Extracting and Implementing List Homomorphisms in Parallel Program Development ⋆

Sergei Gorlatch

University of Passau, D–94030 Passau, Germany

Homomorphisms are functions that match the divide-and-conquer pattern and are widely used in parallel programming. Two problems are studied for homomorphisms on lists: (1) parallelism *extraction*: finding a homomorphic representation of a given function; (2) parallelism *implementation*: deriving an efficient parallel program that computes the function. The proposed approach to parallelism extraction starts by writing two sequential programs for the function, on traditional `cons` lists and on dual `snoc` lists; the parallel program is obtained by generalizing sequential programs as terms. For almost-homomorphic functions, e.g., the maximum segment sum problem, our method provides a systematic embedding into a homomorphism. The implementation problem is addressed by introducing the class of distributable homomorphisms and deriving for them a common parallel program schema. The derivation is based on equational reasoning in the Bird-Meertens formalism, which guarantees the correctness of the parallel target program. The approach is illustrated with the function *scan* (parallel prefix), for which the combination of our two systematic methods yields the optimal hypercube algorithm, usually presented *ad hoc* in the literature.

1 Introduction

The complex problem of developing correct and efficient programs for parallel architectures can only be managed if put on a solid formal basis. The ultimate goal is to liberate the programmer from the difficult task of dealing explicitly with the individual behaviour of numerous parallel processes and their interaction. Program development starts with a specification, which is "obviously"

⋆ The paper combines and expands results which were presented at the international conferences Euro-Par'96 (Lyon) and PLILP'96 (Aachen).

correct but possibly not efficiently implementable in parallel; the development process results in a correct and efficient parallel target program.

As a derivational calculus we use the *Bird-Meertens Formalism* (BMF) [5], where algorithms are specified using a set of higher-order functions over lists. The specification is refined into an executable form by semantically sound transformations which guarantee the *correctness* of the target program. The development process is structured by identifying typically used patterns of parallelism that can be implemented with predictable *performance*. Our approach can be contrasted with methods which address the correctness and performance issues *a posteriori*, relying on program verification, testing, profiling, etc.

In this paper, we study functions called *list homomorphisms*, which represent a particular pattern of parallelism.

Definition 1 *A list function h is a homomorphism iff there exists a binary operator $\circledast$ such that, for all lists x and y:*

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

where $+\!\!+$ is list concatenation. Note that $\circledast$ is necessarily associative on the range of h, because $+\!\!+$ is associative.

The homomorphic property (1) says that the value of h on the concatenated list depends in a particular way, using the *combine operator* $\circledast$, upon the values of h on the pieces of the list. The computations of $h\,x$ and $h\,y$ are independent, so (1) can be viewed as a special case of the divide-and-conquer paradigm. Examples of homomorphisms include such practically relevant functions as *scan* (prefix sums) [6], Fast Fourier transform, etc.

The contributions of the paper are new systematic methods for: (1) extraction of homomorphic parallelism from specifications and (2) subsequent implementation of the extracted parallelism as an efficient target program.

The paper is structured as follows:

- In Section 2, we introduce the Bird-Meertens notation, present the basic homomorphism theorem and formulate open questions.
- Section 3 deals with the extraction problem. In Subsection 3.1, we propose a method, called "`cons` & `snoc`", which extracts the homomorphic parallelism of a given function by generalizing two terms that define the function on `cons` and `snoc` lists. Subsection 3.2 extends this method to so-called almost-homomorphic functions and illustrates it for the popular maximum segment sum problem. Subsection 3.3 studies nested almost-homomorphisms and their application for parsing input-driven languages.

– Section 4 deals with the implementation of parallelism. In Subsection 4.1, we consider homomorphisms whose direct implementation has high communication costs and introduce a subclass DH (distributable homomorphisms), for which these costs can be reduced. In Subsection 4.2, we derive an iterative implementation schema for DH and transform it into a generic hypercube program. Subsection 4.3 demonstrates how this program is specialized for the *scan* function. In Subsection 4.4, a program for a bounded number of processors is obtained by transformations that tackle data (re)distributions. This program is further optimized in Subsection 4.5, which yields the optimal practical implementation for *scan*.
– In Section 5, we compare our results with related work.

We conclude by discussing the findings of the paper and future work.

The power of the approach is illustrated by giving systematic solutions for some problems studied previously elsewhere. Our running example is the *scan* function, used in many parallel applications [6]. First, its homomorphic representation is extracted by the `cons` & `snoc` method; this representation is then systematically adjusted to the DH-format and implemented on the hypercube, yielding optimal algorithms for both unbounded and bounded number of processors. Unlike the usual *ad hoc* presentation of algorithms for *scan* [25], our derivation is systematic and can be exploited for other problems.

2 BMF and Homomorphisms

In the Bird-Meertens formalism (BMF) [5,27], functions including homomorphisms are defined on composite types like lists, trees, etc.; we restrict ourselves to non-empty finite lists. Function application is denoted by juxtaposition, is tightest binding, and associates to the left. The following notation is used :

$[\alpha]$	the type of lists whose elements are of type α;
$\circ$	backwards functional *composition*;
map f	*map* of a unary function f , *map* $f\ [x_1, \ldots, x_n] = [f\ x_1, \ldots, f\ x_n]$;
red $(\odot)$	*reduce* with a binary associative operator, $(\odot) : (\alpha, \alpha) \to \alpha$, *red* $(\odot)\,[x_1, \ldots, x_n] = x_1 \odot x_2 \odot \ldots \odot x_n$;
zip $(\odot)$	combines elements of two lists of equal length with operator $\odot$, *zip* $(\odot)\ ([x_1, \ldots, x_n]\ , [y_1, \ldots, y_n]) = [(x_1 \odot y_1), \ldots, (x_n \odot y_n)]$;
$\langle\ \rangle$	Backus' FP *construction*: $\langle f_1, \ldots, f_n \rangle\, x = (f_1\, x, \ldots, f_n\, x)$.

