Abstraction and Performance in the Design of Parallel Programs

Der Fakultät für Mathematik und Informatik
der Universität Passau vorgelegte
Zusammenfassung der Veröffentlichungen
zur Erlangung der venia legendi

von

Dr. Sergei Gorlatch

Passau, Juli 1997

Contents

1 Introduction **2**

2 Outline of the SAT Approach **6**

2.1 Performance View . . . 6

2.2 Abstraction View . . . 7

2.3 Design in SAT: Stages and Transformations . . . 9

2.4 SAT and Homomorphisms . . . 11

3 List Homomorphisms **12**

4 Extraction and Adjustment **14**

4.1 The CS-Method . . . 14

4.2 Mechanizing the CS-Method . . . 17

4.3 Almost-Homomorphisms: the MSS Problem . . . 19

5 Composition of Homomorphisms **21**

5.1 Rules of Composition . . . 21

5.2 Derivation by Transformation: MSS Revisited . . . 22

6 Implementing Homomorphisms **24**

6.1 Standard Implementation . . . 25

6.2 Distributable Homomorphisms (DH) . . . 26

6.3 Hypercube Implementation . . . 27

6.4 Scan as Distributable Homomorphism . . . 29

6.5 Architecture-Independent Implementation . . . 30

7 General Aspects of Divide-and-Conquer **33**

7.1 A Classification of Divide-and-Conquer . . . 33

7.2 Mutually Recursive Divide-and-Conquer . . . 34

7.3 $\mathcal{N}$-graphs . . . 35

7.4 A Case Study: Polynomial Multiplication . . . 37

8 Conclusion and Outlook **39**

Acknowledgements **43**

References **44**

Parallelism is not obvious.
If you think it is, I can sell you a bridge.

"Drop in" parallel speed-up
is the Holy Grail of high performance computing.
The Holy Grail of programming and software engineering
has been "automatic programming". If you believe
we have either, then I have a big bridge to sell you.

Eugene N. Miya,
NASA Ames Research Center

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 parallel computer development. Advances in parallel programming technology have not kept pace with leaps in the complexity demanded 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 particular families of parallel architectures.

However, a general approach is lacking: parallel programming requires specific technical 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 from the programmer, 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.

Our general goal is *to reconcile performance and abstraction* in parallel programming. Our chosen context is that of functional programming: the underlying semantic simplicity makes it an ideal setting for abstract reasoning about sophisticated programs, and the increasing practical use of functional languages in various fields demonstrates their performance potential.

The development of the last decade in the theory and practice of parallel programming exhibits a tendency towards closing the gap between the requirements of performance and abstraction. Let us mention briefly the main trends on both sides – performance and abstraction – leaving a more detailed discussion of particular approaches for the next section.

On the performance side, there is a trend towards parallel programs which are better structured and, thus, more predictable and portable: (1) new parallel system libraries are gaining a wide acceptance; their purpose is to standardize the set of parallel primitives, which makes programs portable over different parallel architectures; (2) research in parallel algorithms is increasingly aiming at the well-structured computational schemas for classes of problems, rather than at individual, specific solutions; (3) the predictability of the algorithm behaviour on different parallel configurations is getting progressively more attention 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 correctness-preserving transformations on them; (2) these calculi are gradually being equipped with cost information and methods of predicting the performance of the target program; (3) new, so-called "bridging" models of parallelism abstract the architectural details of parallel machines, and establish a structured way of programming and performance evaluation.

This development creates the necessary preconditions for reconciling the requirements of performance and abstraction and, thereby, for advancing towards a general methodology of program design for parallel machines.

Our approach, called SAT (Stages And Transformations), builds on recent progress in the theory and practice of parallelism. It addresses the goals of abstraction and performance in the design of parallel programs and seeks to combine them in a systematic way.

Abstraction is provided by putting the design process on a solid formal basis of higher-order functional programming, which allows the program designer to describe parallelism in an abstract way and to reason formally about it.

Performance is addressed by directing the design process towards programs whose components and overall structure allow a direct, efficient implementation over different parallel architectures.

Reconciliation of abstraction and performance in the SAT approach is provided by developing 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 of stages is captured using the common homomorphism pattern, 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 survey is to present the results obtained within the SAT approach and published in [H1]-[H9]. We comment also on the place of our work in the contemporary research landscape of parallel processing, discuss the lessons learned, and outline our vision of the future development in the field. The presentation is kept succinct, without going deeply into technical detail, which can be found in the papers referred to. In particular, all claims are stated without proof.

The power of the approach is demonstrated on several case studies, which have been suggested in the literature as testbeds for program design methods: scan (parallel prefix), the maximum segment sum (MSS) problem, 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 survey is structured as follows:

Section 2 outlines the SAT approach and comments on related work. We describe the class of target programs, introduce the formal notation used for abstraction, present 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: extraction, composition and implementation of homomorphic parallelism.

Section 4 presents the CS extraction method, based on the generalization of two sequential definitions, shows how it can be mechanized in a term rewriting framework, and applies it for constructing almost-homomorphisms, with the MSS problem as a case study.

Section 5 derives optimizing transformation rules for two practical compositions: reduction with scan and scan with scan. A parallel program derivation using these rules is demonstrated on the MSS problem.

Section 6 deals with the central problem of deriving parallel implementations for homomorphisms. We start with a standard homomorphism implementation, show its drawbacks and introduce the class of distributable homomorphisms (DH), for which efficient generic implementations are derived formally. We illustrate with scan and FFT.

Section7 considers the aspects of divide-and-conquer parallelism which are not covered by homomorphisms. We classify divide-and-conquer algorithms and derive an implementation for a general mutually recursive divide-and-conquer. For binary balanced 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 way of making design decisions 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 the directions of future research in the area.

The references are partitioned into two lists: (a) the surveyed papers by the author, which are submitted for habilitation [H1]-[H9], and (b) other papers cited in the survey.

2 Outline of the SAT Approach

This section, based largely on paper [H6], outlines the SAT approach as a common framework for dealing with performance and abstraction, and states its place in a broader context of the research on parallel program design.

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) [40, 55] is currently 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. The programs of individual processors can be viewed as communicating sequential processes – the view, first adopted in the CSP model [38] and then realized in the transputer language Occam [57].

MPI (Message Passing Interface) [32] is a library of parallel primitives, currently becoming a de facto standard for writing parallel applications in the SPMD style. It is portable across practically all kinds of parallel and distributed platforms. Attempting to cover a possibly wide variety of parallel programming primitives, MPI is becoming too rich and, therefore, hard to use. To improve the situation, an increasing number of people recommend using subsets of the MPI primitives and more structured parallel programming techniques [23]. A new perspective in this direction is shown by recent theoretical results on the coarse-grain algorithms with global communications[29] and also by the practice of MPI-programming with a restricted set of collective communications [56].

Our goal on the performance side is to restrain the complexity of parallel program behaviour, so that abstraction is facilitated and opportunities for good performance are preserved. To liberate the performance view from unimportant details, we represent it in simplified, MPI-like pseudocode.

[1] In [H6], this representation is called *target view*.

Three kinds of statements are allowed:

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

Communications are restricted to the collective communication primitives, like BRCAST 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 communication between them. For example, REDUCE (+) computes a sum of elements across all the processors; another often used primitive is the parallel prefix, e.g., SCAN (+) .

We exploit the group concept of MPI: collective operations can be restricted to a particular group of processors. For example, a reduction can be computed in each row of a virtual processor matrix, for all rows simultaneously. We express this by writing REDUCE (+) in ROW.

Summarizing, the performance view of SAT is: (1) well-structured due to the use of the SPMD discipline and collective communications, (2) portable since it is based on MPI, and (3) suited for high target performance as confirmed by recent experience in different application areas. The performance view might be considered just a good style of parallel programming with competitive performance [20]. Our goal is more ambitious: we aim at the systematic design of parallel programs in the performance view, starting from abstract specifications. Let us now concentrate on the abstraction side and then demonstrate how performance and abstraction are dealt with together in the design process.

2.2 Abstraction View

Similarly to the performance view for target programs, we need an *abstraction view* for hiding the low-level details of parallelism in the design process.[2]

For the purpose of abstraction, SAT makes use of the Bird-Meertens formalism (BMF) on lists [7]. Originally created for the design of sequential programs, BMF is becoming increasingly popular in the parallel setting, largely due to the seminal work of Skillicorn summarized in [60]. In BMF, higher-order functions (functionals) capture, in an architecture-independent way, general idioms of parallel programming, which can be composed for representing algorithms. BMF functionals use elementary operators and functions

[2]It is called *design view* in [H6].

as parameters, so that a BMF expression represents a class of programs which can be reasoned about, either taking into account particular properties of the customizing functions or independently of them. This style of programming is called generic [50] or skeleton-based [13].

Let us briefly 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, is the tightest binding and associates to the left. For the sake of brevity, we define the functionals informally and assume that all functional expressions are correctly typed.

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.,

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

Functional *map* is highly parallel since the computations of f on different elements of the list can be done independently if enough processors are available. Note that the elements of a list can be lists again and that f may be quite a complex function or composition of functions.

