=1,\cdots,n\}$. A tree of type $TREE_i$ is either a single node of type $NODE_i$, or it consists of a root of type $NODE_i$ and a list of n lists, the jth of which contains d_{ij} subtrees of type $TREE_j$.

Recall that d_{ij} is the length of the list produced by $split_{ij}$, i.e., the structure of trees is determined uniquely by the splitting functions of the specification. From now on, we talk, more briefly, of type i rather than type $NODE_i$. Each node of type i has outdegree $d_i = \sum_{j=1}^{n} d_{ij}$, which we represent, like some other subsequent quantifier expressions, in Dijkstra's quantifier notation [DS90]: $d_i = \langle\, \Sigma\, j \,:\, 1 \leq j \leq n \,:\, d_{ij} \,\rangle$.

The evaluation graph of function f_i in (1) is of type $TREE_i$. Because we want to derive a parallel program that computes f_1, we are particularly interested in the type $TREE_1$. It is a partially ordered set: $x \sqsubseteq y$ iff x is a subtree of y. The least upper bound of $TREE_1$ is the infinite tree:

$$tree_1 = \langle \bigsqcup \; tree \;:\; tree \in TREE_1 \;:\; tree \,\rangle$$

such that all trees of type $TREE_1$ are subtrees of $tree_1$.

Tree $tree_1$ is unique for a given specification (1); it represents the communication structure of the parallel implementation we are aiming at. In our definitions, we use the following functions on the node set of $tree_1$, V:

- *father* yields the father of a given node,
- *sons* returns all sons of a given node as a list of n lists: the i-th list contains the sons of type i,

and predicates on V with the following meaning:

- $is.root\, v$: v has no father,
- $is.leaf\, v$: v has no son,
- $is.node_i\, v$: v is of type i,
- $is.son_{jl}\, v$: v is the l-th son of type j of its father.

We introduce an abbreviated notation for conditional functions and predicates that are defined on V:

$$\langle [\!]\, i \,:\, 1 \leq i \leq n \,:\, is.node_i \rightarrow t_i \rangle \;=\; is.node_1 \rightarrow t_1 ; \cdots ; is.node_n \rightarrow t_n ;\; \perp$$

Here, $\perp$ stands for the undefined value. The range predicate $1 \leq i \leq n$ can be omitted if there is no danger of confusion. Then, the notation reduces to $\langle\, [\!]\, i \,::\, is.node_i \rightarrow t_i \,\rangle$.

In particular, this notation has the following property:

$$dmap\,[h_1,\cdots,h_n] \,\circ\, sons \;=\; map^2 \,\langle\, [\!]\, i \,::\, is.node_i \rightarrow h_i \,\rangle \,\circ\, sons \qquad (23)$$

In our further considerations, we also use the following flattened versions of functions and predicates:

- $fsons\ =\ flat \circ sons$,
- $fsplit_i\ =\ flat \circ split_i$,
- $is.son_j\, v\ :\ v$ is the jth son of its father (regardless the type of v).

Let us construct an abstraction function that maps from the concrete type V (the nodes of $tree_1$) to the abstract type τ (the domain parameters). We call our abstraction function div : it "divides" the domain parameters and distributes them among the nodes. We define this function as follows.

- The value of div at the root is defined to be $input$ – the domain parameter for which function f_1 must be computed.
- If the basic predicate holds at some non-leaf node, we set the values of div at its sons to the undefined value $\perp$.
- For other nodes the domain parameter is defined by applying corresponding shift to the parameter of the father: if w is the jth son of a node of type i then we define

  $$div\, w\ =\ (\pi_j \circ fsplit_i \circ div \circ father)\ w$$

  where projection function π_j yields the jth component of a list $(1 \leq i \leq n\ ,\ 1 \leq j \leq d_i)$.

We can now present the definition of div in our higher-order notation:

$$\begin{aligned} div =\quad & is.root \rightarrow input; \\ & p \circ div \circ father \rightarrow \perp; \\ & \langle\, [\,]\, j\ ::\ is.son_j\ \rightarrow\ \pi_j \circ fsplit \circ div \circ father \rangle \end{aligned} \tag{24}$$

where predicate p and function $fsplit$ are such that

$$p \circ div\ =\ \langle\, [\,]\, i\ ::\ is.node_i\ \rightarrow\ p_i \circ div \rangle$$

$$fsplit \circ div\ =\ \langle\, [\,]\, i\ ::\ is.node_i\ \rightarrow\ fsplit_i \circ div \rangle$$

We will also use the following obvious properties of the projection function:

$$\pi_j \circ < h_1, \cdots, h_n > = < h_j > \tag{25}$$

$$\pi_j\ \circ\ map\, h\ =\ h\ \circ\ \pi_j \tag{26}$$

The relation between functions $fsplit$, div and $fsons$, expressed by the equality:

$$(map\ div) \circ fsons\ =\ fsplit \circ div \tag{27}$$

can now be formally deduced. There are three cases according to (24); the first two are simple, for the third one we take an arbitrary projection of the left-hand side and transform it:

$$\begin{aligned} & \pi_j \circ map\ div \circ fsons \\ = & \quad \{\ \text{equality } (26)\ \} \\ & div \circ \pi_j \circ fsons \\ = & \quad \{\ \text{equality } (24)\ \} \\ & \pi_j \circ fsplit \circ div \circ father \circ \pi_j \circ fsons \\ = & \quad \{\ father \circ \pi_j \circ fsons\ =\ id\ \} \\ & \pi_j \circ fsplit \circ div \end{aligned}$$

This holds for all projections, which proves (27).

