Acta Informatica 36, 761–803 (2000)

Abstraction and performance in the design of parallel programs: an overview of the SAT approach

Sergei Gorlatch, Christian Lengauer

Fakultät für Mathematik und Informatik, Universität Passau, 94030 Passau, Germany
(e-mail: {gorlatch,lengauer}@fmi.uni-passau.de)

Received: 6 February 1998 / 2 August 1999

Abstract. SAT stands for *Stages And Tranformations* and is the name of an approach to the high-level, performance-directed design of parallel programs. The target programs obtained with this approach are sequences of internally parallel *stages*, i.e., they fall within the SPMD model. Formal program *transformations* are used for deriving each parallel stage and optimizing the combination of several stages.

The main advantage of the SAT approach is the comparatively high level of abstraction at which performance-relevant design decisions can be made. One consequence is that choices become much clearer than at the code level, enhancing comparability; another is that there is an increased potential for automation of the programming process.

This paper summarizes the approach and assesses its usefulness, with emphasis on one basic parallel programming pattern, the list homomorphism, which captures the divide-and-conquer paradigm. We survey the transformational theory, provide a range of practical examples, and discuss the potential for automation and also the demands made on application programmers and implementers.

1 Introduction

Parallel machines, consisting of hundreds or thousands processors, offer great computational power for complex applications. The community of their potential users is growing tremendously with the emergence of global networks that bring hardware, software and expertise from geographically dispersed sources to bear on large scale problems. However, parallel computing still has not become a routine way of solving problems faster, but rather remains the domain of researchers and highly experienced practitioners.

It is the lack of good software that is holding back the proliferation of parallel computers. Advances in parallel programming technology have not kept pace with leaps in the complexity of applications.

Performance is the foremost goal of parallelism and, at the same time, a major source of its difficulties. To design an efficient parallel solution of a problem, the programmer must decompose the problem into a collection of processes that can run simultaneously, map the processes to the available processors, synchronize the processors, organize their communication, etc. To cope with these low-level, machine-specific details, a rich body of parallel algorithms, languages and implementation techniques has been developed for various parallel architectures. However, parallel programming still requires specific expertise from the user each time one moves to a new parallel machine or even just changes the configuration of a machine.

Abstraction, i.e., hiding low-level details, has been the major way of coping with the software crisis for sequential computers. In the parallel setting, the quest for abstraction is traditionally even stronger, due to the fundamental complexity of the objects studied. While abstraction is wanted badly by the application programmers, implementers of parallel software have often felt that it conflicts with performance.

In the last decade, parallel programming has made progress towards closing the gap between the requirements of performance and abstraction.

On the performance side, there is a trend towards parallel programs which are better structured, more predictable and more portable: (1) program libraries standardize the set of parallel primitives to achieve portability; (2) research in parallel algorithms is increasingly aiming at well-structured schemas for classes of problems, rather than at individual solutions; (3) the predictability of parallel program behaviour on different machines is often viewed as more important than just raw performance.

On the abstraction side, the trend is towards identifying higher-level constructs that enable formal reasoning about parallelism: (1) functional programming calculi offer the power of higher-order functions for specifying parallel algorithms and of equational rewriting rules for algorithm transformation; (2) these calculi are gradually being equipped with methods for predicting the target performance; (3) new, so-called “bridging” models of parallelism abstract from the architectural details of parallel machines.

Our approach, called *SAT (Stages And Transformations)*, builds on this development and seeks to combine the goals of abstraction and performance in the design of parallel programs in a systematic way:

Abstraction is achieved by basing the design process on higher-order functional programming, which facilitates the formal description of and reasoning about parallelism.

Performance is addressed by directing the design process towards programs which have an efficient implementation on different parallel architectures.

Reconciliation of abstraction and performance is achieved by using a common framework for dealing with both:

- Abstract specifications and target programs have a common control structure – a sequential composition of parallel *stages*.
- The parallelism in a stage is captured by the commonly occurring pattern of a *homomorphism*, which plays a key rôle in the extraction of abstract parallelism and in its efficient implementation.
- The design process is based on formal *transformations*, which guarantee the correctness of the transition from abstract specifications to efficient parallel implementations and influence target performance in a predictable way.

The main purpose of this paper is to summarize the results obtained within the SAT approach and published in [28],[32]–[43], place our work in the contemporary research terrain of parallel processing, discuss the lessons learned, and outline our vision of the future development in the field. For much technical detail – in particular, for proofs – we refer to said papers.

The power of the approach is demonstrated in several case studies, which have been suggested in the literature as testbeds for program design methods: scan (parallel prefix), the maximum segment sum (MSS), polynomial multiplication, the Fast Fourier Transform (FFT) and numerical integration on sparse grids. We either give new parallel solutions or, more often, demonstrate a systematic way of arriving at the optimal solutions which were obtained *ad hoc* previously.

The paper is structured as follows:

Section 2 outlines the SAT approach and related work. We describe the class of target programs, the formalism and the programming model based on stages, and explain how abstraction and performance are dealt with in a common transformational framework.

Section 3 introduces our basic parallel pattern, the *list homomorphism,* and formulates the problems to be studied: the extraction, composition and implementation of this pattern.

Section 4 presents a new homomorphism extraction method, based on a generalization of two sequential definitions, shows how it can be mechanized in a term rewriting framework, and applies it to the MSS problem in a case study.

Section 5 derives optimizing transformations for two practical homomorphism compositions, reduction with scan and scan with scan, and demonstrates their use on the MSS problem.

Section 6 deals with the formal derivation of an efficient, generic implementation for the class of distributable homomorphisms (DH) and illustrates its use on scan and FFT.

Section 7 considers specific aspects of the divide-and-conquer parallelism. We classify divide-and-conquer algorithms and derive an implementation for the case of a mutually recursive specification. For binary divide-and-conquer, we propose a new interconnection topology, the $\mathcal{N}$-graph, with the properties of a fixed node degree, balanced load and local communications. Finally, we demonstrate a systematic design method on the case study of polynomial multiplication, for which a time- and cost-optimal solution is developed.

Section 8 summarizes the findings of the SAT approach and outlines future research.

2 Outline of the SAT approach

This section outlines the SAT approach as a common framework for dealing with performance and abstraction, and pinpoints its place in the research area of parallel programming.

2.1 Performance view

To ensure competitive target performance, the design process should result in a program which can be implemented directly and efficiently on a wide variety of parallel machines. We call such a representation of the parallel target program the *performance view*.[1] Following the current practice of parallel programming, the SAT approach adopts a performance view based on the SPMD model and the MPI standard.

SPMD (Single Program Multiple Data) [53, 70] is the most popular and, as is widely believed, the only feasible way of programming massively parallel machines. An identical program drives all the processors, but the path through the program is determined by the logical coordinate of the processor which is a parameter of the program. MPI (Message Passing Interface) [47] is a library of parallel primitives; recently, it has become the *de facto* standard for parallel programming in the SPMD style. It is portable across

[1] Earlier, we called it the *target view* [34].

practically all kinds of parallel and distributed platforms. Attempting to cover a possibly wide variety of parallel primitives, MPI is becoming too rich and, therefore, hard to use. A new perspective is being offered by recent theoretical results on coarse-grain algorithms with global communications [25,30] and also by the practice of MPI programming with a restricted set of collective communications [45,71].

To liberate the performance view from unimportant details, we represent it in simplified, MPI-like pseudocode. This code comes in three types of statements:

Computations are represented as sequential function calls, e.g., `CALL f(a,b)`.

Communications are restricted to the collective communication primitives, like `BCAST` for broadcasting, `ALL-TO-ALL` for the exchange between every pair of processors, etc.

Combinations are collective primitives, which prescribe both computations in the processors and communications between them. Examples are `REDUCE(+)` for computing a sum of elements across the processors, and the parallel prefix `SCAN(+)`.

Note that we try to avoid point-to-point communications in the performance view; we use them mostly for the efficient implementation of collective patterns on particular machines. Collective operations can be restricted to a particular group of processors, e.g., the reduction `REDUCE(+) in ROW` is restricted to a row of a virtual processor matrix, and is applied for all rows simultaneously.

2.2 Abstraction view

For the purpose of abstraction, SAT makes use of the Bird-Meertens formalism (BMF) on lists [8]. Originally created for the design of sequential programs, BMF is becoming increasingly popular in the parallel setting [74]. In BMF, higher-order functions (functionals) capture, independently of the parallel architecture, general idioms of parallel programming which can be composed for representing algorithms. BMF functionals use elementary operators and functions as parameters. A BMF expression usually represents a class of programs which can be reasoned about, either taking into account particular properties of the customizing functions or not. This style of programming is called generic [64] or skeleton-based [4,20].

Let us introduce the BMF notation used in this survey. As the basic data structure, we take the non-empty list which is implemented as an array in the performance view. Function application is denoted by juxtaposition, binds most tightly and associates to the left. For the sake of brevity, we define the BMF functionals informally.

The simplest and at same time the most "parallel" functional of BMF is map, which applies a unary function f, defined on elements, to each element of a list, i.e.,

$$map\ f\ [x_1, x_2, \ldots, x_n] \ = \ [\, f\, x_1, f\, x_2, \ldots, f\, x_n\,] \tag{1}$$

There is also red (for *reduction*) with a binary associative operator $\oplus$:

$$red\,(\oplus)\,[x_1, x_2, \ldots, x_n] \ = \ x_1 \, \oplus \, x_2 \, \oplus \, \ldots, \oplus\, x_n \tag{2}$$

Reduction can be computed on a binary tree, with $\oplus$ in the nodes, optimally in $\log n$ time for an argument list of length n.

There are some other functionals which will be introduced later. Individual functions are composed in BMF by means of functional composition $\circ$, such that $(f \ \circ \ g)\, x \ = \ f\,(g\, x)$, which represents sequential execution order on (parallel) stages.

2.3 Design in SAT: stages and transformations

Our ultimate goal is to mediate between the functional abstraction view and the imperative performance view in the program design process. To do so, the SAT approach is based on the following two concepts which give it its name:

Stages are building blocks of both the abstraction view and the performance view: a program is always a sequence of stages. Each stage encapsulates parallelism of a possibly different kind and involves potentially all processors.

Transformations support program design and optimization. They are correctness-preserving transitions – either between different abstraction views, or from an abstraction view to a performance view.

2.3.1 Stages

The traditional formal model of parallelism, the PRAM [80], facilitates the description and comparison of parallel algorithms, but it does not always reflect the real costs of parallelism. In a more practical model, which also takes account of the network by which processors are connected, a program is viewed as a collection of communicating processes, each with a local memory. It can be called the "PAR-SEQ" model (PARallel composition of SEQuential processes). This model corresponds directly to SPMD; it results usually in efficient programs but involves too many low-level details.

A key feature of SAT is that the PAR-SEQ model is imposed solely on the target program. The program design process is based on the dual programming model, SEQ-PAR. This model, which has its origins in SIMD parallelism [12], is extended in SAT to the class of coarse-grain SPMD-like programs. We adopt the idea of communication-closed layers, introduced originally in the CSP setting [12, 24]: no communication is allowed between different stages. In particular, data (re)distributions are stages themselves.

The restrictions imposed by the SAT model simplify the program structure significantly: the overall control becomes sequential, and parallel pieces of the program become smaller and easier to understand. Similar ideas can be found in the group-SPMD model [71], in the concept of coarse-grain algorithms [30], and in the so-called "bridging" models of parallelism, including the bulk-synchronous parallelism in the BSP model [59], asynchronous activity in LogP [22] and a number of scalable operations in the WPRAM [65].

2.3.2 Transformations

The general idea of transformational programming [14] is to start with an intuitively clear and correct algorithm for a problem and to proceed by step-wise transformation until a semantically equivalent solution with sufficient performance is reached [6]. Although elegant and promising, this approach still has not been a real practical success. One of the reasons is, in our opinion, that the user is expected to become an expert in the transformational formalism.

A way to shield the user from the underlying transformational formalism is offered by the *skeleton* approach, whose original motivation was to capture common schemas of parallelism found in different applications [20]. Skeletons can be viewed as higher-order functions: e.g., the BMF skeleton map, defined by equation (1) with parameter function f, can be customized for a particular application. If a high-quality parallel implementation is offered for a skeleton, then the only task remaining for the user is to cast the particular problem in the given schema. The user need not be aware of which particular steps were used to obtain the implementation. Skeletons have been studied vigorously recently; related work includes experimental skeleton systems [3, 11, 23], different kinds of recursion [50, 81], transformations of skeletons [27, 79], etc.

BMF expressions – and, therefore, also the programs specified by them – can be manipulated by applying rules of the formalism. A simple example of a BMF transformation rule is the map fusion law:

$$map\,(f \circ g) \;=\; map\,f \;\circ\; map\,g \tag{3}$$

If the composition of two stages on the right-hand side of (3) is implemented via a barrier, then it should be transformed into the left-hand side which is more efficient. A cost calculus can be applied to predict the impact of particular choices on the parallel performance [75]. There is a rich body of BMF transformations which can be used in the design process (see Subsection 5.2 and [74] for examples).

2.4 SAT and homomorphisms

The scope of the SAT approach includes algorithms working on recursively constructed data types, such as lists, arrays, trees and so on. The basic parallel skeleton used in this survey is the *homomorphism*. Introduced by Bird [8] in the constructive theory of lists, it has been studied extensively in the category-based theory of data types [60, 74]. Our interest in homomorphisms is due to their direct correspondence to data parallelism [12, 53] and to the divide-and-conquer paradigm which is used extensively in both sequential and parallel algorithm development [13, 72].

