Stages and Transformations in Parallel Programming

Sergei Gorlatch

University of Passau, D–94030 Passau, Germany.
E-mail: gorlatch@fmi.uni-passau.de

Abstract. An approach, called SAT (Stages And Transformations), is introduced to support the derivation of parallel distributed-memory programs. During the design, a program is viewed as a single thread of stages, with parallelism concentrated within stages; the target program is of the SPMD format. The design process is based on the transformation rules of the Bird-Meertens formalism of higher-order functions over lists. The approach is illustrated by three case studies which include: a systematic method of constructing list homomorphisms, a scalable, load-balanced implementation of divide-and-conquer based on a specialized topology and a formal derivation of a time and cost optimal parallel algorithm for straightforward polynomial multiplication.

1 Introduction

The main problem with parallel and distributed systems today seems not to be how to build them, but how to make them work efficiently. The enormous diversity of architectures, together with the specific problems of parallelism, not encountered in sequential computation, make the development of efficient parallel software difficult. At low levels, the design process is already heavily automated and the actual challenge is to expand the range of mechanical assistance to higher levels of the design process by introducing abstract machine and programming models.

PRAM, the widely used parallel machine model, allows to describe and theoretically compare parallel algorithms, by abstracting away synchronization, data locality, machine connectivity and communication capacity [1]. Though it is well understandable, it does not always reflects the real costs of parallelism. More practically oriented models estimate performance by describing the architecture using a small set of characteristics and imposing some discipline on the processor behaviour; they include e.g. the bulk synchronous parallelism in the BSP model [2], asynchronous activity in LogP [3] and a number of scalable operations in WPRAM [4]. Higher-level abstraction mechanisms in parallel programming include skeletons that capture either general [5, 6] or application-oriented [7] algorithmic forms, higher-order functions over structured data in the Bird-Meertens formalism (BMF) [8] and shared abstract data types (SADT) [9].

This paper addresses the process of program design for distributed-memory MIMD machines. We propose a design model, called *SAT* (*Stages And Transformations*), which offers an abstraction from the potentially very complex nature of parallelism. In SAT, a parallel program under development is viewed as a single thread of *stages*; each stage encapsulates parallelism of a possibly different structure. The development process is based on the BMF calculus and consists of semantically sound *transformations*, from a formal specification to an efficient parallel program.

We shall attempt to demonstrate in this paper, that the simple abstraction provided by SAT makes parallel programs conceptually easy to understand and reason about. We briefly report the results of three case studies of quite diverse nature: (1) a transformation of functions on lists to a homomorphic representation and thereby obtaining their standard parallel implementations "for free", (2) a derivation of a scalable and load-balanced divide-and-conquer implementation using a specialized processor topology and (3) a formal development of a time- and cost optimal parallel algorithm for straightforward polynomial multiplication from its mathematical specification.

The structure of the paper is as follows: after an overview of the model, we present our case studies. Then we summarize the experience gained in them and the open questions posed. We conclude with a discussion of the prospects of the approach and comparison to related work.

2 The SAT Model: An Overview

In this section, we describe the idea of *stages* and present a fragment of the Bird-Meertens formalism which provides the second constituent of the model – the calculus of *transformations*.

When developing programs for MIMD machines, we usually aim at a collection of communicating processes, each working on its local memory. Using the terminology of CSP or occam, such a structure can be viewed as "PAR of SEQ" (PARallel composition of SEQuential processes). This view, we call it the *target view*, suits MIMD languages and architectures well but is difficult for programming. Even if we restrict ourselves to the feasible SPMD model, which requires identical code for each processor, the behaviour of the parallel program as a whole may be very complex, mostly due to communication asynchrony.

We propose to use, beside the target view, also an orthogonal kind of structure, "SEQ of PAR", which we call the *design view*. The design view considers a parallel program as a sequence of *stages*, where each stage encapsulates parallelism of a possibly different kind. Under the design view, parallel pieces become smaller and the overall control becomes sequential, i.e. both are easier to understand and to reason about.

In Figure 1, both the target view (a) and the design view (b) consisting of two stages are illustrated for a three-processor program. Note that there are no communications between different stages (this property is called communication closeness [10]).

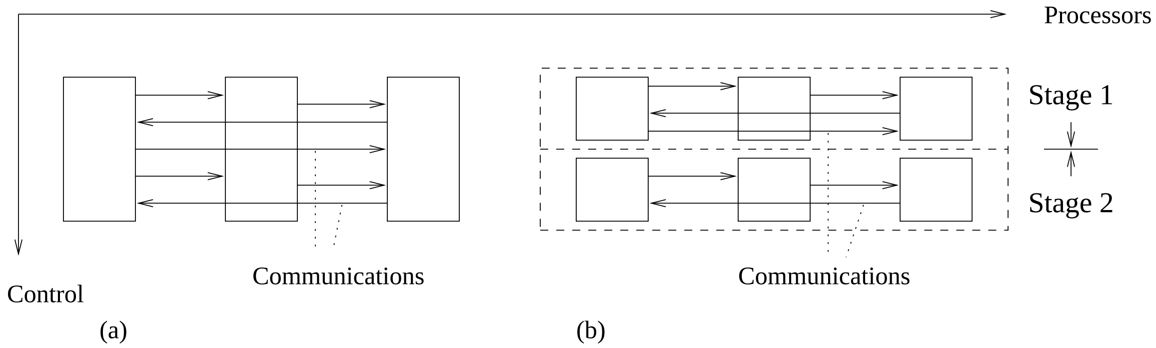

Figure 1: a) Target view and b) design view

The design view in the SAT model comprises two layers. At the top, *inter-stage* layer, stages are sequentially composed and invoked. Parallelism is embodied at the *intra-stage* layer: each stage is a parallel program, which can be a quite complex mixture of computations and communications. The structure of parallelism and communication may substantially change from stage to stage: in Figure 2, an example of a three-stage program with two different communication patterns for stages h and f is shown. Switching from one communication pattern to another often requires a redistribution of data, which is then considered a distinct stage, g.

