Formal Derivation and Implementation of Divide-and-Conquer on a Transputer Network

Sergei GORLATCH

Fakultät für Mathematik und Informatik, Universität Passau
D–94030 Passau, Germany. E-mail: gorlatch@fmi.uni-passau.de

Abstract

This paper considers parallel program development based on functional mutually recursive specifications. The development yields a communication structure linking an arbitrary fixed number of processors and an SPMD program executable on the structure. There are two steps in the development process: first, a parallel functional implementation is obtained through formal transformations in the Bird-Meertens formalism; it is then systematically transformed into an imperative target program with message passing. The approach is illustrated with a divide-and-conquer algorithm for numerical two-dimensional sparse grid integration. The optimization of the target program and the results of experimental performance measurements on a 64-transputer network under OS *Parix* are presented.

1 Introduction

We take the following approach to parallelization: we try to identify certain standard patterns of high-level functional specifications and to associate equivalent parallel programs to them. To that purpose, we construct, first, a communication structure suitable for the specification and second, a program with message passing that works on that structure. Together, they constitute our parallel target program. We aim at parallel programs that can be directly implemented on contemporary multiprocessor architectures, like transputer networks, with the following properties:

- explicit message passing and multithreading,
- an arbitrary but fixed number of processors,
- SPMD programming.

The communication structure of the program is determined, to a large extent, by the form of the underlying data structures and their dependences in the specification. This idea is one of the corner-stones in systolic design [1]. We develop this principle for a more general class of recursive specifications and try to derive both, the communication structure and the program to be executed on each processor of the structure, through formal transformations. The exclusive use of semantically sound transformation rules in the course of the development guarantees the correctness of the target program. We use the rules of two well-known formalisms: Backus' FP [2] and the Bird-Meertens theory of higher-order functions over lists [3].

We consider a class of mutually recursive functional specifications that extends the conventional divide-and-conquer paradigm. Such specifications are used, e.g., in recently developed numerical algorithms on sparse grids [4]. We have used a transformational approach based on the Bird-Meertens formalism to construct formally a *functional* parallel implementation schema for mutually recursive specifications [5]. This schema includes a topology of processors in the form of a communication tree, and a functional program which must be executed on each processor. In this paper, we demonstrate how this schema can be systematically transformed into an *imperative* SPMD program with message passing and consider possible improvements of that program. Transformation and improvements are illustrated with a particular realistic example – a two-dimensional numerical integration algorithm on sparse grid. We derive several versions of the implementation program and discuss their performance on a 64-node *Parsytec* transputer system. Our implementations run under OS *Parix* [6] and make use of virtual topology libraries provided by *Parix*.

In Section 2, the example specification is introduced. Section 3 presents briefly the transformation process that yields the parallel functional implementation of the specification. A systematic transformation to an imperative parallel program with message passing is described in Section 4. In Section 5, this program is being consecutively optimized and the efficiency of the corresponding implementations on a transputer network are discussed. Finally, Section 6 summarizes the results, compares our approach with related work and outlines problems for further study.

2 Example Specification

We start with an example specification which is used throughout the paper. We use the notation of Backus' FP [2] for conditional expressions:

$$p \rightarrow b;\ c \quad \equiv \quad \begin{cases} b\,, & \text{if } p \\ c\,, & \text{if } \neg p \end{cases}$$

As a sample specification we consider an algorithm for numerical two-dimensional non-adaptive integration [4]. The value q of the integral for a given function u vanishing on the boundary, in the domain $[a_1, b_1] \times [a_2, b_2]$, is approximated for a given mesh-width 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:

$$\begin{cases} A(a_1,b_1,a_2,b_2,m) \;=\; (m\!=\!1) \;\rightarrow\; HB(a_1,b_1,a_2,b_2)\,; \\ \quad A(a_1, \frac{a_1+b_1}{2}, a_2, b_2, m-1) + A(\frac{a_1+b_1}{2}, b_1, a_2, b_2, m-1) + N(a_1,b_1,a_2,b_2,m) \\ N(a_1,b_1,a_2,b_2,m) \;=\; (m\!=\!1) \;\rightarrow\; HB(a_1,b_1,a_2,b_2)\,; \\ \quad N(a_1,b_1,a_2,\frac{a_2+b_2}{2},m-1) + N(a_1,b_1,\frac{a_2+b_2}{2},b_2,m-1) + HB(a_1,b_1,a_2,b_2) \end{cases} \tag{1}$$

As we see, there are two mutually recursive functions A and N in the specification.