An important completeness property of the BMF framework is that any formulation of a homomorphism on a recursively constructed data type can be transformed in the formalism into any other formulation of that homomorphism by equational substitution [73]. Whether the world of “all parallel programs” has a similar property is an intriguing open question. Homomorphisms and, more generally, divide-and-conquer algorithms are the main building blocks in SAT and the focus of our interest in the rest of this survey.

3 List homomorphisms

The homomorphism principle is well known and widely used in different areas: intuitively, a homomorphic function preserves the structure of its domain in the codomain.

In the SAT approach, we apply the homomorphism principle to capture the pattern of *data parallelism* in the abstraction view: (1) domains represent data structures, constructed inductively of smaller parts (lists, trees, etc.); (2) codomains represent the results of computing functions on the structured data; (3) a function is a homomorphism if its image of a data structure can be computed from the images of the function on the parts of the structure. Dually, homomorphisms express the *divide-and-conquer* paradigm: to solve a problem, *divide* data into parts, solve the problem on the parts and combine the results to yield the overall solution (*conquer* the problem). Divide-and-conquer is a fundamental principle of algorithm design, both in theory and in practice [72].

Our basic data structure is the non-empty list, with list concatenation $+\!\!+$ as constructor.

Definition 3.1 (Bird [8]) *A list function h is a homomorphism iff there exists a binary operator $\circledast$ such that, for all lists x and y:*

$$h\,(x +\!\!+\, y) \;=\; h\,x \;\circledast\; h\,y \tag{4}$$

In words: the value of h on a concatenated list can be computed by applying the *combine operator* $\circledast$ to the values of h on the pieces of the list. Since the computations of $h\,x$ and $h\,y$ are independent of each other, they can be carried out in parallel. Note that $\circledast$ in equation (4) is necessarily associative on the range of h, because $+\!\!+$ is associative.

As a running example, we take the scan function, also called parallel prefix [10]. This simple function has been extremely useful in many applications; at the same time it is easy to parallelize. These, at first glance, surprising properties can be explained by the fact that scan expresses a general pattern of linear recursion and, thus, inherits its expressive power and potential for parallelization.

Example 1 (Scan as a homomorphism) Function *scan* yields, for an associative binary operator $\odot$ and a list, the list of “prefix sums”. For example, on a list of three elements:

$$\textit{scan}\,(\odot)\,[a, b, c] \;=\; [\,a\,,\; a \odot b\,,\; a \odot b \odot c\,] \tag{5}$$

Scan is a homomorphism with combine operator $\circledast$:

$$\begin{aligned} \textit{scan}\,(\odot)\,(x +\!\!+\, y) \;&=\; S_1 \;\circledast\; S_2 \;=\; S_1 \;+\!\!+\; (\textit{map}\,((\textit{last}\,S_1)\odot)\;S_2) \\ \text{where } S_1 \;&=\; \textit{scan}\,(\odot)\,x\,,\;\; S_2 \;=\; \textit{scan}\,(\odot)\,y. \end{aligned} \tag{6}$$

In (6), we use a shorthand, the operator section: we fix one argument of $\odot$ and obtain a unary function, $((\textit{last}\,S_1)\odot)$, which is supplied as an argument to map.

Theorem 1 (Bird [8]) *A list function h is a homomorphism iff it can be factored into the composition:*

$$h \;=\; \textit{red}\,(\circledast) \;\circ\; \textit{map}\,f \tag{7}$$

where, for any element a, $f\,a \;=\; h\,[a]$, and $\circledast$ is as in (4).

Each homomorphism is uniquely determined by f and $\circledast$. Thus, all homomorphisms can be viewed as instances of a single *homomorphism skeleton*, with customizing functions f and $\circledast$. This skeleton – and, consequently, all its instances – can be computed by a two-stage program, the right-hand side of (7), composed of the highly parallel BMF functionals map and red introduced in Sect. 2.

Thus, the homomorphism skeleton provides the abstraction view of a class of functions on lists, together with a common parallelization schema, given by (7), for this class. Homomorphisms can be positioned between two main kinds of skeletons studied in the literature: elementary skeletons like *map* or *fold*, on the one hand, and more elaborate, application-oriented skeletons, such as diverse flavors of divide-and-conquer, on the other hand.

To use homomorphisms in the SAT design process, the following problems should be addressed:

- *Extraction*: Check whether a given function can be expressed as an instance of the homomorphism skeleton, i.e., construct the customizing functions f and $\circledast$ of (7).
- *Adjustment*: If a function is not a homomorphism, try to make it one, at the expense of additional computations.
- *Composition*: Find an efficient way to compose several homomorphisms as stages in a larger abstract program.
- *Implementation*: For an extracted/adjusted homomorphism, find an efficient performance view, which may depend on the properties of the customizing functions.

In the following sections, we study the problems listed. Each section ends with a brief statement about the impact of its findings on the program design process.

4 Extraction and adjustment

In this section, we present a way of extracting homomorphic parallelism from a given function h, or adjusting the function to the homomorphism skeleton. Function h is a homomorphism if we can construct two customizing functions of (7): function f, which yields the image of h on a singleton list, and the combine operator $\circledast$. Whereas f can be found easily, the problem of constructing $\circledast$ is not trivial: e.g., for the scan function (5), an elaborate correctness proof of a parallel algorithm is presented [66] or some *eureka* steps are applied [46].

4.1 The CS-method

A systematic way of addressing the homomorphism extraction problem is offered in BMF by establishing a connection between two ways of defining functions on lists. In contrast to homomorphisms, in which the constructor is the list concatenation, sequential functional programming is based on the constructor `cons`, which attaches an element to the front of the list. We denote `cons` by $\cdot\!:$ and introduce also its dual, `snoc`, denoted by $:\!\cdot$, which attaches an element at the list's end.

Definition 4.1 *List function h is called leftward (lw) iff there exists a binary operator $\oplus$ such that $h\,(a \cdot\!\!: y) = a \oplus h(y)$, for all elements a and lists y. Dually, function h is rightward (rw) iff, for some $\otimes$, $h\,(y :\!\!\cdot\, a) = h(y) \otimes a$.*

Note that $\oplus$ and $\otimes$ need not be associative, so many functions are either *lw* or *rw*, or both.

Theorem 2 ([29]) *A function on lists is a homomorphism iff it is both lw and rw.*

In earlier work [5,29], the existence of leftward and rightward algorithms for a problem has been used as evidence that a homomorphic algorithm for that problem exists, but no method of constructing the homomorphism has been offered.

In [36], we take a step towards finding the combine operator $\circledast$ from operators $\oplus$ and/or $\otimes$. We start by imposing an additional restriction on the leftward and rightward functions.

Definition 4.2 *Function h is called left-homomorphic (lh) iff there exists $\circledast$ such that, for arbitrary list x and element a, $h\,(a \cdot\!\!: y) = h\,([a]) \circledast h\,(y)$. Likewise, for right-homomorphic (rh) functions, $h\,(x :\!\!\cdot\, b) = h\,(x) \circledast h\,([b])$.*

Evidently, every *lh* (*rh*) function is also *lw* (*rw*), but not vice versa.

Theorem 3 *If function h is a homomorphism then h is both lh and rh with the same combine operator. If function h is lh or rh, and the combine operator is associative, then h is a homomorphism with this combine operator.*

In [32], we prove a slightly stronger property. Theorem 3 indicates that both the left and right homomorphy should be exploited together in the construction of the combine operator. Examples in [36] demonstrate some unsuccessful attempts to arrive at an associative combine operator, starting solely from either a `cons` or a `snoc` representation.

The *CS-method* (for "Cons and Snoc"), proposed originally in [32], is to request from the user two representations of a function, and to bring them into a common form, i.e., to *generalize* them.

Definition 4.3 *A term t_G is called a* generalizer *of terms t_1 and t_2 in the equational theory E (E-*generalizer*) if there are substitutions σ_1 and σ_2, such that $t_G.\sigma_1 \leftrightarrow_E^* t_1$ and $t_G.\sigma_2 \leftrightarrow_E^* t_2$.*

Here, $\leftrightarrow_E^*$ denotes the semantic equality (conversion relation) in the equational theory E, and $t.\sigma$ denotes the result of applying substitution σ to term t. Generalization is the dual of *unification* and is sometimes also called "anti-unification" [49].

Let us assume that function h is a homomorphism, i.e., equation (4) holds, and t_H denotes a term over u and v which defines $\circledast$:

$$u \circledast v \leftrightarrow t_H \qquad (8)$$

$$\begin{tikzcd}
t_B \circledast h(y) \arrow[r, leftrightarrow, "(9)"'] & t_C \arrow[r, leftrightarrow, "*", "E"'] & t_H.\sigma_1 \arrow[r, equal] & t_L \\
u \circledast v \arrow[u, dotted, "\sigma_1"] \arrow[d, dotted, "\sigma_2"] \arrow[r, leftrightarrow, "(8)"'] & t_H \arrow[r, equal] & t_H \arrow[u, dotted, "\sigma_1"] \arrow[d, dotted, "\sigma_2"] & \\
h(x) \circledast t'_B \arrow[r, leftrightarrow, "(9)"'] & t_S \arrow[r, leftrightarrow, "*", "E"'] & t_H.\sigma_2 \arrow[r, equal] & t_R
\end{tikzcd}$$

(a) as a simultaneous generalization problem

$$\begin{tikzcd}
t_B \circledast h(y) \rightharpoonup t_C \arrow[d, leftrightarrow, "*"', "E"] \\
h([a]) \circledast h(y) \rightharpoonup t_L \\
u \circledast v \rightharpoonup t_H \arrow[u, dotted, "\sigma_1"] \arrow[d, dotted, "\sigma_2"] \\
h(x) \circledast h([b]) \rightharpoonup t_R \arrow[d, leftrightarrow, "*"', "E"] \\
h(x) \circledast t'_B \rightharpoonup t_S
\end{tikzcd}$$

(b) using rule term encoding

Fig. 1. The relationships of the terms after a successful generalization

The following two terms, constructed from t_H by the substitutions $t_L = t_H.\{u \mapsto h([a]), v \mapsto h(y)\}$ and $t_R = t_H.\{u \mapsto h(x), v \mapsto h([b])\}$, are semantically equal variants of all `cons` and `snoc` definitions of h, respectively. Let us pick two arbitrary definitions:

$$\begin{aligned} h([a]) &\leftrightarrow t_B \qquad\qquad \text{and} \qquad\qquad & h([b]) &\leftrightarrow t'_B \\ h(a \cdot\!: y) &\leftrightarrow t_C & h(x :\!\cdot\, b) &\leftrightarrow t_S \end{aligned}$$

According to Theorem 3, function h is both *lh* and *rh* with combine operator $\circledast$, so we can rewrite these two definitions as follows:

$$t_B \circledast h(y) \leftrightarrow t_C \qquad\qquad \text{and} \qquad\qquad h(x) \circledast t'_B \leftrightarrow t_S \tag{9}$$

Figure 1(a) illustrates the relationship between terms t_H, t_C, t_S, t_L, and t_R. Dotted arrows indicate applications of substitutions; solid arrows indicate conversion steps. Each substitution is applied to *two* terms – this is a simultaneous generalization problem. In order to work with the defined notion of generalizer on terms, we introduce a fresh binary function symbol, $\rightharpoonup$, and model pairs $s \leftrightarrow t$ of terms as terms $s \rightharpoonup t$, called *rule terms*. With this encoding we get the view of Fig. 1(b). In both subfigures, $\sigma_1 = \{u \mapsto t_B, v \mapsto h(y)\}$, $\sigma_2 = \{u \mapsto h(x), v \mapsto t'_B\}$.

The following theorem states that a generalization of the two rule terms derived of (9) leads to term t_H from (8) – the desired piece of the definition of h as a homomorphism.

Theorem 4 *Let E be the theory of lists and let $t'_B = t_B.\{a \mapsto b\}$. If the two rule terms, $t_B \circledast h(y) \rightharpoonup t_C$ and $h(x) \circledast t'_B \rightharpoonup t_S$, have an E-generalizer $u \circledast v \rightharpoonup t_H$ w.r.t. substitutions σ_1 and σ_2, and the operator $\circledast$ thus defined is associative, then the function defined by t_C and t_S is a homomorphism with $\circledast$ as combine operator.*

The generalization in Theorem 4 is called the *CS-generalization*: it is the key step of the following CS-method of homomorphism extraction.

CS-Method:

1. The user is requested to provide two sequential definitions for a given function: a cons term t_C and a snoc term t_S, after which two mechanizable steps follow.
2. CS-generalization, applied to the rule terms $t_B \circledast h(y) \rightharpoonup t_C$ and $h(x) \circledast t'_B \rightharpoonup t_S$, yields a rule term $u \circledast v \rightharpoonup t_H$.
3. If associativity of $\circledast$ defined by t_H can be proven then, by Theorem 4, $\circledast$ is the desired combine operator.