In addition to the parallelism of *map*, BMF allows to describe tree-like parallelism. This is expressed by functional *red* (for *reduction*) with a binary associative operator $\oplus$:

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

Reduction can be computed on a binary tree with $\oplus$ in the nodes. The time of parallel computation depends on the depth of the tree, which is $\log n$ for an argument list of length n.

We will introduce some other functionals in the sequel, but now let us see how the BMF expressions are constructed. Functions are composed in BMF by means of functional composition $\circ$, such that $(f \,\circ\, g)\, x \;=\; f\,(g\, x)$. Functional composition is associative and represents the sequential execution order.

BMF expressions – and, therefore, also the programs specified by them – can be manipulated by applying semantically sound rules of the formalism. Thus, we can employ BMF not only for programming per se but also, more importantly, for reasoning formally about parallel programs in the design process.

The main features of the BMF-based abstraction view in the SAT approach are summarized as follows: (1) the problem- and implementation-specific details are hidden using the higher-order functionals, (2) expressive power is provided by the variety of parallel patterns exemplified by the functionals and their compositions, (3) formal reasoning about abstraction views is possible using the identities of the formalism.

The next problem is how the functional abstraction view and the imperative performance view can be used together in the program design process.

2.3 Design in SAT: Stages and Transformations

The rôle of a glue between the performance view and the abstraction view of a parallel program is played by the adopted *programming model.*

The traditional formal model of parallelism, the PRAM, allows to describe and compare parallel algorithms, by abstracting from synchronization, data locality, machine connectivity and communication capacity [66]. Though it is well understood, it does not always reflects the real costs of parallelism. This leads to the situation that some efficient PRAM algorithms do not work well on parallel machines. In a more practical, network model, a program is viewed as a collection of communicating processes, each with a local memory. Using the terminology of CSP or Occam [57], it can be called the "PAR-SEQ" model (PARallel composition of SEQuential processes). This model corresponds directly to the SPMD performance view and results usually in efficient programs, but it involves many low-level details and, thus, is too remote from the abstraction view.

A key feature of SAT is to use the PAR-SEQ model for solely representing the target program. The program design process exploits the dual programming model, SEQ-PAR. This model, which has its origins in the SIMD-like data parallelism [10], is extended in SAT to the class of coarse-grain SPMD-like programs. The parallel program under development is viewed as a sequence of parallel *stages.* Each stage encapsulates parallelism of a possibly different kind and involves potentially all the processors. Though abandoning the CSP model in the design process, we borrow the idea of communication-closed layers, which was introduced originally in the CSP setting [10, 22], and apply it to stages: no communication is allowed between different stages. The stage model connects the abstraction view and the performance view: BMF allows to specify directly the single-threaded program structure, which in turn can be easily expressed as an SPMD program due to the communication-closeness of stages.

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 [56], in the coarse-grain algorithms [29], and in the so-called “bridging” models of parallelism, including the bulk synchronous parallelism in the BSP model [45], asynchronous activity in LogP [17] and a number of scalable operations in the WPRAM [51].

The SAT's underlying formalism – BMF – is a framework that facilitates *transformations* of programs into each other. A simple example of a BMF transformation is the map fusion law:

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

If the sequential composition of two parallel steps on the right-hand side of (3) is implemented via a synchronization barrier, which is often the case, then the left-hand side is more efficient and should be preferred. A cost calculus can be exploited to estimate the impact of particular transformations on the parallel performance [61]. There is a rich body of BMF transformations which can be used in the design process (see Subsection 5.2 and [60] for examples).

The general idea of transformational programming [11] is to start with an intuitively clear and correct but probably inefficient algorithm for a problem and to proceed by transformation until a semantically equivalent solution with a sufficient performance is reached (see a case study in [21]). Although very elegant and promising, with the Munich CIP system [4] being a prominent example, such approaches still have not been a real practical success. One of the reasons is, in our opinion, that the user is directly involved into the transformation process and, thus, have to become an expert in the transformational formalism.

A way to shield the user from the transformational framework and from its underlying formalism is offered by the *skeleton* approach, whose original motivation was to describe formally common schemas of parallelism found in different applications [13]. Skeletons can be viewed as higher-order functions: a simple example is the skeleton *map* defined by (1) with parameter function f, which can be customized when used for a particular application. If one or several high-quality parallel implementations are offered for a skeleton, then the only task remaining for the user is to cast the particular problem in the given schema. The user has not to be aware of how the implementations were obtained. The skeleton area has been studied actively recently; related work includes experimental skeleton systems in both the functional [19] and the imperative [2, 9] setting, capturing different kinds of recursion as skeletons [36, 67], transformations of skeletons and their compositions [25, 65], etc.

2.4 SAT and Homomorphisms

In summary, the SAT approach to the design of parallel software is based on the following two concepts which give the approach its name:

Stages structure the abstraction view of parallel programs using BMF functionals and the concept of skeletons and, thereby, hide low-level programming details.

Transformations capture the program design process as the correctness-preserving transitions, either between different abstraction views or from an abstraction view to a performance view.

The scope of the SAT approach includes algorithms working on recursively constructed data types, such as lists, arrays, trees and so on. Algorithmic parallelism is abstracted by higher-order skeletons in the functional context of BMF. The basic parallel skeleton used in the survey is the *homomorphism*. The property of a function to be a homomorphism, familiar in different branches of mathematics, turns out to be helpful for program design in general and parallelization in particular. Introduced by Bird [7] in the constructive theory of lists, it has been studied extensively in the category-based theory of data types [46, 60]. Our interest in homomorphisms is due to their direct correspondence to data parallelism [10, 40] and to the fundamental paradigm of divide-and-conquer, which is used extensively in both sequential and parallel algorithm development [34, 58].

An important reason for our decision to concentrate on homomorphisms is the following completeness property of the BMF transformational framework: any formulation of a homomorphism on a recursively constructed data type can be transformed in the Bird-Meertens formalism into any other formulation of that homomorphism by equational substitution [59].

Homomorphisms and, more generally, divide-and-conquer algorithms are the main building blocks in SAT and the focus of interest in the rest of this survey. In Sections 3-6, we study different subclasses of homomorphisms and develop the methods of extracting abstract homomorphic patterns from specifications, adjusting specifications to the homomorphic skeletons, implementing the extracted/adjusted parallelism with high performance and composing homomorphic patterns in larger applications. Section 7 deals with diverse aspects of abstraction and performance in the parallelization of more general divide-and-conquer algorithms.

3 List Homomorphisms

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

In the SAT approach, we exploit the homomorphism idea 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) co-domains represent the results of computing functions on the structured data; (3) a function is a homomorphism if its value on a data structure can be constructed from the values 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 (*conquer*) the problem solution on the whole. Divide-and-conquer is a fundamental principle of algorithm design, both in theory and in practice [58].

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

Definition 3.1 (Bird [7]) *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 (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 [8]. This simple function has been extremely useful in many applications; at the same time it is well parallelizable. 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 the potential of 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:*

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

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

$$scan\,(\odot)\,(x +\!\!\!+\, y) \;=\; S_1 \circledast S_2 \;=\; S_1 +\!\!\!+ \;(map\,((last\,S_1)\,\odot)\;S_2)\,, \qquad (6)$$
$$\quad where\; S_1 \;=\; scan\,(\odot)\,x\,,\;\; S_2 \;=\; scan\,(\odot)\,y\,.$$

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

Theorem 1 (Bird [7]) *A list function h is a homomorphism iff it can be factored into the composition:*

$$h \;=\; red\,(\circledast)\;\circ\; map\,f \qquad (7)$$

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

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

The homomorphism skeleton provides the abstraction view of a class of functions on lists, together with a common parallelization schema, (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:

1. *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).

2. *Adjustment*: if a function is not a homomorphism, bring it into a homomorphic format, perhaps at the price of some additional computations.

3. *Composition*: find an efficient way to compose several homomorphisms as stages in a larger abstract program.

4. *Implementation*: for a successfully extracted/adjusted homomorphism, find an efficient performance view, which may depend on the properties of the customizing functions.

In the next sections, we study the listed problems.

4 Extraction and Adjustment

In this section, based on papers [H4, H8], we study how to extract homomorphic parallelism of a given function h, or to adjust the function to the homomorphism skeleton.

We must 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 elaborated correctness proof of a parallel algorithm is presented in [52] or some *eureka* steps are applied in [31].

4.1 The CS-Method

A systematic way to address 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 list concatenation, sequential functional programming is based on the constructor `cons`, which attaches an element at 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 $\otimes$ may be non-associative, so many useful functions are either *lw* or *rw*, or both.

Theorem 2 *A function on lists is a homomorphism iff it is both lw and rw.*

In earlier work [3, 27, 28], the existence of leftward and rightward algorithms has been used as an evidence that a homomorphic algorithm exists, but no method of constructing the homomorphism has been offered.

In [H4], 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 [30], we prove a slightly stronger proposition.

While the theorem indicates that the left- or right-homomorphy of a function, together with the associativity of the combine operator, is sufficient for the function to be a homomorphism, the construction of the combine operator is still not provided: examples in [H4] demonstrate some unsuccessful attempts to arrive at an associative combine operator, starting from either a `cons` or a `snoc` representation. The idea of the proposed CS-method (for "Cons and Snoc") is to take both representations and 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" [35].