Function *HB* is not recursive, and rather than defining *HB* precisely, we capture its dependencies in an expression *Expr*:

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

The form of specification (1) resembles functional specifications which are used in systolic design [1]. However, the crucial difference is that A and N in (1) are not arrays with integer coordinates but functions over tuples of reals; furthermore, the dependences between the tuples do not satisfy the restrictions which enable the use of linear algebra and linear programming for the synthesis of a parallel program.

Specification (1) has also explicit features of divide-and-conquer, but it is not a precise fit with the conventional divide-and-conquer scheme with one function defined through the combination of two recursive calls of itself. Instead of one function, as usual, we have here two functions with mutual dependences (A upon N) and, instead of two calls, we have three recursive calls in the definition of A. There is one more slight difference to the conventional case: function *HB* which is not directly used in the combination (in our case, summing up the results) and thus can be computed independently of the recursive calls.

For dealing with specifications like our example, we have considered a general class of such specifications and developed a parallel implementation schema for this class through transformations [5]. For the sake of completeness, we present here the general format of the specifications that we admit and the general idea of the derivation of the parallel schema.

3 Formal Derivation of Parallelism

In the general case, we consider a system of n functions $f = (f_1, \cdots, f_n)$ that are mutually recursive, i.e., there is one defining equation for every function of f:

$$f_i(x) \;=\; p_i(x) \;\rightarrow\; b_i(x)\,;\; E_i\,(g_i,\; f,\; x) \qquad\qquad (i=1,\cdots,n) \tag{3}$$

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 $\tau \rightarrow \sigma$. 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 $\varphi^l_{ij} : \tau \rightarrow \tau$ are called *shifts*.

Example specification (1) is a special case of (3) 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: $\varphi^1_{12} = Id$ (identity), $\varphi^1_{11}(a_1,b_1,a_2,b_2,m) = (a_1, \frac{a_1+b_1}{2}, a_2, b_2, m-1)$. There is no auxiliary non-recursive function for A; the auxiliary function for N is *HB*; *HB* is also the basic function for both A and N.

We view the system of equations (3) 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 $f_1(input)$ and, of course, all necessary values according to the dependences in (3). For our example (1), the target program must compute the value of $A(input)$, where *input* is a tuple of actual parameters (a_1, b_1, a_2, b_2, m).

Let us demonstrate how specification (3) can be parallelized using the rules of the Bird-Meertens formalism and Backus' FP.

We use the following functionals throughout the paper:

- from the Bird-Meertens formalism, we take functional *map* that applies function h to all elements of a list $[a_1, \cdots, a_n]$:

  $$map\ h\ [a_1, \cdots, a_n] = [h\, a_1, \cdots, h\, a_n]$$

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

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

- we introduce a new functional *dap* ("distributed application") which shares the features of both *map* and $<>$; *dap* applies a list of functions to a list of arguments of the same length:

  $$dap\ [h_1, \cdots, h_n]\ [a_1, \cdots, a_n] = [h_1\, a_1, \cdots, h_n\, a_n]$$

General format (3) can now be represented in a higher-order notation using introduced functions and functionals. We define functions $split_i$ which combine the shifts of the specification: $split_i = < split_{i1}, \cdots, split_{in} >$, where $split_{ij} : \tau \to list\,\tau$ yields a list containing all domain parameters with which f_j is called in the equation for f_i of (3). For our example (1), $split_{11} =< \varphi^1_{11}, \varphi^2_{11} >$. We obtain the following higher-order notation of specification (3):

$$f_i \ = \ p_i \to b_i\, ;\ E_i \circ < g_i\ ,\ dap\, (map \ \ map\ \ [f_1, \cdots, f_n]\,)\ \circ\ split_i > \quad (i\!=\!1, \cdots, n) \tag{4}$$

Here, $\circ$ denotes function composition; it has lower precedence than function application.

