Systematic Development of an SPMD Implementation Schema for Mutually Recursive Divide-and-Conquer Specifications

S. Gorlatch and C. Lengauer

Department of Mathematics and Computer Science
University of Passau
D–94030 Passau, GERMANY

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 SPMD 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 SPMD implementation schema that applies to every member of this class.

We choose the *Bird-Meertens* formalism for higher-order functions over lists [3]. 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* [6] and are generally considered as building blocks for composing large application programs. Therefore, people have been trying to identify typical skeletons and to study their parallel implementation. The importance of divide-and-conquer as one of the widely used skeletons has been noted repeatedly [1, 19]. Several approaches to its specification and parallel implementation have been proposed; they are analyzed in Section 7.

These are the main features of our parallel implementation and its construction:

- The class of admitted specifications includes functional mutually recursive 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 [2].
- The final implementation is an imperative distributed SPMD program schema; 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. 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 the centerpiece of the paper, Section 4, the higher-order specification is transformed systematically into a parallel functional program schema. Section 5 is on the generation of a more architecture-related imperative program. Efficiency aspects of this program are discussed in Section 6. Section 7 compares our approach with others. Finally, Section 8 summarizes the results and outlines problems for further study.

0-8186-5602-6/94 © 1994 IEEE

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.

We consider the following system of n mutually recursive functions $f = (f_1, \cdots, f_n)$. Each function f_i $(i = 1, \cdots, n)$ is defined by the equation:

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

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. These calls are 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 (1) as a specification for computing one of functions f_i, say, f_1. Our goal is to generate a parallel program that, given a particular domain parameter *input*, computes $f_1(input)$ and, of course, all values that are necessary for that computation according to the dependencies 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 [12].

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 [10]: it reflects the "divide" aspect of an algorithm.

As a sample specification of the format (1), we consider an algorithm for numerical two-dimensional nonadaptive integration [20]. 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,

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

can be approximated for a given meshwidth 2^{-m}, $m \in$ 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:

$$\begin{aligned} &A(a_1, b_1, a_2, b_2, m) \;=\; \textbf{if } (m = 1) \textbf{ then} \\ &HB(a_1, b_1, a_2, b_2) \textbf{ else } A(a_1, \tfrac{a_1+b_1}{2}, a_2, b_2, m-1) \\ &+A(\tfrac{a_1+b_1}{2}, b_1, a_2, b_2, m-1) + N(a_1, b_1, a_2, b_2, m) \textbf{ fi} \\ &N(a_1, b_1, a_2, b_2, m) \;=\; \textbf{if } (m = 1) \textbf{ then} \\ &HB(a_1, b_1, a_2, b_2) \textbf{ else } N(a_1, b_1, a_2, \tfrac{a_2+b_2}{2}, m-1) \\ &+N(a_1, b_1, \tfrac{a_2+b_2}{2}, b_2, m-1) + HB(a_1, b_1, a_2, b_2) \textbf{ fi} \end{aligned} \quad (2)$$

Specification (2) is a special case of (1) with two recursive functions: $f_1 = A$, $f_2 = N$; domain parameters are from $\tau = \mathbb{R}^4 \times \mathbb{Z}$; some of shifts are: $\varphi^1_{11}(a_1, b_1, a_2, b_2, m) = (a_1, \frac{a_1+b_1}{2}, a_2, b_2, m-1)$, $\varphi^2_{11}(a_1, b_1, a_2, b_2, m) = (\frac{a_1+b_1}{2}, b_1, a_2, b_2, m-1)$, $\varphi^1_{12} = Id$ (identity). There is one basic predicate, we shall 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 is an "empty" function. 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 dependencies in an expression *Expr*:

$$\begin{aligned} &HB(a_1, b_1, a_2, b_2) = Expr\,(a_1, b_1, a_2, u(\tfrac{a_1+b_1}{2}, a_2), \\ &u(a_1, a_2), u(a_1, b_2), u(b_1, a_2), u(b_1, b_2), u(\tfrac{a_1+b_1}{2}, b_2), \\ &b_2, u(a_1, \tfrac{a_2+b_2}{2}), u(b_1, \tfrac{a_2+b_2}{2}), u(\tfrac{a_1+b_1}{2}, \tfrac{a_2+b_2}{2}))) \end{aligned} \quad (3)$$

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) [18] and Backus' FP [2]. 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 has lower precedence than function application.