Example 1 (Continued) With the CS-method, we can extract a homomorphism for the scan function. The rule terms for *scan* are as follows:

$$[a] \;\circledast\; \mathit{scan}\,(\odot)\,y \rightharpoonup a \;\cdot\!\!:\; (\mathit{map}\,(a\,\odot)\,(\mathit{scan}\,(\odot)\,y))$$
$$\mathit{scan}\,(\odot)\,x \;\circledast\; [b] \rightharpoonup (\mathit{scan}\,(\odot)\,x) \;:\!\!\cdot\; (\mathit{last}\,(\mathit{scan}\,(\odot)\,x) \odot b)$$

Their CS-generalization [28] yields: $u \circledast v \;\rightharpoonup\; u \;+\!\!+\; \mathit{map}\,(\mathit{last}\,(u)\,\odot)\,v$. One can prove the associativity of operator $\circledast$ thus defined:

$$u \circledast v \;\leftrightarrow\; u \;+\!\!+\; \mathit{map}\,(\mathit{last}\,(u)\,\odot)\,v \tag{10}$$

Therefore, *scan* is a homomorphism, with $\circledast$ defined by (10) and $f = [.]$, where function [.] wraps its parameter into a singleton list.

4.2 Mechanizing the CS-method

To apply the CS-method in practice, its generalization step must be mechanized, i.e., a generalization algorithm is required. In contrast to unification, properties and methods for generalization in a non-empty equational theory have largely been neglected in research [21]. This subsection, based on joint work with Alfons Geser [28], describes, to the best of our knowledge, the first practical algorithm for generalization in BMF.

We proceed in two steps: first, a *generalization calculus* is designed, which is then turned into a *generalization algorithm*. To have a simple technical apparatus, we restrict ourselves to first-order terms.

The objects of our generalization calculus are triples $(\sigma_1, \sigma_2, t_0)$, satisfying the *generalization property*: t_0 is a generalizer of t_1 and t_2 via the substitutions σ_1 and σ_2, respectively. Starting from the triple $(\{x_0 \mapsto t_1\}, \{x_0 \mapsto t_2\}, x_0)$, where x_0 is the most general generalizer, we specialize by applying inference rules successively until no inference rule applies. The basic idea is to repeat, as long as possible, the following step: extract a common mapping $x_i \mapsto v_i$ in substitutions σ_1 and σ_2 and move it to t_0. Every such step specializes t_0 while maintaining the generalization property. In the end, t_0 is maximally specialized.

The generalization calculus has two inference rules:

– The *ancestor decomposition* rule reconstructs a common top operation symbol of a pair of terms, potentially after a few reverse rewrite steps in the theory E. First, a common variable, x_i, in the domain of the two substitutions is selected. If the corresponding right-hand sides, u_i and v_i, have a common root function symbol, f, then the rule splits mapping $x_i \mapsto f(u'_1, \ldots, u'_m)$ of the first substitution into $x_i \mapsto f(x'_1, \ldots, x'_m)$ and $x'_j \mapsto u'_j$, where x'_j is a fresh variable for each argument position, j, of f. Likewise, $x_i \mapsto f(v'_1, \ldots, v'_m)$ of the second substitution is split, using the same x'_j, to $x_i \mapsto f(x'_1, \ldots, x'_m)$ and $x'_j \mapsto v'_j$. The common part of the results, $x_i \mapsto f(x'_1, \ldots, x'_m)$, is transferred to t_0.
– The *agreement* rule joins variables that map to the same term. Mappings $x_i \mapsto u$ and $x_j \mapsto u$ with common right-hand sides in the first substitution can be transformed into $x_j \mapsto x_i$ and $x_i \mapsto u$. Moreover, if there are $x_i \mapsto v$ and $x_j \mapsto v$ in the second substitution, then they are transformed into $x_j \mapsto x_i$ and $x_i \mapsto v$. These transformations are applied only if both are applicable, and the common part of their results, $x_j \mapsto x_i$, is transferred to t_0.

To turn the generalization calculus into an algorithm, we propose a strategy of applying the inference rules: we prefer the agreement rule, apply a rule always to the smallest possible index, and restrict the ancestor decomposition rule to at most one reverse rewrite step applied at the top of terms.

The following results, proven in [28], justify the proposed mechanization of the CS-method:

Soundness of the calculus: every derivation yields a generalizer of the given two terms.

Reliability of the calculus: if the associativity of the generalized operator can be proven then this operator indeed customizes the function to the homomorphism skeleton.

Termination of the algorithm: the generalization process terminates under the imposed strategy.

Although the worst-case time complexity of the generalization algorithm is exponential, it is linear on practical examples. The homomorphic representation of the scan function is extracted automatically in six steps [28].

4.3 Almost-homomorphisms: the MSS problem

In this subsection, we deal with functions which are not homomorphisms, and we attempt to adjust them to the homomorphic format. A powerful method of adjustment is based on the notion of an *almost-homomorphism* [16] – a function which can be turned into a homomorphism by phrasing it in terms of a number of auxiliary functions.[2] Our contribution is a systematic way of constructing suitable auxiliary functions, based on the CS-method (Subsection 4.1).

We demonstrate our approach by considering the famous *maximum segment sum* (MSS) problem – a *programming pearl* [7], studied by many authors [8,16,74,76,78].

The MSS Problem. Given a list of numbers, function *mss* returns the largest sum across all contiguous list segments, e.g.,

$$\mathit{mss}\,[\,2, -4, 2, -1, 6, -3\,] \;=\; 7$$

where the result is contributed by the segment $[2, -1, 6]$.

The CS-method starts with defining function *mss* over `cons` lists, i.e., writing a sequential functional program for *mss*. Let function $\uparrow$ return the larger of its two arguments. For some element a and list y, it may well be the case that $\mathit{mss}\,(a \cdot\!: y) = a \uparrow (\mathit{mss}\, y)$. But the true segment of interest may also include both a and some initial segment of y; so we have to introduce an auxiliary function *mis*, which yields the sum of the *maximum initial segment*. The next step of the CS-method, the `snoc` definition, requires the introduction of a new auxiliary function, *mcs* (for *maximum concluding segment*).
We obtain the following definitions of *mss*:

$$\begin{aligned}\mathit{mss}\,(a \cdot\!: y) &= a \uparrow \mathit{mss}\, y \uparrow (a + \mathit{mis}\, y)\\ \mathit{mss}\,(x :\!\cdot\, b) &= \mathit{mss}\, x \uparrow (\mathit{mcs}\, x + b) \uparrow b\end{aligned}$$

Augmenting function *mss* with both auxiliary functions, we obtain a triple: $\langle \mathit{mss}, \mathit{mis}, \mathit{mcs}\rangle$. For each function of the triple, the CS-method requires both `cons` and `snoc` definitions, and this leads to one more auxiliary function, *ts*, which computes the total sum over the list [39].

[2] By using the identity function, every function can be turned into a trivial homomorphism; however, no useful parallelism results from that.

In all, we have introduced three auxiliary functions which, together with mss, constitute a quadruple: $\langle \mathit{mss}, \mathit{mis}, \mathit{mcs}, \mathit{ts} \rangle$. The cons and snoc definitions of the four functions are closed, i.e., they refer exclusively to the functions of the quadruple. Our generalization algorithm, applied for each function of the quadruple, requires 27 steps in total, and yields the following combine operator [28]:

$$\begin{aligned} &(\mathit{mss}\, x, \mathit{mis}\, x, \mathit{mcs}\, x, \mathit{ts}\, x) \;\circledast\; (\mathit{mss}\, y, \mathit{mis}\, y, \mathit{mcs}\, y, \mathit{ts}\, y) \;= \\ &\quad \big(\, \mathit{mss}\, x \uparrow (\mathit{mcs}\, x + \mathit{mis}\, y) \uparrow \mathit{mss}\, y \,,\; \mathit{mis}\, x \uparrow (\mathit{ts}\, x + \mathit{mis}\, y) \,, \\ &\quad\;\; \mathit{mcs}\, y \uparrow (\mathit{mcs}\, x + \mathit{ts}\, y) \,,\; (\mathit{ts}\, x + \mathit{ts}\, y) \,\big) \end{aligned} \tag{11}$$

Since $\circledast$ is associative, the quadruple is a homomorphism with two customizing functions: operator $\circledast$ defined by (11), and function f defined as follows:

$$f\, a \;=\; \langle \mathit{mss}, \mathit{mis}, \mathit{mcs}, \mathit{ts} \rangle\, [a] \;=\; (a \,,\, a \,,\, a \,,\, a) \tag{12}$$

The abstract program for the *mss* problem is obtained immediately using schema (7):

$$\mathit{mss} \;=\; \pi_1 \,\circ\, \mathit{red}\, (\circledast) \,\circ\, \mathit{map}\, f \tag{13}$$

where projection π_1 yields the first component of a tuple, in our case, mss. Program (13) coincides with the results of [16,76] but, unlike them, our solution has been obtained systematically, by using the CS-method and the standard homomorphic parallelization, rather than by relying on our intuition about parallelism in the derivation process. The mechanization of the adjustment process is discussed in the next subsection.

Let us estimate the parallel time complexity of the derived homomorphic algorithm, whose performance view in MPI-like pseudocode is given in Subsection 6.1. Since both f and $\circledast$ require a constant number of communicated elements and executed operations, the total time on n processors is $O(\log n)$. The number of processors can be reduced to $n/\log n$ by simulating lower levels of the processor tree sequentially, based on Brent's theorem [70]. Therefore, program (13) is both time- and cost-optimal.

4.4 Impact on program design

The CS-method of homomorphism extraction and adjustment, graphically depicted in Fig. 2, has two practical advantages: (1) it requires from the user only sequential representations of the problem in question; (2) it offers a potential for mechanization. For proofs of associativity, we have used the induction prover TIP [26]. For the scan example, TIP produced an

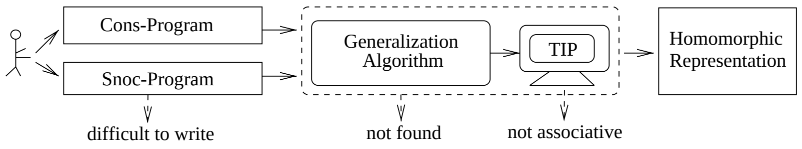

Fig. 2. Application of the CS-Method

inductive proof of associativity without any user interaction. Of course, there are cases in which the method does not succeed, as indicated in the figure by the downarrows which are labeled with the reasons for the failure.

The rewriting approach seems to be especially useful in the case of almost-homomorphisms, i.e., when a non-trivial adjustment is necessary. The maximum segment sum problem requires only three auxiliary functions but, in other cases, there may be more. For example, there are eleven functions in the two-dimensional case of MSS [76].

The CS-method applies also to the adjustment of so-called *nested almost-homomorphisms*, whose combine operators turn out to be, again, almost-homomorphisms. An example is the multi-bracket matching problem [32], for which we obtain two levels of homomorphic parallelism, with an overall time complexity of $O(\log^2 n)$.

5 Composition of homomorphisms

This section addresses the composition of homomorphisms in the abstraction view – an important issue in the SAT design process, where programs are typically composed of several stages.

5.1 Rules of composition

We study two compositions which are often used in practice: scan with reduction and scan with scan. Since both composed functions are homomorphisms, it is natural to try and fuse them into one, more complex parallel stage. We consider both the conventional scan, or prefix, defined by (5), and its symmetric version, called suffix:

$$suf\,(\odot)\,[x_1, x_2, x_3] \;=\; [\,x_1 \odot x_2 \odot x_3\,,\; x_2 \odot x_3\,,\; x_3\,]$$

We do not proceed by inventing a solution and then verifying it, but rather by systematic adjustment to the homomorphism skeleton using formal transformations in BMF [38]: (1) the scan-reduction composition is defined for a concatenation of two lists; (2) since the obtained format is not homomorphic, the composition is augmented with reduction as an auxiliary function;

(3) the resulting pair of functions is transformed into a homomorphic format under additional assumptions about the customizing operators.

Thus, the composition of scan with reduction is shown to be an almost-homomorphism. Further BMF-calculations yield an almost-homomorphic format for the composition of scan with scan. Both results are combined in the following theorem:

Theorem 5 (Composition Rules [38]) *For arbitrary associative binary operators, $\oplus$ and $\otimes$:*

(a) if $\otimes$ is left-distributive over $\oplus$ then

$$red\,(\oplus) \ \circ\ scan(\otimes) \ = \ \pi_1 \ \circ\ red\,(\langle\oplus,\otimes\rangle) \ \circ\ map\,pair \tag{14}$$
$$scan\,(\oplus) \ \circ\ scan\,(\otimes) \ = \ map\,\pi_1 \ \circ\ scan\,(\langle\oplus,\otimes\rangle) \ \circ\ map\,pair \tag{15}$$

(b) if $\otimes$ is right-distributive over $\oplus$ then

$$red\,(\oplus) \ \circ\ suf(\otimes) \ = \ \pi_1 \ \circ\ red\,(<\oplus,\otimes>) \ \circ\ map\,pair \tag{16}$$
$$suf\,(\oplus) \ \circ\ suf\,(\otimes) \ = \ map\,\pi_1 \ \circ\ suf\,(<\oplus,\otimes>) \ \circ\ map\,pair \tag{17}$$

where functions $pair$, $\langle\oplus,\otimes\rangle$ and $<\oplus,\otimes>$ are defined as follows:[3]

$$pair\,a \ \stackrel{\text{def}}{=}\ (a,a) \tag{18}$$
$$(p_1,r_1)\,\langle\oplus,\otimes\rangle\,(p_2,r_2) \ \stackrel{\text{def}}{=}\ (p_1 \oplus (r_1 \otimes p_2)\,,\, r_1 \otimes r_2) \tag{19}$$
$$(s_1,r_1) <\oplus,\otimes> (s_2,r_2) \ \stackrel{\text{def}}{=}\ ((s_1 \otimes r_2) \oplus s_2\,,\, r_1 \otimes r_2) \tag{20}$$

The theorem extends previous results in the general theory of homomorphisms [60] and in sequential program design [9, 78]. To the best of our knowledge, the transformation of two scans is new; a version of the scan-reduction composition was used in [15] for parallelizing linear recurrences. Our formulation is arguably more convenient for direct use in practical parallel programming. In the sequential case [9], the scan-reduction fusion is expressed more simply, due to the use of the *foldl* functional which, however, prescribes a particular order of computation.