4.2. Example

For our example specification, type $TREE_1$ corresponds to function A which is to be computed. An example tree of this type with 12 nodes is shown in Figure 1. There are two types of nodes in the tree: ternary nodes of type 1 correspond

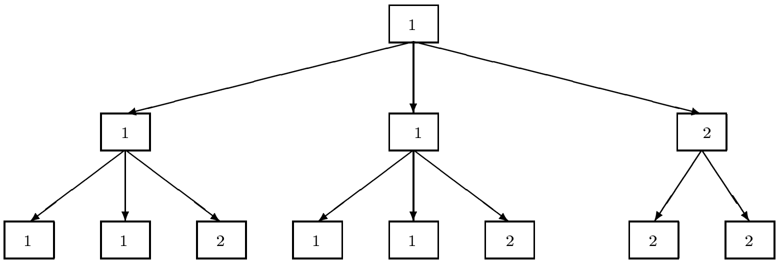

Fig. 1. Tree of type $TREE_1$ – example

to A and binary nodes of type 2 correspond to N. Function *sons* yields a list of two lists at each node; the first of them is empty at the nodes of type 2.

5. Derivation in the Bird-Meertens Formalism

In this section, we present a calculational way of deriving a functional parallel implementation from the higher-order specification, using the abstraction function introduced in the previous section.

5.1. General Case

Using the abstraction function *div*, we introduce a new function $F : V \to \sigma$ as follows:

$$F \quad = \quad \langle \,[\!]\, i \,::\, \mathit{is.node}_i \to f_i \,\circ\, \mathit{div} \rangle \tag{28}$$

Function F combines functions $f_1, \cdots, f_n$ of (1); however, it is defined not on the domain τ but on the nodes of tree_1. From (24) and (28), we see immediately that the value of F at the root of tree_1 is equal to $f_1(\mathit{input})$—the value which is the solution of the initial specification. Therefore, we have reduced the task of implementing the specification to the task of computing function F at the root of tree_1.

Our aim is to parallelize the latter task by distributing the computation of F over the nodes of the tree. We run into two problems. First, tree tree_1 is infinite by definition. Second, the computation of F at the root is not yet parallelized: according to (28), we must compute $f_1(\mathit{input})$ as before, i.e., sequentially recursively. Using the introduced functions and their properties, we can cope with both problems.

First, when dealing with real-life communication structures, we can pick a fixed finite tree $\mathit{tree} \in TREE_1$ whose number of nodes does not exceed the number of processors available to us. This tree is determined by the predicate *is.leaf*, defined on V, which selects the leaves of the tree. For each particular finite tree *tree*, we shall use the restrictions of all functions originally defined on V – like F, *div*, etc. – to the node set of *tree*, without giving them special names. All properties of these functions also hold for their restrictions. Note that we do not have to make our considerations for every particular finite tree, rather all results are valid for each such tree: necessary specializations are "hidden" in predicate *is.leaf*.

Second, we can now parallelize the expression $f_i \circ div$ of (28) via transformation:

$$
\begin{array}{ll}
 & f_i \circ div \\
= & \quad \{ \text{ equality (13) } \} \\
 & is.leaf \to f_i \circ div \,;\; f_i \circ div \\
= & \quad \{ \text{ equalities (21), (14) } \} \\
 & is.leaf \to f_i \circ div \,; \\
 & p_i \circ div \to b_i \circ div \,; \\
 & E_i \circ \prec g_i \,,\, \circ flat \circ dmap\,[f_1, \cdots, f_n] \circ split_i \succ \circ div
\end{array}
$$