The derivation proceeds as follows (see [5] for details):

- A new type of so-called communication trees is introduced, which is uniquely determined by functions *split*. One communication tree for our example is shown in Figure 1. It has nodes of two types. Type 1 corresponds to function A; all nodes of this type are ternary because A has three recursive calls on the right-hand side of its equation. The nodes of type 2 correspond to function N and are binary. There are infinitely many possible communication trees for a specification, and we can choose an arbitrary finite one as a communication structure of the target parallel program, taking into account the number of available processors in the system.

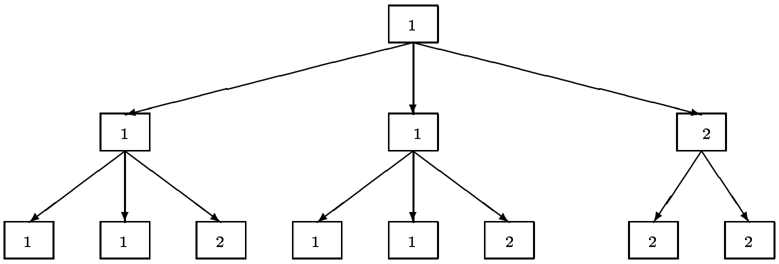

Figure 1: Communication tree – example

- Recursive functions f_i, which are defined on τ in the specification, can now be aggregated into one function F defined on the nodes of the communication tree. The expression for F is derived from higher-order representation (4) using equational properties of the Bird-Meertens formalism as transformation rules [5]:

$$F = \langle\, [\!]\, i \,::\, \mathit{is.node}_i \rightarrow \mathit{conq}_i \,\rangle \,\circ\, \mathit{div} \tag{5}$$

$$\begin{aligned} \mathit{div} = {} & \mathit{is.root} \rightarrow \mathit{input}\,;\; p \circ \mathit{div} \circ \mathit{father} \rightarrow \perp\,; \\ & \langle\, [\!]\, j :: \mathit{is.son}_j \rightarrow P_j \circ \mathit{fsplit} \circ \mathit{div} \circ \mathit{father}\,\rangle \end{aligned} \tag{6}$$

$$\mathit{conq}_i = \mathit{is.leaf} \circ \mathit{node} \rightarrow f_i;\; p_i \rightarrow b_i;\; E_i \circ {<} g_i, (\mathit{map}\, F \circ \mathit{fsons} \circ \mathit{node}) {>} \tag{7}$$

We use here the following functions and predicates defined on the nodes of the tree. Predicate $\mathit{is.node}_i$ signals that the node is of type i, $\mathit{is.son}_j$ – that it is the j-th son of some other node; the meaning of *is.root* and *is.leaf* is obvious; *fsons* yields the flattened list of all sons of the given node. Function *node* computes the node position in the tree for the given value of the domain parameter. Function *fsplit* behaves as split_i in a node of type i and then flattens the resulting list, P_j yields the j-th element of it.

In functional notation (5), function F is the composition of two functions which are called *div* for "divide" and *conq* for "conquer". We will see that these names really express the intuitive meaning of divide-and-conquer.

The important property of function F is that its value at the root of the tree is exactly the value $f_1(\mathit{input})$ which is to be computed from the initial specification. Therefore, we have reduced the problem of implementing the specification (3) to the problem of computing $F(\mathit{root})$. From (7) we see immediately that, in order to compute F at a node, we need the values of this very function F at the sons of the node. This leads to the following parallel implementation schema for the initial specification: assign processors to the nodes of the communication tree and compute function F at each node. The correctness of the implementation is guaranteed by the semantically sound way it has been obtained.

It remains now to develop an imperative program for each processor in the communication structure, including communication with other processors.

4 Derivation of the Imperative Program

In this section, we convert the functional notation (5), describing the computation of function F, into parallel imperative program with explicit message passing primitives. We express the imperative program in pseudocode with evident semantics.