5.2 Derivation by transformation: MSS revisited

This subsection demonstrates program derivation in SAT: a problem is first specified in an intuitively clear, but inefficient abstraction view; this view is then transformed into a more efficient view by means of the composition rules, developed in the previous subsection, and other BMF–based transformations. This approach can be viewed as an alternative to an extraction

[3] Please note that two similarly looking notations – $<\,,\,>$ and $\langle\,,\,\rangle$ – are used for different functions.

via the CS-method of Subsection 4.3, which proceeds by generalizing two sequential programs.

The following equational rules of BMF will be used:

$$\begin{aligned} scan\,(\oplus) &= map\,(red\,(\oplus)) \circ inits && (21)\\ suf\,(\oplus) &= map\,(red\,(\oplus)) \circ tails && (22)\\ inits \circ map\,f &= map\,(map\,f) \circ inits && (23)\\ map\,f \circ red\,(+\!\!+) &= red\,(+\!\!+) \circ map\,(map\,f) && (24)\\ red\,(\oplus) \circ red\,(+\!\!+) &= red\,(\oplus) \circ map\,(red\,(\oplus)) && (25) \end{aligned}$$

Functions $inits$ and $tails$ are introduced informally:

$$\begin{aligned} inits\,[x_1, x_2, \ldots, x_n] &= [\,[x_1], [x_1, x_2], \ldots, [x_1, x_2, \ldots, x_n]\,]\\ tails\,[x_1, x_2, \ldots, x_n] &= [\,[x_1, x_2, \ldots, x_n], \ldots, [x_{n-1}, x_n], [x_n]\,] \end{aligned}$$

For comparison with the CS-method, let us consider again the MSS problem formulated in Subsection 4.3. We start with the following intuitive abstraction view of function *mss*:

$$mss = red\,(\uparrow) \circ map\,(red\,(+)) \circ segs \qquad (26)$$

Specification (26) consists of three stages, from right to left:

$segs$: yields the list of all segments of the original list;

$map\,(red\,(+))$: for each segment, computes the sum of its elements;

$red\,(\uparrow)$: computes the largest sum by reducing with $\uparrow$ (maximum).

A constructive method for producing all segments of a list, i.e., for implementing function $segs$, is given by:

$$segs = red\,(+\!\!+) \circ map\,tails \circ inits \qquad (27)$$

Specification (26) with function $segs$ defined by (27) is obviously correct. However, it has a poor time complexity: $O(n^3)$ for a list of length n [75].

Note that our concrete operators, $\uparrow$ and $+$, are both associative; moreover, $+$ distributes backwards over $\uparrow$, because $(a \uparrow b) + c = (a + c) \uparrow (b + c)$. Therefore, we can use Theorem 5 for composing suffix with reduction in the following derivation:

$$\begin{aligned} & mss = red\,(\uparrow) \circ map\,(red\,(+)) \circ red\,(+\!\!+) \circ map\,tails \circ inits\\ = \quad & \{\ \text{Eq. (24),(25)}\ \}\\ & red\,(\uparrow) \circ map\,(red\,(\uparrow)) \circ map\,(map\,(red\,(+))) \circ map\,tails \circ inits\\ = \quad & \{\ \text{Eq. (3),(22)}\ \}\\ & red\,(\uparrow) \circ map\,\big(\,red\,(\uparrow) \circ suf\,(+)\,\big) \circ inits \end{aligned}$$

$$\begin{array}{ll}
= & \{\text{ Theorem 5 (b), i.e., Eq. (16) with } \otimes = + \text{ and } \oplus = \uparrow \} \\
& \mathit{red}\,(\uparrow) \circ \mathit{map}\,\big(\pi_1 \circ \mathit{red}\,\big(<\uparrow,+>\big) \circ \mathit{map\,pair}\,\big) \circ \mathit{inits} \\
= & \{\text{ Eq. (3),(23),(21) }\} \\
& \mathit{red}\,(\uparrow) \circ \mathit{map}\,\pi_1 \circ \mathit{scan}\,(<\uparrow,+>) \circ \mathit{map\,pair}
\end{array}$$

We have thus arrived at a program whose complexity is $O(n)$ sequentially and $O(\log n)$ in parallel.

In [15], the derivation stops at this point, but we can proceed further:

– Extend $\uparrow$ to pairs: $(a,b) \Uparrow (c,d) \overset{\text{def}}{=} (a\uparrow c\,, b\uparrow d)$, and note that:

$$\pi_1 \circ \mathit{red}\,(\Uparrow) = \mathit{red}\,(\uparrow) \circ \mathit{map}\,\pi_1 \tag{28}$$

– Note that $<\uparrow,+>$ distributes (forward) over $\Uparrow$, which allows us to apply Theorem 5 (a), with $\otimes = <\uparrow,+>$ and $\oplus = \Uparrow$.

After three more transformation steps [38], we arrive at:

$$\mathit{mss} = \pi_1^2 \circ \mathit{red}\,\big(\langle \Uparrow, <\uparrow,+> \rangle\big) \circ \mathit{map}\,(\mathit{pair}^2) \tag{29}$$

where $\pi_1^2 \overset{\text{def}}{=} \pi_1 \circ \pi_1$, and $\mathit{pair}^2 \overset{\text{def}}{=} \mathit{pair} \circ \mathit{pair}$. The first stage of program (29) creates a pair of pairs (quadruple) of each element, and the third stage picks the first component of a quadruple. These data management stages obviously require constant parallel time.

The central stage of (29) is the reduction with $\langle \Uparrow, <\uparrow,+> \rangle$; that operator is expressed in terms of $\uparrow$ and $+$ as follows:

$$\begin{array}{l}
\big(\,(r_1,s_1),(t_1,u_1)\,\big)\ \langle \Uparrow, <\uparrow,+> \rangle\ \big(\,(r_2,s_2),(t_2,u_2)\,\big) = \\
\quad \Big(\,\big(r_1 \uparrow r_2 \uparrow (t_1+s_2)\,, s_1 \uparrow (u_1+s_2)\big)\,, \big(t_2 \uparrow (t_1+u_2)\,, u_1+u_2\big)\,\Big)
\end{array} \tag{30}$$

Our result (30) is equivalent to solution (11) with logarithmic time complexity, which we have obtained in Sect. 4.3 from two sequential programs by means of the CS-method. The components of the quadruple have a problem-specific meaning – maximum initial segment sum, total sum, etc. – but this meaning is not referred to in our derivation. Again, the solution for the sequential case can be expressed more simply [9,78].

Our derivation establishes a connection between two parallel solutions known previously: an application of Theorem 5 for suffix-reduction leads to a solution similar to [15], which, by a one more application of Theorem 5, this time for scan-reduction, is improved and transformed into the solution from [16,76].

5.3 Impact on program design

The proposed composition rules (14)–(17) fuse two consecutive program stages, each including a global communication, into one stage. Thereby, extra communication and synchronization in the performance view are eliminated. We have developed analytical estimates of the influence of both rules on the target performance for various target models [44].

In addition to program derivation as demonstrated with the MSS example, composition rules can be applied to optimize target programs. E.g., the rules can be readily reformulated as source-to-source transformations of MPI programs [44].

6 Implementing homomorphisms

This section addresses the problem of implementing a homomorphism– either a given one, or one obtained as a result of extraction or adjustment. In SAT, implementation means search for an efficient performance view. We start with a simple standard performance view for an arbitrary homomorphism and proceed to the cases where additional restrictions must be imposed on the homomorphism skeleton in order to improve the efficiency of the implementation.

6.1 Standard implementation

The standard method of homomorphism implementation is to use program (7), which consists of two stages: a map followed by a reduction. This program may be time-optimal, but only under an assumption which makes it impractical: the required number of processors must grow linearly with the size of the data.

A more practical approach is to consider a bounded number p of processors, with a data block assigned to each of them. We introduce the type $[\alpha]_p$ of lists of length p, and affix functions defined on such lists with the subscript p, e.g., map_p. Partitioning of an arbitrary list into p sublists, called *blocks*, is done by the *distribution function*, $\mathit{dist}^{(p)} : [\alpha] \to [\,[\alpha]\,]_p$. The following obvious equality relates distribution with its inverse, flattening: $\mathit{red}\,(+\!\!+) \circ \mathit{dist}^{(p)} = \mathit{id}$.

Our further considerations are valid for arbitrary partitions but, in practice, one tries to obtain blocks of approximately the same size.

Theorem 6 (Promotion [8]) *For a $\circledast$-homomorphism h:*

$$h \circ \mathit{red}\,(+\!\!+) = \mathit{red}\,(\circledast) \circ \mathit{map}\,h \tag{31}$$

This general result about homomorphisms is useful for the practical issue of parallelization via data partitioning: from (31), we obtain the following standard abstraction view of homomorphism h on p processors:

$$h \;=\; red\,(\circledast) \;\circ\; \mathit{map}_p\, h \;\circ\; \mathit{dist}^{(p)} \tag{32}$$

To illustrate the use of schema (32), let us return to the MSS example, studied in Subsections 4.3 and 5.2. Since the quadruple $\langle \mathit{mss}, \mathit{mis}, \mathit{mcs}, \mathit{ts}\rangle$ is a homomorphism with $\circledast$ defined by (11), the promotion theorem can be applied to it:

$$mss \;=\; \pi_1 \;\circ\; red\,(\circledast) \;\circ\; \mathit{map}_p\,\langle \mathit{mss}, \mathit{mis}, \mathit{mcs}, \mathit{ts}\rangle \;\circ\; \mathit{dist}^{(p)} \tag{33}$$

To simplify the presentation of the performance view, we make the assumption, which is not uncommon in the parallel setting, that the program accepts the result of $\mathit{dist}^{(p)}$, i.e., the input list is distributed among the processors. In the performance view below, program `MSS-Dist`, the partitioned input is expressed by the HPF-like annotation `(BLOCK)` [25], which means that array `list` is distributed blockwise, with `list-block` being the block stored locally in a processor:

```
Program MSS-Dist (mypid);
/* Input: list (BLOCK) */
  CALL quadruple (list-block);
  REDUCE (⊛, root);
  IF mypid == root THEN OUTPUT (mss);
```

The three stages of program `MSS-Dist` are obtained directly from the stages of the abstraction view (33), which provides confidence that the generation of this performance view can be mechanized:

$\mathit{map}_p\,\langle \mathit{mss}, \mathit{mis}, \mathit{mcs}, \mathit{ts}\rangle$ is implemented by calling the sequential program `quadruple` in each of the p processors on its block of the input list;

$red\,(\circledast)$ is implementable by the MPI-like primitive `REDUCE` with the user-defined operation $\circledast$ which, in this case, is given by (11); the resulting quadruple is stored in processor `root`;

projection, π_1, is performed by processor `root` by picking the first component of the quadruple, thus yielding the result of the program.

The performance of program `MSS-Dist` can be predicted by estimating the complexity of each of its three stages: (1) sequential program `quadruple` has linear complexity and works on a block of size n/p, which takes time $O(n/p)$; (2) reduction over p processors, with a constant-time basic operation $\circledast$, takes time $O(\log p)$; (3) the last stage, projection, is obviously constant-time. Therefore, the time complexity of our target program, `MSS-Dist`, is $O(n/p{+}\log p)$ – the best one can expect in practice – with the constants depending on the characteristics of the concrete parallel machine.

6.2 Distributable homomorphisms (DH)

The standard implementation works well on examples like function *mss*, where the chunks of data, communicated in the reduction stage, remain constant. The situation is different if a function yields a composite data structure (list, array, etc.). E.g., the combine operator for scan, (6), contains concatenation, which induces linear communication costs in the reduction stage. Thus, we lose the optimality of a logarithmic-time solution. As shown in [75], this cannot be improved by just increasing the number of employed processors; moreover, this kind of reduction cannot be implemented using `MPI_Reduce`.

To improve the implementation, the class of homomorphisms is specialized in [35] to the subclass DH (for Distributable Homomorphisms), defined on *powerlists* [61], i.e., lists of length 2^k $(k = 0, 1, \dots)$, with balanced concatenation. The following DH definition makes use of the BMF functional *zip*, which combines elements of two lists of equal length with operator $\odot$:

$$\mathit{zip}\,(\odot)\ ([x_1, \dots, x_n], [y_1, \dots, y_n]) \ = \ [(x_1 \odot y_1), \dots, (x_n \odot y_n)]$$

Definition 6.1 *For binary operators, $\oplus$ and $\otimes$, the Distributable Homomorphism (DH) on lists, denoted $\oplus \updownarrow \otimes$, is defined as follows:*

$$\begin{aligned}(\oplus \updownarrow \otimes)\ [a] \ &= \ [a] \\ (\oplus \updownarrow \otimes)\ (x +\!\!+\, y) \ &= \ \mathit{zip}\,(\oplus)\,(u, v) \ +\!\!+\ \mathit{zip}\,(\otimes)\,(u, v) \qquad (34)\end{aligned}$$
$$where\ \ \mathit{length}\, x \ = \ \mathit{length}\, y \ = \ 2^k,\ u \ = \ (\oplus \updownarrow \otimes)\, x\,,\ v \ = \ (\oplus \updownarrow \otimes)\, y\,.$$

Figure 3 contrasts how a general homomorphism (on the left) and a distributable homomorphism (on the right) are computed on a concatenation of two powerlists. The main difference is the specific, pointwise format of the combine operator in a DH.