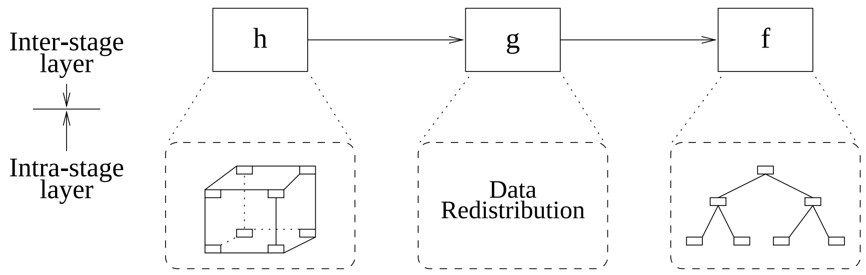

Figure 2: An example of stages and layers

The SAT approach consists of, first, specifying an algorithm in the design view in a natural and obviously correct way and, second, transforming/optimizing the specification into the provably correct target view. The intra-stage layer in the specification is usually implicitly parallel, i.e. a stage can be viewed as a higher-order skeleton [5]; in the target view, we must express parallelism explicitly.

The feasibility of the approach relies on the following arguments:

- In the design view, an algorithm is specified as if the data were globally addressable and there is a single logical execution thread. It has been demonstrated in both CSP [10] and data-parallel programming [11], that programming in such "SEQ of PAR" view is much easier than attempting to directly deal with data on many processors and with many different execution threads.

- Many practical parallel applications are naturally specifiable as a sequence of potentially parallel steps, where each step is a particular algorithm or a logically closed part thereof [12].

- Optimal parallel algorithms have been developed and their performance characteristics were formally derived using a (sequential) composition of (implicitly parallel) higher-order functions as an overall structure [8].

To use the presented model in the design process, we put it into a mathematically founded environment where different designs can be transformed and compared.

As a formal framework, we use the Bird-Meertens formalism over lists; extensions for arrays, trees, etc. also exist [8]. BMF introduces higher-order functions (functionals) over data structures; algorithms are specified as expressions of the formalism. The functionals of BMF are the source of parallelism in the model: we use them to represent the stages of the program under development. Functional composition facilitates a sequential control flow at the inter-stage level. Algebraic identities of functionals are considered as transformation rules: using equational reasoning, a specification is transformed into a form suitable for an efficient parallel implementation. The transformational style of BMF is an argument for the correctness of the design: there is no need to construct a separate proof.

The BMF-notation used in our case studies is, informally, as follows:

$[\alpha]$	the type of lists whose elements are of type α (α-lists);
$[\alpha]_k$	α-lists of length k;
$+\!\!+$	list concatenation;
$\circ$	backward functional composition;
$map\, f$	*map* of an unary function f, i.e. $map\, f\,[x_1, \cdots, x_n] \;=\; [f x_1, \cdots, f x_n]$;
$red \odot$	*reduce* over a binary associative operation $\odot \;:\; \alpha \times \alpha \;\to\; \alpha\,$, $red\; \odot\; [x_1, \cdots, x_n] \;=\; x_1 \odot x_2 \odot \cdots \odot x_n$;
$< >$	Backus' FP *construction*: $< f_1, \cdots, f_n > \; x \;=\; [f_1\, x, \cdots, f_n\, x]$;
zip	applied to a pair of lists of equal length, yields the list of pairs: $zip\; ([x_1, \cdots, x_n]\,,\, [y_1, \cdots, y_n]\,) \;=\; [\,(x_1, y_1), \cdots, (x_n, y_n)\,]$;
$zipl \odot$	"long *zip*": combines elements of two lists with operation $\odot$, returning a list as long as its longest argument, i.e.: $zipl \odot ([\,], x\,) \;=\; x$.

We enhance the original BMF a bit by allowing explicit parameters for a size of data. In addition to the usual type of lists denoted by $[\alpha]$, we consider a new type $[\alpha]_n$, where n is the length of the list. The consequence is that the result type of a functional may depend on the size of its argument, as exploited e.g. in the T-Ruby system [13]. Variable n is then used as a subscript, so we write, e.g. map_n.

3 First Case Study: Homomorphisms

We begin with a case study of a comparatively simple program structure, which, however, is quite typical in the practice of parallel programming.

3.1 Homomorphisms: A Standard Implementation

A list function h is a *homomorphism* iff there exists a binary operation $\oplus$ such that, for all lists x and y and the list concatenation $+\!\!+$, holds:

$$h\,(x +\!\!+ y) \;=\; h\,(x) \;\oplus\; h\,(y) \tag{1}$$

Intuitively this means that the value of h for a list depends in a particular way (using combining operation $\oplus$) on the values of h applied to the pieces of the list. The computations of $h(x)$ and $h(y)$ are independent and can be carried out in parallel. Equation (1) can be viewed as expressing the well-known divide-and-conquer paradigm and thus the property of a function being a homomorphism is often equal to the property of being well suitable for parallelization. Examples of homomorphisms are simple functions, such as *sum*, which, for a list of numbers, yields the sum of all its elements, and also more complicated and important functions, e.g. *scan* (prefix sums) [14].

The widely used in practice *scan*-function computes, for an associative operation $\odot$ and a list, the list of "running totals", e.g.:

$$scan\; \odot\; [a, b, c] = [a,\; (a \odot b),\; (a \odot b \odot c)]$$

According to Bird's first homomorphism theorem, every homomorphism can be uniquely represented as follows:

$$h = red\,(\oplus) \;\circ\; map\,(f),$$

where $f\,a = h\,[a]$, so we use the notation $h = hom\,(f, \oplus)$.