We use brackets $\langle\ \rangle$ for construction rather than [ ] as in FP [2], because the latter are traditionally reserved in BMF for lists.

The inherent parallelism of the BMF functionals allows parallel programs to be represented as expressions of the formalism during the design process. We follow the SAT (Stages and Transformations) approach [14]: the program under development has the form of a sequential composition of parallel *stages*, dually to the traditional parallel composition of communicating sequential processes. The transition to the latter form, which is directly implementable on a parallel machine, is postponed until the end of the development process.

Theorem 2 (Bird [5]) *Function h on lists is a homomorphism iff:*

$$h \;=\; red(\circledast) \;\circ\; map\, f \tag{2}$$

where, for an element a, $f\ a \;=\; h\,[a]$, and $\circledast$ is associative.

Theorem 2 is known as the first homomorphism theorem. Expression (2) provides a standard parallelization pattern for all homomorphisms as a sequential composition of two stages. Whereas the first stage in (2), *map*, is totally parallel, the reduction can be computed in parallel on a tree-like structure, with combine operator $\circledast$ applied in the nodes.

Our running example is the *scan*-function, also called parallel prefix. Function *scan* yields, for associative $\odot$ and a list, the list of "prefix sums". For instance, on a list of four elements,

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

Function *scan* is a $\circledast$-homomorphism, i.e. its combine operator is $\circledast$:

$$u \circledast v \;=\; u \;+\!\!+\; map\,((last\,u) \odot)\ v \tag{3}$$

We use so-called operator sectioning, in which one argument of $\odot$ is fixed, so that the resulting unary function can be *map*ped; function *last* yields the last element of a list. Every homomorphic function is completely determined by its combine operator and its actions on singleton lists, so we write $hom(f, \circledast)$ for the unique $\circledast$-homomorphism h, such that $h\,[a] \;=\; f\ a$, for arbitrary a.

The paper addresses the following two problems:

(i) *Parallelism extraction*: for a given function, find the corresponding combine operator $\circledast$ that satisfies property (1). For functions like *length* this construction is simple, but already for the *scan* function it requires a formal proof [23] or some *eureka* steps [15]. For non-homomorphic functions, the problem is how to transform them into homomorphisms [8].

(ii) *Parallelism implementation*: for a given homomorphism, develop its efficient parallel implementation systematically. The reduction stage in (2) is a potential source of inefficiency: its direct tree-like implementation

may impose high communication costs which cannot be compensated for by just increasing the number of processors [28].

3 Extracting Homomorphisms

This section deals with the question of whether a given function is a homomorphism and if so, how its combine operator can be constructed. The proposed method is based on two inherently sequential representations of the function.

3.1 The "`cons` & `snoc`" Method

Whereas homomorphisms use list concatenation, traditional (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 $:\!\cdot$, which attaches an element at the list's end.

Definition 3 *List function h is called leftwards (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 rightwards (rw) iff, for some $\otimes$, $h\,(x :\!\cdot\, b) = h\,x \otimes b$.*

Since operators $\oplus$ and $\otimes$ need not be associative , many functions are either leftwards or rightwards or both. The importance of a function being both left- and rightwards is shown by the following theorem.

Theorem 4 *Function h is a homomorphism iff it is leftwards and rightwards.*

The theorem's necessary condition, known also as the second homomorphism theorem, is easy to see. The sufficient condition is sometimes referred to as the third homomorphism theorem: it was conjectured by Bird, proved by Meertens and presented systematically by Gibbons [12,11]. Gibbons says in [11] about this theorem with respect to the problem of extracting a suitable combine operator:

> "The existence of a suitable operator is guaranteed, but the theorem does not address the question of the existence – let alone the construction – of a direct and efficient way of computing it."

Solving this problem will be our first goal. Let us introduce a new definition.

Definition 5 *Function h is called left-homomorphic (lh) iff there exists binary operator $\oplus$, such that $h\,(a \cdot\!: y) = h\,[a] \oplus h\,y$. The dual definition of right-homomorphic (rh) function is obvious.*

Note that $\oplus$ in Definition 5 need not be associative. Every *lh* (*rh*) function is also *lw* (*rw*), but, e.g., the following function g is *lw* but not *lh*:

$$\begin{aligned} g\,[a] &= |a| \\ g\,(a \mathbin{\cdot\!:} y) &= \texttt{if}\quad a \le g\,y\quad \texttt{then}\quad |a + g\,y|\quad \texttt{else}\quad |a - g\,y| \end{aligned}$$

Let us study the relation of *lh* and *rh* functions to homomorphisms.

Theorem 6 *If function h is a homomorphism with combine operator $\circledast$, i.e., $h = hom(f, \circledast)$, then h is both lh and rh with the same combine operator. If function h is lh or rh, and the corresponding combine operator is associative, then h is a homomorphism with this combine operator.*

PROOF. The first part of the theorem is proved by a simple specialization of the homomorphism definition. We prove the second part for the case $h = lh(f, \oplus)$, showing that $h\,(x +\!\!+\, y) = h\,x \oplus h\,y$, by induction on the length of list x, for arbitrary y.

Induction base, $x = [a]$, is obvious.

Induction hypothesis: for a list x of length at most k and an arbitrary list y, the following holds: $h\,(x +\!\!+\, y) = h\,x \oplus h\,y$.