Next, we develop a common implementation schema of the DH skeleton and illustrate its applications.

6.3 Hypercube implementation

In [35], a generic hypercube implementation of DH is developed by introducing an architectural skeleton, *swap*, which describes a pattern of the hypercube (more generally, butterfly) behavior:

$$\mathit{hyp}\ \ (\mathit{swap}\ d\ (\oplus, \otimes)\ x)\ i \ \stackrel{\text{def}}{=} \begin{cases} (\mathit{hyp}\ x\ i)\ \oplus\ (\mathit{hyp}\ x\ (\mathit{xor}\,(i,\ 2^{d-1})))\,, & \text{if } i < \mathit{xor}\,(i,\ 2^{d-1}) \\ (\mathit{hyp}\ x\ (\mathit{xor}\,(i,\ 2^{d-1})))\ \otimes\ (\mathit{hyp}\ x\ i)\,, & \text{otherwise}\end{cases}$$

where $\mathit{length}\,(x) \ = \ 2^k\,,\ 1 \le d \le k\,,\ 0 \le i < 2^k.$

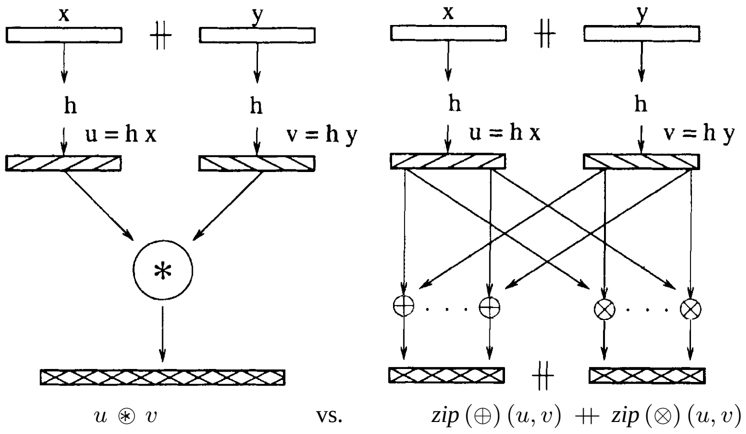
x
y
h
h
u = h x
v = h y
*
$u \circledast v$
vs.
$\mathit{zip}\,(\oplus)\,(u,v) \;+\!\!\!+\; \mathit{zip}\,(\otimes)\,(u,v)$

Fig. 3. Homomorphism (left) vs. Distributable Homomorphism (right)

Here, function hyp yields, for list x and index i, the ith element of x; function *xor* is the bitwise exclusive OR. Therefore, *swap* specifies a pairwise, bidirectional communication in one dimension of the hypercube, followed by a computation with one of the two customizing operators.

The following result relates the abstraction view of DH with the performance view expressed by $swap$.

Theorem 7 (DH on Hypercube [39]) *Every DH over a list of length $n = 2^k$ can be computed on an n-node hypercube by a sequence of swaps, with the dimensions counting from 1 to k:*

$$\oplus \updownarrow \otimes \;=\; \bigcirc_{d=1}^{k} (\mathit{swap}\; d\,(\oplus,\otimes)) \tag{35}$$

Here, expression $\bigcirc_{d=1}^{k} (\mathit{swap}\; d\,(\oplus,\otimes))$ is defined as follows:

$$\bigcirc_{d=1}^{k} (\mathit{swap}\; d\,(\oplus,\otimes)) \;\stackrel{\text{def}}{=}\; (\mathit{swap}\; k\,(\oplus,\otimes)) \circ \cdots \circ (\mathit{swap}\; 1\,(\oplus,\otimes))$$

To address the practical situation of a bounded number of processors, we introduce, for a function $h : [\alpha] \to [\alpha]$, its *p-distributed version,* $(\,\widetilde{h}\,)_p : [[\alpha]]_p \to [[\alpha]]_p$, so that

$$h \;=\; \mathit{red}\,(+\!\!\!+) \;\circ\; (\,\widetilde{h}\,)_p \;\circ\; \mathit{dist}^{(p)}$$

Programs for parallel machines are often of type $(\,\widetilde{h}\,)_p$: either it is assumed that the input and output data distribution is taken care of by the operating system, or the distributed data are produced and consumed by other stages of a larger application. There are many possible implementations of $(\widetilde{h})_p$; our task is to find an efficient parallel version in case of a distributable homomorphism.

Theorem 8 (Distributed DH [39]) *For a p-partitioned input list,*

$$\left(\widetilde{\oplus \updownarrow \otimes}\right)_p \;=\; \left((\mathit{zip}\,\oplus) \updownarrow (\mathit{zip}\,\otimes)\right)_p \;\circ\; \mathit{map}_p\,(\oplus \updownarrow \otimes) \tag{36}$$

To map the abstraction view (36) onto a hypercube of p processors, we apply equality (35) with $k = \log p$, which yields:

$$\left(\widetilde{\oplus \updownarrow \otimes}\right)_p \;=\; \left(\mathop{\bigcirc}_{d=1}^{\log p} \mathit{swap}_p\, d\; (\mathit{zip}\,(\oplus)\,, \mathit{zip}\,(\otimes)\,) \right) \;\circ\; \mathit{map}_p\,(\oplus \updownarrow \otimes) \tag{37}$$

Program (37) provides a generic, provably correct implementation of the DH skeleton on a p-processor hypercube. It consists of two stages: a sequential computation of the function in all p processors on their blocks simultaneously, and then a sequence of swaps on the hypercube.

Let $T_1(n)$ denote the sequential time complexity of a DH function on a list of length n. Then the first stage of program (37) requires time $T_1(n/p)$. The *swap* stage requires $\log p$ steps, with blocks of size n/p to be sent and received and sequential pointwise computations on them at each step; its time complexity is $O\left((n/p) \cdot \log p\right)$. For functions whose sequential time complexity is $O(n \log n)$, e.g., FFT, the first stage dominates asymptotically, so that program (37) is both time- and cost-optimal.

6.4 Scan as a distributable homomorphism

Let us apply the results on DH from the previous subsection to scan. By expressing scan as a distributable homomorphism, and then applying rule (35), we obtain the following hypercube program for an unbounded number of processors [35]:

$$\mathit{scan}\,(\odot) \;=\; \mathit{map}\;\pi_1 \;\circ\; \mathop{\bigcirc}_{d=1}^{k} (\mathit{swap}\; d\,(\oplus, \otimes)\,) \;\circ\; \mathit{map}\; \mathit{pair} \tag{38}$$

where function pair is defined by (18), and

$$\begin{aligned} (s_1, r_1) \oplus (s_2, r_2) &\overset{\text{def}}{=} (s_1\,,\; r_1 \odot r_2) \\ (s_1, r_1) \otimes (s_2, r_2) &\overset{\text{def}}{=} (r_1 \odot s_2\,,\; r_1 \odot r_2) \end{aligned} \tag{39}$$

Program (38) is the “folklore” implementation [70]. In Fig. 4, it is illustrated by the two-dimensional hypercube which computes *scan* $(+)\;[1, 2, 3, 4]$.

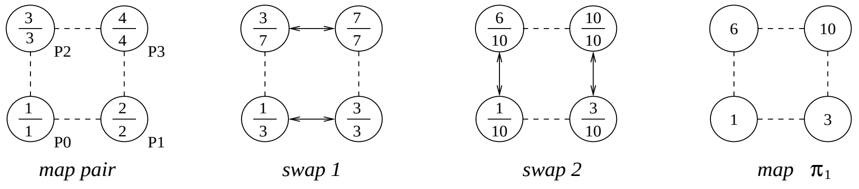

Fig. 4. Computing scan on a hypercube

Unfortunately, for scan on a fixed number of processors, schema (37) yields a suboptimal time complexity, $O\left((n/p) \cdot \log p\right)$. This indicates that both skeletons, homomorphism and DH, are too general for scan. A further specialization, called *localization schema* [35], yields the following program for *scan* $(\odot)$ on p processors:

$$\left(\widetilde{\mathit{scan}\,(\odot)}\right)_p\ x \;=\; \mathit{zip}_p\ (\odot\!\!\lhd)\ (y\,,\,z)\,,$$

$$\text{where}\quad z \;=\; \mathit{map}_p\ (\mathit{scan}\,(\odot))\ x\,,\ \text{and} \tag{40}$$

$$y \;=\; \left(\mathit{map}_p\,\pi_1 \;\circ\; \bigcirc_{d=1}^{\log p} (\mathit{swap}_p\ d\,(\oplus,\otimes)) \;\circ\; \mathit{map}_p\ (\mathit{prepair} \,\circ\, \mathit{last})\right) z$$

with $\oplus, \otimes$ from (39), $a \odot\!\!\lhd\, u \;\overset{\text{def}}{=}\; \mathit{map}\ (a\,\odot)\ u$, and $\mathit{prepair}\, a \;\overset{\text{def}}{=}\; (0_\odot, a)$.

Abstraction view (40) can be expressed directly as a three-stage SPMD program which computes scan on a list x, partitioned blockwise across p processors:

– *Compute* z: Each processor computes scan on its block of x; this yields block z.
– *Compute* y: Each processor creates a pair consisting of $0_\odot$ and the last element of its block z. Then, each processor performs $\log p$ steps, communicating at step i with its neighbour in dimension i of the hypercube and performing computations $\oplus$ and $\otimes$ defined by equations (39). Finally, the first element of the resulting pair is assigned to variable y.
– *Compute the result*: Each processor computes $y \odot\!\!\lhd\, z$ independently.

This is exactly the implementation of scan on a hypercube, with a bounded number of processors used in practice [70]. The time complexity of the first and third stage is $O(n/p)$; the second stage consists of $\log p$ steps, with communications and computations on pairs of elements, which yields time $O(\log p)$. Therefore, the time complexity of program (40) is $O(n/p + \log p)$, the best one can expect for scan on p processors. Our localization schema can be viewed as a constructive version of the compress-and-conquer technique [62]. Divide-and-conquer parallelism on powerlists has been also treated algebraically in [2].

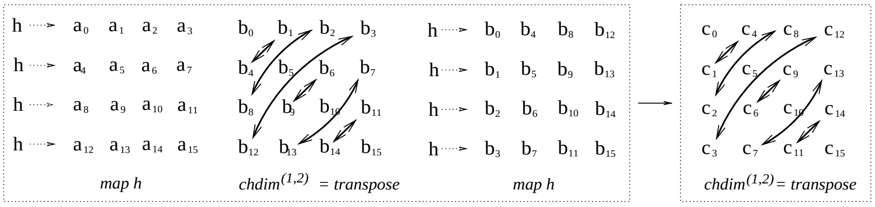

Fig. 5. Implementation (41) in the two-dimensional case, i.e., $d = 1$

6.5 Architecture-independent implementation

In this subsection, based on joint work with Holger Bischof [42], we derive an architecture-independent, generic implementation of DH, which makes considerable use of global communication primitives.

We use the type of d-nested lists with an equal number m of sublists at all levels: $[\alpha]^d_m \overset{\text{def}}{=} [\ldots[\alpha]_m\ldots]_m$. A generalization of *map* denoted by *map*^{l} $h \overset{\text{def}}{=}$ *map* $($*map*$^{l-1}$ $h)$, where $map^0\ h \overset{\text{def}}{=} h$, allows us to map a list function along an arbitrary dimension of a multidimensional list.

For a function h defined on lists, its d-distributed version, denoted $\widetilde{h^{(d)}}$, takes the input and yields the result in the $(d{+}1)$-dimensional form, i.e., $\widetilde{h^{(1)}}$ is an analogue of $(\,\widetilde{h}\,)_p$ from Subsection 6.3, with the restriction of equal length in both dimensions.

Theorem 9 (Distributed Implementation of DH [42])
For an arbitrary DH function h on a list of length 2^l, and arbitrary $d : 0 \le d \le l-1$:

$$\widetilde{h^{(d)}} \;=\; \bigcirc_{i=1}^{d} chdim^{(i,i+1)} \;\circ\; \bigcirc_{i=1}^{d+1} \left(map^{d}\ h \;\circ\; chdim^{(i,d+1)}\right) \qquad (41)$$

where chdim$^{(a,b)}$ swaps dimensions a and b of a nested list.

For an input list of length 2^l, Theorem 9 provides a family of l abstraction views, one for each $d \in [0, l-1]$. Case $d=0$ covers sequential computation. Case $d = 1$ prescribes the 2-dimensional data arrangement, i.e., a matrix shown in Fig. 5. The computation consists of two *map*s and two transpositions since, for $i = 2$, we have *chdim*$^{(2,2)}$ $=$ *id*. Note that the notation $\bigcirc_{i=1}^{d}$ is used in [42] and, consequently, in (41) in the opposite order compared to Subsection 6.3.

The other extreme special case of schema (41) is $d = l-1$: the input list is represented as a virtual l-dimensional hypercube, and the computation proceeds in l steps, similarly to the implementation based on *swap* in Subsection 6.3.