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

$$u \circledast v \leftrightarrow t_H \tag{8}$$

The following two terms, constructed from t_H by 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:

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

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

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

Figure 1(a) illustrates the relations between terms t_H, t_C, t_S, t_L, and t_R. Dotted arrows indicate application 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 established notion of a generalizer, 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 Figure 1(b).

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

(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

Figure 1: The relationships of the terms after a successful generalization; here: $\sigma_1 = \{u \mapsto t_B, v \mapsto h(y)\}$, $\sigma_2 = \{u \mapsto h(x), v \mapsto t'_B\}$.

The following theorem states that generalization of the rule terms constructed 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 asked to provide two sequential definitions for a given function: a `cons` term t_C and a `snoc` term t_S.

2. A successful 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 inductively then, by Theorem 4, $\circledast$ is the desired combine operator.

With the CS-method, we can extract a homomorphism for the scan function.

Example 1 (Continued) *For scan, the corresponding rule terms are:*

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

Their CS-generalization (see [H8]) yields: $u \circledast v \rightharpoonup u +\!\!+ \; map\,(last\,(u) \odot)\, v$*. One can prove the associativity of operator* $\circledast$ *thus defined:*

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

Therefore, scan is a homomorphism, with $\circledast$ *defined by* (10) *and* $f = [.]$*, where function* $[.]$ *creates singleton list of an element.*

4.2 Mechanizing the CS-Method

To apply the CS-method practically, the generalization step of the method must be mechanized, i.e., a generalization algorithm is required. In contrast to unification, a field studied extensively, properties and methods for generalization in a nonempty equational theory has been a largely neglected topic [16]. This subsection, based on joint work with Alfons Geser [H8, 26], 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 in that we suppress higher-order parameters, by considering a fixed associative operator $\odot$.

The objects of our generalization calculus are triples $(\sigma_1, \sigma_2, t_0)$, maintaining the invariant that 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 more 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, while preserving the generalization property of t_0, makes it more special until, finally, t_0 is maximally special.

We introduce two inference rules: ancestor decomposition and agreement. The *ancestor decomposition* rule reconstructs a common top 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 $x_i \mapsto u$. If, moreover, 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 $x_i \mapsto v$. The common part of the two, $x_j \mapsto x_i$, is transferred to t_0.

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

The following results, proved formally in [26], substantiate the proposed mechanization of the CS-method:

Soundness of the generalization calculus: every derivation yields a generalizer of the given `cons` and `snoc` 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 generalization algorithm: the algorithm stops under the imposed strategy of rule application.

The generalization algorithm is tested on the scan function, for which the homomorphic representation is extracted successfully [26]. In that example, we have used the semi-automatic inductive prover TIP [24], which produced a proof of associativity by rewriting induction, without any user interaction.

4.3 Almost-Homomorphisms: the MSS Problem

This subsection, based on paper [H4], deals with functions, which are not homomorphisms, but which can be adjusted to the homomorphic format.

A powerful method of adjustment is based on the notion of an *almost-homomorphism* [14] – a function that can be turned into a homomorphism by augmenting it with a number of auxiliary functions.[3] Our contribution is a systematic way of constructing suitable auxiliary functions by means of the CS-method (Subsection 4.1), extended for tackling the adjustment problem.

We demonstrate our approach by considering the famous *maximum segment sum* (MSS) problem – a *programming pearl* [5], studied by many authors [7, 14, 60, 62, 64].

The MSS Problem. Given a list of numbers, function *mss* finds a contiguous list segment, whose members have the largest sum among all such segments, and returns this sum. For example:

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

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

The CS-method starts with defining function *mss* over `cons` lists. Let function $\uparrow$ returns the larger of its two arguments. For some element a and list y, it may well be the case that $mss\,(a \mathbin{\cdot\!\!:} y) \;=\; a \uparrow (mss\, y)$. But the true segment of interest may also include both a and some initial segment of y; so we have to introduce 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*:

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

[3]By augmenting with the identity function, every function can be turned into a trivial homomorphism, but since no parallelism is possible in this case, we do not address it.

Augmenting function *mss* with both auxiliary functions, we obtain a triple: $\langle mss, mis, mcs\rangle$. For each function of the triple, the CS-method requires both `cons` and `snoc` definitions, which lead to one more auxiliary function, *ts* (for *total sum*).

Finally, we arrive at a quadruple of functions: $\langle mss, mis, mcs, ts\rangle$, with closed `cons` and `snoc` definitions (see [H4] for details). Generalization applied for each function of the quadruple yields the following combine operator:

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

Since $\circledast$ is associative, the quadruple is an instance of the homomorphism skeleton with two customizing functions: operator $\circledast$, defined by (11), and

$$f\ a \;=\; \langle mss, mis, mcs, ts\rangle\,[a] \;=\; (a\ ,\ a\ ,\ a\ ,\ a)$$

The abstraction view of the *mss* problem is obtained immediately from (7):

$$mss \;=\; \pi_1 \,\circ\, red\,(\circledast) \,\circ\, map\,f \tag{12}$$

where π_1 yields the first component of a tuple.

Program (12) coincides with the results of [14, 62] but, unlike them, our solution is obtained systematically, by applying the CS-method, without using our intuition about parallelism in the derivation process.

Let us estimate the parallel time complexity of the derived homomorphic algorithm, whose performance view in MPI-like pseudocode will be 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 [55]. Therefore, algorithm (12) is both time- and cost-optimal.

In [H4], we apply the CS-method also to the adjustment problem for so-called *nested almost-homomorphism*, whose combine operators turn to be, again, almost-homomorphisms. Therefore, we obtain two levels of the homomorphic parallelism, with overall time complexity $O(\log^2 n)$.

5 Composition of Homomorphisms

This section, based on [H9], addresses the composition of homomorphisms in the abstraction view – a very important issue in the SAT design process, where parallel programs are 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 a one, more complex parallel stage. We consider both the conventional scan, or prefix, defined by (5), and its symmetric version, called suffix:

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

We proceed not by inventing a solution and then verifying it, but rather by systematic adjustment to the homomorphism skeleton using formal transformations in BMF [H9]: (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 scan-reduction composition 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) *For arbitrary associative $\oplus$ and $\otimes$:*

- *if $\otimes$ distributes over $\oplus$ then*

$$\mathit{red}\,(\oplus) \,\circ\, \mathit{scan}(\otimes) \;=\; \pi_1 \,\circ\, \mathit{red}\,(\langle\!\langle \oplus, \otimes \rangle\!\rangle) \,\circ\, \mathit{map\,pair} \tag{13}$$

$$\mathit{scan}\,(\oplus) \,\circ\, \mathit{scan}\,(\otimes) \;=\; \mathit{map}\,\pi_1 \,\circ\, \mathit{scan}\,(\langle\!\langle \oplus, \otimes \rangle\!\rangle) \,\circ\, \mathit{map\,pair} \tag{14}$$

- *if $\otimes$ distributes backwards over $\oplus$ then*

$$\mathit{red}\,(\oplus) \,\circ\, \mathit{suf}\,(\otimes) \;=\; \pi_1 \,\circ\, \mathit{red}\,(<\oplus, \otimes>) \,\circ\, \mathit{map\,pair} \tag{15}$$

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

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

$$
\begin{aligned}
\mathit{pair}\ a &\stackrel{\text{def}}{=} (a, a) && (17)\\
(p_1, r_1)\,\langle\oplus,\otimes\rangle\,(p_2, r_2) &\stackrel{\text{def}}{=} (p_1 \oplus (r_1 \otimes p_2)\,,\ r_1 \otimes r_2) && (18)\\
(s_1, r_1) <\oplus,\otimes> (s_2, r_2) &\stackrel{\text{def}}{=} ((s_1 \otimes r_2) \oplus s_2\,,\ r_1 \otimes r_2) && (19)
\end{aligned}
$$

The proposed optimizing composition rules (13)-(16) fuse two consecutive program stages, each including a global communication, into one stage. Thereby, extra communication and synchronization in the performance view are eliminated.

Our results build on the general theory of homomorphisms in the category-theoretical setting [46] and in the sequential program design [6, 64]. Whereas, to the best of our knowledge, the transformation of two scans is new, a version of the scan-reduction composition was used in [12] for parallelizing linear recurrences. Our formulation is arguably more convenient for direct use in the practical parallel programming. In the next subsection, we show how our optimizing composition rules can be used in the program derivation process.

5.2 Derivation by Transformation: MSS Revisited

This subsection, based on paper [H9], demonstrates the 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 the CS extraction method from Subsection 4.3, which proceeds by generalizing two sequential programs.