Induction step. List x of length $\le k{+}1$ can be represented as follows: $x = a \mathbin{\cdot\!:} xs$, where a is an element and xs is of length $\le k$. By calculation:

$$\begin{aligned} & h\,(x +\!\!+\, y) && \{\ x = a \mathbin{\cdot\!:} xs\ \} \\ &= h\,((a \mathbin{\cdot\!:} xs) +\!\!+\, y) && \{\ \text{list constructors' property}\ \} \\ &= h\,(a \mathbin{\cdot\!:} (xs +\!\!+\, y)) && \{\ h \text{ is leftwards-homomorphic}\ \} \\ &= h\,[a] \oplus h\,(xs +\!\!+\, y) && \{\ \text{induction hypothesis}\ \} \\ &= h\,[a] \oplus (h\,xs \oplus h\,y) && \{\ \text{associativity of } \oplus\ \} \\ &= (h\,[a] \oplus h\,xs) \oplus h\,y && \{\ h \text{ is leftwards-homomorphic}\ \} \\ &= h\,([a] +\!\!+\, xs) \oplus h\,y && \{\ [a] +\!\!+\, xs = x\ \} \\ &= h\,x \oplus h\,y \end{aligned}$$

The theorem is proved. In [13], the requirement of associativity is loosened to so-called left- and right-associativity.

Theorem 6 suggests a simple way to find a homomorphic representation of a given function: construct a `cons` definition of the function in the *lh* format (or, dually, find an *rh* representation on `snoc` lists), such that the combine operator is associative. Sometimes this direct method succeeds, as the following example demonstrates. Let us consider function *length*, which yields the length of a given list. Function *length* on a singleton list is: $length\,[a] = 1$, so $f = one$,

where $one\,x = 1$. The `cons`-definition is obvious: $length\,(a : x) = f\ a + length\,x$, thus *length* is *lh* with $+$ as combine operator. From associativity of $+$, by theorem 6 it follows that $length = hom\,(one, +)$.

The following example (courtesy of J. Gibbons) demonstrates that the test for (at least weakened) associativity in this simple direct method is indeed necessary. The identity function on lists, *id*, can be defined as both *lh* and *rh* with combine operator $\circledast$: $u \circledast v = [head\,u] +\!\!+ init\,v +\!\!+ tail\,u +\!\!+ [last\,v]$, (here functions *init* and *tail* yield the list without the first and without the last element respectively, function *head* yields the first element). However, function *id* is clearly not a homomorphism with this non-associative operator.

Let us try the direct method on the *scan* function. The `cons` definition is:

$$scan\,(\odot)\,(a \mathbin{:\!\cdot} y) \;=\; a \mathbin{:\!\cdot} (map\,(a \odot)\,(scan\,(\odot)\,y)) \tag{4}$$

Representation (4) does not match the *lh* format because a is used where only $scan\,(\odot)\,[a]$ is allowed. Since $scan\,(\odot)\,[a] = [a]$, there are different ways to express a via $scan\,(\odot)\,[a]$, e.g., $a = head\,(scan\,(\odot)\,[a])$ or $a = last\,(scan\,(\odot)\,[a])$. Since there are two occurrences of a, we obtain four possible terms for $\circledast$; however, none of these terms defines an associative operator! Let us try to use the right-homomorphic property in the `snoc` definition:

$$scan\,(\odot)\,(x \mathbin{\cdot\!:} b) \;=\; (scan\,(\odot)\,x) \mathbin{\cdot\!:} (last\,(scan\,(\odot)\,x) \odot b) \tag{5}$$

Alas, we run into a similar problem: both obvious substitutions for b, namely $head\,(scan\,(\odot)\,[b])$ and $last\,(scan\,(\odot)\,[b])$, lead to a non-associative operator, and thus we are still unable to express the *scan*-function as a homomorphism.

Note that the direct method is actually an intuitive search for an associative combine operator, using one of the extreme partitions of the list. This search can be difficult, as our unsuccessful attempts with *scan* demonstrate. Our idea now is to use both `cons` and `snoc` representations together in a more systematic way. The approach is based on the notion of term generalization.

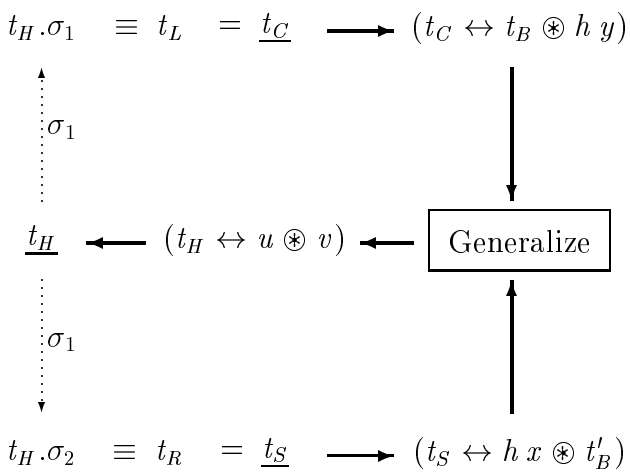
A *substitution* is an assignment of variables to terms, which we will write as $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$. For term t and substitution $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$, let $t.\sigma$ denote the result of simultaneously substituting in t each variable x_i, $1 \leq i \leq n$, by term t_i.