The last alternative, which applies in the case of $\neg\,(p_i \circ div \vee is.leaf)$, is transformed further:

$$
\begin{array}{ll}
 & E_i \circ \prec g_i \,,\, flat \circ dmap\,[f_1, \cdots, f_n] \circ split_i \succ \circ div \\
= & \quad \{ \text{ equality (8) } \} \\
 & E_i \circ \prec (g_i \circ div) \,, (flat \circ dmap\,[f_1, \cdots, f_n] \circ split_i \circ div) > \\
= & \quad \{ \text{ equality (27) } \} \\
 & E_i \circ \prec (g_i \circ div) \,, \big(flat \circ dmap\,[f_1, \cdots, f_n] \circ map^2\; div \circ sons\big) \succ \\
= & \quad \{ \text{ equalities (7), (23) } \} \\
 & E_i \circ \prec (g_i \circ div) \,, \big(flat \circ map^2\, \langle\, [\!]\, i :: is.node_i \to f_i \circ div \rangle \circ sons\big) \succ \\
= & \quad \{ \text{ definition (28) } \} \\
 & E_i \circ \prec (g_i \circ div) \,, \big(flat \circ map^2\, F \circ sons\big) \succ \\
= & \quad \{ \text{ equality (5), definition of } fsons \ \} \\
 & E_i \circ \prec g_i \circ div \,,\; map\, F \circ fsons \succ
\end{array}
$$

The following transformations are applied again to the entire expression:

$$
\begin{array}{ll}
 & f_i \circ div \\
= & is.leaf \to f_i \circ div \,; \\
 & p_i \circ div \to b_i \circ div \,; \\
 & E_i \circ \prec (g_i \circ div) \,,\, map\, F \circ fsons \succ \\
= & \quad \{ \text{ equalities (14), (8), (25) } \} \\
 & (\, is.leaf \circ \pi_1 \to f_i \circ \pi_2 \,; \\
 & p_i \circ \pi_2 \to b_i \circ \pi_2 \,; \\
 & E_i \circ \prec g_i \circ \pi_2 \,,\, map\, F \circ fsons \circ \pi_1 \succ) \\
 & \circ < id \,,\, div >
\end{array}
$$

The obtained expression is a composition of two parts, the second of which, $< id\,,\, div >$, is essentially a "divide" step. We call the first one $conq_i$ (for "conquer"). The "divide"-part is a construction of the identity function *id* and *div* and it thus yields a list of two values: the first is the node itself, the second is the domain parameter for this node. To make the intuitive meaning evident, we use the names *node* for π_1 and *par* for π_2.

Now, substituting expression for $f_i \circ div$ into (28), we arrive at the final expression for F:

$$F = \langle [\!]\, i :: \mathit{is.node}_i \circ \mathit{node} \rightarrow \mathit{conq}_i \,\rangle \circ <\mathit{id}\,,\ \mathit{div}> \tag{29}$$

$$\begin{aligned} \mathit{div} = \ & \mathit{is.root} \rightarrow \mathit{input}; \\ & p \circ \mathit{div} \circ \mathit{father} \rightarrow \perp; \\ & \langle [\!]\, j :: \mathit{is.son}_j \rightarrow \pi_j \circ \mathit{fsplit} \circ \mathit{div} \circ \mathit{father} \rangle \end{aligned} \tag{30}$$

$$\begin{aligned} \mathit{conq}_i = \ & \mathit{is.leaf} \circ \mathit{node} \rightarrow f_i \circ \mathit{par}; \\ & p_i \circ \mathit{par} \rightarrow b_i \circ \mathit{par}; \\ & E_i \circ \prec g_i \circ \mathit{par}\,,\ \mathit{map}\, F \circ \mathit{fsons} \circ \mathit{node} \succ \end{aligned} \tag{31}$$

There are three important observations to be made about (29)–(31). First, for computing function F at some node of *tree*, we can use the results of computing F at its sons as in (31) and the result of computing *div* (which is a part of F) at its father as in (30). Therefore, the computation of F at the root of *tree* can be distributed over the nodes of *tree* with function F to be computed at each node. Second, the computations at different nodes can be performed in parallel: F is *map*ped to the sons in (31), i.e., the computations at the sons are independent of each other. They are also independent of the computation of the auxiliary function g_i, because of $\prec\succ$. Third, we arrived at this parallel implementation from the higher-order representation (21) of the specification in a calculational way using sound formal rules. Therefore, (29)–(31) correctly implements the initial specification.

Let us summarize the way of implementing a specification of format (1). Recall that we must generate a program that, given a particular parameter *input*, computes $f_1(\mathit{input})$. We construct type TREE_1, which captures the communication structure needed by the specification, and choose an arbitrary tree of this type, such that each of its nodes can be mapped onto a processor. If all processors simultaneously compute function F according to (29)–(31) and *input* is available at the root processor, then the result obtained at the root is the desired value $f_1(\mathit{input})$. Function F is computed at each node of the tree in two steps:

1. Apply $<\mathit{id}, \mathit{div}>$. Applying the identity function simply means that we need some information about the node itself, not only the value of *div* at it. Function *div* computes the corresponding domain parameter for a node. Equation (30) says that, for the root, this parameter is *input* and, for each other node, it is determined by the result of *div* at its father. Thus, in the computation of *div*, data is flowing from the root to the leaves of the tree. Note that if the domain parameter at some node makes predicate p_i true, i.e., the basic case is reached, then the domain parameters for all descendants of this node are undefined: no computations at those nodes are needed. This reflects the situation, when the number of processors exceeds the degree of parallelism in the specification.

