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.
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 \( f = \{ f_1, \ldots, f_n \} \). Each function \( f_i \) (\( i = 1, \ldots, n \)) is defined by the equation:

\[
f_i(x) = \text{if } p_i(x) \text{ then } b_i(x) \text{ else } E_i(g_i, f, x) \text{ fl }
\]

Here \( g = (g_1, \ldots, g_n) \) 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 \( \tau \rightarrow \sigma \). The domain \( \tau \) and the range \( \sigma \) are arbitrary sets; they may be structured but we ignore their structural properties. Elements of \( \tau \) are called domain parameters; \( p_i \) the basic predicates and \( b_i \) the 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 \( f_j(\varphi_j(x)) \), where functions \( \varphi_j : \tau \rightarrow \tau \) 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 \( \text{input} \), computes \( f_1(\text{input}) \) 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 \( \varphi(z) = z + a \), 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 non-adaptive 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,

\[
q = \int_{a_1}^{b_1} \int_{a_2}^{b_2} u(x_1, x_2) \, dx_1 \, dx_2
\]

can be 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:

\[
A(a_1, b_1, a_2, b_2, m) = \begin{cases} f_1 \text{ if } (m=1) \text{ then } & \text{else } A(a_1, a_1+b_2, b_2, m-1) + A(a_1+b_2, b_2, a_2, b_2, m-1) + N(a_1, b_1, a_2, b_2, m) \text{ fl } \end{cases}
\]

\[
N(a_1, b_1, a_2, b_2, m) = \begin{cases} f_1 \text{ if } (m=1) \text{ then } & \text{else } N(a_1, a_1+b_2, b_2, m-1) + N(a_1+b_2, b_2, a_2, b_2, m-1) + HB(a_1, b_1, a_2, b_2, m) \text{ fl } \end{cases}
\]

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: \( \varphi_1^i(a_1, b_1, a_2, b_2, m) = (a_1, a_1+b_2, a_2, b_2, m-1) \), \( \varphi_2^i(a_1, b_1, a_2, b_2, m) = (a_1+b_2, b_2, a_2, b_2, m-1) \), \( \varphi_1^i = \text{id} \) (identity). There is one basic predicate, we shall name it \( m \text{ is } l \). It is defined by \( (m \text{ is } l)(a_1, a_2, b_2, b_2, m) = (m = 1) \). 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 \(Expr:\)

\[
HB(a_1, b_1, a_2, b_2) = Expr(a_1, b_1, a_2, b_2, u(a_1+b_2, a_2), u(a_1, a_1+b_2, u(b_1, b_2), u(a_1+b_2, b_2), (3) b_2, u(a_1, a_1+b_2), u(b_1, a_1+b_2), u(a_1+b_2, a_1+b_2))
\]

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 Backs’ 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:

- \( \text{map} \) applies function \( h \) to all elements of a list \([a_1, \ldots, a_n] \):

\[
\text{map } h [a_1, \ldots, a_n] = [h a_1, \ldots, h a_n]
\]

- \( \text{red} \) (reduction) computes a value of some type from a list \([a_1, \ldots, a_n] \) of values of that type by applying an associative binary operation \( @ \):

\[
\text{red } @ [a_1, \ldots, a_n] = a_1 @ \cdots @ a_n
\]

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 \textit{generalized map}, \textit{gmap}, for a function \( h \) and a list of lists of arguments:

\[
gmap h [l_1, \ldots, l_n] = [\text{map } h l_1, \ldots, \text{map } h l_n]
\]

- The \textit{distributed map}, \textit{dmap}, is the following functional on a list \([h_1, \ldots, h_n] \) of \( n \) functions and a list \([l_1, \ldots, l_n] \) of \( n \) lists:

\[
dmap [h_1, \ldots, h_n] [l_1, \ldots, l_n] = [\text{map } h_1 l_1, \ldots, \text{map } h_n l_n]
\]