Definition 7 *A term t_G is called a* generalizer *of terms t_1 and t_2 in the equational theory E if there are substitutions σ_1 and σ_2, such that $t_G.\sigma_1 \stackrel{E}{=} t_1$ and $t_G.\sigma_2 \stackrel{E}{=} t_2$, where $\stackrel{E}{=}$ is the semantic equality of E.*

We use BMF as our equational theory, with semantic equality $=$ and syntactic equality $\equiv$. Note that there is always a trivial (most general) generalizer

for two terms, a variable; it is obvious that people prefer the *most special* generalizer, provided there is one. In this respect, generalization is the dual to *unification* where a most general common special case is wanted. For this reason, generalization is sometimes also called "anti-unification"[16].

Let us first study the equality- and substitution-relations between the terms we deal with. Assume that function h is a $\circledast$-homomorphism, and t_H denotes the goal of extraction – a "homomorphic" term over u and v that defines $\circledast$. This term is shown underlined in the left part of the diagram below. We move now from t_H along the dotted arrows (for substitutions) and equalities $=$ and $\equiv$. The right part of the diagram, in particular the bold arrows, should be temporarily ignored. The following two terms, built from t_H by substitutions: $t_L \equiv t_H.\{u \mapsto h\,[a], v \mapsto h\,y\}$ and $t_R \equiv t_H.\{u \mapsto h\,x, v \mapsto h\,[b]\}$, are obviously in the *lh* and *rh* format respectively. Terms t_L and t_R are semantically equal variants to all `cons` and `snoc` representations, respectively, of function h. Let t_C and t_S denote some particular `cons` and `snoc` representations.



Let us now exploit generalization to solve the extraction problem, i.e. for arriving at t_H from t_C and t_S. For an operator $\circledast$, we keep its defining term t_H and expression $u \circledast v$ together. Such a pair is viewed as a term by introducing a fresh binary symbol $\leftrightarrow$ with the weakest binding. Terms of the form $(t_H \leftrightarrow u \circledast v)$ are called *rule terms*. Our starting terms t_C and t_S define operator $\circledast$ in two special cases: $(t_C \leftrightarrow t_B \circledast h\ y)$ and $(t_S \leftrightarrow h\,x \circledast t'_B)$, where base terms t_B, t'_B define h on singletons.

For example, the rule terms for *scan* are obtained directly from (4),(5):

$$a \mathbin{\cdot\!:} (map\,(a \odot)\,(scan\,(\odot)\,y)) \leftrightarrow [a] \circledast scan\,(\odot)\,y \tag{6}$$

$$(scan\,(\odot)\,x) \mathbin{:\!\cdot} (last\,(scan\,(\odot)\,x) \odot b) \leftrightarrow scan\,(\odot)\,x \circledast [b] \tag{7}$$

The following fact relates rule terms with the homomorphism extraction.

Theorem 8 *If two rule terms:* $t_C \leftrightarrow t_B \circledast h\, y$ *and* $t_S \leftrightarrow h\, x \circledast t'_B$*, both for function* h*, have a generalizer,* $t_H \leftrightarrow u \circledast v$*, which defines an associative operator* $\circledast$*, then* h *is a homomorphism with* $\circledast$ *as combine operator.*

The use of rule terms ensures that the variables are substituted by the same terms in the generalization process.

If a generalization algorithm for BMF is available, then Theorem 8 provides us with the following method of homomorphism extraction. The method can be traced by moving along the bold arrows in the diagram above.

The `cons` & `snoc` method:

(i) The user is required to define function h on `cons` and `snoc` lists which gives a `cons` term t_C, a `snoc` term t_S and base terms t_B, t'_B.
(ii) The generalization algorithm, applied to the corresponding rule terms, $t_C \leftrightarrow t_B \circledast h\, y$ and $t_S \leftrightarrow h\, x \circledast t'_B$, yields a rule term $t_H \leftrightarrow u \circledast v$.
(iii) If the resulting term t_H defines an associative operator then this is the desired combine operator.

Applied to *scan*, the method generalizes rule terms (6) and (7), yielding

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

The resulting operator $\circledast$ is associative, so by Theorem 8 function *scan* is the homomorphism: $scan\,(\odot) = hom\,([.], \circledast)$, with $\circledast$ from (3).

The `cons` & `snoc` method provides a systematic solution to the homomorphism extraction problem. Its applicability depends on the power of the available generalization algorithm. We have developed a provably correct, terminating generalization algorithm for BMF. The algorithm was successfully tested on several examples; the details go far beyond the scope of this paper.

3.2 Almost-Homomorphisms

In this subsection, we deal with the situation that a function is not a homomorphism. Many practical non-homomorphic functions are so-called *almost-homomorphisms* (name coined by M. Cole): they are convertible to a composition of a homomorphism and some adjusting function.

Actually, every function h can be tupled together with the identity function, resulting in the function $g = \langle\, h\,,\; id \,\rangle$. Obviously, g is a homomorphism: $g\,(x \mathbin{+\!\!+} y) = g\, x \circledast g\, y$, where $(u, x) \circledast (v, y) = \Big(\; h\,(x \mathbin{+\!\!+} y)\,,\; x \mathbin{+\!\!+} y \;\Big)$. The original function is computed from g by projection, $h = \pi_1 \circ g$, where π_1 yields the first component of a tuple. This seems to provide an amazingly simple way of

computing every function in parallel as a homomorphism, followed by a simple projection. A closer look at operator $\circledast$ reveals the snag: it does not make use of the computed values, u and v, and computes function h from scratch!

Fortunately, there are also examples where a conversion to a "true" tuple homomorphism exists. Cole reports several case studies [8]; the main difficulty is to guess which *auxiliary functions* must be included in a tuple and then to find the combine operator. Usually, this requires a lot of ingenuity from the developer, hence a more systematic approach is desired. Cole says:

> "It is of interest to ask how easily the resulting algorithms might have been derived in a more strictly formal setting."