We present first the correspondence between the constructs used in the functional implementation and the statements which will be used in the target program:

- *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 for a multiprocessor system. The communication tree which has been obtained in the course of the development will describe an architecture of the system. Our target program `Node`, which is presented in Figure 2, is executed at every node of the tree. One implicit parameter of program `Node` is the id of the associated processor (variable `my_id`).

- *Functions.* 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, depend on the type of node $node_i$: $split_i$, p_i, b_i, f_i, g_i and E_i. In the imperative program, they are implemented by the corresponding procedures `Split`, `P_holds`, `Compute_b`, `Compute_f`, `Compute_g` and `Compute_E`. These procedures have two parameters: the type of the node and the value of domain parameter in it. For brevity, we have not listed the procedure interfaces in our import list.

- *Input-output.* Input domain parameter *input* must be available in the root processor. We implement this using statement `READ`. Moreover, the final result in the root processor is the desired output of the whole target program; we use statement `WRITE` for output.

- *Local memory.* We consider an SPMD model; every processor has its own local memory. Whereas in the functional representation functions get parameters and transfer results implicitly among each other, in the imperative program we use variables and assignments for this purpose. The target program contains the following variables:

  – Variable `param` stores the domain parameter (result of *div*) for the current processor. Since, according to (6), the domain parameter for a node is computed by its father, we need also arrays `type[]` and `param[]` for storing corresponding values computed for the sons.

  – Function *node* is used in (7) in order to coordinate the type of result of function *div*, which is the domain parameter, with the input type of functions *is.leaf* and *sons* which is a tree node. In the imperative case, we can use the fact that for each processor holds: `node(param)=my_id` and that `my_id` is directly accessible in the processor.

- The type of the node, i, is stored in variable `type`; it is computed and transmitted together with `param`.
- Variable `res_aux` contains the result of computing the auxiliary function in the processor and array `res[]` – the results of F from its sons.
- Finally, `result` contains the value of F at the current node.

- *Communication.* Communications between processors include sending and receiving data 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. For the sake of simplicity, we consider here synchronous communication; asynchronous communication can yield better performance.

- *Conditionals.* The conditional of Backus' FP is implemented directly by an `IF-THEN-ELSE` statement; for conditionals with more than two alternatives, we use the `ELSE-IF` construct.

- *Sequential composition.* The composition of the functions in (5) is implemented by sequential composition of imperative statements.

- *Parallel composition.* There are two candidates for parallel execution in (5): *map* and $<>$. We implement *map* 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 $<>$. We use two kinds of parallel statements in the pseudocode: a fork `PARBEGIN-PAREND` and a parallel loop `FORALL`.

Function F is computed at each node of the communication tree in two steps: first *div* is applied, and then its result is passed to *conq*. Therefore, the processor program we derive consists also of two consecutive parts that correspond to these functions.

Let us demonstrate how these two functions are implemented in the target program which is shown in Figure 2.

Function *div* computes the corresponding domain parameter for a node. For the root, this parameter is *input* and, for each other node, it is determined by the result of *div* at its father. This dependence implies that 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, the parameters for all descendants of this node are undefined: no computations at the descendants are needed. In this case, the number of available processors exceeds the degree of parallelism in the specification. For the first alternative, i.e., for the root processor, the value of *div* is simply *input* which is obtainable by a `READ` statement. For all other processors, according to (6), the value of *div* is computed at the father, using function *split*. So, the current processor must receive its `type` and `param` from the father.

Let us now transform function $conq_i$ to an imperative form. It takes the domain parameter returned by *div*. Further computations depend on the type of the node

```
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,param)
    ELSE RECV (type,param) FROM Father (my_id);
   END_IF;
   IF P_holds (type,param)
    THEN result := Compute_b (type, param)
    ELSE-IF Is_leaf (my_id)
     THEN  result := Compute_f (type,param)
     ELSE
      FORALL k:=1 TO Outdegree (my_id) DO
        (type (k), param (k)) := Split (type,param,k);
        SEND (type (k), param (k)) TO Son (my_id,k)
      END_FORALL;
      PARBEGIN;
       res_aux := Compute_g (type,param);
       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;
    IF Is_root (my_id)
      THEN WRITE (result)
      ELSE SEND (result) TO Father (my_id)
    END_IF;
END_Node;
```

Figure 2: Target SPMD program

(index i) and on its position in the tree. In the leaf nodes, f_i for the domain parameter must be computed (we do not care about parallelizing it). 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.