It can be proved that:

$$scan \odot\ = hom\,([.], \oplus),\ \text{where}\ \ u \oplus v = u\ +\!\!\!+\ (map\,(last\,(u) \odot)\ v) \tag{2}$$

An important property of $h = hom\,(f, \oplus)$ is the so-called promotion law:

$$h\ \circ\ red\,(+\!\!\!+) = red\,(\oplus)\ \circ\ map\,(h) \tag{3}$$

which describes how h can be computed over a distributed input (list of lists). The right-hand side of (3) is interpreted in the SAT model as consisting of two stages: the parallel application of function h to all sublists, followed by the combination of the results using reduction with $\oplus$. We call such a two-stage program the *standard* parallel implementation of a given homomorphism; this implementation is immediately available for every homomorphic function as soon as we know the combining operation $\oplus$.

To illustrate some aspects of the design process within the SAT model, let us outline how the standard two-stage design view is transformed into an efficient target view:

- Each stage can be analyzed and transformed separately. For the *map* stage, we can directly develop an imperative implementation in form of a simple SPMD program, where each processor computes h sequentially; there is no communication between processors.

- Stages can be split into smaller stages: e.g., the reduce stage for the *scan* function is usually transformed into a composition of so-called up-sweep and down-sweep stages [15] or a shift stage is introduced [16].

- In order to get the optimal parallel time complexity, which in case of *scan* is logarithmic, the reduce stage is designed so, that the output list is distributed over processors. This property is formulated as the *postcondition* of the design, with the concluding "flattening" stage left unimplemented. An algorithm producing a monolithic result list would require at least linear time if communication costs are accounted for.

- A particular imperative realization of the reduce stage of a homomorphism depends on the available programming platform: it may suffice to use `MPI-Reduce`, the global communication routine in the MPI standard, or it may be necessary to use explicit, well-placed pairwise communications. The latter is more cumbersome but can also be done systematically as demonstrated in [17].

- After both stages are transformed into an imperative form with explicit parallelism, we must transform their sequential composition into a monolithic SPMD program. The semantics of functional composition prescribes a global synchronization between stages: all processors must finish the current stage before the next stage can be started. Due to the communication closeness property of the stages, we can safely compose them: the design view of the form $(P1||P2); (S1||S2)$ is semantically equivalent to the target view $(P1; S1)||(P2; S2)$ if stages $P1||P2$ and $S1||S2$ are communication closed [10]. This prevents unnecessary oversynchronization. Asynchronous behaviour of the SPMD target program may therefore lead to the situation where different processors execute different stages at the same time.

The standard homomorphic implementation is not always optimal [8] but often yields a scaled speed-up [18].

3.2 Constructing Homomorphisms

The challenge is often not only how to implement a homomorphic function efficiently, but also how to construct the homomorphism, i.e. for a given function h, how to find the corresponding factorizing function f and combining operation $\oplus$ of (3). This is simple for functions like *sum*, but already for the *scan* function the construction of $\oplus$ in the form (2) is not obvious; some published *scan* algorithms contain errors, as pointed out in [15].

In [19], we prove a modification of the third homomorphism theorem [20]. This modification provides a simple method of constructing homomorphisms based on two well-known (and inherently sequential) representations of list functions: over *cons*- and *snoc*-lists. We use the symbol $\cdot\!:$ for *cons*, which attaches an element at the front of the list, and $:\!\cdot$ for *snoc*, which attaches the element at the list's end.

The method requires the user to provide both the *cons* and the *snoc* definition of the function in question. These definitions of function h should be then transformed into a *matching* format: $h(a \cdot\!: x) = h([a]) \oplus h(x)$ and $h(x :\!\cdot a) = h(x) \oplus h([a])$, with the same associative $\oplus$. If the transformation succeeds, then $\oplus$ is the operator we are looking for.

For the *scan* function, two sequential functional definitions are obvious:

$$\begin{aligned}
\mathit{scan} \odot\ (a \cdot\!: x) &= a \cdot\!:\ (\mathit{map}\,(a\,\odot)\,(\mathit{scan} \odot\ x)) \\
\mathit{scan} \odot\ (x :\!\cdot a) &= (\mathit{scan} \odot\ x)\ :\!\cdot\ (\mathit{last}\,(\mathit{scan} \odot\ x) \odot a)
\end{aligned}$$

Using the following equalities as rewrite rules for arbitrary elements a, b and list y:

$$\begin{aligned}
a &= \mathit{last}\,[a] \\
[a] &= \mathit{scan} \odot\ [a] \\
a \cdot\!:\ y &= [a] +\!\!+\ y \\
y :\!\cdot\ a &= y +\!\!+ [a] \\
[(b \odot a)] &= \mathit{map}\,(b\odot)\,[a]
\end{aligned}$$

the sequential definitions are transformed into the matched representations:

$$\begin{aligned}
[a] &\ +\!\!+\ (\mathit{map}\,(\mathit{last}\ \ ([a]) \odot)\ \ (\mathit{scan} \odot x)) \\
(\mathit{scan} \odot x) &\ +\!\!+\ (\mathit{map}\,(\mathit{last}\ (\mathit{scan} \odot x) \odot)\ \ [a])
\end{aligned}$$

which are obviously the instantiations of the expression $u +\!\!+ (\mathit{map}(\mathit{last}\,(u) \odot)\ v)$, where u is substituted by $[a]$ and v by $(\mathit{scan} \odot x)$ in the *cons* definition and the other way round in the *snoc* definition. This yields the required associative operator $\oplus$ of (2).