We will demonstrate that the `cons` & `snoc` method allows us to systematically construct almost-homomorphisms, known from the literature.

We consider the *maximum segment sum* (*mss*) problem – a *programming pearl* [4], studied by many authors [5,8,27,29]. Given a list of integers, function *mss* finds the contiguous list segment whose members have the largest sum among all such segments and returns this sum. For example, in the notation of [8]:

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

where the result is contributed by the segment $[2, -1, 6]$. The empty segment is defined to have sum 0, so the result is always non-negative.

Let us first express function *mss* over `cons` lists. For some element a and list y, it may well be the case that $mss\,(a \cdot\!: y) = (mss\,[a]) \uparrow (mss\,y)$, where $\uparrow$ returns the larger of its two arguments. But we must not overlook the possibility that the true segment of interest includes 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 method, `snoc` definition, requires the introduction of auxiliary function *mcs*, yielding the sum of the *maximum concluding segment*. The obtained definitions of *mss* are as follows:

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

To get a closed definition, we consider *mss* together with both auxiliary functions: $\langle mss, mis, mcs\rangle$. For this triple function, the `cons` & `snoc` method requires both `cons` and `snoc` definitions. Trying to find them, we see that the concluding segment of $a \cdot\!: y$ may be the whole list, so we need its sum, which no (combination) of the functions from the triple can yield. Therefore, we have to introduce one more auxiliary function, *ts* (for *total sum*).

Our triple becomes a quadruple $\langle mss, mis, mcs, ts\rangle$, which has the following closed `cons` and `snoc` definitions:

$$\begin{aligned} mss\,(a \mathbin{\cdot\!:} y) &= mss\,[a] \uparrow (a + mis\,y) \uparrow mss\,y \\ mis\,(a \mathbin{\cdot\!:} y) &= mis\,[a] \uparrow (a + mis\,y) \\ mcs\,(a \mathbin{\cdot\!:} y) &= mcs\,y \uparrow (a + ts\,y) \\ ts\,(a \mathbin{\cdot\!:} y) &= ts\,[a] + ts\,y \end{aligned}$$

$$\begin{aligned} mss\,(x \mathbin{:\!\cdot} b) &= mss\,x \uparrow (mcs\,x + b) \uparrow mss\,[b] \\ mis\,(x \mathbin{:\!\cdot} b) &= mis\,x \uparrow (ts\,x + b) \\ mcs\,(x \mathbin{:\!\cdot} b) &= mcs\,[b] \uparrow (mcs\,x + ts\,[b]) \\ ts\,(x \mathbin{:\!\cdot} b) &= ts\,x + ts\,[b] \end{aligned}$$

Generalizing each function of the quadruple separately, we arrive at the following combine operator:

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

Since $\circledast$ is associative, our tuple is the homomorphism: $\langle mss, mis, mcs, ts\rangle = hom\,(f, \circledast)$, where f yields the result of the tuple on singleton lists:

$$f\ a \ = \ \langle mss, mis, mcs, ts\rangle\,[a] \ = \ (a \uparrow 0\,,\ a \uparrow 0\,,\ a \uparrow 0\,,\ a)$$

The target function *mss* is therefore computable as follows:

$$mss \ = \ \pi_1 \circ red\,(\circledast) \circ map\,f \tag{8}$$

Let us estimate the parallel time complexity of the derived homomorphic algorithm. Since both function f and operator $\circledast$ require a constant number of communicated elements and executed operators, 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 tree sequentially, based on Brent's theorem [25]. Therefore, the direct tree-like algorithm is both time and cost optimal.

The `cons` & `snoc` method is therefore applicable to almost-homomorphisms if extended by "tupling" all auxiliary functions which arise in the process of building closed `cons` and `snoc` definitions. For the *mss* problem, the result of systematically applying the method coincides with the result obtained by Cole in [8] and by Smith in [29], but unlike them we have not used our intuition about parallelism in the derivation process.

3.3 Nested Almost-Homomorphisms

In this subsection, we apply our methodology to an example from [8] with so-called *nested parallelism*. The idea is that if the combine operator of a homomorphism is costly then the operator itself should be parallelized.

We consider the problem of determining, for a given string, whether the brackets of several types, e.g., ( ), [ ] { }, are correctly matched. There exists a straightforward linear-time sequential algorithm which maintains a stack during scanning the input. Opening brackets are pushed, and closing brackets are matched with the stack top. Failure is indicated by a mismatch, by an empty stack when a match is required, or by a non-empty stack at the end of the scan.

The idea of parallelization is to exclude all matching brackets and then to test the remaining list for emptiness. Let us apply the `cons` & `snoc` method to function *exmatch*, which yields the unmatched brackets in a list which, for simplicity, consists exclusively of brackets. Using predicate *match* for comparing two brackets, the `cons`- and `snoc`-representations are as follows:

$$\begin{aligned} exmatch\,(a \cdot\!: y) &= \texttt{if}\ match\,(a, head\,(exmatch\,y)) \\ &\qquad \texttt{then}\ exmatch\,(tail\,(exmatch\,y))\ \texttt{else}\ (a \cdot\!: (exmatch\,y)) \\ exmatch\,(x :\!\cdot\ b) &= \texttt{if}\ match\,(last\,(exmatch\,x), b) \\ &\qquad \texttt{then}\ exmatch\,(init\,(exmatch\,x))\ \texttt{else}\ ((exmatch\,x) :\!\cdot\ b) \end{aligned}$$

where x is non-empty. Generalization yields the following combine operator:

$$u \circledast' v \;=\; \texttt{if}\ match\,(last\,u, head\,v)\ \texttt{then}\ exmatch\,(init\,u +\!\!+ tail\,v)\ \texttt{else}\ (u +\!\!+ v)$$