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 work with lists of functions and lists of lists of arguments. This gives rise to the following generalized versions of BMF functionals.

- We define the *generalized map*, *gmap*, for a function h and a list of lists of arguments:

$$gmap\; h\; [l_1, \cdots, l_n] = [map\; h\; l_1, \cdots, map\; h\; l_n]$$

- 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:

$$\begin{aligned} dmap\; [h_1, \cdots, h_n]\;\; [l_1, \cdots, l_n] = \\ [map\; h_1\; l_1,\; \cdots, map\; h_n\; l_n] \end{aligned}$$

- The *generalized reduction*, *gred*, is defined on a binary associative operation and a list of lists:

$$gred \oplus [l_1, \cdots, l_n] = red \oplus [red \oplus l_1, \cdots, red \oplus l_n]$$

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

$$dmap\,[h, \cdots, h] = gmap\,h \quad (4)$$
$$gmap\,h = map\,h \circ flat \quad (5)$$
$$gred \oplus = red \oplus \circ flat \quad (6)$$
$$dmap[h_1, \cdots, h_n] \circ gmap\,h = dmap[h_1 \circ h, \cdots, h_n \circ h] \quad (7)$$

We use FP's *construction* functional for applying a list of functions to one argument; we denote it by <>:

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

The following properties hold:

$$< h > = h \quad (8)$$
$$< h_1, h_2 > \circ h = < h_1 \circ h\,,\ h_2 \circ h > \quad (9)$$

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

$$p \rightarrow h\,;\,h = h \quad (10)$$
$$(p \rightarrow b\,;\,c) \circ h = p \circ h \rightarrow b \circ h\,;\,c \circ h \quad (11)$$

To simplify the exposition, we use the notation $\neg p$ for $\neg \circ p$ and $f + g$ for $+ \circ [f, g]$.

Let us rephrase the specification (1) in our higher-order notation. Each function E_i in (1) takes a list of functional calls, which we denote by $calls_i$, applied via construction <> to the domain parameter. The higher-order representation of system (1) is then:

$$f_i = p_i \rightarrow b_i\,;\ E_i \circ < calls_i > \quad (i = 1, \cdots, n) \quad (12)$$

Take any i $(i = 1, \cdots, n)$. List $calls_i$ consists of two elements: the auxiliary function g_i and the function representing recursive function calls in (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} $(l = 1, \cdots, d_{ij})$ in the function $split_{ij} : \tau \rightarrow list\,\tau$; it yields a list of length d_{ij} containing all domain parameters with which f_j is called in the equation for f_i of (1). The list produced by $split_{ij}$ is always flat, therefore:

$$gmap\,h \circ split_{ij} = map\,h \circ split_{ij} \quad (13)$$

Using function $split_i = < split_{i1}, \cdots, split_{in} >$ of type $split_i : \tau \rightarrow list\ list\,\tau$, we can represent the recursive calls in $calls_i$ by $dmap\,[f_1, \cdots, f_n] \circ split_i$. Substituting this expression into (12) and flattening the argument list of E_i, we obtain the following higher-order notation of (1):

$$f_i = p_i \rightarrow b_i\,;\ E_i \circ flat \circ < g_i\,,\ dmap\,[f_1, \cdots, f_n] \circ split_i > \quad (i = 1, \cdots, n) \quad (14)$$

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

$$A = (m.is.1) \rightarrow HB;\ gred + \circ\ dmap\,[A, N] \circ split_1$$
$$N = (m.is.1) \rightarrow HB;\ gred + \circ < HB, mapN \circ split_{22} > \quad (15)$$

4 Functional parallel implementation

The presence of higher-order functions <>, *map* or *dmap* in (14) points already to divide-and-conquer parallelism in the specification: all 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 from f; the nodes at one level can be evaluated in parallel.