369
• The generalized reduction, $\text{gred}$, is defined on a binary associative operation and a list of lists:

$$\text{gred } \odot [h_1, \ldots, h_n] = \text{red } \odot [\text{red } @ h_1, \ldots, \text{red } @ h_n]$$

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

$$\text{dmap}[h_1, \ldots, h_n] = \text{gmap } h$$

$$\text{gmap } h = \text{map } h \circ \text{flat}$$

$$\text{dmap}[h_1, \ldots, h_n] \circ \text{gmap } h = \text{dmap}[h_1 \circ h, \ldots, h_n \circ h]$$

We use FP's construction functional for applying a list of functions to one argument; we denote it by $\langle \rangle$:

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

The following properties hold:

$$< \emptyset > = h$$

$$< h_1, h_2 > \circ h = < h_1 \circ h, h_2 \circ h >$$

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

$$p \rightarrow h; h = h$$

$$p \rightarrow b; c \circ h = p \circ h \rightarrow b ; c \circ h$$

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 $\text{calls}_i$, applied via construction $\langle \rangle$ to the domain parameter. The higher-order representation of system (1) is then:

$$f_i = p_i \rightarrow b_i; E_i \circ \text{calls}_i \quad (i = 1, \ldots, n) \quad (12)$$

Take any $i$ $(i = 1, \ldots, n)$. List $\text{calls}_i$ consists of two elements: the auxiliary function $y_i$ and the function representing recursive function calls in (1). Any $f_j$ $(j = 1, \ldots, n)$ may be called by $f_i$, possibly more than once, with specific, "shifted" domain parameters. We combine all corresponding shifts $\psi^j_i$ $(i = 1, \ldots, d_i)$ in the function $\text{split}_{ij}$: $\tau \rightarrow \text{list } \tau$; it yields a list of length $d_i$ containing all domain parameters with which $f_j$ is called in the equation for $f_i$ of (1). The list produced by $\text{split}_{ij}$ is always flat, therefore:

$$\text{gmap } h \circ \text{split}_{ij} = \text{map } h \circ \text{split}_{ij}$$

Using function $\text{split}_{ij} = \langle \text{split}_{i1}, \ldots, \text{split}_{in} \rangle$ of type $\text{split}_{ij} : \tau \rightarrow \text{list } \tau$, we can represent the recursive calls in $\text{calls}_i$ by $\text{dmap}[f_1, \ldots, f_n] \circ \text{split}_{ij}$. Substituting this expression into (12) and flattening the argument list of $E_i$, we obtain the following higher-order notation of (1):

$$f_i = p_i \rightarrow b_i; E_i \circ \text{flat} \circ < g_i, \text{dmap}[f_1, \ldots, f_n] \circ \text{split}_{ij} > \quad (i = 1, \ldots, n) \quad (14)$$

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

$$A = (m \cdot \text{is.1}) \rightarrow \text{HB}; \text{gred } + \circ \text{dmap}[A, N] \circ \text{split}_{i}$$

$$N = (m \cdot \text{is.1}) \rightarrow \text{HB}; \text{gred } + \circ < \text{HB}, \text{map } N \circ \text{split}_{22} >$$

$$\text{div } w_j = \langle [i] \circ \text{is } node, v \rightarrow (p_i \circ \text{div } v \rightarrow \perp; P_j \circ \text{flat } \circ \text{split}_{ij} \circ \text{div } v) \rangle$$

4 Functional parallel implementation

The presence of higher-order functions $\langle \rangle$, $\text{map}$ or $\text{dmap}$ 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 $\tau$ is established and used for obtaining a parallel program schema that implements the initial specification.

We introduce $n$ types of trees, $\text{Tree}_i$, $(i = 1, \ldots, n)$, one for each function $f_i$ in (1). The nodes are taken from the set $\{\text{node}_i \mid i = 1, \ldots, n\}$. A tree of type $\text{Tree}_i$ is either a single node, $\text{node}_i$, or it is of the form $\text{node}_i(v_1^i, \ldots, v_{d_i}^i) = \text{Tree}_1, \ldots, \text{Tree}_n$ ($\text{Tree}$, $\text{Tree}_i$), i.e., its root is $\text{node}_i$ and the number of its sons that are of type $\text{Tree}_i$ is $d_i$ – the length of the list produced by $\text{split}_{ij}$. The outdegree of $\text{node}_i$ is $d_i = \{\Sigma j : 1 \leq j \leq n : d_j\}$. We use Dijkstra's quantifier notation [8].