The following equational rules of BMF will be used:

$$
\begin{aligned}
\mathit{scan}\,(\oplus) &= \mathit{map}\,(\mathit{red}\,(\oplus)) \circ \mathit{inits} && (20)\\
\mathit{suf}\,(\oplus) &= \mathit{map}\,(\mathit{red}\,(\oplus)) \circ \mathit{tails} && (21)\\
\mathit{inits} \circ \mathit{map}\, f &= \mathit{map}\,(\mathit{map}\, f) \circ \mathit{inits} && (22)\\
\mathit{map}\, f \circ \mathit{red}\,(+\!\!\!+) &= \mathit{red}\,(+\!\!\!+) \circ \mathit{map}\,(\mathit{map}\, f) && (23)\\
\mathit{red}\,(\oplus) \circ \mathit{red}\,(+\!\!\!+) &= \mathit{red}\,(\oplus) \circ \mathit{map}\,(\mathit{red}\,(\oplus)) && (24)
\end{aligned}
$$

where functions *inits* and *tails* are introduced informally:

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

For comparison purposes, we consider again the MSS problem formulated in Subsection 4.3.

Let us start with the following, intuitive abstraction view of function *mss*:

$$mss \;=\; red\,(\uparrow) \;\circ\; map\,(red\,(+)) \;\circ\; segs \tag{25}$$

which 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 is given by:

$$segs \;=\; red\,(+\!\!\!+) \;\circ\; map\; tails \;\circ\; inits \tag{26}$$

Specification (25) with function *segs* defined by (26) is obviously correct, but it has a poor time complexity: $O(n^3)$ for a list of length n [61].

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

$$
\begin{array}{ll}
 & mss \;=\; red\,(\uparrow) \;\circ\; map\,(red\,(+)) \;\circ\; red\,(+\!\!\!+) \;\circ\; map\; tails \;\circ\; inits \\
= & \quad \{\text{ Eq. (23),(24) }\} \\
 & red\,(\uparrow) \;\circ\; map\,(red\,(\uparrow)) \;\circ\; map\,(map\,(red\,(+))) \;\circ\; map\; tails \;\circ\; inits \\
= & \quad \{\text{ Eq. (3),(21) }\} \\
 & red\,(\uparrow) \;\circ\; map\;\big(\,red\,(\uparrow) \;\circ\; suf\,(+)\,\big) \;\circ inits \\
= & \quad \{\text{ Theorem 5 for suf-red, i.e., Eq. (15) with } \otimes = + \text{ and } \oplus = \uparrow \}\\
 & red\,(\uparrow) \;\circ\; map\;\big(\pi_1 \;\circ\; red\,\big(<\uparrow, + >\big) \;\circ\; map\; pair\,\big) \;\circ inits \\
= & \quad \{\text{ Eq. (3),(22),(20) }\} \\
 & red\,(\uparrow) \;\circ\; map\,\pi_1 \;\circ\; scan\;(<\uparrow, + >) \;\circ\; map\; pair
\end{array}
$$

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

In [12], the derivation stops here, but we can proceed further:

- Extend $\uparrow$ to pairs: $(a, b) \Uparrow (c, d) \stackrel{\mathrm{def}}{=} (a \uparrow c\,, b \uparrow d)$, and note that:

$$\pi_1 \;\circ\; \mathit{red}\,(\Uparrow) \;=\; \mathit{red}\,(\uparrow) \;\circ\; \mathit{map}\,\pi_1 \tag{27}$$

- Note that $<\uparrow, + >$ distributes (forwards) over $\Uparrow$, which allows us to apply Theorem 5 for scan-reduction, with $\otimes = <\uparrow, + >$ and $\oplus = \Uparrow$.

After three more transformation steps [H9], we arrive at:

$$\mathit{mss} \;=\; \pi_1^2 \;\circ\; \mathit{red}\,\big(\langle \Uparrow, <\uparrow, + > \rangle\big) \;\circ\; \mathit{map}\,(\mathit{pair}^2) \tag{28}$$

where $\pi_1^2 \stackrel{\mathrm{def}}{=} \pi_1 \circ \pi_1$ and $\mathit{pair}^2 \stackrel{\mathrm{def}}{=} \mathit{pair} \circ \mathit{pair}$. The first stage of (28) creates a pair of pairs (quadruple) of each element, and the third stage picks the first component of a quadruple. Both stages require constant parallel time.

The central stage of (28) is the reduction with operator $\langle \Uparrow, <\uparrow, + > \rangle$, expressed in terms of $\uparrow$ and $+$ as follows:

$$\begin{array}{l} \big(\, (r_1, s_1), (t_1, u_1) \,\big) \;\; \langle \Uparrow, <\uparrow, + > \rangle \;\; \big(\, (r_2, s_2), (t_2, u_2) \,\big) = \\ \qquad \big(\, \big(r_1 \uparrow r_2 \uparrow (t_1 + s_2)\,, s_1 \uparrow (u_1 + s_2)\big)\,, \big(t_2 \uparrow (t_1 + u_2)\,, u_1 + u_2\big) \,\big) \end{array} \tag{29}$$

Our result (29) is equivalent to solution (11) with logarithmic time complexity, which we have obtained in Section 4.3 by means of the CS-method from two sequential programs. 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.

In the sequential setting, the program derivation for the MSS problem was done in [6, 64]. 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 [12], which, by a one more application of Theorem 5, this time for scan-reduction, is improved and transformed into the solution from [14, 62].

6 Implementing Homomorphisms

This section, based on [H4, H7], addresses the problem of implementing a homomorphism, i.e., finding an efficient performance view for it. We start with a simple standard performance view 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 unpractical: the required number of processors grows linearly with the size of the data.

A more practical approach is to consider a bounded number of processors, p. 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*, $dist^{(p)} : [\alpha] \;\to\; [\,[\alpha]\,]_p$. The following obvious equality relates distribution with its inverse, flattening: $red\,(+\!\!+) \;\circ\; dist^{(p)} \;=\; id$.

Theorem 6 (Promotion [7]) *For a $\circledast$-homomorphism h :*

$$h \;\circ\; red\,(+\!\!+) \;=\; red\,(\circledast) \;\circ\; map\,h \tag{30}$$

From (30), the standard abstraction view of homomorphism h on p processors reads as follows:

$$h \;=\; red\,(\circledast) \;\circ\; map_p\,h \;\circ\; dist^{(p)} \tag{31}$$

To illustrate the use of schema (31), let us return to the MSS example, studied in Subsections 4.3 and 5.2. Since quadruple $\langle mss, mis, mcs, ts\rangle$ is a homomorphism with $\circledast$ defined by (11), the promotion theorem can be applied to it, which yields:

$$mss \;=\; \pi_1 \;\circ\; red\,(\circledast) \;\circ\; map_p\,\langle mss, mis, mcs, ts\rangle \;\circ\; dist^{(p)} \tag{32}$$

To simplify the presentation of the performance view, we make the assumption, usual in the parallel setting, that the program accepts the result of $dist^{(p)}$, i.e., the input list is distributed among the processors. In the performance view below, program `MSS-Dist`, this is expressed by the HPF-like [23] annotation `(BLOCK)` which means that array `list` is distributed block-wise, with `list-block` being a 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 the performance view are obtained directly from the stages of the abstraction view (32) which follow after $dist^{(p)}$:

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

$red\ (\circledast)$ is implemented by the MPI 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 implemented by picking the first component of the quadruple in the root processor and outputting it as the result of the program.

The performance of program `MSS-Dist` can be predicted by estimating the complexity of each of its three stages: sequential program `quadruple` has linear complexity and works on a block of size n/p, which gives time $O(n/p)$; reduction over p processors, with a constant-time basic operation $\circledast$, costs $O(\log p)$; the last stage, projection, is obviously constant. Therefore, the time complexity is $O(n/p + \log p)$ – the best one can expect in practice – with 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. As shown in [61], this leads to poor performance, which cannot be improved by just increasing the number of employed processors.

To improve the implementation, the class of homomorphisms is specialized in [H4] to the subclass DH (for Distributable Homomorphisms), defined on *powerlists* [47] of length 2^k, $k = 0, 1, \ldots$, with balanced concatenation. The DH definition makes use of the BMF functional *zip*, which combines elements of two lists of equal length with operator $\odot$:

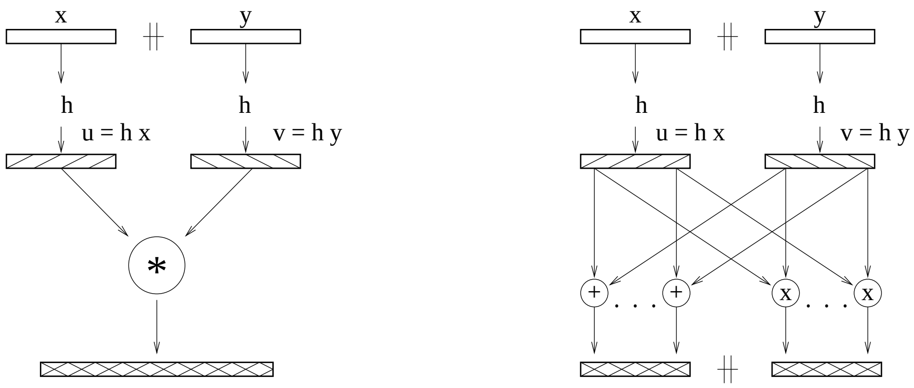
$$zip\,(\odot)\ (\,[x_1, \ldots, x_n]\,,\,[y_1, \ldots, y_n]\,) \;=\; [\,(x_1 \odot y_1), \ldots, (x_n \odot y_n)\,]$$

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

$$\begin{aligned} (\oplus\updownarrow\otimes)\ [a] \;&=\; [a] \\ (\oplus\updownarrow\otimes)\ (x +\!\!\!+\, y) \;&=\; zip\,(\oplus)\,(u,v) \;+\!\!\!+\; zip\,(\otimes)\,(u,v) \end{aligned} \qquad (33)$$

where $length\,x \;=\; length\,y \;=\; 2^k,\ u \;=\; (\oplus\updownarrow\otimes)\,x\ ,\ v \;=\; (\oplus\updownarrow\otimes)\,y\ .$

Figure 2 compares 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, point-wise format of the combine operator in a DH.