The next problem is what to do if a function is not a homomorphism. An idea of Cole [21] is to find, for a non-homomorphic function h, a collection of auxiliary functions $h_1, \cdots, h_k$, such that tuple $[h, h_1, \cdots, h_n]$ is a homomorphism. The program for computing h consists then of three stages: two stages implement the tuple homomorphism, the third stage applies a projection function, say *fst*, which obviously does not change the overall performance of the algorithm.

The required auxiliary functions can be constructed systematically using the described method of matching *cons* and *snoc* representations. We derived cost-optimal algorithms for the one-dimensional maximum segment sum problem and parsing of input-driven languages [19] and thus proposed an answer to the problem posed by Cole of finding auxiliary functions in a systematic way, rather than based on intuition.

4 Second Case Study: $\mathcal{N}$-graphs

The *map* stage of the standard homomorphic implementation assumes that the corresponding list is distributed between processors, which makes parallel computation possible. If, however, the list is monolithic, an additional stage, usually called divide, is necessary. Our next case study considers the traditional divide-and-conquer schema.

This schema can be specified in BMF using additionally the conditional and the construction functionals from the Backus' FP:

$$f \;=\; p \rightarrow b \;;\; E \,\circ\, (map\, f) \circ \;< \varphi_1, \varphi_2 > \tag{4}$$

Here, function f is defined recursively: in the base case, when the condition p is satisfied, the base function b is computed; otherwise, two new arguments are computed by functions φ_1 and φ_2, function f is evaluated for them and the results are combined by function E. A simple example of such a schema is the mergesort algorithm.

The typical communication pattern of divide-and-conquer is a complete binary tree. Trees and other topologies can be viewed as index domains, with functions (communication operators) and predicates defined on the nodes. A mapping from the index domain into problem related values determines how the data are distributed between the nodes (this is known as a data field in the Crystal system [22] or as an abstraction function used in [23, 17]). The program which implements the specification (4) can be presented in the SAT model as consisting of three stages which we call divide, compute and combine (see the top of Figure 4):

$$tree.program \;=\; tree.bottom.up\,(E) \,\circ\, (map\, f.leaves) \,\circ\, tree.top.down\,(<\varphi_1, \varphi_2>)$$

At the compute stage, the *map* functional of BMF is used, but now it is defined not on the abstract type of lists but on the concrete type of complete binary trees of an arbitrary but fixed size. Function *f.leaves* is an implementation of f which effectively performs computations only in the leaf processors. The divide and combine stage consist of either top-down or bottom-up traversal of the tree, with corresponding computations on the way; in [17], these traversals are derived formally.

In practice, if the processor number is fixed, the base case is not reached during the divide process, and the leaf processors may have to perform comparatively coarse-grained computations. The idleness of the non-leaf processors in the complete binary tree is therefore a serious source of inefficiency. Alternative topologies with a better load balance, like the binomial tree [18] or hypercube, have another drawback: the growing number of links per node prevents scalability.

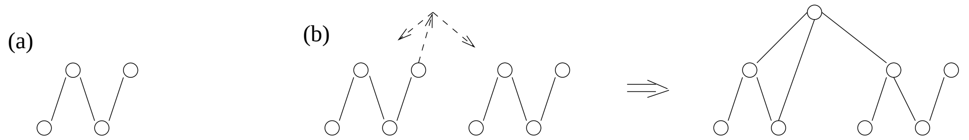

Figure 3: $\mathcal{N}$-graph : (a) the base case, (b) the construction

We introduce a new virtual topology for divide-and-conquer, the $\mathcal{N}$-graph [24], with the intention to improve the load balance of the compute stage, preserving at the same time the scalability. Figure 3 shows how this topology is constructed, starting with the base case of 4 nodes. Compared to the complete binary tree of the same depth, the $\mathcal{N}$-graph has one extra node and some extra edges, but there are not more than 4 links per node, so it is scalable.

In the SAT model, the compute stage, effectively performed at the leaves of the tree, is transformed into the following three stages:

$$map\, f.leaves \;=\; graph.down\,(E) \;\circ\; (map\, f.nodes) \;\circ\; graph.up\,(<\varphi_1, \varphi_2>)$$

These stages work on the $\mathcal{N}$-graph as follows (see Figure 4):

(a) *graph.up*: in the leaves, tasks are divided one more time, each leaf keeps one of subtasks for itself and sends the other one to its partner along the solid arrows;

(b) *map f.nodes*: each node executes its subtask (computes f) sequentially; nodes which perform computations are shown shaded: the load is balanced now;

(c) *graph.down*: results of the computations are sent back.