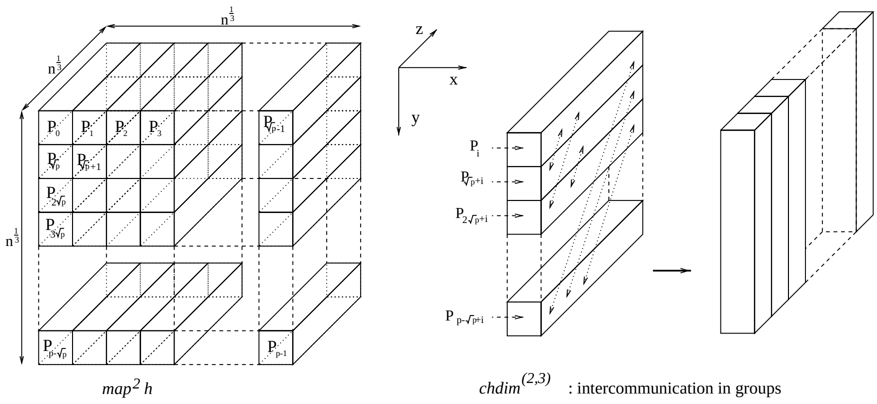

Fig. 6. Three-dimensional case, $d = 2$: two first steps

Schema (41) can be mapped onto a bounded number of processors, p, by partitioning each of the first d dimensions evenly among $p^{\frac{1}{d}}$ processors and leaving the last dimension undistributed. The initial data distribution in p blocks and the redistribution between two consecutive steps in the three-dimensional case are shown in Fig. 6, adapted from [53]. Case $d = \log p$ corresponds to the p-processor hypercube. More details are in [42].

The implementation given by Theorem 9 is always a composition of two stages. For example, in the two-dimensional case these stages are indicated in Fig. 5 by dotted rectangles. The second stage ensures solely that the output is distributed the same way as the input. Following the customary practice, we skip this stage in the performance view, and annotate the altered output data distribution.

The performance view for schema (41) reads as follows:

```
Program DH-Generic (d, h, mypid);
/* Input:(BLOCK,...,BLOCK, * ) */
  For i := d+1 To 1 Do
    ALL-TO-ALL in GROUP (i,d,mypid);
    CALL local-transpose (i);
    CALL map-h;
  End-Do;
/* Output:(BLOCK,...,BLOCK, * ,BLOCK) */
```

Program `DH-Generic` implements the first stage of schema (41). The input and output data distributions are annotated as in Sect. 6.1, with * denoting an undistributed dimension. The program proceeds in a sequence of d+1 stages. The first part of each stage is a transposition of two dimensions, which is implemented by a personalized all-to-all communication, followed by an intraprocessor rearrangement, `local-transpose`. Function `map-h` computes h along the undistributed dimension for the whole block.

For the FFT case study, the experiments with the target program on a 64-transputer network show a competitive speed-up between 20 and 45, depending on the problem size [42].

6.6 Impact on program design

The implementation schemas presented in this section are of a general nature and, at the same time, allow the development of optimal and practically usable parallel algorithms. The key design decisions, e.g., to work on pairs of values for scan, have become the result of a systematic adjustment to the corresponding skeleton, rather than of an intuitive process.

Our case studies demonstrate the use of different schemas. A comparatively laborious process of adjusting scan to the DH skeleton indicates that this skeleton is too general for scan. This suggests that scan could itself be viewed as a primitive skeleton [10], which is the case, e.g., in MPI [47]. For FFT, the mathematical specification can be adjusted systematically to the DH format and then the generic schema (41) can be applied [38,42]. The power of the approach is demonstrated by the fact that all three algorithms used for computing FFT in practice [53] – the two-dimensional and three-dimensional transposition and the binary exchange algorithm – are described in our framework by one abstraction view, schema (41).

7 General aspects of divide-and-conquer

The divide-and-conquer paradigm is often used as a natural way of specifying problems and algorithms, without explicit consideration of their data-parallel execution. This section extends our previous considerations by addressing several aspects of divide-and-conquer not covered by homomorphisms:

– How can the diversity of divide-and-conquer algorithms and their potential for parallelization be classified?
– How can a divide-and-conquer specification be transformed into a parallel abstraction view and, further, into a performance view?
– What is the most suitable processor interconnection topology for an efficient implementation of divide-and-conquer?
– How can the process of making design decisions in the parallelization of a divide-and-conquer specification be captured and understood in the SAT framework?

Although not all of the results presented here fall in the scope of the SAT approach, they have a direct relation either to the stage-based programming model or to formal transformations in BMF. The presentation is kept more cursory than in the previous sections.

7.1 A classification of divide-and-conquer

Let us sketch the classification of divide-and-conquer proposed in [41]:

Static. We identify a class of divide-and-conquer algorithms whose control structure can be determined at compile time; we call them *SDC*-algorithms (Static Divide-and-Conquer). This class is described by a higher-order function whose parameters are the division degree, the recursion depth and the tuples of both the divide and the conquer function. Thus, *SDC*-algorithms allow different divide and/or conquer functions to be applied in the course of a computation.

Binary. The most well known and widely used divide-and-conquer schema is the binary, symmetric version of the *SDC*-skeleton which, in our classification, is called DC.

One-Sided. The separation of data rearrangements from a DC algorithm, which we have already used for DH, can be captured in two specialized schemas. For instance, if the divide phase of an algorithm is a pure data arrangement, then we view this algorithm as a C-algorithm, for "Conquer (without Divide)". The list homomorphisms of Sect. 3 are computable by C–algorithms: their divide stage can be captured as a partitioning of the input list, according to the promotion property.

Nested. Divide-and-conquer algorithms may have a nested control structure, with their divide or their conquer phase being a divide-and-conquer algorithm itself. For example, the problem of parsing many-bracket languages is a so-called nested almost-homomorphism (mentioned in Subsection 4.3): it is a C-algorithm, whose conquer function is again a C-function; we denote this class by $C(C)$. Another well-known example is the bitonic sort [1,63], which belongs to the class $C(D)$.

An alternative classification of divide-and-conquer in Haskell is presented in [51].

7.2 Mutually recursive divide-and-conquer

This subsection deals with a case which is relevant in practice: mutually recursive divide-and-conquer specifications, with numerical integration on sparse grids as a case study.

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

$$f_i(x) \;=\; \texttt{if}\; p_i(x) \;\texttt{then}\; b_i(x) \;\texttt{else}\; E_i(g_i,\, f,\, x) \;\texttt{fi} \quad (i = 1, \ldots, n) \tag{42}$$

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

We view equations (42) as a specification for computing one of the functions f_i, say, f_1. Our goal is to generate a parallel program which, given a particular domain parameter *input*, computes f_1(*input*). Specification (42) includes special cases that have been studied extensively in the literature. Systolic algorithms are often specified in this format, where $\tau = \mathbb{Z}^m$ and the shifts are of the form $\varphi(i) = i + a$, for some fixed $a \in \mathbb{Z}^m$. These and other restrictions enable the use of linear algebra and linear programming for the synthesis of a parallel program [55]. The conventional divide-and-conquer algorithms correspond to system (42) with a single function f_1 and two recursive calls of it [43].

To develop a SAT abstraction view for specification (42), we construct an abstract type of trees, which captures the communication structure of the specification, and choose an arbitrary tree of this type, such that each of its nodes can be mapped onto a processor. A sequence of BMF transformations [43] leads to a function F such that, if all processors compute F simultaneously and *input* is available at the root processor, then the result obtained at the root is the desired value $f_1(\textit{input})$. Therefore, we obtain an SPMD performance view on a tree of processors.

The following, efficiency-related aspects are studied in [43]: (1) an additional, intraprocessor level of parallelism is implemented by multithreading; (2) the redundant computations of common values on different processors are eliminated by introducing additional interprocessor communication; (3) the load balance is improved using additional transformations at the functional level.

Our experiments for the case study of numerical integration on a 64-transputer machine yielded an efficiency ranging from 0.6 to 0.9 (ratio speed-up/number of processors).

7.3 $\mathcal{N}$-graphs

In this subsection, we present a new interconnection topology, the $\mathcal{N}$-graph, whose purpose is to implement efficiently the binary divide-and-conquer on a bounded number of processors.

In the literature, mainly two topologies with a bounded number of nodes have been considered [13, 48, 70]. The complete binary tree has the drawback that the load is not balanced: only half of the processors are busy with computation of the coarse-grain base case; this entails a 50% loss of speedup. An alternative, the binomial tree, has a better load balance, but the node degree becomes non-constant, which makes the topology non-scalable.

In [37], we introduce a new topology, the *$\mathcal{N}$-graph*, which combines the advantages of both binary and binomial trees, i.e., balanced load, locality, and a constant node degree. Locality means that the communications in the divide and combine stages happen only between neighbours in the graph.

Informally, the $\mathcal{N}$-graph can be viewed as the result of augmenting the complete binary tree with one auxiliary node and connecting every right leaf of the tree to its inorder successor; the last leaf is connected to the auxiliary node.

Figure 7 (the upper part) shows the $\mathcal{N}$-graph with 16 nodes, whose additional edges are dashed and whose additional node is shaded.

As proved in [37], the node degree of an $\mathcal{N}$-graph is not greater than 4 (e.g., the left son of the root has four edges including the dashed one), and this fits nicely on, e.g., transputer networks. The balanced load and locality are demonstrated in the lower part of Fig. 7: (a) shows two leaves of a complete binary tree, with a common predecessor and a task to be executed at each leaf; (b) illustrates that, in an $\mathcal{N}$-graph, every leaf can divide its task into two sub-tasks, keep one of them and dispatch the other subtask to its ancestor; (c) shows that after this transformation, every node of the $\mathcal{N}$-graph is responsible for exactly one subtask.

This idea is realized in the SAT framework by transforming a general divide-and-conquer schema into an abstraction view on an $\mathcal{N}$-graph. The abstraction view is then transformed into a performance view, i.e., an SPMD program with message passing. The target program behaves similarly to the function F of the previous subsection, with the only difference in the leaf nodes: rather than performing their tasks, the leaves divide them into two parts and redistribute one of them to their neighbours in the $\mathcal{N}$-graph as illustrated in Fig. 7(b)-(c).

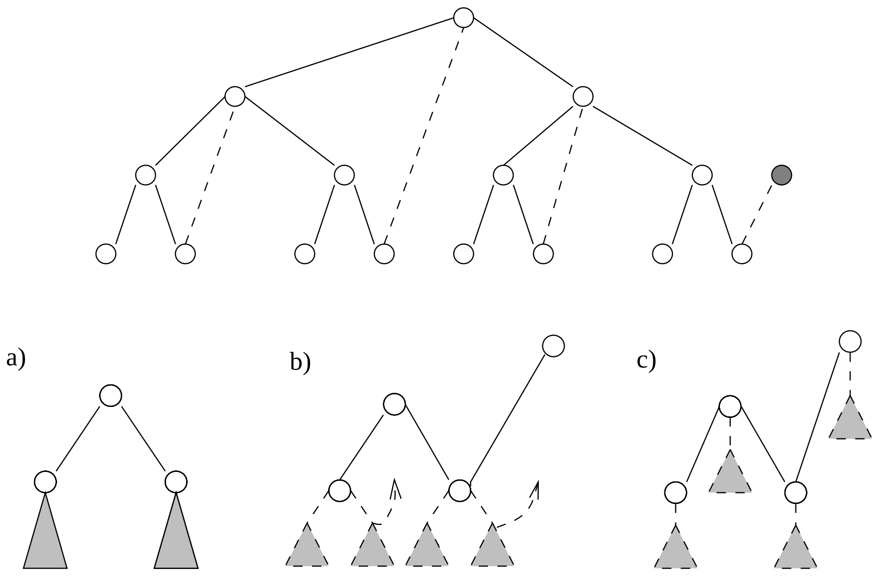

Fig. 7. Upper: $\mathcal{N}$-graph; lower: distribution of tasks amongst processors

We have studied the performance of the $\mathcal{N}$-graph topology for a case study of the mergesort on a 64-node *Parsytec* transputer system under OS Parix [69]. Both the complete binary tree and the $\mathcal{N}$-graph topology were implemented as virtual topologies. Even with the $\mathcal{N}$-graph, simulated on the physical two-dimensional mesh of the machine, we obtain a speed-up close to 2, with respect to the complete binary tree. While this is an upper bound on the expected improvement, it provides, in practice, reasonable gains at low cost.

Though quite simple, the idea of $\mathcal{N}$-graph seems to be new. There are explicit differences to the, at first glance, similar concept of *threaded trees* [58]. In particular, our topology introduces the additional node and also only half as many additional edges as compared with threaded trees. Moreover, the $\mathcal{N}$-graph is not a tree at all, due to the loops introduced.

7.4 A case study: polynomial multiplication

In this subsection, we apply the SAT approach to a case study: we take the straight-forward cumulative sum algorithm for polynomial multiplication and go through all development steps, from the mathematical specification towards the parallel target program. We pursue two goals: (1) to make all design decisions systematically, using formal transformations and performance considerations within SAT, rather than making the decisions *ad hoc*,

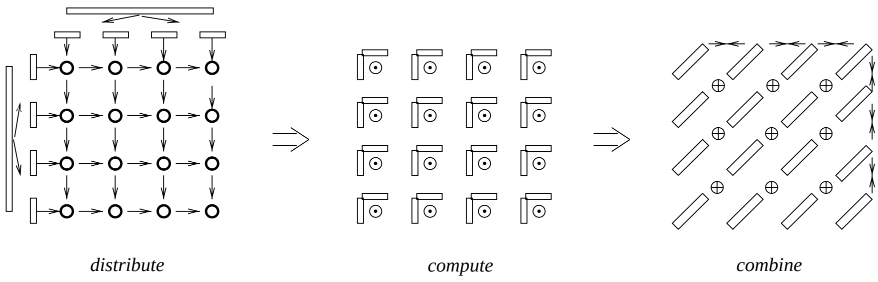

Fig. 8. Three stages of the polynomial product

and (2) to compare the resulting performance with the results achieved by another popular approach to parallelization, systolic design.