$u \circledast v$ vs. $zip\,(\oplus)\,(u,v) \;+\!\!+\; zip\,(\otimes)\,(u,v)$

Figure 2: General vs. distributable homomorphism: an illustration

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

6.3 Hypercube Implementation

In [H4], a generic hypercube implementation of DH is developed by introducing an architectural skeleton, *swap*, which describes a pattern of the hypercube behavior:

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

where $length\,(x) \;=\; 2^k\,,\ 1 \leq d \leq k\,,\ 0 \leq i < 2^k$.

Here, function *hyp* yields, for list x and index i, the ith element of x; function *xor* is the bit-wise exclusive OR. Therefore, *swap* consists of pair-wise, bidirectional communication in one dimension of the hypercube, followed by a computation with one of the two customizing operators. Let us write:

$$\mathop{\bigcirc}_{d=1}^{k}(swap\ d\ (\oplus,\otimes)) \ \stackrel{\mathrm{def}}{=}\ (swap\,k\,(\oplus,\otimes)) \circ \cdots \circ (swap\,1\,(\oplus,\otimes))$$

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

$$\oplus\updownarrow\otimes \;=\; \bigcirc_{d=1}^{k} (swap\; d\, (\oplus, \otimes)) \tag{34}$$

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

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

Many programs for parallel machines are actually of type $(\widetilde{h})_p$: it is usually assumed that the input and output data distribution is taken care of by the operating system, or that the distributed data are produced and consumed by other stages of a larger application.

Theorem 8 (Distributed DH) *For a distributed argument of type $[\,[\alpha]\,]_p$:*

$$\left(\widetilde{\oplus\updownarrow\otimes}\right)_p \;=\; \left((zip\,\oplus)\updownarrow(zip\,\otimes)\right)_p \;\circ\; map_p\,(\oplus\updownarrow\otimes) \tag{35}$$

To map the abstraction view (35) onto a hypercube of p processors, we apply equality (34) with $k = \log p$, which yields:

$$\left(\widetilde{\oplus\updownarrow\otimes}\right)_p \;=\; \Big(\bigcirc_{d=1}^{\log p} swap_p\; d\;(zip\,(\oplus)\,, zip\,(\otimes)\,)\Big) \;\circ\; map_p\,(\oplus\updownarrow\otimes) \tag{36}$$

Program (36) provides a generic, provably correct implementation of the DH skeleton on the 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 the function on a list of length n. Then the first stage of (36) 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 point-wise 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., the Fast Fourier transform (FFT), the first stage dominates asymptotically, so that program (36) is both time- and cost-optimal.

6.4 Scan as Distributable Homomorphism

Let us apply the results on DH from the previous subsection to scan. By adjusting scan to the DH skeleton and applying (34), we obtain the following hypercube program on the unbounded number of processors [H4]:

$$scan\,(\odot) \;=\; map\,\pi_1 \;\circ\; \bigcirc_{d=1}^{k} (swap\; d\; (\oplus, \otimes)\,) \;\circ\; map\; pair \tag{37}$$

where function *pair* is defined by (17), and

$$\begin{aligned} (s_1, r_1) \oplus (s_2, r_2) &\stackrel{\text{def}}{=} (s_1\,,\; r_1 \odot r_2) \\ (s_1, r_1) \otimes (s_2, r_2) &\stackrel{\text{def}}{=} (r_1 \odot s_2\,,\; r_1 \odot r_2) \end{aligned} \tag{38}$$

Program (37) is the well-known "folklore" implementation [55]. In Figure 3, its performance view is illustrated by the 2-dimensional hypercube which computes $scan\,(+)\,[1, 2, 3, 4]$.

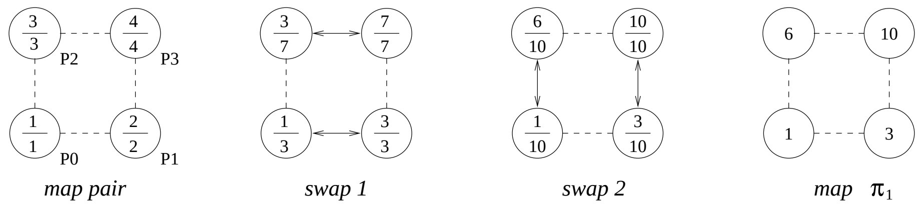


Figure 3: Computing scan on a hypercube

For scan on a fixed number of processors, the general schema (36) yields a suboptimal solution with time complexity $O\,((n/p) \cdot \log p)$, as explained in the previous subsection. This indicates that both skeletons, homomorphism and DH, are too general for scan.

A further specialization, called *localization schema* [H4], yields the following program for $scan\,(\odot)$ on p processors:

$$\begin{aligned} (\;\widetilde{scan\,(\odot)}\;)_p \;\; x \;&=\; zip_p\;\; (\odot\!\!\lhd)\;\; (y\;,\; z)\,, \\ \text{where}\quad z \;&=\; map_p\;(scan\,(\odot))\; x \\ y \;&=\; \big(map_p\,\pi_1 \;\circ\; \bigcirc_{d=1}^{\log p} (swap_p\; d\,(\oplus, \otimes)) \;\circ\; map_p\;(prepair \circ last)\big)\; z \end{aligned} \tag{39}$$

with $\oplus, \otimes$ from (38), $a \odot\!\!\lhd\, u \stackrel{\text{def}}{=} map\;(a\,\odot)\; u$, and $prepair\, a \stackrel{\text{def}}{=} (0_\odot, a)$.

Abstraction view (39) can be expressed directly as a three-stage program which computes scan on a list x, partitioned block-wise across p processors:

- *Compute* z: Each processor computes scan independently 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 (38). The first element of the result 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 [55]. 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 (39) is $O(n/p + \log p)$, the best one can expect for scan on p processors. Our localization schema is a constructive version of the compress-and-conquer technique [49].

The case study of scan confirms the power of the homomorphism approach: we arrive at the optimal and practically usable parallel algorithm. The key design decision of working on pairs of values is a result of the systematic adjustment to the DH skeleton, rather than an *ad hoc* trick. At the same time, a comparatively laborious process of adjustment indicates that the DH skeleton is too general for scan. This suggests that scan should be viewed as a primitive skeleton itself [8], which is the case, e.g., in the MPI standard [32].

6.5 Architecture-Independent Implementation

In this subsection, based on joint work with Holger Bischof [H7], we derive an architecture-independent, generic implementation of DH, which makes a considerable use of global communication primitives.

We use the type of d-nested lists with an equal number m of sublists at all levels: $[\alpha]_m^d \stackrel{\text{def}}{=} [\ldots[\alpha]_m\ldots]_m$. A generalization of *map* denoted by $map^l\, h \stackrel{\text{def}}{=} map\ (map^{l-1}\, h)$, where $map^0\, h \stackrel{\text{def}}{=} h$, 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 the notation $(\,\widetilde{h}\,)_p$ from Subsection 6.3, with the restriction of equal length in both dimensions.

Theorem 9 (Distributed Implementation of DH) *For an arbitrary DH function h on a list of length $2^{\,l}$, and arbitrary $d : 0 \leq d \leq l-1$:*

$$\widetilde{h^{(d)}} \;=\; \bigcirc_{i=1}^{d} chdim^{(i,i+1)} \;\circ\; \bigcirc_{i=1}^{d+1} \left(map^{\,d}\, h \;\circ\; chdim^{(i,d+1)} \right) \tag{40}$$

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

For an input list of length 2^l, Theorem 9 provides a family of exactly 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 Figure 4. Computation consists of two *map*s and two transpositions; for $i = 2$, both *chdim*s disappear since $chdim^{(2,2)} = id$. Note that notation $\bigcirc_{i=1}^{d}$ is used in [H7] and, consequently, in (40) in the opposite order compared to Subsection 6.3.

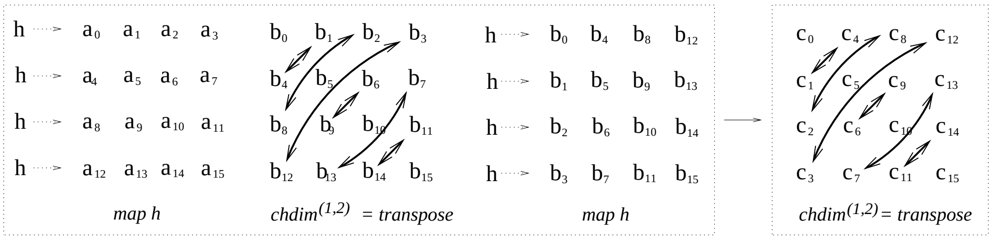


Figure 4: Implementation (40) in the two-dimensional case, $d = 1$.