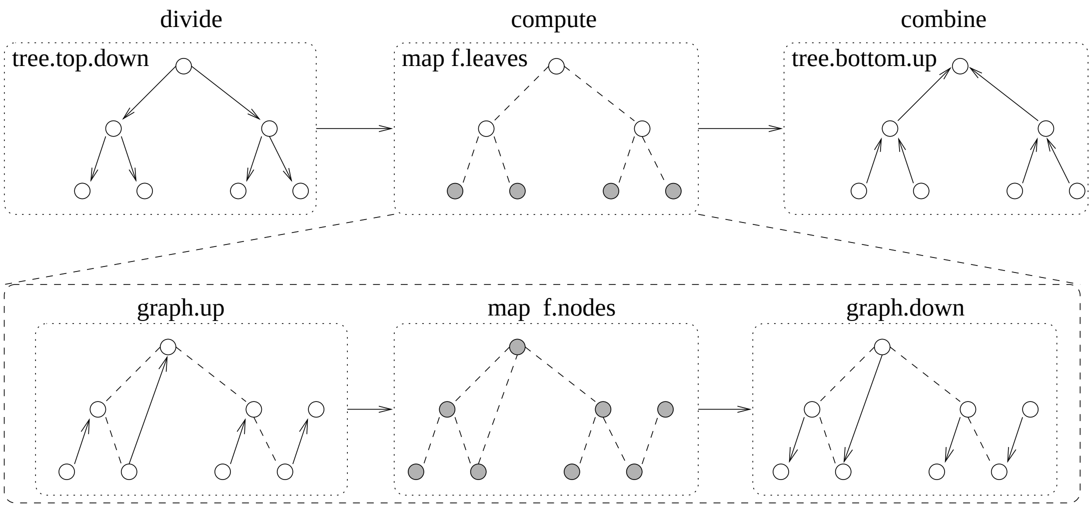

Figure 4: Transformation of the compute stage

The program under development consists now of five stages. Further improvements are made by merging two first and two last stages: this transformation for the combine stages is shown in Figure 5; nodes which are combining results are shaded. This transformation, expressed as: $graph.down\,(E) \circ tree.bottom.up\,(E) \;=\; graph.bottom.up\,(E)$, saves one communication of each leaf node with its father.

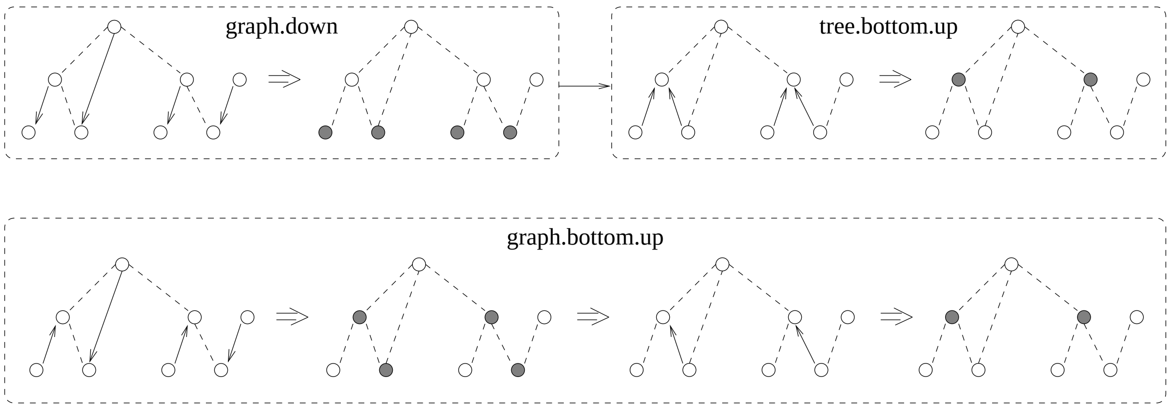

Figure 5: Collecting and combining results: before and after optimization.

On the way to an imperative SPMD implementation, collective communications in the stages can be expressed in the target view, using corresponding functions and predicates of the index domain.

The processor program of the divide stage contains, e.g., local statements like:

```
if (non.root)
 then RECV x (father);
end-if;
x1 = Fi1 (x); x2 = Fi2 (x);
if (non.leaf)
 then (SEND x1 (left.son); SEND x2 (right.son));
end-if;
```

where `Fi1, Fi2` are implementations of functions φ_1, φ_2 respectively; predicates and functions `non.root, father`, etc. are defined on the index domain of binary trees.

The communication closeness of the imperative implementation guarantees that the stages can be safely composed. Note that the stages may work asynchronously in the target program: during the bottom-up traversal, some leaves may become free and proceed to the next stage of the program if any, whereas the root may still be busy with the compute stage.

Experiments with the mergesort on a transputer network [24] show an almost two-fold efficiency improvement over the complete binary tree topology; this was to be expected because twice as many processors are exploited at the compute stage.

5 Third Case Study: Polynomial Multiplication

In the previous case studies, the inter-stage structure is standard (two stages for homomorphisms or three stages for divide-and-conquer), and the problem is either to instantiate the stages or to optimize them. Our final case study demonstrates, how stages arise in the development process, which starts from a mathematical specification and targets at a parallel program. The example, straightforward polynomial multiplication, is popular in the systolic community where the polytope method is used [25].

Let $A(z)$ and $B(z)$ be two polynomials of degree $n-1$, in Dijkstra's notation:

$$A(z) \;=\; \langle\, \Sigma\, k \;:\; 0 \le k \le n-1 \;:\; a_k z^k \,\rangle$$

$$B(z) \;=\; \langle\, \Sigma\, k \;:\; 0 \le k \le n-1 \;:\; b_k z^k \,\rangle$$

Desired is the polynomial $C(z)$ of degree $2(n-1)$ defined by:

$$C(z) = \langle\, \Sigma\, k \;:\; 0 \le k \le 2(n-1) \;:\; c_k z^k \,\rangle, \quad \text{where:}$$

$$\langle\, \forall\, k \;:\; 0 \le k \le 2(n-1) \;:\; c_k = \langle\, \Sigma\, i,j \;:\; 0 \le i,j \le n-1 \wedge i+j=k \;:\; a_i * b_j \,\rangle\,\rangle$$

We use $\otimes$ to denote polynomial multiplication, i.e., $C = A \otimes B$. Polynomial coefficients are represented by lists, two-dimensional structures will be modeled by lists of lists.

The development process is a sequence of *design decisions* which are made based on formal transformation, type analysis and performance evaluation.

The first design decision concerns the granularity: for a problem of size n, the input lists are partitioned in p parts; both n and p are parameters. Thus, granularity is treated explicitly by parameters; for simplicity, we assume that p divides n, such that $n = p * k$. This partitioning is realized by function $distr_{np} : [\alpha]_n \to [[\alpha]_k]_p$.