2. Function conq_i takes the domain parameter returned by *div*. Equation (31) prescribes that, in the leaf nodes, f_i for the domain parameter must be computed using the initial specification, i.e., recursively and sequentially. In the basic case, the basic function is computed. Otherwise, the results from the sons and from computing the auxiliary function figure into the computation of E_i. Therefore, at this step, data is flowing from the leaves to the root.

5.2. Example

It is not necessary to repeat the whole derivation process for our example: its parallel implementation is obtained by a specialization of the general schema (29)–(31). We write A^F for the specialization of function F. Expressions for specific functions A (for f_1) and N (for f_2) can be taken from (22). Functions $fsplit_1$ and $fsplit_2$ are just the flattened versions of $split_1$ and $split_2$ from Subsection 3.3. Function *fsons* is a construction of components that yields the respective sons: $< son_1, son_2, son_3 >$ for nodes of type 1 and $< son_1, son_2 >$ for nodes of type 2.

The specialization of *conq* can be optimized, e.g., by "lifting" addition to the functional level:

$$red(+) < h_1, \cdots, h_n > = \ h_1 + \cdots + h_n$$

The specialized schema reads as follows:

$$A^F \ = \ (\, is.node_1 \circ node \ \to \ conq_1 \ ; \ conq_2 \,) \circ \ < id \, , \, div > \tag{32}$$

$$\begin{aligned} div \ = \ & is.root \to input; \\ & m.is.1 \circ div \circ father \to \perp; \\ & \langle \, [] \, j : 1 \leq j \leq 3 : \ is.son_j \to fsplit \circ div \circ father \rangle \end{aligned} \tag{33}$$

$$\begin{aligned} conq_1 \ = \ & is.leaf \circ node \to A \circ par; \\ & m.is.1 \circ par \to HB \circ par; \\ & A^F \circ son_1 \circ node + A^F \circ son_2 \circ node + A^F \circ son_3 \circ node \end{aligned} \tag{34}$$

$$\begin{aligned} conq_2 \ = \ & is.leaf \circ node \to N \circ par; \\ & m.is.1 \circ par \to HB \circ par; \\ & HB \circ par + A^F \circ son_1 \circ node + A^F \circ son_2 \circ node \end{aligned} \tag{35}$$

6. Imperative parallel implementation

In the previous section, we have "refined" the general form of specification into a higher-order functional parallel schema. This implementation schema consists of a communication structure defined by type $TREE_1$, and a function F, defined by (29)–(31), which is to be computed simultaneously at all nodes of the structure. In this section, we take the functional implementation and convert it to an imperative program with explicit message passing. The imperative program prescribes the computation and communication for processors that are assigned to the nodes of the tree.

6.1. General Case

We describe first the correspondence between the constructs used in the functional implementation and the statements which will be used in the target program. We express the imperative program in pseudocode with evident semantics; see Figure 2.

Program structure. Since there is a single function F to be computed at all nodes of the tree, the expression for this function can be viewed as an SPMD parallel program: our target program `Node` is executed at every node of the tree. One implicit parameter of program `Node` is the id (identifier, not identity function *id*) of the associated processor (variable `my_id`).

```
PROGRAM Node;
 IMPORT Father, Son, Outdegree, Is_leaf, Is_root, P_holds,
        Split, Compute_E, Compute_f, Compute_g, Compute_b;
  BEGIN_Node;
   IF Is_root( my_id)
    THEN READ (type,par)
    ELSE RECV (type,par) FROM Father (my_id)
   END_IF;
   IF Is_leaf (my_id)
    THEN
     IF type \= 0
      THEN  result := Compute_f (type,par)
     END_IF
    ELSE
     FORALL k:=1 TO Outdegree (my_id) DO
      IF ( type = 0 OR Pholds(type,par) )
       THEN (type (k), par (k)) := (0,0)
       ELSE (type (k), par (k)) := Split (type,par,k);
      END_IF;
      SEND (type (k), par (k)) TO Son (my_id,k)
     END_FORALL;
     IF P_holds (type,par)
      THEN result := Compute_b (type, par)
      ELSE
       PARBEGIN
        res_aux := Compute_g (type,par) //
        FORALL k:=1 TO Outdegree (my_id) DO
         RECV (res[k]) FROM Son (my_id,k)
        END_FORALL
       PAREND;
       result := Compute_E (type, res_aux, res[1], ..., res[n])
     END_IF
    END_IF;
    IF Is_root (my_id)
      THEN WRITE (result)
      ELSE SEND (result) TO Father (my_id)
    END_IF;
END_Node;
```

Fig. 2. SPMD program schema

Procedures. The imperative program uses procedures that implement the functions and predicates defined on the communication tree: `Is_root`, `Is_leaf`, `Outdegree`, `Father`, `Son`. All procedures take the processor id as a parameter; `Son` has an additional parameter k, specifying the (k-th) son to be computed. The following functions, used in the specification and then in function F, are defined on τ (the domain parameter): $split_i$, p_i, b_i, f_i, g_i and E_i; they depend on the type of node, i. We implement them by the corresponding procedures `Split`, `P_holds`, `Compute_b`, `Compute_f`, `Compute_g` and `Compute_E`. These procedures have the type of the node and the domain parameter as parameters. `Split` has an additional parameter corresponding to projection function π_j in (31). For brevity, we have not listed the procedure interfaces in our import list.