We start from the mathematical specification of the product of two polynomials, represented by the lists of their coefficients. Eight design decisions [33] result in the three-stage program structure shown in Fig. 8. We sketch the design decisions and explain how they are made in SAT (see [33, 56] for more details):

1. Partition both lists of coefficients into p segments and apply the standard SAT implementation described in the previous section.
2. Compose the program structure of three stages: $distribute$, $compute$ and $combine$ as prescribed by the standard implementation schema (32).
3. Employ p^2 processors, due to the two-dimensional nature of the available homomorphism.
4. Broadcast input data segments between the processors in order to exploit the two-dimensional parallelism.
5. Assume distributed input data, thus arriving at a C-algorithm (see Subsection 7.1) and liberating the program structure from an explicit data distribution step.
6. Based on the performance prediction, organize the reduction step in the $combine$ stage along the diagonals.
7. As a consequence of the previous design decision, connect the processors as a mesh of trees with diagonal trees [54].
8. Assume distributed output, and implement the concluding step of $combine$ as a pairwise exchange between neighbouring processors.

The obtained performance view reads as follows:

```
/* Input: A, B (BLOCK) */
BCAST (A) in ROW;
BCAST (B) in COLUMN;
CALL PolyProd (A,B);
REDUCE (+) in DIAGONAL;
EXCHANGE-NEIGHBOURS;
```

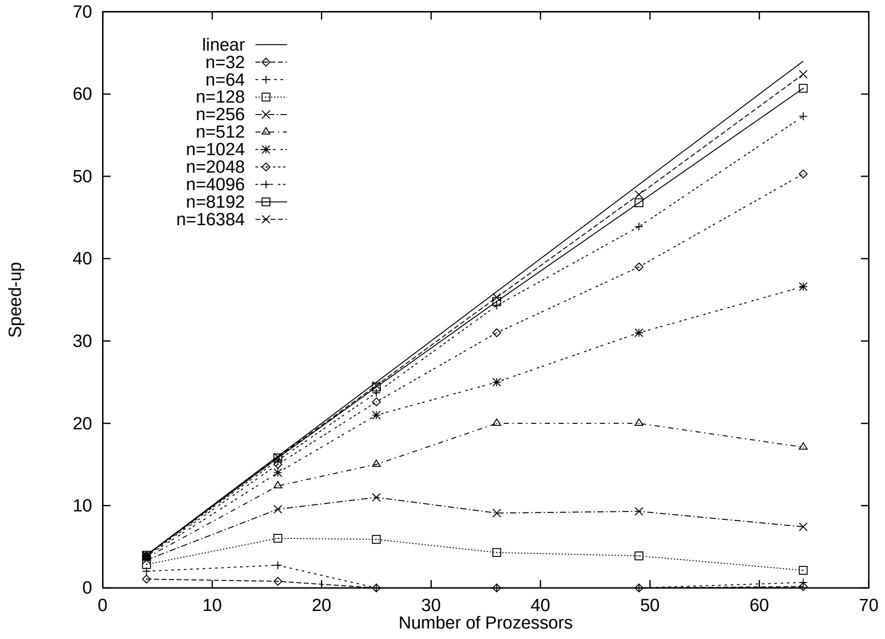

Fig. 9. Speed-up of the polynomial product

```
/* Output: C (BLOCK) */
```

Here, sequential program `PolyProd` is called in each processor for multiplying two chunks of the input polynomials. We can choose the number of processors between 1 and n, where n is the length of the polynomials. If $p = n/\sqrt{\log n}$ then the optimal time complexity $t = O(\log n)$ is achieved on $p^2 = (n^2/\log n)$ processors. This cost-optimal solution is not possible in the systolic setting. In practice, the number of processors is bounded, and the problem size n is relatively large. Then the term $(n/p)^2$ dominates in the expression of the time complexity, which guarantees a so-called scaled linear speed-up [70].

A detailed comparison of some systematic parallelization approaches on the example of the polynomial multiplication can be found in [57].

Experiments on our Parsytec GCel64 showed quite good performance for sufficiently large polynomials; see Fig. 9.

7.5 Impact on program design

A classification of divide-and-conquer helps in choosing the right set of algorithmic skeletons for different application areas. E.g., the usefulness of our one-sided divide-and-conquer has been demonstrated recently for a bitonic sort [17].

A particular algorithm can be adjusted to the general format (42) in more than one way [43]. E.g., a different adjustment for the integration example changes the performance view from a heterogeneous to a binary communication tree, which can be implemented more efficiently on most multiprocessors.

The rôle of the communication topology in program design is demonstrated by the $\mathcal{N}$-graph as an idealized structure for binary divide-and-conquer, and by the mesh of trees with diagonal trees for the polynomial product. The latter case study also demonstrates that design decisions, which are usually made by a software designer based on his or her intuition and experience, can be explained and substantiated formally, without "handwaving" arguments.

8 Conclusion and outlook

The diversity of parallel computers and the complexity of their software are calling for portable, tractable and efficiently implementable parallel programming models and languages. The SAT approach is an attempt to propagate the use of higher-order programming constructs as the building blocks of such models.

An analogy can be drawn with the historical development of sequential programming, in which simple, relatively unstructured mechanisms, closely tied to the underlying architecture, have given way to more powerful, structured and abstract concepts. Similar progress in the parallel setting should raise the level of abstraction from models with explicit communication to a world in which complex patterns of computation and interaction are combined and presented as parameterized program-forming constructs.

The SAT approach focuses on two orthogonal aspects of parallel programming: abstraction and performance. They are reconciled within a programming model, which recasts a traditional parallel composition of sequential processes into a sequential composition of actions on parallel objects. The price paid is a possible loss in expressiveness and performance. Let us demonstrate how this is outweighed by major gains for the two main communities dealing with parallelism.

8.1 Gains for application programmers

Application programmers gain from abstraction, which hides much of the complexity of managing massive parallelism. They are provided with a set of basic abstract skeletons, whose parallel implementations have a well-understood behavior and predictable efficiency. To express an application in terms of skeletons is usually simpler than to develop a low-level parallel program for it. The CS-method demonstrates an opportunity of automatically supporting this task for the homomorphism skeleton.

This higher-order approach changes the program design process in several aspects. First, it liberates the user from the practically unmanageable task of making the right design decisions based on numerous and mutually influencing low-level details of a particular application and a particular machine. Second, by providing standard implementations, it increases the confidence in the correctness of the target programs, for which traditional debugging is too hard to be practical on massively parallel machines. Third, it offers predictability instead of an *a posteriori* approach to performance evaluation, in which a laboriously developed parallel program may have to be abandoned because of inadequate efficiency. Fourth, it provides semantically sound methods for program composition and refinement, which open new perspectives in software engineering (in particular, in reusability). Last but not least, abstraction, i.e., going from the specific to the general, gives new insights into the basic principles of parallel programming.

An important feature of the SAT approach is that the underlying formal framework – the Bird-Meertens formalism – remains largely invisible to the application programmers. The programmers are given a set of methods for instantiating, composing and implementing diverse homomorphic skeletons, but the BMF-based development of these methods is delegated to the community of implementers.

8.2 Gains for implementers

This group includes the experts which develop algorithmic skeletons and their implementations, as well as the implementers of the basic parallel programming tools like compilers, communication libraries, etc. The main concern of this community is performance.

The SAT approach is an example of a programming model developed largely independently of the parallel execution model. By abstracting from the details of a particular machine, we unavoidably give up some portion of potential program efficiency. However, we believe strongly in the feasibility of this approach, for two reasons: (1) there are positive results in the structured sequential programming, where programs are compiled to codes which are often faster than programs with `goto` or hand-written assembler versions; (2) performance estimation and machine experiments with structured parallel solutions demonstrate their competitive performance.

Even more important is the fact that possible losses in absolute performance are traded for portability and ease of programming. The design of parallel skeletons becomes simpler due to the structure imposed on both skeleton languages (abstraction view) and target languages (performance view). The structured performance view simplifies also the task of the implementers of parallel software: they can concentrate on a standard set of global operations which have to be implemented on each target architecture. This increases the chance of finding high-performance solutions, which are portable across different architectural classes.

Thus, the task of the implementer can be formulated more precisely, and alternative solutions can be compared more systematically than in the case of an unrestricted variety of parallel architectures, programming styles and implementation tricks. This opens the way for a coming-of-age of the implementation area and for a gradual transition from largely *ad hoc* implementation efforts to an integrated compiler technology for parallel machines.

8.3 Directions of further research

The promising results in the field of parallel programming with higher-order constructs raise issues which deserve further study.

Expressiveness. The question is: what might be the right set of types and operators (or does such a bounded set even exist) and what is the appropriate language framework for expressing skeletal schemas? Whereas the first skeletons in the literature were chosen with an efficient implementation in mind, it is likely that, with growing experience, designers will want to include skeletons with greater expressive power, whose implementations are not obvious.

Cost calculus. The challenge is to build a tractable, compositional cost calculus. The fundamental problem of skeletons is that a common yardstick by which to evaluate them has not been found as of yet. A promising direction is to identify and study the relevant properties of cost-based transformation systems, such as confluence and convexity as proposed recently by Skillicorn [18].

Techniques. Practical tools should help a programmer to decide what to do next in a derivation and what would be the right level of abstraction for making a particular design decision. Possible approaches are based on abstract parallel machines [67] or on collective operations [31, 45]. Two other important issues are portability and data distribution, for which approaches like shapes [52] and data distribution algebras [77] have been suggested.

Applications. Irregular computations remain a challenge; progress in this direction is achieved in the framework of parallel abstract data types at Imperial College [79] and in the Proteus project [68].

Component-based programming. In the SAT approach, we concentrate mostly on programming-in-the-small. The major way of tackling software complexity should be by means of combining predefined components, which range from straight-forward libraries to elaborate application frameworks. Programming-in-the-large, which still receives most attention from industry, should be addressed also by academia.

8.4 The bottom line

Parallel programming is and will remain a non-trivial task, requiring a fair amount of ingenuity from the user. The complex trade-offs often reduce the design process to a black art. The challenge is to support program designers in their creative activity by providing a formally sound, practically useful notation, together with tools for making design decisions. In well-understood cases, the user will be provided with exact rules, or the design process can be mechanized entirely.

The results presented in this survey illustrate a way of combining abstraction and performance, in order to make the design process tractable and improve the quality of the resulting programs. The higher-order, formally based approach to parallelism is finding an increasing number of supporters, and a research community has been forming recently [18,19,40].

Acknowledgements. Many people have helped with advice during the development of the SAT approach; their names are mentioned in the acknowledgements of the corresponding technical papers. The first author is especially indebted to his coauthors Alfons Geser and Holger Bischof. The three anonymous referees helped to imrove the quality of presentation. The DFG-supported project RecuR2 has been a source of inspiration. We gratefully acknowledge the financial support received from the EU (INTAS Programme) and the DAAD (exchange programme ARC with the Universities of Edinburgh and Glasgow, and exchange programme VIGONI with the University of Pisa).