The evaluation graph of function $f_i$ in (1) is of type $\text{Tree}_i$. Because we want to derive a parallel program that computes $f_i$, we are particularly interested in the type $\text{Tree}_1$; it is a partially ordered set: $x \preceq y$ iff $x$ is a subtree of $y$. The least upper bound of $\text{Tree}_i$ is the infinite tree: $\text{Tree} = (\lambda i \in \text{Tree}_i : \text{Tree}_i) : \text{Tree}_1$, whose subtrees are all trees of type $\text{Tree}_i$.

$\text{Tree}_1$ 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 $\text{Tree}_i$, $\mathcal{V}$, with the evident semantics: $\text{is.root}$, $\text{is.node}$, and $\text{is.son}$.

We introduce an abbreviated notation for conditional functions and predicates on $\mathcal{V}$:

$$[i] : 1 \leq i \leq n : \text{is.node}_i \rightarrow t_i = \text{is.node}_1 \rightarrow t_1; \ldots; \text{is.node}_n \rightarrow t_n$$

We omit range $1 \leq i \leq n$ if there is no danger of confusion and use the notation $[[i] : \text{is.node}_i \rightarrow t_i]$.

Let us construct an abstraction function [10] that maps from the concrete type $\mathcal{V}$ (the nodes of $\text{Tree}_i$) to the abstract type $\tau$ (the domain parameters). We call our abstraction function $\text{div}$: it "divides" the domain parameters and distributes them among the nodes. We define $\text{div}$ using $\text{input}$ – the domain parameter for which function $f_1$ must be computed: the value of $\text{div}$ at the root is defined to be $\text{input}$ and, for the $j$-th son, $w_j$, of any node $v \in \mathcal{V}$:

$$\text{div } w_j = (\langle [i] : \text{is.node}_i, v \rightarrow (p_i \circ \text{div } v \rightarrow \perp ; P_j \circ \text{flat } \circ \text{split}_{ij} \circ \text{div } v) \rangle$$

370
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 \(\text{div}\) in our higher-order notation. We define function \(\text{node} : \tau \rightarrow V\) as \(\text{div}^{-1}\):

\[
\text{node} \circ \text{div} = \text{Id}
\]

We give special names to some functions on \(\tau\):

\[
p = (\{i :: \text{is.node}_i \circ \text{node} \rightarrow p_i\})
\]

\[
\text{split} = (\{i :: \text{is.node}_i \circ \text{node} \rightarrow \text{split}_i\})
\]

\[
\text{fsplit} = \text{flat} \circ \text{split}
\]

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

\[
\text{div} = \text{is.root} \rightarrow \text{input}; (\{j :: (\text{is.son}_j \wedge 
\neg \text{p} \circ \text{div} \circ \text{father}) \rightarrow P_j \circ \text{fsplit} \circ \text{div} \circ \text{father}\})
\]

Here, the range of \(j\) is \(1 \leq j \leq d\) where \(d = (\max i : 1 \leq i \leq n : d_i)\).

Using the abstraction function \(\text{div}\), we define function \(F : V \rightarrow \sigma\) as follows:

\[
F = (\{i :: \text{is.node}_i \rightarrow f_i \circ \text{div}\})
\]