Local memory. Each processor has its own local memory. Whereas in the functional representation functions get parameters and transfer results implicitly between each other, in the imperative program we use variables and assignments for this purpose. Variable par stores the domain parameter (result of *div*) for the current processor. The type of the node, i, is stored in variable type; it is computed and transmitted together with par. For nodes, where the domain parameter is undefined, we set type=0. Since, according to (30), the domain parameter for a node is computed by its father, we need also arrays type[] and par[] for storing corresponding values computed for the sons. Function *node* is implemented by variable my_id which is directly accessible in the processor. Variable res_aux contains the result of computing the auxiliary function in the processor and array res[], the results of F from the sons. Finally, result stores the value of F at the current node.

Input-output is implemented by the statements READ and WRITE.

Communication. Processors send and receive data by the statements SEND (<data>) TO <partner> and RECV (<data>) FROM <partner>. Here, <partner> is the id of the processor with whom the communication takes place. For the sake of simplicity, we consider synchronous blocking communication; asynchronous communication can sometimes yield better performance.

Conditionals of Backus' FP are implemented by the IF–THEN–ELSE statement.

Sequential composition. The composition of the functions in (29)–(31) is implemented by sequential composition of imperative statements.

Parallelism. There are two candidates for parallel execution in the functional schema: *map* and $\prec\succ$. We implement *map* in (31) by distributing the computation of F over all processors in the tree; it remains only to receive the results from the sons, and *map* tells us that this can be done in parallel. Furthermore, it can be done simultaneously with the computation of the auxiliary function due to $\prec\succ$. We use two kinds of parallel statements in our pseudocode: PARBEGIN // PAREND and the parallel FORALL loop.

Formulae (29)–(31) are implemented in the program Node as follows. The domain parameter *input* and the type type of the root node are the input to the program. The root receives them by READ (<data>). According to (30), the domain parameters at non-root nodes can be obtained at their father by applying Split to its own domain parameter. Then, the current processor must receive its type and par from the father.

Having obtained the domain parameter, i.e., having computed function *div*, it remains to compute $conq_i$. In the program, there is a conditional statement corresponding to the condition in (31). For a leaf node, function f_i is computed sequentially by procedure Compute_f. A non-leaf node computes a pair (type,par) and sends it to each of its sons (communications at the *divide* and *conquer* steps are coordinated by that). Note that even if type=0, this value must be sent to all descendants, otherwise they would wait infinitely trying to execute their RECV statements, i.e., there would be a deadlock in the system. If the basic predicate holds, it remains only to compute the basic function by Compute_b.

We come now to the case of a non-leaf, non-basic node—the third alternative in (31). Subexpression $mapF \circ fsons$ in it is in fact already implemented by assuming that F is computed by all nodes of the tree. It remains only to receive the results of computing F from all sons of the current node. We do that by a parallel loop (due to *map*) which produces array res[i]. The computation of the auxiliary function is independent of this loop (due to $\prec\succ$), and thus we can use `PARBEGIN // PAREND`. After all threads of the parallel statement have finished, we compute E_i implemented by `Compute_E` which yields the result of F at the current node. Communication is coordinated by executing `SEND (result) TO Father` at all non-root processors. At the root, `result` is the output of the whole parallel program, it is given away by `WRITE`.

6.2. Example

The imperative parallel program for the example specification has the same structure as in the general case; specialization is realized in the corresponding procedures where particular functions used in the specification are computed. We omit the program text of the example for brevity (see [GL93]).

7. Efficiency issues

In this section, we touch briefly on some questions concerning the efficiency of our parallel imperative program schema.