There are three alternatives in the formula for *conq*.

The two first (in the leaf processor and if the basic predicate is true) require simple computation of the corresponding functions which are implemented imperatively as procedure calls.

The third alternative is more interesting. First, there is a part corresponding to the computation of *div*. Remember that, during the computation of *div*, each processor which has a father, receives data from it. Since communications are pairwise, for each such `RECV` statement, we must place a corresponding `SEND` statement in the father. Therefore, each processor which is a father, i.e., not a leaf of the tree, must send `param` and `type`, after they have been computed, to all sons of this processor. The computation is realized in the program by procedure `Split` which has three parameters: the type of the node, the domain parameter and the number of the son; the latter corresponds to projection function P_j of (6). The computation of the parameters and their transmission to the sons can be coded in a parallel loop.

Moreover, the third alternative includes receiving the results of computing F from all sons of the current node. Subexpression $mapF \circ fsons$ in the third alternative 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 $<>$), and thus we can use `PARBEGIN-PAREND`. After all threads in the fork have finished, we can compute E_i which is implemented, like all other computational functions, by a procedure call. Function E_i yields the result of F in the current node. Again, we coordinate the communication statements by using `SEND (result) TO Father` in all non-root processors. In the root, `result` is the output of the whole parallel program, it is given away by `WRITE`.

5 Implementation on a Transputer Network

In this section, we present experiments with different versions of the parallel program for our example specification (1)-(2).

The experiments have been carried out on a 64-nodes *Parsytec* transputer system. The nodes of the system are T805 transputers that are physically connected in a two-dimensional grid. They run the *Parix* operating system [6], which provides a set of easily usable functions for building and using communication networks matching particular problems and for defining suitable routines managing data operations. *Parix* offers different types of synchronous and asynchronous communication available. We use virtual topology libraries that relieve the programmer from having to build communication networks for some standard problems. The libraries provide calls for building grids, trees, hypercubes, etc. In addition to building a virtual topology, the library calls also optimize the mapping of the virtual links to the physical communication structure. We use *C*, extended with message passing, as our programming language; our pseudocode is easily convertible to this language.

We start with the program derived in the previous section and systematically improve it with respect to efficiency. For each version of the program we briefly explain the idea of the improvement made and provide the results of performance measurements for a fixed problem size, which is reflected by parameter m in (1), on different processor numbers. These results are presented in Figure 3 as a semi-log plot, i.e., we show the dependency of $\log_2(time)$ on processor number, where *time* is an elapsed time in seconds for the problem with the depth of recursion $m=14$.

- **Version 1**. The first version of the program is a direct specialization of the general schema in Figure 2; it works on the communication tree from Figure 1.