We introduce (for brevity, informally) a new operation, $shift_k$, on lists of lists, which moves elements of the inner lists to the right by inserting $0,\ k,\ 2k,\ \cdots$ neutral elements at the beginning of the first, second, third, $\cdots$ inner lists correspondingly, e.g.:

$$shift_k\,[\,[a_1,\cdots,a_n],[b_1,\cdots,b_n],[c_1,\cdots,c_n]\,] =$$
$$[\,[a_1,\cdots,a_n],[\underbrace{0,\cdots,0}_{k},b_1,\cdots,b_n],[\underbrace{0,\cdots,0}_{2k},c_1,\cdots,c_n]\,]$$

The properties of $\otimes$ when one of lists is partitioned can be formulated in BMF:

$$\otimes\,(a,b) \;=\; (\,red_p\,(zipl+)\circ shift_k\circ map_p\,(\otimes\, b)\circ distr_{np}\,)\;a \tag{5}$$
$$\otimes\,(a,b) \;=\; (\,red_p\,(zipl+)\circ shift_k\circ map_p\,(a\,\otimes)\circ distr_{np})\;b \tag{6}$$

Using these properties, the original expression $\otimes\,(a,b)$ can be parallelized by transformation, see [26] for details. We use $\overset{n}{\otimes}_{pk}$ to denote the parallel version of $\otimes$ which multiplies n-polynomials using the (n-p-k)-partition, and $\overset{k}{\otimes}$ for a (sequential) multiplication of k-polynomials.

The transformed expression for $\overset{n}{\otimes}_{pk}$ reads as follows:

$$\overset{n}{\otimes}_{pk} = combine\circ compute\circ distribute \tag{7}$$

where

$$\begin{aligned} distribute &= map_p\,(zip_p)\circ zip_p\,\circ \\ &\quad <copy_p\circ distr_{np}\circ fst,\ map\,(copy_p)\circ distr_{np}\circ snd> \end{aligned} \tag{8}$$
$$compute = map_p(map_p\,(\overset{k}{\otimes})) \tag{9}$$
$$combine = red_p\,(zipl+)\circ shift_k\circ map_p\,(red_p\,(zipl+)\circ shift_k\,) \tag{10}$$

where fst and snd yield the first and the second component of a pair, respectively.

We have obtained three stages. The second design decision fixes the inter-stage layer of our parallel design (see Figure 6).

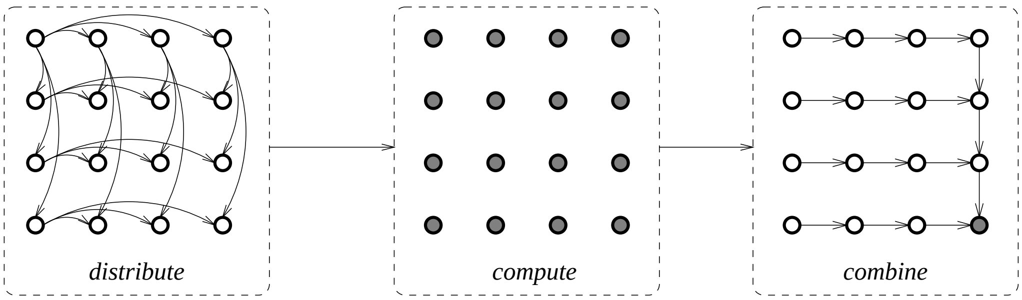

Figure 6: Three stages of the program: intermediate version

Now we can analyze and optimize each stage separately.

The second stage, *compute*, is obviously the simplest with respect to parallelization. The semantics of *map* tells immediately that we have p^2 potentially parallel processes, each multiplying two polynomials of length k.

Consequently, the next design decision is to use p^2 processors to implement our program. The processor number remains parameterized: we can choose p arbitrarily, the only restriction being the assumption that p divides n, which can be rectified easily. For convenience, we consider a two-dimensional matrix of processors but do not make any assumption about their interconnection yet.

The *distribute* stage can be split into two: first, partitioning each of input lists in p segments, placing the segments of a in the processors of the first row and b in the first column, and second, broadcasting along the columns and along the rows of the processor matrix. Both are examples of a data redistribution stage, without computation.

The partitioning stage can be implemented in one of two ways: (1) pipelining requires p steps, with time $O(n/p)$ at each step; (2) using a tree structure we need $\log p$ steps with the varying length of lists to be transmitted at each step: $n/2$, $n/4$, $\cdots$, n/p. Both implementations have time complexity $O(n)$. To achieve logarithmic time complexity, the design decision is made to separate this stage and leave it unimplemented. The result of this stage, the input lists, partitioned and placed in the first row and the first column of the processor matrix, is now required as the input of the program. This requirement constitutes the *precondition* of the design in analogy to the introduction of a postcondition in Section 3.1. Note that assumptions about distributed input and output are common also in systolic algorithms which, however, achieve only linear time complexity.

The logarithmic time for the broadcasting stage is provided by the design decision to connect processors in the mesh of trees topology [27].

The *combine* stage is viewed as consisting of two stages: $\mathit{combine} = \mathit{comb}_2 \circ \mathit{comb}_1$. The first performs reductions in parallel along the rows of the matrix and the second reduces these partial results to the final value. Both induce a linear time complexity because of communications. In contrast to our previous practice, we transform their composition as a whole into a composition of two new stages:

$$\mathit{comb}_2 \circ \mathit{comb}_1 \;=\; \mathit{combine}_2 \circ \mathit{combine}_1$$

where

$$\begin{aligned}
\mathit{combine}_1 &= \mathit{red}\,(\mathit{zipl}\,(\mathit{zipl}\,+)) \circ \mathit{shift}_1 \\
\mathit{combine}_2 &= \mathit{red}\,(\mathit{zipl}+) \circ \mathit{shift}_k
\end{aligned}$$

Stage $\mathit{combine}_1$ can be implemented in logarithmic time by making the design decision to enrich the processor topology: we arrive at the mesh of trees with diagonal trees, which has been known to be useful for other applications [27]. Stage $\mathit{combine}_2$ is split further: its first part requires constant time but leaves the result distributed; according to the already described practice we formulate this as the postcondition of the design.