There are, in general, various levels of parallelism that can be detected in a specification, extracted from it and implemented in a parallel program. Our considerations in this paper have been limited to the "generic" parallelism which is determined by the dependences in (1) and which is not influenced by the properties of particular functions, g and E, and particular domains τ and σ. All this parallelism has been preserved during the development of the functional implementation (29)–(31). But we have to watch our step in the transition to the imperative implementation. The following efficiency aspects should be taken into account:

- **Restricted dividing**. The amount of parallelism is governed by the recursion depth of function *div*. In the parallel schema, dividing is additionally controlled by the predicate *is.leaf*. This way, the parallelism is matched with the available number of processors.
- **Multithreading**. There are two kinds of parallelism in the derived schema: inter-processor parallelism (between nodes) and intra-processor parallelism (inside one node). The latter is expressed by `FORALL` loops and `PARBEGIN // PAREND` brackets, and can be advantageous on some architectures, e.g., on transputer networks where it is implemented by multithreading.
- **Communication structure**. The communication tree may be non-homogeneous. E.g., in our example, the nodes may vary in outdegree. The architecture of the multiprocessor must cope with that. On the other hand, the synthesized structure does not change during program execution, and the communications are only between direct neighbours (father and sons).
- **Redundant computations**. We see from (3) that the computation of *HB* for different arguments uses common values of function u. In the imperative program, this leads to redundant computations. They can be prevented by introducing additional communication [Gor92].
- **Load Balancing**. The amount of computational work in a processor depends on its position in the tree. The load balance can be improved using

additional transformations at the functional level that are explained subsequently. There is also an adaptive variant of the integration algorithm where the basic predicate is some accuracy condition which has to be fulfilled. In this case, the actual amount of work is known only at run time, and dynamic load-balancing should be used.

Let us discuss two ways of improving the performance of the imperative program. First, a particular specification may be matched in different ways with format (1). Another match for our example (2) is to make A the single recursive and N its auxiliary function. In this case, the target program has a binary communication tree that can be efficiently implemented on most multiprocessors. The node program computes sequentially the corresponding value of N, i.e., the granularity of parallelism becomes higher and the load is balanced better. Second, we can stick to our match of Section 2—and, thus, the original communication structure—but execute one of *map*'s components in the processor itself. E.g., in our example, the processor might not use the link to the left son and perform the corresponding computations sequentially. Then, parallelism is also balanced better, and the outdegree of each node is reduced by 1. These and other optimizations can be realized by transformations.

Our experiments with a parallel implementation of the example on a 64-node transputer system yielded an efficiency (speed up/number of processors) ranging from 0.6 to 0.9, depending on the input domain parameter and on the processor number. See [Gor94] for details.

8. Related work

Recently, there has been a lot of work investigating parallelism with the Bird-Meertens formalism for lists. Researchers in this field are Skillicorn [Ski90, Ski93], Jones [JG92], de Moor [SdM], Cole [Col93], Fokkinga [Fok91] and others. They mostly use the formalism for specifying and parallelizing problems in which lists are the basic data structure. We extend this application area of BMF by using lists and the higher-order functions on them in quite a different way, namely to describe and parallelize the computational structure of an algorithm, regardless of the basic data structure. Our extension to BMF combines generalized versions of *map* and *red* with the FP's construction functional and makes use of transformation rules for them. Our approach does not stop at the functional level but proceeds down to the target program with explicit message passing.

Much attention has been paid to the formal parallelization of conventional (i.e., not mutually recursive) divide-and-conquer.

Mou and Hudak describe divide-and-conquer in an algebraic model [MH88], Carpentieri and Mou study communication issues in this model [CM91]. Our approach differs from this work in three main aspects. First, we consider a more general case by allowing a specification to consist of several mutually recursive functions and also of non-recursive auxiliary functions. Second, we propose a systematic, semantically sound way of deriving a distributed-memory SPMD program schema for this class of specifications. Third, we ignore the structural properties of the data domain τ what enables a more general treatment of divide-and-conquer. On the other hand, this has the drawback that we do not consider the effect of the data size on communication and parallelism in our performance analysis.

Smith develops an approach in which the structure of a class of algorithms is represented as a theory. E.g., divide-and-conquer can be treated as a homomorphism from a decomposition algebra on the input domain to a composition algebra on its output domain; design tactics for divide-and-conquer algorithms have been implemented in the KIDS system [SL89].

A research group at Imperial College is developing a skeletal approach, initiated by Cole [Col88], in the context of functional programming languages [DFH+93]. The non-linear recursion of the divide-and-conquer skeleton is transformed into tail recursion and is then pipelined [dGHM93, Har92]. Pipelining reduces the parallelism of divide-and-conquer but is claimed to be more suitable for parallel architectures with a static communication structure (we are not aware of any experimental results on performance). In contrast, we preserve the initial tree parallelism of divide-and-conquer and show that it can be implemented with static and local communication. The idea of abstract data type transformation that was used by Harrison for parallelizing linear recursion [Har92] is applied here in a broader context: we implement both stages of divide-and-conquer and show that the abstraction function expresses the essence of the dividing stage.

A group around Pepper is trying to incorporate the notion of data partitioning into formal program development [Pep93]. Parallelism is represented using recursive networks, whereas we are aiming at an imperative target program which is directly executable on contemporary multiprocessor systems.