From (18) and (19), we see immediately that the value of \(F\) at the root of \(\text{tree}_1\) is \(f_1(\text{input})\), i.e., we have reduced the problem of implementing the specification to the problem of computing function \(F\) at the root of \(\text{tree}_1\). This function combines functions \(f_1, \ldots, f_n\); however, it is defined not on the domain \(\tau\) but on the nodes of \(\text{tree}_1\). We would like to distribute the computation of \(F\) among the nodes of the tree, but run into two problems. First, \(\text{tree}_1\) is infinite. Second, the computation of \(F\) at the root is not yet parallelized: according to (19), we must compute \(f_1(\text{input})\) 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 \(\text{tree}_1 \in \text{Tree}\), whose number of nodes does not exceed the number of processors available to us. This tree is determined by the predicate \(\text{is.leaf}\) which selects the leaves of the tree. For each particular finite tree, we shall use the restrictions of all functions originally defined on \(V\) - like \(\text{div}\), 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 \(\text{sons}\) 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 \(\text{node}_i\). Function \(\text{sons}\) is defined for such \(v \in V\) that \(\text{is.leaf}(v) = \text{false}\;\); it has the following properties:

\[
\text{dmap}[h_1, \ldots, h_n] \circ \text{sons} = \text{gmap}(\{i :: \text{is.node}_i \rightarrow h_i\}) \circ \text{sons}
\]

\[
(\text{gmap} \circ \text{div}) \circ \text{sons} = \text{split} \circ \text{div}
\]

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

\[
f_i \circ \text{div} = \text{is.leaf} \rightarrow f_i \circ \text{div}; f_i \circ \text{div} = \{\text{equality}\}
\]

\[
is.leaf \rightarrow f_i \circ \text{div}; P_i \circ \text{div} \rightarrow b_i \circ \text{div};
\]

\[
E_i \circ \text{flat} \circ <g_i, \text{gmap}[f_1, \ldots, f_n] \circ \text{split}, > \circ \text{div}
\]

The last alternative, which applies in the case of \(\text{is.leaf} \wedge (\neg \text{p} \circ \text{div})\), is transformed further:

\[
E_i \circ \text{flat} \circ <g_i, \text{gmap}[f_1, \ldots, f_n] \circ \text{split}, > \circ \text{div} = \{\text{equality}\}
\]

\[
\text{is.leaf} \rightarrow f_i \circ \text{div}; P_i \circ \text{div} \rightarrow b_i \circ \text{div};
\]

\[
E_i \circ \text{flat} \circ <g_i, \text{gmap}[f_1, \ldots, f_n] \circ \text{sons} > \circ \text{div}
\]

The following transformations are applied again to the entire expression:

\[
f_i \circ \text{div} = \text{is.leaf} \rightarrow f_i \circ \text{div}; P_i \circ \text{div} \rightarrow b_i \circ \text{div};
\]

\[
E_i \circ \text{flat} \circ <g_i, \text{gmap}[f_1, \ldots, f_n] \circ \text{sons} > \circ \text{div}
\]

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

\[
F = (\{i :: \text{is.node}_i \rightarrow \text{conq}_i \circ \text{div}\})
\]

\[
\text{div} = \text{is.root} \rightarrow \text{input}; (\{j :: (\text{is.son}_j \wedge 
\neg \text{p} \circ \text{div} \circ \text{father}) \rightarrow P_j \circ \text{fsplit} \circ \text{div} \circ \text{father}\})
\]

\[
\text{conq}_i = \text{is.leaf} \circ \text{node} \rightarrow f_i; P_i \rightarrow b_i;
\]

\[
E_i \circ \text{flat} \circ <g_i, \text{gmap}[f_1, \ldots, f_n] \circ \text{node}\>
\]

There are three important observations to be made about (22)–(24). First, for computing function \(F\) at some node of \(\text{tree}\), we can use the results of computing \(F\) at its sons and the result of computing \(\text{div}\) (which is a part of \(F\)) at its father. Therefore, the computation of \(F\) at the root of \(\text{tree}\) can be distributed over the nodes of \(\text{tree}\) 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 \(<\>\). 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 \( \text{input} \), computes \( f_1(\text{input}) \). We construct type \( \text{Tree}_1 \), 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 \( \text{input} \) is available at the root processor, then the result obtained at the root is the desired value \( f_1(\text{input}) \).