The other extreme special case of schema (40) 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.

Implementation (40) is 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 distribution in p blocks and the redistribution between two consecutive steps in the 3-dimensional case are shown in Figure 5, adapted from [40]. Case $d = \log p$ corresponds to the p-processor hypercube.

The implementation given by Theorem 9 is a composition of two stages, which are indicated in Figure 4 by dotted rectangles. The goal of the second stage is solely to ensure that the output is distributed the same way as the input. Following the custom in practice, we skip this stage in the performance view, and annotate the altered output data distribution.

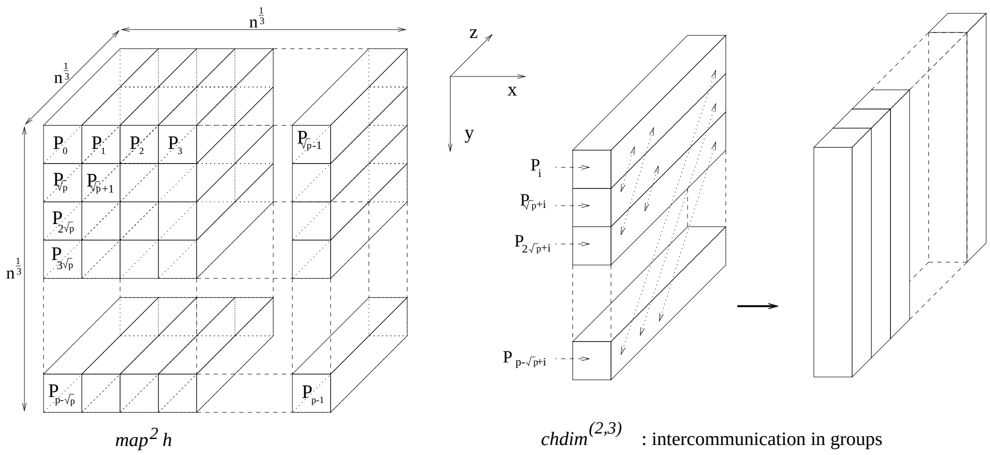


Figure 5: Three-dimensional case, $d = 2$: the transition between the first and the second stage.

The performance view 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 (40). The input and output data distributions are annotated as before, with * denoting the 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 by `local-transpose`. Function `map-h` computes function h along the undistributed dimension for the whole block.

As a case study, we have considered the FFT computation: in [H5], it is adjusted to the DH format and then generic schema (40) is applied. The power of the approach is demonstrated by the fact that all three algorithms used for computing FFT in practice [40] – the 2-dimensional and 3-dimensional transposition and the binary exchange algorithm – are described by one abstraction view (40). 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.

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 explicitly having in mind their data parallel execution. This section, based on papers [H1]-[H3] and [H5], addresses several aspects of parallelism in divide-and-conquer:

- How can the diversity of divide-and-conquer algorithms and their potential for parallelization be classified?
- How can a general 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 were obtained within 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 [H5]:

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 conquer functions. Thus, the *SDC* specifications allow different divide or/and conquer functions to be applied in the course of a computation.

Binary. The most well known and widely used divide-and-conquer scheme 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, is captured using our DC acronym: we call one-sided schemes D– and C–algorithms, for "Divide (without Conquer)" and "Conquer (without Divide)", correspondingly. For example, the list homomorphisms of Section 3 are computable by C–algorithms: their divide stage can be separated as a partitioning of the input list, according to the promotion property.

Nested. Divide-and-conquer specifications may have a nested control structure. 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, which belongs to the class $C(D)$; see [1, 48] for details.

An alternative top-down classification of divide-and-conquer by skeletons expressed in the functional language Haskell is presented in [37].

7.2 Mutually Recursive Divide-and-Conquer

This subsection, based on joint work with Chris Lengauer [H1], considers the practically relevant case of 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 recursive, i.e., there is a functional definition with one equation for every function of f:

$$f_i(x) \quad = \quad \texttt{if } p_i(x) \texttt{ then } b_i(x) \texttt{ else } E_i(g_i,\, f,\, x) \textbf{ fi} \qquad (i=1, \cdots, n) \tag{41}$$

Here, $g = (g_1, \cdots, g_n)$ is a collection of what we call *auxiliary functions*: g_i represents the non-recursive part of the equation for f_i. We 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 l-th 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 (41) as a specification for computing one of the functions f_i, say, f_1. Our goal is to generate a parallel program that, given a particular domain parameter *input*, computes $f_1(input)$. Specification (41) includes special cases that have been studied extensively in the literature: (1) Systolic algorithms are often specified in this format, where $\tau = \mathbb{Z}^m$ and the shifts are of the form $\varphi(i) = i + a$, for some fixed $a \in \mathbb{Z}^m$. These and other restrictions enable the use of linear algebra and linear programming for the synthesis of a parallel program [42]. (2) The class DC of conventional divide-and-conquer algorithms correspond to the case of a system with a single function f_1 and two recursive calls of it.

To develop a SAT abstraction view for specification (41), 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 [H1] leads to function F such that, if all processors simultaneously compute F and *input* is available at the root processor, then the result obtained at the root is the desired value $f_1(input)$. Therefore, we obtain an SPMD performance view on a tree of processors.

The following efficiency aspects are studied: (1) an additional, intra-processor level of parallelism is implemented by multi-threading; (2) the redundant computations of common values in different processors are eliminated by introducing additional interprocessor communication; (3) the load balance is improved using additional transformations at the functional level.

The study in [H1] illustrates also some specific features of skeleton-based programming. First, a particular algorithm can be adjusted in more than one way to the general format (41). E.g., a different match for the integration example changes the performance view from a heterogeneous communication tree to a binary one, which can be implemented more efficiently on most multiprocessors. Second, an optimization of the performance view results in the reduced outdegree of the tree nodes.

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

7.3 $\mathcal{N}$-graphs

In this subsection, based on paper [H3], we develop a new interconnection topology, called $\mathcal{N}$-graph, whose purpose is to implement efficiently the binary divide-and-conquer on a bounded number of processors.

In the literature, mainly two tree-like topologies with a bounded number of nodes have been considered [33, 34, 55]. 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, binomial tree, has a better balanced load, but we run into another problem: the node degree becomes non-constant, which makes the topology non-scalable.

In [H3], we introduce a new topology, the $\mathcal{N}$*-graph*, which is shown to combine 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 physical network. 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 descendant; the last leaf is connected to the auxiliary node. Figure 6 (left) shows the $\mathcal{N}$-graph with 16 nodes, whose additional edges are dashed and the additional node is shaded.

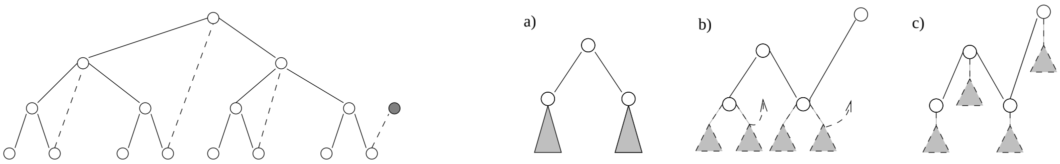


Figure 6: Left: $\mathcal{N}$-graph; right: distribution of tasks amongst processors

The node degree of an $\mathcal{N}$-graph is not greater than 4, which fits nicely on, e.g., transputer networks. The balanced load and locality are demonstrated on the right half of Figure 6: (a) shows a piece of a complete binary tree, with two leaves which have a common ascendant 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 subtasks, keep one of them and dispatch the other subtask to its descendant; (c) shows that after such transformation, every node of the $\mathcal{N}$-graph has exactly one subtask to execute.

This idea is realized in the SAT framework by transforming a general DC schema into an abstraction view on an $\mathcal{N}$-graph. The abstraction view is then transformed into a performance view, i.e., 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, leaves divide them into two parts and redistribute one of them to their neighbours in the $\mathcal{N}$-graph as illustrated in Figure 6(b)-(c).

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 [54]. 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 2D-mesh of the machine, we yield a speed-up near to 2, as compared with 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 similarities with *threaded trees* [44] which are used for optimizing tree traversal algorithms. Our topology differs from them in introducing an additional node and also in half as many additional edges. Moreover, the $\mathcal{N}$-graph is not a tree anymore, due to the loops introduced.