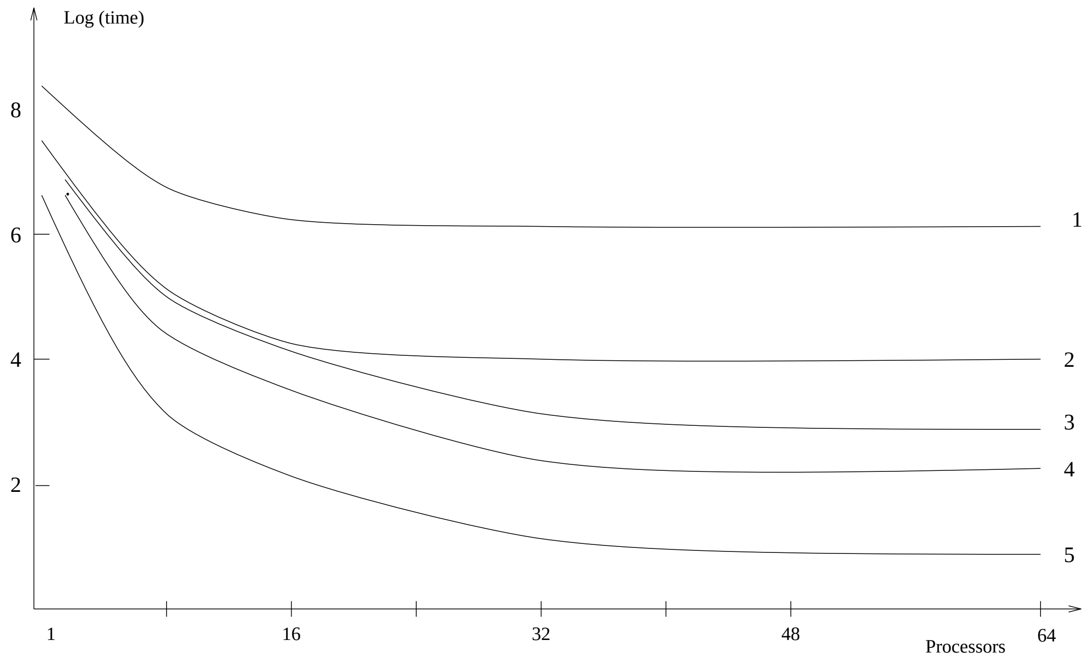

Figure 3: Experimental results

We can see that parallel speed-up increases for up to 16 processors; for larger configurations the elapsed time stabilizes.

- **Version 2**. The first improvement of the program is based on the fact that a particular specification may be matched in different ways with format (3). Another possible match for our example (1) is to view A as the single recursive and N as its auxiliary function. In this case, the target program has a binary communication tree that can be directly and efficiently implemented under *Parix* using virtual topology call `MakeTree`. This call yields the optimal embedding of the tree on the physical topology for an arbitrary processor number. The program at each node of the tree computes sequentially the corresponding recursive call of N, i.e., the granularity of parallelism increases and the load is balanced better. This version provides better performance than Version 1.

- **Version 3**. Analysis shows that the root processor is the "bottle-neck" in Version 2: its volume of computation for recursive call $N(a_1, b_1, a_2, b_2, m)$ is almost twice as high as that of the processors at the next level of the tree where N is called with the last parameter $m-1$. On the other hand, we use $p = 2^k - 1$ processors in a complete binary tree topology, whereas there are usually 2^k processors in the configuration which the user of our transputer network gets from the operating system. So, we have the following situation: there is one processor in the system which is overloaded with work, and there is one processor which does not belong to the tree and thus does no work at all (we call it the outer processor).

  The most natural solution would be to distribute the amount of work at the root between the root processor itself and the outer processor. There is only one technical problem: how do the two processors "find each other" in any arbitrary configuration? A possible solution in *Parix* is to build a virtual topology with a fixed processor as the root (e.g., we can always make the processor with logical number 0 a root), and to use a receive statement in the root processor that does not specify a particular sender. This way, the root processor "finds" the outer processor, and both execute the corresponding recursive calls of N with $m-1$ as the last parameter. After receiving the results from its sons, the root processor receives the result from the outer processor and computes the final result. For large numbers of processor, we achieve almost double the speed-up Version 2.

- **Version 4**. The next optimization of the parallel program strikes at the heart of the program's structure. We try, on the one hand, to avoid restrictions implied by the divide-and-conquer nature of the algorithm and, on the other hand, to exploit its parallelism fully. Namely, we try to save the divide phase of the algorithm by computing the domain parameters for corresponding recursive call statically in each processor itself, without communication with its father.

  For our example, as for many other divide-and-conquer algorithms, this can be realized by using two facts. Firstly, we suppose the number of processors to be a power of 2 which is very often the case. Secondly, there are exactly two recursive calls of the function itself on the right-hand side of each equation in (1). Therefore, the number of recursive calls of both A and N to be executed at each level of the tree is always divided by 2^k. For every number of processors $p=2^k$, we can find a level in the tree of recursive calls such that there are $s*p$ calls of A and $q*p$ calls of N at this level. Since all A-calls at one level are equally complex, and the same applies for the N-calls, we can distribute the load very regularly between processors: each processor has to make exactly s calls of A and q calls of N. We have derived formulae that give us explicitly the parameters of calls to be executed in each processor of the system. They are omitted here for brevity.