In this section, we present a systematic way of deriving a functional parallel implementation of (1). Informally, we proceed as follows. First we define a new data type that represents possible evaluation trees for a given specification; these trees are further used as structures for parallel computation, with processors associated to the nodes. Then a correspondence between this type and the original domain type τ is established and used for obtaining a parallel program schema that implements the initial specification.

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, $node_i$, or it is of the form $node_i\,(v^{(1)}_1, \cdots, v^{(1)}_{d_{i1}} : Tree_1, \cdots, v^{(n)}_1, \cdots, v^{(n)}_{d_{in}} : Tree_n)$, i.e., its root is $node_i$ and the number of its sons that are of type $Tree_j$ is d_{ij} – the length of the list produced by $split_{ij}$. The outdegree of $node_i$ is $d_i = \langle\ \Sigma\ j\ :\ 1 \le j \le n\ :\ d_{ij}\ \rangle$. We use Dijkstra's quantifier notation [8].

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: $tree_1 = \langle\ \sqcup\ tree\ :\ tree \in Tree_1\ :\ tree\ \rangle$, whose subtrees are all trees of type $Tree_1$.

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 predicates on the node set of $tree_1$, V, with the evident semantics: $is.root$, $is.node_i$ and $is.son_j$.

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

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

We omit range $1 \le i \le n$ if there is no danger of confusion and use the notation $\langle [\!]\ i :: is.node_i \rightarrow t_i \rangle$.

Let us construct an abstraction function [10] 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 div using $input$ – the domain parameter for which function f_1 must be computed: the value of div at the root is defined to be $input$ and, for the j-th son, w_j, of any node $v \in V$:

$$div\,w_j = \langle [\!]\ i :: is.node_i\,v \rightarrow (p_i \circ div\,v \rightarrow \perp\ ;\ P_j \circ flat \circ split_i \circ div\,v)\ \rangle \quad (16)$$

Here, $\perp$ stands for the undefined value and P_j is the projection function yielding the j-th element of a list.

Let us reformulate *div* in our higher-order notation. We define function ***node*** : $\tau \rightarrow V$ as div^{-1}:

$$node \circ div \;=\; Id \tag{17}$$

We give special names to some functions on τ:

$$\begin{aligned} p &= \langle [] i :: is.node_i \circ node \rightarrow p_i \rangle \\ split &= \langle [] i :: is.node_i \circ node \rightarrow split_i \rangle \\ fsplit &= flat \circ split \end{aligned}$$

Introducing function *father* on V that yields the father of a given node, we can reformulate (16) as follows:

$$\begin{aligned} div \;=\; & is.root \rightarrow input; \langle [] j :: (is.son_j \wedge \\ & \neg p \circ div \circ father) \rightarrow P_j \circ fsplit \circ div \circ father \rangle \end{aligned} \tag{18}$$

Here, the range of j is $1 \leq j \leq d$ where $d = \langle \max i : 1 \leq i \leq n : d_i \rangle$.

Using the abstraction function *div*, we define function $F: V \rightarrow \sigma$ as follows:

$$F \;=\; \langle [] i :: is.node_i \rightarrow f_i \circ div \rangle \tag{19}$$

From (18) and (19), we see immediately that the value of F at the root of $tree_1$ is $f_1(input)$, i.e., we have reduced the problem of implementing the specification to the problem of computing function F at the root of $tree_1$. This function combines functions $f_1, \cdots, f_n$; however, it is defined not on the domain τ but on the nodes of $tree_1$. We would like to distribute the computation of F among the nodes of the tree, but run into two problems. First, tree $tree_1$ is infinite. Second, the computation of F at the root is not yet parallelized: according to (19), we must compute $f_1(input)$ as before. Using the introduced functions and their properties, we can cope with both problems.