Since u and v are in the range of *exmatch*, we can simplify:

$$\begin{aligned} & exmatch\,(init\,u +\!\!+ tail\,v) \\ = \quad & \{\ \text{Definition of } \circledast'\ \} \\ & exmatch\,(init\,u) \circledast' exmatch\,(tail\,v) \\ = \quad & \{\ u \text{ and } v \text{ are results of idempotent } exmatch\ \} \\ & init\,u \circledast' tail\,v \end{aligned}$$

Thus $exmatch = hom\,(f, \circledast)$, with $f\ a = [a]$ and the combine operator:

$$u \circledast v \;=\; \texttt{if}\ match\,(last\,u, head\,v)\ \texttt{then}\ (init\,u \circledast tail\,v)\ \texttt{else}\ (u +\!\!+ v) \qquad (9)$$

To allow comparison with the derivation by Cole, we note that our operator $\circledast$, applied to a pair (u, v), does the same as the composition $combine \circ$

dropmatches does in [8] to $(reverse\, u, v)$. Function *dropmatches* is, not surprisingly, recursive as is our $\circledast$ in (9):

$$dropmatches\,(u, v) = \texttt{if}\ match\,(head\, u, head\, v) \\ \texttt{then}\ dropmatches\,(tail\, u, tail\, v)\ \texttt{else}\ (u, v)$$

Function *dropmatches* requires both computation and communication time which is linear in the length of the leftover strings. In the best case, for strings with short, shallow-nested segments, the homomorphism *exmatch* provides logarithmic time complexity. In the worst case of one deeply nested segment, however, we cannot do better than the sequential algorithm.

This is the point where the nested parallelism is helpful: let us try to express *dropmatches* again as a homomorphism. First we massage it from a function on pairs of lists to a function on lists of pairs. The new function *dropmatches* (following [8], we do not change its name) accepts a list of pairs, where the first element is from u and the second is from v. If u and v are of different lengths, a special symbol is used to supplement the shorter list. Function *dropmatches* yields a list of pairs, with leading matching pairs removed. We apply the `cons` & `snoc` method to *dropmatches*. If it happens in $dropmatches\,((a, b) \cdot\!: x)$ that a and b are not matched then the result is just the input list, i.e. we need to know not only the value of $dropmatches\, x$ but also x itself. The auxiliary function is *original* which is the identity on lists of pairs. Now, the tuple $\langle dropmatches, original\rangle$ has a closed definition, whose generalization yields:

$$\begin{aligned} &\langle dropmatches, original\rangle \;=\; hom(g, \circledast), \\ &\qquad \text{where} \\ &g\,[(a, b)] \;=\; \Big(\texttt{if}\ match\,(a, b)\ \texttt{then}\ [\,]\ \texttt{else}\ [(a, b)]\ ,\ [(a, b)]\Big) \\ &(u, u') \circledast (v, v') \;=\; \Big(\ u +\!\!+ (\texttt{if}\ u = [\,]\ \texttt{then}\ v\ \texttt{else}\ v')\ ,\ u' +\!\!+ v'\ \Big) \end{aligned}$$

The parallel implementation of *dropmatches* is then expressed by:

$$dropmatches = \pi_1 \;\circ\; red\,(\circledast) \;\circ\; map\, g$$

Since the computation of *dropmatches* takes place in every application of $\circledast$ in the homomorphism *exmatch*, we get a tree, whose nodes are trees again. By embedding into a homomorphism, we achieved logarithmic computation time for *dropmatches*. Thus, our nested homomorphic algorithm for bracket matching has parallel complexity $O(\log^2 n)$ on shared memory. The communication time on distributed memory is still linear because of $+\!\!+$. A possible improvement for the bracket matching problem is presented in [8]; a general case is addressed in the next section.

4 Implementing Homomorphisms

Now, after we have extracted the combine operator of an (almost-) homomorphism, let us address the subsequent problem: how the constructed homomorphism can be implemented efficiently on a parallel machine.

4.1 Concatenating and Distributable Homomorphisms

Definition 9 *A list homomorphism is called a "concatenating homomorphism" if its combine operator has* $+\!\!+$ *as the top function:* $\circledast = (+\!\!+) \circ \langle f, g \rangle$.

The reduction stage of a concatenating homomorphism starts from the singleton lists after the map stage of (2) and arrives at a "long" result list at the root of the tree. The communication of lists of growing length induces linear execution time on machines with distributed memory, independently of the number of processors [28]. In the previous section, we faced this problem for function *dropmatches*. From (3) it follows that *scan* is a concatenating homomorphism. However, there exist parallel logarithmic-time algorithms for *scan* with good performance on parallel machines [25]: they both consume and produce lists which are distributed between processors. Our goal is to derive such algorithms systematically.

From now on, we restrict ourselves to *powerlists* [21] of length 2^k, $k = 0, 1, \ldots$, with *balanced* concatenation: $x +\!\!+ y$ is defined iff $length\, x = length\, y = 2^k$.