Axford has suggested viewing the divide-and-conquer paradigm as a fundamental design principle [Axf92], and, together with Joy have proposed divide-and-conquer list processing primitives for parallel computation on shared memory [AJ93]. Our imperative schema yields an SPMD implementation on distributed memory. Other approaches for distributed memory are the design of specialized architectures for divide-and-conquer, e.g., by McBurney and Sleep [MS87], and explicit annotation of functional programs with directives for parallelism and communication, as by Kelly [CHK+92].

Unfolding the recursion is proposed by Lengauer and Huang [LH86] e.g., for the bitonic sort [HL86] – also a divide-and-conquer algorithm. The derived parallel algorithm has logarithmic complexity and is provably optimal. The disadvantage of the method is that the computational complexity of the derivation process depends on the problem size. In our approach, the computational complexity of the derivation process is independent of the problem size.

The important problem of how to map the parallel program onto particular communication topologies (3D mesh, hypercube, etc.) is considered, e.g., in [MCH92]. We have derived a logical communication structure and shown how to adapt it to the available number of processors; the problem of mapping it onto a physical interconnection topology is a point of further research. There is much work on this matter , e.g. [NSS93], but we are not aware of any that considers non-homogeneous trees, as in our example.

Preliminary performance estimates for our example have been done in [Gor94]. These questions are being investigated in depth for various skeleton functions on a transputer-based system at the University of British Columbia [FSWC92, WSC93].

9. Conclusions and future work

Our paper takes the approach in which the starting point in the program development is a problem-oriented, often non-procedural, formal specification of an algorithm. The specification describes *what* is to be done but not *how* it is to be done. Aspects of the how—in our case, parallelism—are introduced by formal transformations. Procedural aspects enter the development when the implementation is mapped to a language executable on existing processor networks.

We have presented a parallel implementation of a functional specification of mutually recursive divide-and-conquer. First, a parallel functional schema is obtained via transformation of the specification. Its correctness is guaranteed by the semantical soundness of the transformation rules, which are taken from the Bird-Meertens formalism, extended for our purposes. The functional schema consists of a communication tree with processors at the nodes and a common higher-order function associated with each node. The communication structure has two important properties: it is static, i.e., it does not change during program execution, and it is local, i.e., each processor in the tree communicates only with its father and sons. The functional schema is then transformed into an imperative SPMD schema with coarse-grained parallelism and message passing. The target program adapts itself to the available number of processors.

Our future work will include detailed performance studies of parallel divide-and-conquer by means of both analytical and experimental methods. The goal is to study the influence of various transformation rules used in the derivation process on the efficiency of the resulting parallel program. Another problem is how to formalize the transition from a parallel functional implementation to an imperative one.

10. Acknowledgement

We are very grateful to one of the anonymous referees whose outstandingly extensive and insightful review had a significant impact on the final version of the paper.

References

[ACG89] M. J. Atallah, R. Cole, and M. T. Goodrichs. Cascading divide-and-conquer: A technique for designing parallel algorithms. *SIAM J. Computing*, 18(3):499–532, 1989.

[AJ93] T. Axford and M. Joy. List processing primitives for parallel computation. *Computer Languages*, 19(1):1–12, 1993.

[Axf92] T. Axford. The divide-and-conquer paradigm as a basis for parallel language design. In L. Kronsjo and D. Shumsheruddin, editors, *Advances in Parallel Algorithms. Chapter 2.* Blackwell, 1992.

[Bac78] J. W. Backus. Can programming be liberated from the von Neumann style? *Communications of the ACM*, 21:613–641, 1978.

[Bir88] R. S. Bird. Lectures on constructive functional programming. In M. Broy, editor, *Constructive Methods in Computing Science*, NATO ASO Series F: Computer and Systems Sciences. Vol. 55, pages 151–216. Springer Verlag, 1988.

[CHK+92] S. Cox, S. Huang, P. Kelly, J. Liu, and F. Taylor. An implementation of static functional process networks. In D. Etiemble and J.-C. Syre, editors, *PARLE'92 Parallel Architecture and Languages Europe*, Lecture Notes in Computer Science 605, pages 497–512, 1992.

[CM91] B. Carpentieri and G. Mou. Compile-time transformations and optimizations of parallel divide-and-conquer algorithms. *ACM SIGPLAN Notices*, 20(10):19–28, 1991.

[Col88] M. Cole. Algorithmic skeletons: A structured approach to the management of parallel computation. Ph.D. Thesis. Technical Report CST-56-88, Department of Computer Science, University of Edinburgh, 1988.

[Col93] M. Cole. Parallel programming, list homomorphisms and the maximum sum problem. Technical Report CSR–25–93, Department of Computer Science, University of Edinburgh, May 1993.

[DFH+93] J. Darlington, A. Field, P. Harrison, P. Kelly, D. Sharp, Q. Wu, and R. While. Parallel programming using skeleton functions. In A. Bode, M. Reeve, and G. Wolf, editors, *PARLE '93 Parallel Architectures and Languages Europe*, Lecture Notes in Computer Science 694, pages 146–160. 1993.