First, when dealing with real-life communication structures, we pick a fixed finite tree $tree \in Tree_1$ whose number of nodes does not exceed the number of processors available to us. This tree is determined by the predicate *is.leaf* 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.

We define function *sons* on V to return all sons of a given node as a list of n lists: the i-th list contains the sons of type $node_i$. Function *sons* is defined for such $v \in V$ that $is.leaf(v) = false$; it has the following properties:

$$\begin{aligned} & dmap\,[h_1, \cdots, h_n] \circ sons = \\ & \qquad gmap\,\langle [] i :: is.node_i \rightarrow h_i \rangle \circ sons \end{aligned} \tag{20}$$

$$(gmap\; div) \circ sons \;=\; split \circ div \tag{21}$$

Second, we can now parallelize the expression $f_i \circ div$ of (19) via transformation:

$$\begin{aligned} & f_i \circ div \\ = & \quad \{ \text{ equality (10) } \} \\ & is.leaf \rightarrow f_i \circ div;\; f_i \circ div \\ = & \quad \{ \text{ equalities (14), (11) } \} \\ & is.leaf \rightarrow f_i \circ div;\; p_i \circ div \rightarrow b_i \circ div; \\ & E_i \circ flat \circ \langle g_i,\; dmap\,[f_1, \cdots, f_n] \circ split_i \rangle \circ div \end{aligned}$$

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

$$\begin{aligned} & E_i \circ flat \circ \langle g_i,\; dmap\,[f_1, \cdots, f_n] \circ split_i \rangle \circ div \\ = & \quad \{ \text{ equality (9) } \} \\ & E_i \circ flat \circ \langle (g_i \circ div),\; (dmap\,[f_1, \cdots, f_n] \circ split_i \circ div) \rangle \\ = & \quad \{ \text{ equality (21), definition (19) } \} \\ & E_i \circ flat \circ \langle (g_i \circ div),\; (dmap\,[f_1, \cdots, f_n] \circ gmap\; div \circ sons) \rangle \\ = & \quad \{ \text{ equalities (7), (20) } \} \\ & E_i \circ flat \circ \langle (g_i \circ div),\; (gmap\,(is.node_i \rightarrow f_i \circ div) \circ sons) \rangle \\ = & \quad \{ \text{ definition (19), equality (4) } \} \\ & E_i \circ flat \circ \langle (g_i \circ div),\; (gmap\; F \circ sons) \rangle \end{aligned}$$

The following transformations are applied again to the entire expression:

$$\begin{aligned} & f_i \circ div \\ = & \; is.leaf \rightarrow f_i \circ div; p_i \circ div \rightarrow b_i \circ div; \\ & E_i \circ flat \circ \langle (g_i \circ div),\; (gmap\; F \circ sons) \rangle \\ = & \quad \{ \text{ composition with } Id \text{ from (17) } \} \\ & (is.leaf \rightarrow f_i \circ div\,; p_i \circ div \rightarrow b_i \circ div;\; E_i \circ \\ & flat \circ \langle g_i \circ div,\; gmap\; F \circ sons \rangle) \circ node \circ div \\ = & \quad \{ \text{ equalities (11), (9) } \} \\ & (\,is.leaf \circ node \rightarrow f_i\,; p_i \rightarrow b_i\,;\; E_i \circ flat \circ \\ & \langle g_i,\; (gmap\; F \circ sons \circ node) \rangle\,) \circ div \end{aligned}$$

The obtained expression is a composition of two functions, the second of which is *div*. We call the first one $conq_i$ (for "conquer"). Substituting expression for $f_i \circ div$ into (19), we arrive at the final expression for F:

$$F \;=\; \langle [] i :: is.node_i \rightarrow conq_i \circ div \rangle \tag{22}$$

$$\begin{aligned} div \;=\; & is.root \rightarrow input; \langle [] j :: (is.son_j \wedge \\ & \neg p \circ div \circ father) \rightarrow P_j \circ fsplit \circ div \circ father \rangle \end{aligned} \tag{23}$$

$$\begin{aligned} conq_i \;=\; & is.leaf \circ node \rightarrow f_i\,; p_i \rightarrow b_i\,; \\ & E_i \circ flat \circ \langle g_i, (gmap\; F \circ sons \circ node) \rangle \end{aligned} \tag{24}$$