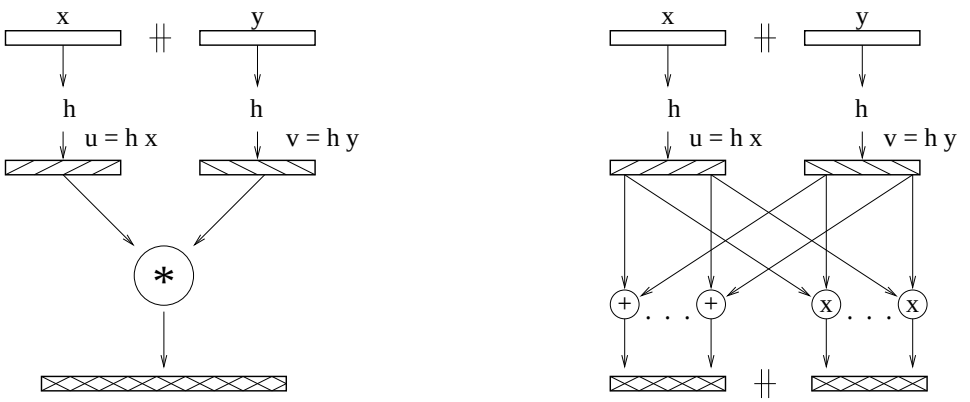
Definition 10 *For a pair of binary operators* $\oplus$ *and* $\otimes$*, the Distributable Homomorphism (DH) on powerlists, denoted* $\oplus \updownarrow \otimes$*, is defined as follows:*

$$\begin{array}{l} (\oplus \updownarrow \otimes)\,[a] \;=\; [a] \\ (\oplus \updownarrow \otimes)\,(x +\!\!+ y) \;=\; zip\,(\oplus)\,(u, v) \;+\!\!+\; zip\,(\otimes)\,(u, v) \\ \text{where } length\, x \;=\; length\, y \;=\; 2^k,\; u \;=\; (\oplus \updownarrow \otimes)\, x\,,\; v \;=\; (\oplus \updownarrow \otimes)\, y\,. \end{array} \tag{10}$$

In the homomorphism notation, $\oplus \updownarrow \otimes \;=\; hom\Big([.]\,,\; (+\!\!+) \circ \langle zip\,(\oplus), zip\,(\otimes)\rangle\Big)$, where function [.] yields, for an element a, the singleton list $[a]$.

Figure 1 compares how a general homomorphism (on the left) and a distributable homomorphism (on the right) are computed on a concatenation of two powerlists.

The first question is: how representative is the class of DH, i.e. which functions can be expressed in the form $\oplus \updownarrow \otimes$ by a suitable choice of the customizing operators $\oplus$ and $\otimes$?



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

Fig. 1. General vs. distributable homomorphism: an illustration

As a simple example, let us consider the function called "distributed reduction", informally defined as $redd\,(\odot)\,x \;=\; [red\,(\odot)\,x, \ldots, red\,(\odot)\,x]$. It is easy to express *redd* in the DH-format (10):

$$redd\,(x \;+\!\!+\; y) \;=\; zip\,(\odot)\,(redd\,x, redd\,y) \;+\!\!+\; zip\,(\odot)\,(redd\,x, redd\,y)$$

Therefore, *redd* belongs to the class DH:

$$redd\,(\odot) \;=\; \odot\updownarrow\odot \tag{11}$$

Function *redd* is widely used in parallel programming and is implemented as the `AllReduce` primitive in the MPI standard [30]. We will see further examples of DH in the sequel; let us first concentrate on their parallel implementation.

4.2 Towards an Efficient Parallel Implementation

Our goal is to find a provably correct and efficient parallel implementation for all DH functions. In this subsection, we first develop an architecture-independent implementation schema and then map it onto the hypercube topology. For the sake of efficiency, we aim at an iterative, rather than recursive implementation.

Let us introduce some functions on powerlists which we will use later.

Our first two functions do simple rearrangements:

$$att : nat \to \alpha \to (nat, \alpha) \qquad glue : \alpha \to (nat, \alpha) \to (nat, \alpha, \alpha)$$
$$att\; i\; x \;=\; (i, x) \qquad glue\; a\,(i, b) \;=\; (i,\; a,\; b)$$

The next function, *permute*, interchanges pair-wise elements which have a given distance, $k = 2^i$, between their positions in the powerlist x whose length is greater than k. The function attaches to each element a flag which is equal to 0 if the element has moved to the left and 1, otherwise:

$$
\begin{array}{l}
permute : nat \to [\alpha] \to [(nat, \alpha)] \\
permute\, k\, (x +\!\!+ y) \;=\; permute\, k\, x \;+\!\!+\; permute\, k\, y\,, \quad \text{if } k < length\, x \\
\qquad\qquad\qquad\qquad\quad map\, (att\, 0)\, y \;+\!\!+\; map\, (att\, 1)\, x\,, \quad \text{if } (k = length\, x
\end{array}
$$

Figure 2 illustrates how function *permute* and function *step*, introduced later, work on a 4-element list.

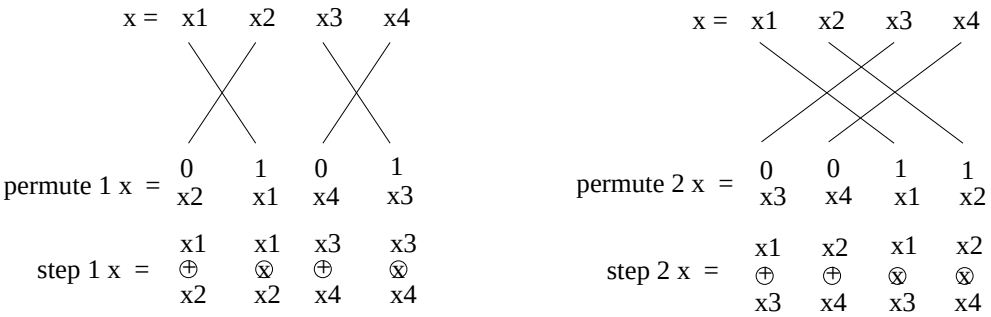


Fig. 2. Functions *permute* and *step*: an illustration

Function *triples* combines a list with a permutation of itself

$$
\begin{array}{l}
triples : nat \to [\alpha] \to [(nat, \alpha, \alpha)] \\
triples\, k\, x \;=\; zip\, glue\, (x,\, permute\, k\, x)
\end{array}
$$