References

1. Achatz, K., Schulte, W.: K Architecture independent massive parallelization of divide-and-conquer algorithms. In: B. Möller (ed), Mathematics of Program Construction (MPC'95), Lecture Notes in Computer Science 947, pp. 97–127, Berlin Heidelberg New York: Springer 1995
2. Achatz, K., W.Schulte: Massive parallelization of divide-and-conquer algorithms over powerlists. Sci Comput Program, 26:59–78 (1996)
3. Bacci, B., Danelutto, M., Orlando, S., Pelagatti, S., Vanneschi, M.: P^3L: A structured high level programming language and its structured support. Concurrency: Practice and Experience, 7(3):225–255 (1995)
4. Bacci, B., Gorlatch, S., Lengauer, C., Pelagatti, S.: Skeletons and transformations in an integrated parallel programming environment. In: Parallel Computing Technologies (PaCT-99), LNCS 1662, pp. 13–27. Berlin Heidelberg New York: Springer 1999

5. Barnard, D., Schmeiser, J., Skillicorn, D.: Deriving associative operators for language recognition. Bulletin of the EATCS, 43:131–139 (1991)
6. Bauer, F., (ed): The Munich Project CIP. Lecture Notes in Computer Science 183 Berlin Heidelberg New York: Springer, 1985
7. Bentley, J.: Programming pearls. Comm. ACM, 27:865–871 (1984)
8. Bird, R.: Lectures on constructive functional programming. In: Broy, M., (ed): Constructive Methods in Computing Science, NATO ASI Series F: Computer and Systems Sciences. Vol.55, pp. 151–216. Berlin Heidlberg New York: Springer, 1988
9. Bird, R.: Algebraic identities for program calculation. Comput J., 32(2):122–126 (1989)
10. Blelloch, G.: Scans as primitive parallel operations. IEEE Trans. Comput, TC-38(11):1526–1538, (1989)
11. Botorog, G., Kuchen, H.: Efficient parallel programming with algorithmic skeletons. In: Bougé, L. et al. (eds), Euro-Par'96: Parallel Processing, Lecture Notes in Computer Science 1123, pp. 718–731, Berlin Heidelberg New York: Springer 1996
12. Bougé, L.: The data-parallel programming model: a semantic perspective. In: A.Darte, G.-R. Perrin, (eds), The Data-Parallel Programming Model, Lecture Notes in Computer Science 1132, pp. 4–26, Berlin Heidelberg New York: Springer 1996
13. Brinch-Hansen, P.: Model programs for computational science: A programming methodology for multicomputers. Concurrency: Practice and Experience, 5:407–423 (1993)
14. Burstall, R., Darlington, J.: A transformation system for developing recursive programs. J. ACM, 25(1):44–67 (1977)
15. Cai, W., Skillicorn, D.: Calculating recurrences using the Bird-Meertens formalism. Parallel Process. Lett., 5(2):179–190 (1995)
16. Cole, M.: Parallel programming with list homomorphisms. Par. Proc. Lett., 5(2):191–204 (1994)
17. Cole, M.: On dividing and conquering independently. In: C.Lengauer, Griebl, M., Gorlatch, S. (eds): Parallel Processing. Euro-Par'97, Lecture Notes in Computer Science 1300, pp. 634–637. Berlin Heidelberg New York: Springer 1997
18. Cole, M., Gorlatch, S., Lengauer, Skillicorn, D, C., (eds): Theory and Practice of Higher-Order Parallel Programming. Dagstuhl-Seminar Report 169, Schloß Dagstuhl. 1997
19. Cole, M., Gorlatch, S., Prins, J., Skillicorn, D. (eds): High Level Parallel Programming: Applicability, Analysis and Performance. Dagstuhl-Seminar Report 238, Schloß Dagstuhl. 1999
20. Cole, M.I.: Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computation. London: Pitman, 1989
21. Comon, H., Haberstrau, M., Jouannaud, J.-P., Syntacticness, cycle-syntacticness, and shallow theories. Inform. Comput., 111:154–191 (1994)
22. Culler, D. et al.: LogP: Towards a realistic model of parallel computation. In: Proc. 4th ACM Conf. on Principles and Practice of Parallel Prog. (PPoPP'93), pp. 1–12. San Diego, CA, USA, ACM Press, 1993
23. Darlington, J. et al.: Parallel programming using skeleton functions. In: Bode, A., Reeve, M., Wolf, G., (eds), Parallel Architectures and Languages Europe (PARLE'93), Lecture Notes in Computer Science 694, pp. 146–160. Berlin Heidelberg New York: Springer 1993
24. Elrad, T., Francez, N.: Decomposition of distributed programs into communication-closed layers. Sci. Comput. Program, 2:155–173 (1982)
25. Foster. I.: Designing and Building Parallel Programs. Reading, Ma: Addison-Wesley (1995)

26. Fraus, U., Hußmann, H.: Term induction proofs by a generalization of narrowing. In: Rattray, Clark, C, R.G.,, (eds), The Unified Computation Laboratory — Unifying Frameworks, Theories and Tools. Oxford: Clarendon Press 1992
27. Geerling, M.: Transformational Development of data-parallel algorithms. PhD thesis, Katholieke Universiteit Nijmegen, 1996
28. Geser, A., Gorlatch, S.: Parallelizing functional programs by generalization. In: Hanus, M., Heering, J., Meinke, K. (eds): Algebraic and Logic Programming (ALP'97), Lecture Notes in Computer Science 1298, pp. 46–60, Berlin Heidelberg New York: Springer 1997
29. Gibbons, J.: The third homomorphism theorem. J. Funct. Program., 6(4):657–665 (1996)
30. Goodrich, T.: Communication-efficient parallel sorting. In: Proc. 28th ACM Symp. on Theory of Computing (STOC '96). Philadephia, PA, USA: ACM Press 1996
31. Gorlatch, S.: Towards formally-based design of message passing programs. IEEE Trans. on Software Engineering. To appear
32. Gorlatch, S.: Constructing list homomorphisms. Technical Report MIP-9512, Universität Passau, August 1995 Available at http://www.fmi.uni-passau.de/cl/papers/Gor97b.html.
33. Gorlatch, S.: From transformations to methodology in parallel program development: a case study. Microprocessing and Microprogramming, 41:571–588 (1996)
34. Gorlatch, S.: Stages and transformations in parallel programming. In: M.Kara et al., (eds), Abstract Machine Models for Parallel and Distributed Computing, pp. 147–162. Amsterdam Oxford Washington DC Tokyo: IOS Press 1996
35. Gorlatch, S.: Systematic efficient parallelization of scan and other list homomorphisms. In: Bougé, L., Fraigniaud, P., Mignotte, A., Robert, Y., (eds): Euro-Par'96: Parallel Processing, Vol.II, Lecture Notes in Computer Science 1124, pp. 401–408, Berlin Heidelberg New York: Springer 1996
36. Gorlatch, S., Systematic extraction and implementation of divide-and-conquer parallelism. In: Kuchen, H., Swierstra, D. (eds): Programming Languages: Implementation, Logics and Programs (PLILP'96), Lecture Notes in Computer Science 1140, pp. 274–288, Berlin Heidelberg New York: Springer 1996
37. Gorlatch, S.: N-graphs: scalable topology and design of balanced divide-and-conquer algorithms. Parallel Comput., 23(6):687–698 (1997)
38. Gorlatch, S.: Optimizing compositions of scans and reductions in parallel program derivation. Technical Report MIP-9711, Universität Passau, May 1997. Available at http://www.fmi.uni-passau.de/cl/papers/Gor97b.html
39. Gorlatch, S.: Extracting and implementing list homomorphisms in parallel program development. Sci. Comput. Program. 33(1):1–27 (1998)
40. Gorlatch, S. (ed).: First Int. Workshop on Constructive Methods for Parallel Programming (CMPP'98), Techreport MIP-9805. University of Passau, May 1998
41. Gorlatch, S.: Programming with divide-and-conquer skeletons: an application to FFT. J. Supercomputing, 12(1-2):85–97 (1998)
42. Gorlatch, S., Bischof, H.: A generic MPI implementation for a data-parallel skeleton: Formal derivation and application to FFT. Parallel Processing Letters, 8(4):447–458 (1998)
43. Gorlatch, S., Lengauer, C.: Parallelization of divide-and-conquer in the Bird-Meertens formalism. Formal Asp. Comput., 7(6):663–682 (1995)
44. Gorlatch, S., Lengauer, C.: (De)Composition rules for parallel scan and reduction. In: Proc. 3rd Int. Working Conf. on Massively Parallel Programming Models (MPPM'97), pp. 23–32. IEEE Computer Society Press, 1998 Available at http://brahms.fmi.uni-passau.de/cl/papers/GorLe97c.html

45. Gorlatch, S., Wedler, C., Lengauer, C.: Optimization rules for programming with collective operations. In: Atallah, M., (ed), 13th Int. Parallel Processing Symp. & 10th Symp. on Parallel and Distributed Processing (IPPS/SPDP'99), pp. 492–499. IEEE Computer Society Press, 1999
46. Grant-Duff, Z., Harrison, P.: Parallelism via homomorphisms. Parallel Process. Lett., 6(2):279–295 (1996)
47. Gropp, W., Lusk, E., Skijellum, A.: Using MPI: Portable Parallel Programming with the Message Passing. MIT Press 1994
48. Gupta, A., Hambrusch, S.: Load balanced tree embeddings. Parallel Computi., 18:595–614 (1992)
49. Heinz, B.: Lemma discovery by anti-unification of regular sorts. Technical Report 94-21, TU Berlin, May 1994
50. Herrmann, C.A., Lengauer, C.: On the space-time mapping of a class of divide-and-conquer recursions. Parallel Process. Lett., 6(4):525–537 (1996)
51. Herrmann, C.A., Lengauer, C.: Parallelization of divide-and-conquer by translation to nested loops. J. Functional Programming, 9(3):279–310 (1999)
52. Jay, C., Clarke, D., Edwards, J.: Exploiting shape in parallel programming. In: Proc. 2nd IEEE Int. Conf. on Algorithms and Architectures for Parallel Processing, pp. 295–302, IEEE Press (1996)
53. Kumar, V. et al.: Introduction to Parallel Computing. Redwood City, CA, USA: Benjamin/Cummings 1994
54. Leighton, F.T.: Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.Los Alosca: Morgan Kaufmann 1992
55. Lengauer, C.: Loop parallelization in the polytope model. In: Best, E. (ed), CONCUR '93, Lecture Notes in Computer Science 715, pp. 398–416. Berlin Heidelberg New York: Springer 1993
56. Lengauer, C., Gorlatch, S., Herrmann, C.: The static parallelization of loops and recursions. In: Proc. 11th Int. Symp. on High Performance Computing Systems (HPCS'97), pp. 3–15, 1997
57. Lengauer, C., Gorlatch, S., Herrmann, C.: The static parallelization of loops and recursions. J. Supercomputing, 11:333–353 (1997)
58. Lewis, H., Denenberg, L.: Data structures and their algorithms. New York: Harper Collins 1991
59. McColl, W.F.: Scalable computing. In: van Leeuwen, J. (ed), Computer Science Today, Lecture Notes in Computer Science 1000, pp. 46–61, Berlin Heidelberg New York: Springer 1995
60. Meijer, E., Fokkinga, M., Paterson, R.: Functional programming with bananas, lenses, envelopes and barbed wire. In: Hughes, J. (ed), Proc. 5th ACM Conf. on Functional Prog. and Comp. Architecture (FPCA'91), pp. 124–144, Berlin Heidelberg New York: Springer 1991
61. Misra, J.: Powerlist: a structure for parallel recursion. ACM TOPLAS, 16(6):1737–1767 (1994)
62. Mou, Z.G., Constantinescu, C., Hickey, J, T.: Divide-and-conquer on three-dimensional meshes. In: Joosen, E.Milgrom, W., (eds), Parallel Computing: From Theory to Sound Practice, pp. 344–355. IOS Press 1992
63. Mou, Z.G., Hudak, P.: An algebraic model for divide-and-conquer algorithms and its parallelism. J. Supercomputing, 2(3):257–278 (1988)
64. Musser, D., Stepanov, A.: Algorithm-oriented generic libraries. Software – Practice and Experience, 24(7):623–642 (1994)

65. Nash, J., Dyer, M., Dew, P.: Designing practical parallel algorithms for scalable message passing machines. In: Cook, B. et al. (eds), Transputer Applications and Systems ’95, pp. 529–541. IOS Press 1995
66. O’Donnell, O.: A correctness proof of parallel scan. Parallel Process. Lett., 4(3):329–338 (1994)
67. O’Donnell, J., Rünger, G.: A methodology for deriving parallel programs with a family of abstract parallel machines. In: Lengauer, C., Griebl, M., Gorlatch, S. (eds), Parallel Processing. Euro-Par’97, Lecture Notes in Computer Science 1300, pp. 661–668, Berlin Heidelberg New York: Springer 1997
68. Palmer, D., Prins, J., Westfold, S.: Work-efficient nested data-parallelism. In: Proc. Fifth Symp. on the Frontiers of Massively Parallel Processing (Frontiers’ 95), pp. 186–193. IEEE Computer Society Press 1995
69. Parix. Software Documentation 1.2. Parsytec Computer GmbH, 1993
70. Quinn, M.J.: Parallel Computing. McGraw-Hill 1994
71. Rauber, T., Rünger, G.: Integrating library modules into special purpose parallel algorithms. In: Agha, G., Russo, S. (eds), Proc. 2nd Int. Workshop on Software Engineering for Parallel and Distributed Systems (PDSE’97), pp. 162–173. IEEE Computer Society Press 1997
72. Sedgewick, R.: Algorithms. Reading, MA, USA: Addison-Wesley, second edition, 1988
73. Skillicorn, D.: The Bird-Meertens Formalism as a parallel model. In: Kowalik, L. Grandinetti, J. (eds), Software for Parallel Computation, Vol 106 of NATO ASI Series F, pp. 120–133. Berlin Heidelberg New York: Springer 1993
74. Skillicorn, D.: Foundations of Parallel Programming. Cambridge University Press, 1994
75. Skillicorn, D., Cai, W.: A cost calculus for parallel functional programming. J. Parallel Distrib Comput, 28:65–83 (1995)
76. Smith, D.: Applications of a strategy for designing divide-and-conquer algorithms. Sci. Comput. Program., 8(3):213–229 (1987)
77. Südholt, M.: Data distribution algebras — a formal basis for programming using skeletons. In: Olderog, E.-R. (ed), Programming Concepts, Methods and Calculi (PROCOMET’94), pp. 19–38. Amsterdam Elsevier, 1994
78. Swierstra, D., deMoor, O.: Virtual data structures. In: Möller, B., Partsch, H., Schuman, S. (eds), Formal Program Development, Lecture Notes in Computer Science 755, pp. 355–371. Berlin Heidelberg New York: Springer 1993
79. To, H.W.: Optimising the Parallel Behaviour of Combinations of Program Components. PhD thesis, Imperial College, 1995
80. Vishkin, U.: A case for the PRAM as a standard programmer’s model. In: auf der Heide, M. et al. (eds), Parallel Architectures and their Efficient Use, Lecture Notes in Computer Science 678, pp. 11–19. Berlin Heidelberg New York: Springer 1993
81. Wedler, C., Lengauer, C.: On linear list recursion in parallel. Acta Informatica, 35(10):875–909 (1998)