The design process described so far yields a target view implementation, which is time-optimal (logarithmic) on n^2 processors; thus the cost (time-processor product) is worse than in the sequential case. Aiming at both time and cost optimality, we make our final design decision: both broadcasting in the distribute stage and gathering in the combine stage are organized with pipelining.

The complexity estimate of the parallel target program is then as follows:

$$t = O\left(\frac{n \cdot \log p}{p \cdot m(n,p)} + (n/p)^2\right)$$

where $m(n,p) = \mathit{min}\,\{n/p, (\log p + 1)\}$ is the pipeline length. The value of p can be chosen between 1 and n. If $p = \left(n/\sqrt{\log n}\right)$, then $m(n,p) = \sqrt{\log p}$, which yields $t = O\,(\log n)$.

We have the logarithmic time complexity which is achieved on $p^2 = (n^2/\log n)$ processors. Therefore, our parallel implementation of the straightforward polynomial multiplication on $(n^2/\log n)$ processors is both *time and cost optimal.*

The following cases are also of interest:

- $p = 1$: we get $t = O(n^2)$, so we have not worsened the sequential situation;
- $p = n$: $t = O(\log n)$ on n^2 processors: time- but not cost optimal;
- $p = n/\log n$: cost optimal with $t = O\left(\log^2 n\right)$ on $(n/\log n)^2$ processors;
- $p = \sqrt{n}$: $t = O(n)$ on n processors: cost optimal and equal to the performance of the systolic solution [25].

6 SAT Design: A Summary

Our case studies demonstrate the potential of the SAT (Stages And Transformations) approach for parallel program design. In this section, we briefly summarize the design process and discuss some open questions.

• *The designed program* is viewed as a sequence of stages, with parallelism incorporated within each stage. The number of stages in our case studies varies from two to six.

• *Stages* either are provided by a standard implementation of a class of algorithms, like in case of homomorphisms, or arise as the result of transformations as in the polynomial multiplication example. They can also be specified by the designer as a natural representation of an algorithm consisting of logically closed parts. All three case studies demonstrate how stages can be split, composed and transformed in the design process.

• *The inter-stage layer* in the design view requires mechanisms of stage invocation and sequential composition which are expressed in BMF by functional application and composition. Although functional composition is a universal mechanism well suited for transformations, its exclusive use for expressing sequential control flow may sometimes lead to quite awkward expressions (many examples exist, e.g., in the Backus' FP). We view this as a serious impediment for the practical use of a functional approach which has to be addressed in the future.

• *The development process* proceeds completely in the design view: it is more suitable for transformations and is easier to comprehend. Note that, even if we have developed an explicit (imperative) implementation of a stage, the design view, i.e. the "SEQ of PAR" structure, is maintained. At the final step of development, all stages are simply "glued" together to form a single SPMD program, without any additional global synchronization. The correctness of the result program due to the communication closeness property has been discussed in Section 3.1.

• *A stage program* is initially an implicitly parallel expression over the functionals of BMF whose order of evaluation is not specified. Explicit parallelism at the intra-stage layer is obtained either as some standard skeleton implementation as in case of homomorphisms, or it is derived directly from the implicit representation together with the corresponding virtual processor topology. Whereas the choice of a suitable topology is a performance-driven decision, usually made by the designer (the $\mathcal{N}$-graph or the mesh of trees with diagonals are examples), the individual node program can be obtained by transformation as in the $\mathcal{N}$-graph example, which guarantees its correctness.

• *Unimplemented stages* : we can separate the first and/or the last stage of a program and develop an implementation of the remaining program. The input of the residual program is then required to be provided by the environment; we call this requirement the *precondition* of the design. Symmetrically, the output of the program becomes the input of the unimplemented last stage; we call this property of the designed program the *postcondition*. In our case studies, pre- and postconditions express assumptions about initial and target data distributions.

• *The processor number* is assumed to be arbitrary but fixed; it is usually an explicit parameter of the design. Contrary to the approaches which specify the maximal parallelism and then rely on the run-time system in mapping it to the available processors, our examples show that this can be done efficiently at higher design levels.

• *Granularity* is controlled explicitly by parameterizing the design in both the problem size and the processor number. It has been reported repeatedly that this ability is critical to high performance and portability. Our case studies exploit it for making design decisions and predicting performance.

• *Cost estimation* is realized using granularity information in the BMF-expressions: we can estimate both the left-hand and the right-hand side of a transformation rule and thus have an understanding of possible (dis)advantages of applying the rule. In the polynomial multiplication case study, we made estimations based on the approach of [16], accounting for both computations and inter-processor communications depending on the data size. Bridging models [2, 3], which take into account the latency and the bandwidth of the communication network, could be also used.

• *Portability* is ensured in that the global communication mechanism is kept transparent to the designer, relieving her/him from explicitly designing communication schemes of the algorithms. However, sometimes the explicit treatment of communication issues leads to more efficient solutions, as the $\mathcal{N}$-graph and the polynomial multiplication case studies demonstrate. In the process of stage transformation we usually aim at a uniform pattern of communication for each stage, which simplifies the analysis.

• *Transformations* make the design more secure by placing the design decisions on a formal basis of BMF. Although some experience with transformations, useful for parallelization, has been accumulated, a self-contained system of transformations and strategies of their application remains an area of future research.

7 Conclusion and Related Work

The SAT approach aims ultimately at a structured methodology for developing correct and efficient parallel programs. Stage abstraction prescribes a single-threaded overall structure of the program under development, which simplifies its comprehensibility. It allows also to develop and evaluate the performance of each stage in isolation, and yet to compose the target program without any additional global synchronization. Design decisions are made based on a type analysis and performance considerations, rather than *ad hoc*. The correctness of the target program is substantiated by the semantical soundness of transformations in the Bird-Meertens formalism. The endeavor for efficiency is reflected in the careful estimation of each transformation step regarding its influence on the performance.