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

1. Apply \( \text{div} \). This function computes the corresponding domain parameter for a node. Equation (23) says that, for the root, this parameter is \( \text{input} \) and, for each other node, it is determined by the result of \( \text{div} \) at its father. Thus, in the computation of \( \text{div} \), data is flowing from the root to the leaves of the tree. Note that if the domain parameter at some node makes \( p_1 \) 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 \( \text{conq} \) takes the domain parameter returned by \( \text{div} \). 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 Tree1 – example](image)

In our example, type \( \text{Tree}_2 \) 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 \( \text{son}_1 \), \( \text{son}_2 \) and \( \text{son}_3 \), which are the components of the general function \( \text{sons} \). We denote the negation of \( m.\text{is}.1 \) by \( m.\text{not}.1 \).

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

\[
A = \text{is}.\text{node}_1 \rightarrow \text{conq}_1 \circ \text{div}; \text{conq}_2 \circ \text{div}
\]

\[
div = \text{is}.\text{root} \rightarrow \text{input}; ([j : 1 \leq j \leq 3; (is.\text{son}_j \land m.\text{not}.1 \circ \text{div} \circ \text{father}) \rightarrow P_j \circ \text{fsplit} \circ \text{div} \circ \text{father})
\]

\[
\text{conq}_1 = \text{is}.\text{leaf} \circ \text{node} \rightarrow A; A \circ \text{son}_1 \circ \text{node} + A \circ \text{son}_2 \circ \text{node} + A \circ \text{son}_3 \circ \text{node}
\]

\[
\text{conq}_2 = \text{is}.\text{leaf} \circ \text{node} \rightarrow N;
\]

\[
HB + A \circ \text{son}_1 \circ \text{node} + A \circ \text{son}_2 \circ \text{node} + A \circ \text{son}_3 \circ \text{node}
\]

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 \( \text{Tree}_1 \), 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 \( \text{is}.\text{leaf} \), \( \text{is}.\text{node}_1 \) and \( \text{is}.\text{son}_j \). In other words, the behavior depends on the type of the node and on its position in the tree.

Our imperative target program \( \text{Node} \), which is presented in Figure 2, is therefore executed at every node of the tree. One implicit parameter of program \( \text{Node} \) is the id of the associated processor (variable \( \text{my}._{\text{id}} \)).

The imperative program uses procedures that implement the functions and predicates of the functional implementation: \( \text{is}.\text{root} \), \( \text{is}.\text{leaf} \), \( \text{Outdegree} \), \( \text{Father} \), \( \text{Son} \). 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; \( \text{Son} \) has an additional parameter \( k \), specifying the \((k-\text{th})\) son to be computed.

The following functions, used by \( F \), depend on the type of node \( \text{node} \): \( \text{split} \), \( f_i \), \( g_i \) and \( E_i \). They are defined on the domain parameters. In the imperative program, they are implemented by the corresponding procedures \( \text{Split} \), \( \text{Compute}_f \), \( \text{Compute}_g \) and \( \text{Compute}_E \). The first parameter of these procedures is the type, \( \text{type} \), of the node and the second is the domain parameter, \( \text{param} \).

Communications between processors include sending and receiving data. They are implemented by the statements \( \text{SEND} \langle \text{data} \rangle \) TO \langle \text{partner} \rangle \) and \( \text{RECV} \langle \text{data} \rangle \) FROM \langle \text{partner} \rangle. Here, \langle \text{partner} \rangle is the id of the processor with whom the communication takes place.

Formulas (22)-(24) are implemented in the program as follows. The domain parameter \( \text{input} \) and the type \( \text{type} \) 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
END_IF;
ELSE RECV (type,param) FROM Father(my-id)
END_IF
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 := 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 cong. 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 cong 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 $f_i(input)$ 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 $\sigma$ and $\tau$. 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.

- **Sequentalizing.** 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 $HB(a_1, b_1, a_2, b_2, m) < c$. In this case, the actual amount of work is known only at run time, and good performance must be achieved by dynamic load-balancing.
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 \( \tau \). 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 reduce 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