[dGHM93] I. P. de Guzman, P. G. Harrison, and E. Medina. Pipelines for divide-and-conquer functions. *The Computer Journal*, 36(3):254–268, 1993.

[DS90] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag, 1990.

[Fok91] M. Fokkinga. An exercise in transformational programming: Backtracking and branch-and-bound. *Science of Computer Programming*, 16:19–48, 1991.

[FSWC92] D. Feldcamp, H. Sreekantaswamy, A. Wagner, and S. Chanson. Towards a skeleton–based parallel programming environment. In A. Veronis and Y. Parker, editors, *Transputer Research and Applications*, pages 104–115. IOS Press, 1992.

[GL93] S. Gorlatch and C. Lengauer. Parallelization of divide-and-conquer in the Bird-Meertens formalism. Technical Report MIP–9315, Fakultät für Mathematik und Informatik, Universität Passau, December 1993.

[Gor92] S. Gorlatch. Parallel program development for a recursive numerical algorithm: A case study. Technical Report SFB 342/7/92, Institut für Informatik, TU München, March 1992.

[Gor94] S. Gorlatch. Formal derivation and implementation of divide-and-conquer on a transputer network. In A. De Gloria, M. Jane, and D. Marini, editors, *Transputer Applications and System '94*, pages 763–776. IOS Press, 1994.

[Har92] P. G. Harrison. A higher-order approach to parallel algorithms. *The Computer Journal*, 35(6):555–566, 1992.

[HL86] C.-H. Huang and C. Lengauer. The automated proof of a trace transformation for a bitonic sort. *Theoretical Computer Science*, 46(2–3):261–284, 1986.

[JG92] G. Jones and J. Gibbons. Linear-time breadth-first tree algorithms: An exercise in the arithmetic of folds and zips. Technical Report TR-31-92, Oxford University Computing Laboratory, 1992.

[Len93] C. Lengauer. Loop parallelization in the polytope model. In E. Best, editor, *CONCUR '93*, Lecture Notes in Computer Science 715, pages 398–416. Springer-Verlag, 1993.

[LH86] C. Lengauer and C.-H. Huang. A mechanically certified theorem about optimal concurrency of sorting networks. In *Proc. 13th Ann. ACM Symp. on Principles of Programming Languages (POPL)*, pages 307–317, 1986.

[MCH92] Z. G. Mou, C. Constantinescu, and T. J. Hickey. Divide-and-conquer on three-dimensional meshes. In W. Joosen and E. Milgrom, editors, *Parallel Computing: From Theory to Sound Practice*, pages 344–355. IOS Press, 1992.

[MH88] Z. G. Mou and P. Hudak. An algebraic model for divide-and-conquer algorithms and its parallelism. *Journal of Supercomputing*, 2(3):257–278, 1988.

[MS87] D. McBurney and M. Sleep. Transputer-based experiments with the ZAPP architecture. In J. de Bakker, A. Nijman, and P. Treleaven, editors, *PARLE Parallel Architectures and Languages Europe*, Lecture Notes In Computer Science 258, pages 242–259. 1987.

[NSS93] W. G. Nation, G. Saghi, and H. J. Siegel. Properties of inteconnection networks for large-scale parallel processing systems. In P. Tvrdík, editor, *ISIPCALA'93*, pages 51–79, 1993.

[Pep93] P. Pepper. Deductive derivation of parallel programs. In R. Paige, J. Reif, and R. Wachter, editors, *Parallel Algorithm Derivation and Program Transformation*, pages 1–53. Kluwer Academic Publishers, 1993.

[SdM] D. Swiestra and O. de Moor. Virtual data structures. In B. Moeller, H. Partsch, and S. Schuman, editors, *Formal Program Development*, Lecture Notes in Computer Science 755, pages 355–371.

[Ski90] D. B. Skillicorn. Architecture-independent parallel computation. *IEEE Computer*, 23(12):38–50, 1990.

[Ski93] D. B. Skillicorn. Deriving parallel programs from specifications using cost information. *Science of Computer Programming*, 26:205–221, 1993.

[SL89] D. Smith and M. Lowry. Algorithm theories and design tactics. In J. L. A. van de Snepscheut, editor, *Mathematics of Program Construction*, Lecture Notes in Computer Science 375, pages 379–398, 1989.

[Sto87] Q. F. Stout. Supporting divide-and-conquer algorithms for image processing. *Journal of Parallel and Distributed Computing*, 4:95–115, 1987.

[WSC93] A. Wagner, H. Sreekantaswamy, and S. Chanson. Performance issues in the design of a processor farm template. In R. Grebe et al., editors, *Transputer Applications and Systems. Vol. 1*, pages 438–450. IOS Press, 1993.

[Zen90] C. Zenger. Sparse grids. Technical Report SFB-Nr. 342/18/90 A, Institut für Informatik, TU München, October 1990.