There are three important observations to be made about (22)–(24). First, for computing function F at some node of *tree*, we can use the results of computing F at its sons and the result of computing *div* (which is a part of F) at its father. 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 *mapped* to the sons, i.e., the computations at the sons are independent of each other. They are also independent of the computation of auxiliary function g_i, because of $\langle\rangle$. Third, we arrived at this parallel implementation from the specification by calculation using formal rules.

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 $f_1(input)$. We construct type $Tree_1$, which captures the communication structure needed by the specification, and choose a particular 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 (22)–(24) and *input* is available at the root processor, then the result obtained at the root is the desired value $f_1(input)$.

Expressions (22)–(24) tell us that function F is computed at each node of *tree* in two steps:

1. Apply *div*. This function computes the corresponding domain parameter for a node. Equation (23) 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. In other words, the number of processors exceeds in this case the degree of parallelism in the specification.

2. Function *conq* takes the domain parameter returned by *div*. Equation (24) prescribes that further computations depend on the type of the node (index i) and on its position in the tree. In the leaf nodes, f_i for the domain parameter must be computed (sequentially). In the non-leaf nodes, 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.

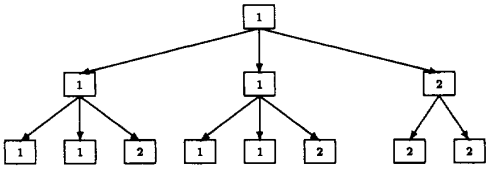

Figure 1: Tree of type $Tree_1$ – example

In our example, type $Tree_1$ corresponds to the function A to be computed. An example tree of this type with 12 nodes of two types is shown in Figure 1. The maximal outdegree of a node is 3, we use concrete functions son_1, son_2 and son_3 which are the components of the general function *sons*. We denote the negation of *m.is.1* by *m.not.1*.

Parallel implementation of A is obtained as a specialization of the general schema (22)–(24):

$$
\begin{aligned}
A &= is.node_1 \rightarrow conq_1 \circ div\,;\ conq_2 \circ div \\
div &= is.root \rightarrow input;\ \langle [] j : 1 \le j \le 3 : (is.son_j \wedge \\
&\quad m.not.1 \circ div \circ father) \rightarrow P_j \circ fsplit \circ div \circ father \rangle \\
conq_1 &= is.leaf \circ node \rightarrow A\,;\ A \circ son_1 \circ node \\
&\quad + A \circ son_2 \circ node + A \circ son_3 \circ node \\
conq_2 &= is.leaf \circ node \rightarrow N; \\
&\quad HB + A \circ son_1 \circ node + A \circ son_2 \circ node
\end{aligned}
$$

5 Imperative parallel implementation

In the previous section, we have demonstrated how the specification of an algorithm can be "refined" into a higher-order functional parallel implementation that consists of a communication structure defined by type $Tree_1$, and a function F, defined by (22)–(24), which is to be computed simultaneously at all nodes of the structure. In this section, we take this functional implementation and convert it to an imperative program with explicit message-passing. The target program prescribes the computation and communication for processors that are assigned to the nodes of the tree.

According to (22)–(24), F specifies a computation in the SPMD (single-program-multiple-data) model: the same function applies to all nodes of the tree. The variations in computation at the different nodes are expressed by predicates *is.leaf*, $is.node_i$ and $is.son_j$. In other words, the behaviour depends on the type of the node and on its position in the tree.

Our imperative target program `Node`, which is presented in Figure 2, is therefore executed at every node of the tree. One implicit parameter of program `Node` is the id of the associated processor (variable `my_id`).