7.4 A Case Study: Polynomial Multiplication

In this subsection, based on paper [H2], we apply the SAT approach to a case study: we take the straight-forward sum accumulation algorithm for polynomial multiplication, and go through all development steps, from the mathematical specification towards the parallel target program. We pursue two major goals: (1) to make all design decisions systematically, using type analysis, formal transformations and performance considerations within SAT, rather than making the decisions *ad hoc*, and (2) to compare the resulting performance of the parallelization with the results achieved by another popular approach to parallelization, systolic design. We use a BMF-based

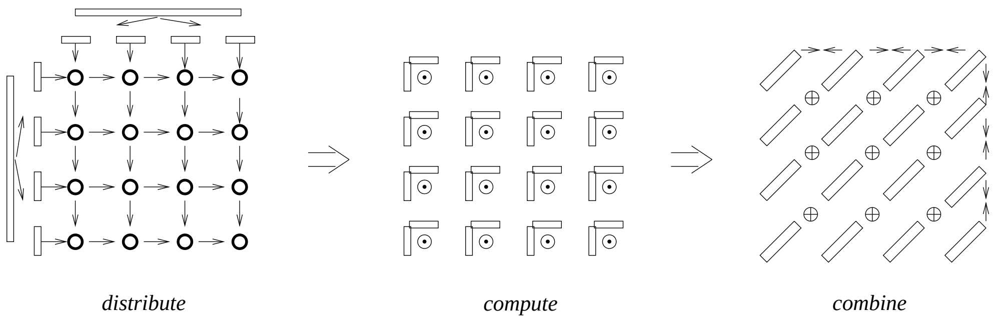


Figure 7: Three stages of the polynomial product

design process, starting from the mathematical specification of the product of two polynomials, represented by the lists of their coefficients. Eight design decisions have been made, resulting in the program structure shown in Figure 7.

We sketch the design decisions briefly (see [H2] for details):

1. Partition both lists of coefficients in p segments.
2. Construct the program as consisting of three stages: *distribute*, *compute* and *combine*.
3. Employ p^2 processors.
4. Broadcast data segments between the processors.
5. Assume distributed input data.
6. Organize the reduction in the *combine* stage along the diagonals.
7. Connect processors as a mesh of trees with diagonal trees [41].
8. Assume distributed output and implement the concluding step of *combine* as a pair-wise exchange between neighbouring processors.