Our case studies account for the communication costs and yield either asymptotic time and cost optimality or constant time improvements confirmed by machine tests.

Our approach extends the idea of communication closed layers in imperative programming [10, 11] by putting the stage abstraction into the transformational calculus of BMF, which makes stages themselves and their interface an object of design and ensures the design correctness.

Our work is inspired and heavily influenced by the seminal research on applying BMF to parallel computing [8]. The SAT model combines the BMF transformation calculus with the practice of parallel program development by addressing the decision making process and the problems of granularity, scalability, etc.

We see our approach as complementary to related work on skeletons [5, 6] and their performance estimation [4, 7]. Stages or compositions thereof may be sometimes immediately expressed by means of skeletons; this can also be a goal at the final steps of the design process, when some standard presentation of the found efficient solution is desirable. As our case studies demonstrate, the particular design goals may require to differentiate smaller entities which are easier to analyze separately, as e.g. divide, compute and combine stages instead of a monolithic divide-and-conquer skeleton.

The long-term aim of the SAT approach is to find a suitable compromise between abstraction and efficiency, between formality and established practice in parallel programming, in order to arrive at programs which are more portable and reliable but whose efficiency is competitive with hand-written, target-specific programs. Our case studies can be viewed as a preliminary justification of the approach, but much work still remains to be done both in the theory of a semantically sound transition from the design to the target view and in the practice of performance estimation.

8 Acknowledgments

Thanks to Murray Cole, Chris Lengauer and anonymous referees for valuable comments.

This work profited from the exchange grants under an ARC project from the DAAD and under the INTAS project "Efficient Symbolic Computations".

References

[1] U. Vishkin. A case for the PRAM as a standard programmer's model. In Meyer auf der Heide et al., editors, *Parallel architectures and their efficient use*, Lecture Notes In Computer Science 678, pages 11–19. 1993.

[2] W. McColl. Scalable computing. In J. van Leeuwen, editor, *Computer Science Today*, Lecture Notes In Computer Science 1000, pages 46–61. 1995.

[3] D. Culler, R. Karp, D. Patterson, et al. LogP: towards a realistic model of parallel computation. In *Proceedings of the 4th ACM PPOPP*, pages 1–12, 1993.

[4] J. Nash, M. Dyer, and P. Dew. Designing practical parallel algorithms for scalable message passing machines. In B. Cook et al., editors, *Transputer Applications and Systems'95*, pages 529–541. IOS Press, 1995.

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

[6] J. Darlington et al. Parallel programming using skeleton functions. In A. Bode, M. Reeve, and G. Wolf, editors, *Parallel Architectures and Languages Europe, PARLE '93*, Lecture Notes in Computer Science 694, pages 146–160. 1993.

[7] H. Deldarie, J. Davy, and P. Dew. The performance of parallel algorithmic skeletons. Technical Report 95.6, University of Leeds, March 1995.

[8] D. Skillicorn. *Foundations of Parallel Programming*. Cambridge University Press, 1994.

[9] D. Goodeve, J. Davy, and C. Wadsworth. Shared accumulators. In B. Cook et al., editors, *Transputer Applications and Systems'95*, pages 518–528. IOS Press, 1995.

[10] T. Elrad and N. Francez. Decomposition of distributed programs into communication-closed layers. *Science of Computer Programming*, 2:155–173, 1982.

[11] L. Bouge. The data-parallel programming model: a semantic perspective. Research Report 92-45, Ecole Normale Superieure de Lyon, 1992.

[12] L. Snyder. A practical parallel programming model. DIMACS Series in Discrete Math. and Theor. Comp. Sci., Vol. 18, pages 143–160. 1994.

[13] R. Sharp and O. Rasmussen. Using a language of functions and relations for VLSI specification. In *Proceedings of FPCA'95*, pages 45–54, La Jolla, CA, USA, 1995.

[14] G. Blelloch. Scans as primitive parallel operations. *IEEE Trans. on Computers*, 38(11):1526–1538, November 1989.

[15] J. O'Donnell. A correctness proof of parallel scan. *Parallel Processing Letters*, 4(3):329–338, 1994.

[16] D. Skillicorn and W. Cai. A cost calculus for parallel functional programming. *Journal of Parallel and Distributed Computing*, 28:65–83, 1995.

[17] S. Gorlatch and C. Lengauer. Parallelization of divide-and-conquer in the Bird-Meertens formalism. *Formal Aspects of Computing*, 7(6):663–682, 1995.

[18] M. J. Quinn. *Parallel Computing.* McGraw-Hill, Inc., 1994.

[19] S. Gorlatch. Constructing list homomorphisms. Technical Report MIP-9512, Universität Passau, 1995.

[20] J. Gibbons. The third homomorphism theorem. *J. Fun. Programming.* To appear.

[21] M. Cole. Parallel programming with list homomorphisms. *Parallel Processing Letters*, 5(2):191–204, 1994.

[22] M. Chen, Y. Choo, and J. Li. Crystal: theory and pragmatics of generating efficient parallel code. In B. Szymanski, editor, *Parallel Functional Languages and Compilers*, chapter 7. ACM Press, 1991.

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

[24] S. Gorlatch and C. Lengauer. N-graphs: A topology for parallel divide-and-conquer on transputer networks. In B. Cook et al., editors, *Transputer Applications and Systems'95*, pages 394–409. IOS Press, 1995.

[25] 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.

[26] S. Gorlatch. From transformations to methodology in parallel program development: A case study. Technical Report MIP-9508, Universität Passau, May 1995.

[27] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann Publ., 1992.