The imperative program uses procedures that implement the functions and predicates of the functional implementation: `Is_root`, `Is_leaf`, `Outdegree`, `Father`, `Son`. For brevity, we have not listed the procedure interfaces in our import list; a strongly typed language would, of course, have to do so. 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 by F, depend on the type of node $node_i$: $split_i$, f_i, g_i and E_i. They are defined on the domain parameters. In the imperative program, they are implemented by the corresponding procedures `Split`, `Compute_f`, `Compute_g` and `Compute_E`. The first parameter of these procedures is the type, `type`, of the node and the second is the domain parameter, `param`.

Communications between processors include sending and receiving data. They are implemented 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.

Formulae (22)–(24) are implemented in the program as follows. The domain parameter *input* and the type `type` of the root node are the input to the

```
PROGRAM Node;
 IMPORT Father,Son,Outdegree,Is_leaf,Is_root,
        Split,Compute_E,Compute_f,Compute_g;
  BEGIN_Node
   IF Is_root(my_id)
    THEN READ (type,param)
    ELSE RECV (type,param) FROM Father(my_id)
   END_IF;
   IF Is_leaf(my_id)
    THEN
     result := Compute_f(type,param)
    ELSE
     son_param := Split(type,param);
     FOR k:=1 TO Outdegree(my_id) DO
       SEND (son_param[k]) TO Son(my_id,k)
     END_FOR;
     res_aux := Compute_g(type,param);
     FOR k:=1 TO Outdegree(my_id) DO
      RECV (res[k]) FROM Son(my_id,k)
     END_FOR;
     result := Compute_E(type, res_aux,
                         res[1], ..., res[n])
   END_IF;
   IF Is_root(my_id)
    THEN WRITE (result)
    ELSE SEND (result) TO Father(my_id)
   END_IF;
  END_Node
```

Figure 2: SPMD program schema

program. The root receives them by `READ (<data>)`. According to (23) for *div*, the domain parameters at non-root nodes are obtained by applying *split* to the domain parameter of the father. In the program, this is realized by procedure `Split`, which yields the domain parameters and the type values for the sons. For simplicity of the exposition, our imperative implementation presumes that there are strictly fewer processors than are required to realize all parallelism. That is, the basic case is not reached during the computation of the domain parameters for the sons.

Having obtained the domain parameter, i.e., having computed function *div*, it remains to compute *conq*. In the program, there is firstly a conditional statement corresponding to the FP-condition in (24). For a leaf node, function f_i is computed sequentially by procedure `Compute_f`. For a non-leaf node, there are different ways of implementing the constructions with <> and *gmap*. Our node program is sequential; computation starts with the second component of <> which has the form $gmap\ F \circ sons$. This means computing F at all sons of the current node independently. To this effect, the program computes array `son_param` by procedure `Split`. Each element of `son_param` is a pair (`type`,`param`) which is then sent to the corresponding son. This way, the computation of *conq* at the father is synchronized with the computations of *div* at the sons. After sending the necessary data to all sons, we compute the first component of <>, the auxiliary function, by calling procedure `Compute_g` whose result is `res_aux`. We must then receive the results from the sons; they are used by procedure `Compute_E`. This procedure yields the result which, in fact, represents the value of F at the current node. This value must either be sent to the father or, in case of the root node, it is the output $f_1(input)$ of the whole program.

For lack of space, we do not present the specialization of this imperative parallel program schema for the example.

6 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 dependencies 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 (22)–(24). In the development of the imperative program, this parallelism is converted to a programming language. 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.

- **Sequentializing**. In the functional parallel implementation, there are potentially parallel threads inside one node: communicating with the sons and/or computing the auxiliary function. In the imperative implementation, we execute them sequentially; on some architectures, however, the use of multithreading can improve efficiency.

- **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).