The performance view reads as follows:

```
/* Input: A, B (BLOCK) */
   BRCAST (A) in ROW;
   BRCAST (B) in COLUMN;
   CALL PolyProd(A, B);
   REDUCE (⊕) in DIAGONAL;
   EXCHANGE-NEIGHBOURS;
/* 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 [55].

The case study 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 which are often used in the design of parallel programs. A detailed comparison of some systematic parallelization approaches to the polynomial multiplication can be found in [43].

8 Conclusion and Outlook

The diversity of parallel computers and the complexity of their software drive the quest for portable, tractable and efficiently implementable parallel programming models and languages. The SAT approach to this problem addresses the design and exploitation 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 in which communications are primitive, to a world in which more 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 to be 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.

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 well-understood behavior and predictable efficiency.

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 to instantiate, compose and implement diverse homomorphic skeletons, but the BMF-based development of these methods is delegated to the community of implementers.

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 is a positive experience of the structured sequential programming, where programs are compiled to codes which are often better than what programs with *goto*'s or hand-written assembler programs can provide; (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 restrict themselves to a standard set of global operations which have to be implemented on each target architecture. This increases the chance for finding high-performance solutions, which are portable across different architectural classes.

Thus, the very task of implementers becomes formulated more precisely, and different solutions can be compared more systematically than for an unrestricted variety of parallel architectures, programming styles and implementation tricks. This opens the way for the maturing of the implementation area and for a gradual transition from largely odd implementation efforts to an integrated compiler technology for parallel machines.

Directions of Further Research

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

Expressiveness. The question is: what might constitute the right set of types and operators (or does such a bounded set even exist) and what might be the most appropriate language framework for expressing the base level of a skeletal scheme? Very general skeletons may be inadequate for expressing certain algorithms and may lead to implementations that are not very efficient, whereas too specialized skeletons may degenerate to plain parallel library functions. To maximize opportunities for efficient implementation and reuse, we need to provide language mechanisms that can capture the similarities as well as the differences of various skeletons. Whereas the first skeletons were chosen with efficient implementation in mind, it is likely that, with growing experience, designers will want to include skeletons with a greater expressive power, whose implementations are not obvious.

Cost calculus. The challenge is to build a tractable, compositional cost calculus for higher-order programming models. One of the strengths of the recent models like BSP, compared to alternative models of parallel computation, is their simple yet *realistic* cost model that decomposes cost into a communication part and a computational part. 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 the confluence and convexity as proposed by Skillicorn [15].

Techniques. Practical methods and tools for the derivation and transformation of parallel programs must be developed further. In particular, they should help a programmer to decide what to do next in a derivation and what would be the right level of abstraction for making the design decisions. A possible approach are abstract machines by O'Donnell and Rünger [53]. Two other important issues are portability and data distributions, for which approaches like shapes [39] and the data distribution algebras [63] are suggested.

Implementations. The implementers of skeleton-based systems should be sensitive to the needs of application programmers, who have to be convinced that the idea of abstraction offers real advantages in practice. For example, the Ruby system of hardware design offers many of the tools that the skeletons community plans to provide for high level parallel programming: graphical interface, checkers, compilers, etc. Nevertheless, finding industrial users for Ruby still remains a challenge [15].

Applications. To assert its practical relevance, parallel programming in general and program design methods in particular need to be tested on convincing applications. A real challenge is, for example, irregular computations which underly many important applications; progress in this direction is achieved in the framework of parallel abstract data types at Imperial College [65] and in the Proteus project [18].

Component-based programming. In the SAT approach, we concentrate mostly on programming-in-the-small and simple functional compositions of homomorphisms. The major way of tackling the software complexity is by means of gluing together pre-written components, which range from straightforward libraries to elaborated application frameworks. A higher-order approach is clearly possible here: for example, an application framework can be thought of as a function, parameterized by a description of the behavior of the application. Programming-in-the-large, which still receives most effort from industry, is a natural candidate for more attention from academia.

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 that are necessary in the development and evaluation of parallel programs often reduce the design process to a black art. The challenge is to support program designers in their creative activity by providing them with formally sound and practically useful notation, together with tools for making design decisions. Of course, in well-understood cases the user will be provided with a set of exact rules, or the design process can be mechanized entirely.

The results presented in this survey show a way to combine abstraction and performance in parallel program design, in order to make the design process tractable and improve the quality of the resulting programs. The motto of a higher-order, formally based approach to parallelism finds an increasing number of supporters, and a research community has been forming recently [15]. Its goal is the development of programming methods and tools that will help to liberate parallel computing from the "selling a bridge" image (see the epigraph on p. 2) and turn it into an economically reasonable component of future information technology.

Acknowledgements

No research can be performed in a vacuum, and none of the work presented here could have been done without the support of people whose help I appreciate very much.

First and foremost I would like to thank my advisor Christian Lengauer, who invited me to Passau and provided constant support, guidance and encouragement over the past few years. I am grateful for the freedom he gave me in selecting my research topics as well as for his genuine interest in my work and many critical questions that have been instrumental in the development of my ideas. Christian has tried his best to remove the rough edges in my presentation; the jagged bits that remain are of course solely my responsibility.

It was a real pleasure and a valuable experience to work with my coauthors Alfons Geser and Holger Bischof. I am indebted to Christoph Wedler for bravely proof-reading almost all of my papers in the draft version and for his comments which greatly improved the presentation. Christoph A. Herrmann deserves a special mention for many helpful discussions. Other members of our group at the University of Passau have offered their help and advice, including Martin Griebl, Ulrike Lechner, Nils Ellmenreich, Johanna Bucur and Ulrike Peiker.

I am grateful to Manfred Broy who initiated my first visit to Germany under the generous support of the Alexander von Humboldt Stiftung, and who has been supporting me since then.

Many colleagues explained their work and mine to me, and helped me a lot with their advice: D.K. Arvind, Richard Bird, Arndt Bode, Luc Bougé, Murray Cole, Jean-François Collard, John Darlington, Walter Dosch, Jeremy Gibbons, Zully Grant-Duff, Herbert Kuchen, Lambert Meertens, Alberto Pettorossi, John O'Donnell, Gudula Rünger, David Skillicorn, Hing Wing To, Christoph Zenger and Eberhard Zehendner.

Finally, I would like to thank my family for their love, understanding and steady support.

The DFG-supported project RecuR2 has been a source of inspiration. I gratefully acknowledge the financial support received from the EU (INTAS Programme) and the DAAD (exchange programmes ARC and PROCOPE).

References

1. Publications submitted for habilitation

Refereed journal papers:

[H1] (with C. Lengauer) "Parallelization of divide-and-conquer in the Bird-Meertens formalism", *Formal Aspects of Computing*, 7(6):663–682, 1995.

[H2] "From transformations to methodology in parallel program development: A case study", *Microprocessing and Microprogramming*, 41:571–588, 1996.

[H3] "N-graphs: scalable topology and design of balanced divide-and-conquer algorithms", *Parallel Computing*, 23(6):687–698, 1997.

[H4] "Extracting and implementing list homomorphisms in parallel program development", *Science of Computer Programming.* To appear.

[H5] "Programming with Divide-and-Conquer Skeletons: A Case Study of FFT", *The Journal of Supercomputing.* To appear.

Refereed conference papers:

[H6] "Stages and transformations in parallel programming", In M. Kara et al., editors, *Abstract Machine Models for Parallel and Distributed Computing*, pages 147–162. IOS Press, 1996.

[H7] (with H. Bischof) "Formal derivation of divide-and-conquer programs: A case study in the multidimensional FFT's", In D. Mery, editor, *Formal Methods for Parallel Programming: Theory and Applications. Workshop at IPPS'97*, pages 80–94, 1997.

[H8] (with A. Geser) "Parallelizing functional programs by generalization", In *Algebraic and Logic Programming. ALP'97*, Lecture Notes in Computer Science. Springer-Verlag, 1997. To appear.

Technical report:

[H9] "Optimizing Compositions of Scans and Reductions in Parallel Program Derivation", Technical Report MIP-9711, Universität Passau, May 1997.

2. Other references

[1] K. Achatz and W. Schulte. Architecture independent massive parallelization of divide-and-conquer algorithms. In B. Moeller, editor, *Mathematics of Program Construction*, Lecture Notes in Computer Science 947, pages 97–127, 1995.

[2] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vaneschi. The P3L approach to massively parallel programming. In B. Buchberger and J. Volkert, editors, *Parallel Processing: CONPAR 94 – VAPP VI*, 1994.

[3] D. Barnard, J. Schmeiser, and D. Skillicorn. Deriving associative operators for language recognition. *Bulletin of EATCS*, 43:131–139, 1991.

[4] In F. Bauer, editor, *The Munich Project CIP*, Lecture Notes in Computer Science 183. 1985.

[5] J. Bentley. Programming pearls. *Communications of the ACM*, 27:865–871, 1984.

[6] R. Bird. Algebraic identities for program calculation. *The Computer Journal*, 32(2):122–126, 1989.

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

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

[9] G. Botorog and H. Kuchen. Efficient parallel programming with algorithmic skeletons. In L. Bougé et al., editors, *Parallel Processing. Euro-Par'96*, Lecture Notes in Computer Science 1123, pages 718–731. Springer-Verlag, 1996.

[10] L. Bougé. The data-parallel programming model: a semantic perspective (final version). Research Report 3044, INRIA, November 1996.

[11] R. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 25(1):44–67, 1977.

[12] W. Cai and D. Skillicorn. Calculating recurrences using the Bird-Meertens formalism. *Parallel Processing Letters*, 5(2):179–190.

[13] M. Cole. *Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computation.* PhD thesis, University of Edinburgh, 1988.

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

[15] M. Cole, S. Gorlatch, C. Lengauer, and D. Skillicorn, editors. *Theory and Practice of Higher-Order Parallel Programming.* Dagstuhl-Seminar-Report Nr.169, Schloß Dagstuhl. 1997.

[16] H. Comon, M. Haberstrau, and J-P. Jouannaud. Syntacticness, cycle-syntacticness, and shallow theories. *"Inform. and Computation*, 111:154–191, 1994.

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

[18] J. Prins, D. Palmer and S. Westfold. Work-efficient nested data-parallelism. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Processing (Frontiers 95).* IEEE, 1995.

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

[20] X. Deng and N. Gu. Good programming style on multiprocessors. In *Proc. IEEE Symposium on Parallel and Distributed Processing*, pages 538–543, 1994.

[21] C. Delgado Kloos, W. Dosch, and B. Möller. Design and proof of multipliers by correctness-preserving transformation. In *Proceedings Comp-Euro 92*, pages 238–243. IEEE Press, 1992.

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

[23] I. Foster. *Designing and Building Parallel Programs.* Addison-Wesley, 1995.

[24] Ulrich Fraus and Heinrich Hußmann. Term induction proofs by a generalization of narrowing. In C. Rattray and R. G. Clark, editors, *The Unified Computation Laboratory — Unifying Frameworks, Theories and Tools*, Oxford, UK, 1992. Clarendon Press.

[25] M. Geerling. Program transformations and skeletons: formal derivation of parallel programs. Technical Report CSI-R9411, Katholike Universiteit Nijmegen, October 1994.

[26] A. Geser and S. Gorlatch. Parallelizing functional programs by term rewriting. Technical Report MIP-9709, Universität Passau, April 1997. Available from http://brahms.fmi.uni-passau.de/cl/papers/GesGor97a.html.

[27] J. Gibbons. The third homomorphism theorem. Technical report, U. Auckland, 1994.

[28] J. Gibbons. The third homomorphism theorem. *J. Fun. Programming*, 6(4):657–665, 1996.

[29] T. Goodrich. Communication-efficient parallel sorting. In *Proc. 28-th ACM Symp. on Thoery of Computing*, 1996.

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

[31] Z. Grant-Duff and P. Harrison. Parallelism via homomorphisms. *Parallel Processing Letters*, 6(2):279–295, 1996.

[32] W. Gropp, E. Lusk, and A. Skijellum. *Using MPI: Portable Parallel Programming with the Message Passing*. MIT Press, 1994.

[33] A. K. Gupta and S. E. Hambrusch. Load balanced tree embeddings. *Parallel Computing*, 18:595–614.

[34] P. B. Hansen. Model programs for computational science: A programming methodology for multicomputers. *Concurrency: Practice and Experience*, 5:407–423, aug 1993.

[35] B. Heinz. Lemma discovery by anti-unification of regular sorts. Technical Report 94-21, TU Berlin, May 1994.

[36] C. A. Herrmann and C. Lengauer. On the space-time mapping of a class of divide-and-conquer recursions. *Parallel Processing Letters*, 6(4):525–537, 1996.

[37] C. A. Herrmann and C. Lengauer. Parallelization of divide-and-conquer by translation to nested loops. Technical Report MIP-9705, Fakultät für Mathematik und Informatik, Universität Passau, March 1997.

[38] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[39] C.B. Jay, D.G. Clarke, and J.J. Edwards. Exploiting shape in parallel programming. In 1996 IEEE Second International Conference on Algorithms and Architectures for Parallel Processing: Proceedings, pages 295–302. IEEE, 1996.

[40] V. Kumar et al. *Introduction to Parallel Computing.* Benjamin/Cummings Publ., 1994.

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

[42] C. Lengauer. Loop parallelization in the polytope model. In E. Best, editor, *CONCUR '93*, Lecture Notes in Computer Science 715, pages 398–416. Springer-Verlag, 1993.

[43] C. Lengauer, S. Gorlatch and C. Herrmann "The static parallelization of loops and recursions", In Proc. 11th Int. Symp. on High Performance Computing Systems (HPCS'97). To appear.

[44] H. Lewis and L. Denenberg. *Data structures and their algorithms.* Harper Collins, 1991.

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

[46] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *FPCA'91 Proceedings*, pages 124–144. Springer-Verlag, 1991.

[47] J. Misra. Powerlist: a structure for parallel recursion. *ACM TOPLAS*, 16(6):1737–1767, 1994.

[48] Z. G. Mou and P. Hudak. An algebraic model for divide-and-conquer algorithms and its parallelism. *Journal of Supercomputing*, 2(3):257–278, 1988.

[49] Z. G. Mou, C. Constantinescu, and T. Hickey. Optimal mapping of divide-and-conquer algorithms to mesh connected parallel architectures. In *Proceedings Int. Computer Symposium*, pages 273–284. Taiwan, 1992.

[50] D. Musser and A. Stepanov. Algorithm-oriented generic libraries. *Software – Practice and Experience*, 24(7):623–642, 1994.

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

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

[53] J. O'Donnell and G. Rünger. A methodology for deriving parallel programs with a family of abstract parallel machines. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *Parallel Processing. Euro-Par'97*, Lecture Notes in Computer Science 1300, pages 661–668. Springer-Verlag, 1997.

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

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

[56] T. Rauber and G. Rünger. Integrating library modules into special purpose parallel algorithms. In G. Agha and S. Russo, editors, *Proc. 2nd Int. Workshop on Software Engineering for Parallel and Distributed Systems (PDSE'97)*, pages 162–173.

[57] A.W. Roscoe and C.A.R. Hoare. The laws of Occam programming. Technical Monograph PRG–53, Oxford University, February 1986.

[58] R. Sedgewick. *Algorithms.* Second edition, 1988.

[59] D. Skillicorn. The bird-meertens formalism as a parallel model. In *Software for Parallel Computation*, NATO ARW, 1992.

[60] D. Skillicorn. *Foundations of Parallel Programming.* Cambridge U. P., 1994.

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

[62] D. Smith. Applications of a strategy for designing divide-and-conquer algorithms. *Science of Computer Programming*, 8(3):213–229, 1987.

[63] M. Südholt. Data distribution algebras — a formal basis for programming using skeletons. In *Programming Concepts, Methods and Calculi (PROCOMET'94)*, pages 38–57, 1994.

[64] D. Swierstra and O. de Moor. Virtual data structures. In B. Moeller, H. Partsch, and S. Schuman, editors, *Formal Program Development*, Lecture Notes in Computer Science 755, pages 355–371.

[65] Hing Wing To. *Optimising the Parallel Behaviour of Combinations of Program Components.* PhD thesis, Imperial College, 1995.

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

[67] C. Wedler and C. Lengauer. Parallel implementations of combinations of broadcast, reduction and scan. In G. Agha and S. Russo, editors, *Proc. 2nd Int. Workshop on Software Engineering for Parallel and Distributed Systems (PDSE'97)*, pages 108–119.