  In this version, we presume associativity and commutativity of addition. Since these properties are not guaranteed by the floating point arithmetic, that leads to a substantial deterioration of the accuracy of the computations. The situation can be corrected by using a sorting procedure to enforce a summation in ascending order of values. In this version, the efficiency is improved so substantially, that it makes well up for the cost of additional sorting.

- **Version 5**. The last and the most serious transformation of the program requires a total change of the very nature of the algorithm: we transform a recursive divide-and-conquer scheme into an iterative one. It is not always possible to do that (see [7] for details).

However, for our example, it can be proved by induction that the equation for $N(a_1, b_1, a_2, b_2)$ from (1) can be implemented as follows:

```
S := 0;
FOR i:= 0 TO m-1 DO
 FOR j:= 1 TO 2**i DO
   S := S + HB(a1, b1, (j*a2+(2**i-j)*b2)/2**i,
                      ((j-1)*a2+(2**i+1-j)*b2)/2**i);
 END_FOR;
END_FOR;
```

Similar transformation from recursion to iteration can be made for function A.

Beside the fact that iteration can be implemented more efficiently than recursion, there is another very important source of efficiency in this transformation. From (2) one can see that, for computing HB at different points of the domain, we often need some common values of function u. It can be shown that some values of u are recomputed nine times, either at different processors or in the same processor. The iterative form of the algorithm makes it possible to save some of these redundant computations. This allows for a substantial increase in efficiency, see Figure 3.

6 Conclusions and future work

Our paper takes the approach in which the starting point of 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.

Divide-and-conquer algorithms have long been recognized as very useful in many applications and naturally suitable for parallel implementation. Much of the recent work on them has been done in the context of higher-order functional programming. Divide-and-conquer is considered as an *algorithmic skeleton* (due to Cole [8]), the parallelism of which is to be studied. Mou and Hudak have introduced an algebraic model [9], that gave rise to the language *Divacon* [10], and formal transformations [11]. The group around Harrison transforms the non-linear divide-and-conquer recursion into tail recursion and then pipelines it [7, 12]. Axford views the divide-and-conquer paradigm as a fundamental design principle [13] and considers its parallel implementation on a shared memory architecture. Approaches for distributed memory are the design of specialized architectures, e.g., by McBurney and Sleep [14], and explicit annotations for parallelism and communication [15]. The skeletal approach and formal transformations have been used by Pepper [16] for deriving parallel networks from functional specifications.

In comparison with related work, we consider a more general case by allowing a specification to consist of several mutually recursive functions and also of non-recursive auxiliary functions. We propose a systematic, semantically sound way of deriving a distributed-memory SPMD program schema with explicit message passing. The Bird-Meertens formalism has been traditionally used for deriving new parallel algorithms that work on lists as a basic data structure. In order to admit specifications with arbitrary data types, we use lists in the formalism for representing units of computations and their interdependences rather than elementary data dependences.

Our experiments on improving the initial naive implementation have shown that due to the proper choice of the optimization method, the efficiency of divide-and-conquer can be drastically increased.

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 find transformation rules that allow the formalization of all improvements reported here and to study the influence of various transformation rules on the efficiency of the resulting parallel program.

7 Acknowledgements

I am indebted to Prof. Christian Lengauer who has actively supported this work. Thanks are due to the anonymous referees for their helpful comments.

References

[1] 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.

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

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

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

[5] S. Gorlatch and C. Lengauer. Systematic development of an SPMD implementation schema for mutually recursive divide-and-conquer specifications. In *Proc. Eighth Internat. Parallel Programming Symposium (IPPS'94)*, pages 368–375. IEEE Computer Society Press, 1994.

[6] Parix. *Release 1.2. Software Documentation.* Parsytec Computer GmbH, 1993.

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

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

[9] 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.

[10] 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.

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

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

[13] 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.

[14] D. McBurney and M. Sleep. Transputer-based experiments with the ZAPP architecture. *Lecture Notes In Computer Science*, 258:242–259, 1987.

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

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