- **Processor number**. The maximal number of processors for a given value of *input* can sometimes be determined analytically, as in our example. However, there is an adaptive variant of the integration algorithm where the basic predicate is $HB(a1, b1, a2, b2, m) < \varepsilon$. In this case, the actual amount of work is known only at run time, and good performance must be achieved by dynamic load-balancing.

- **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 [9].

- **Load Balancing.** The amount of computational work in a processor strongly depends on its position in the tree. The load balance can be improved using additional transformations at the functional level that are explained subsequently.

Let us discuss briefly two ways of improving the imperative program performance.

First, a particular specification may be matched in different ways with format (1). Another match for our example (2) 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 better balanced, and the outdegree of each node is reduced by 1.

These and other improvements of the target program, can be realized by additional 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. More on this in a different paper.

7 Related work

There has been a lot of work on formal parallelization of conventional (not mutually recursive) divide-and-conquer.

There is an algebraic model for describing divide-and-conquer and a language, *Divacon*, based on it [14, 16]; communication issues in this model have also been studied [4]. The approach is based on the theory of pseudomorphisms which has much in common with the Bird-Meertens formalism. Our approach differs from this work in three main aspects. First, we consider a more general case, allowing a specification to consist of several, possibly mutually recursive functions and also 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. This enables a more general treatment of divide-and-conquer, since we need not ensure that recursive calls are applied always to smaller chunks of data as in [16]. In our example, there are no chunks at all! Of course, this has the drawback that we do not consider the effect of the data size on communication and parallelism in our performance analysis.

In [7, 10], the higher-order approach was used for transforming non-linear recursion, typically divide-and-conquer, into tail recursion and then pipelining the latter. Pipelining reduces the parallelism inherent in 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-like parallelism of divide-and-conquer and show that it can be realized with static and local communication.

Our paper has much in common with recent work investigating parallelism with the Bird-Meertens formalism [18]. Our extension to BMF consists of generalized versions of *map* and *red* and transformation rules for them.

The idea of abstract data type transformation that was used in [10] for parallelizing linear recursion is applied here in a broader context. We derive an implementation for both stages of divide-and-conquer and show that the abstraction function expresses the essence of the dividing stage in divide-and-conquer.

An approach based on unfolding the recursion is described in [11, 13] for the bitonic sort, which is also a divide-and-conquer algorithm. The derived parallel algorithm has logarithmic complexity and was proved to be optimal. The disadvantage of this approach is that the computational complexity of the derivation process depends (quadratically) on the problem size. In our present 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 [15]. There are development systems, like *PARSE* [5], that support parallel program derivation and performance evaluation 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., [17]), but we are not aware of any that considers non-homogeneous trees, as in our example.

8 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 (semi-)automatic 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

non-procedural (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. The target program is adapted 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.

Acknowledgements

Thanks to all three anonymous referees for many useful comments.

References

[1] 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.

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

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

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

[5] T. Casavant, H. Dietz, P. Sheu, and H. Siegel. The PARSE approach to programming non-shared memory parallel computers. In *Int. Conf. Paral. Proc.*, pages 380–389, 1989.

[6] M. I. Cole. A "skeletal" approach to the exploitation of parallelism. In *Proc. CONPAR 88*, pages 667–675. British Computer Society Workshop Series, 1989.

[7] I. P. de Guzman, P. G. Harrison, and E. Medina. A higher-order approach to parallel algorithms. *The Computer Journal*, 36(3):254–268, 1993.

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

[9] S. Gorlatch. Parallel program development for a recursive numerical algorithm: A case study. Technical Report SFB 342/7/92, Institute for Computer Science, Technical University of Munich, March 1992.

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

[11] 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.

[12] 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.

[13] 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.

[14] Z. G. Mou. Divacon: A parallel language for scientific computing based on divide and conquer. In *Proc. 3rd Symposium on the Frontiers of Massively Parallel Computation*, pages 451–461, October 1990.

[15] 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.

[16] 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.

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

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

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

[20] C. Zenger. Sparse grids. Technical Report SFB-Nr. 342/18/90 A, Techn. Univ. Muenchen, October 1990.