Function *apply* performs one of two binary operations ($\oplus$ or $\otimes$) on the elements of a list of triples, depending on the value of the flag:

$$
\begin{array}{l}
apply : (((\alpha, \alpha) \to \alpha), ((\alpha, \alpha) \to \alpha)) \to (nat,\, \alpha,\, \alpha) \to \alpha \\
apply\, (\oplus, \otimes)\, (i,\, a,\, b) \;=\; \text{if } (i = 0) \text{ then } (a \oplus b) \text{ else } (b \otimes a)
\end{array}
$$

Now we introduce function *step*, whose behaviour is illustrated in Figure 2.

$$
\begin{array}{l}
step : nat \to (((\alpha, \alpha) \to \alpha), ((\alpha, \alpha) \to \alpha)) \to [\alpha] \to [\alpha] \\
step\ k\ (\oplus, \otimes) \;=\; map\, (apply (\oplus, \otimes)) \circ (triples\, k)
\end{array}
$$

The next function, *iter*, performs k consecutive applications of function *step*:

$$
\begin{array}{l}
iter : nat \to (((\alpha, \alpha) \to \alpha), ((\alpha, \alpha) \to \alpha)) \to [\alpha] \to [\alpha] \\
iter\, k\, (\oplus, \otimes) \;=\; (\, step\ 2^{(k-1)}\ (\oplus, \otimes)\,) \circ (\, iter\, (k-1)\, (\oplus, \otimes)\,), \quad \text{if } k > 0 \\
\qquad\qquad\qquad\qquad\quad id\,, \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{if } k = 0
\end{array}
$$

Function *iter* has an obvious iterative implementation, in contrast to the *cascading recursion* in the DH definition (10). The following theorem establishes the equivalence of these two forms.

Theorem 11 *For an arbitrary list x of length 2^k, the following holds:*

$$(\oplus\updownarrow\otimes)\, x \quad = \quad iter\, k\, (\oplus, \otimes)\, x \tag{12}$$

PROOF. By induction on k.

The theorem provides a common computation schema for all DH functions. The next step is to map this architecture-independent solution onto a particular processor topology. As an example, let us consider the hypercube.

Our lists of length $n = 2^k$ are stored in a k-dimensional hypercube with n nodes. The standard encoding is used: the position i, $0 \leq i < n$, of a list is stored in the node i, whose index is the k-bit binary representation of i. The access function on hypercubes, $hyp : [\alpha] \to nat \to \alpha$, yields, for list x and index i, the element of x stored in the node i of the hypercube. Each processor of the hypercube can communicate directly with its k neighbours, whose indices differ in one bit position; this position determines the dimension in which the communication takes place. In each dimension, $n/2$ pairs of processors can communicate simultaneously, without dilation or congestion. For processor i, its partner in the dimension $d = 1, 2, \ldots, k$ is denoted $p_i^d = xor(i, 2^{d-1})$, where xor is "bit-wise exclusive OR".

Function *swap* expresses a pattern of hypercube behaviour:

$$\begin{aligned}
&swap : nat \to (((\alpha,\alpha) \to \alpha), ((\alpha,\alpha) \to \alpha)) \to [\alpha] \to [\alpha] \\
&hyp\ \ (swap\ d\ (\oplus, \otimes)\ x)\ \ i \ = \ (hyp\ x\ i) \oplus (hyp\ x\ p_i^d)\ , \quad \text{if } i < p_i^d \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad (hyp\ x\ p_i^d) \otimes (hyp\ x\ i)\ , \quad \text{otherwise} \\
&\quad \text{where}\ \ length\, x \ = \ 2^k\ ,\ 1 \leq d \leq k\ ,\ 0 \leq i < 2^k .
\end{aligned}$$

From the definition it follows that $swap\, d$ consists of pair-wise, two-directional communications between all neighbours in dimension d of the hypercube, followed by a computation in each processor. The following theorem establishes the correspondence between one step of (12) and one application of *swap*.

Theorem 12 *For a list x of length 2^k and $1 \leq d \leq k$, the following holds:*

$$step\ 2^{d-1}\ (\oplus, \otimes)\ x \quad = \quad swap\ d\ (\oplus, \otimes)\ x \tag{13}$$

Denoting: $\bigcirc_{d=1}^{k} (swap\ d\ (\oplus, \otimes)) \ = \ swap\, k\ (\oplus, \otimes) \ \circ \ \cdots \ \circ \ swap\, 1\ (\oplus, \otimes)$, we obtain from Theorems 11, 12 the following:

Corollary 13 (DH on Hypercube) *For a list x of length 2^k:*

$$(\oplus \updownarrow \otimes)\ x \ = \ \Big(\bigcirc_{d=1}^{k} (swap\ d\ (\oplus, \otimes)) \Big)\ x \tag{14}$$

Therefore, every DH on a list of length 2^k can be computed on the 2^k-node hypercube by a sequence of *swap*s, with the dimensions counted from 1 to k.

4.3 Scan as DH: Adjustment and Implementation

In this subsection, we apply the generic DH implementation (14) to the *scan*-function. Let us first check whether *scan* is DH. Its combine operator, $\circledast$, extracted by our `cons` & `snoc` method, reads as follows:

$$\begin{aligned} &scan\,(\odot)\,(x +\!\!+ y) \ = \ S_1 \circledast S_2 \ = \ S_1 +\!\!+ \ map\ ((last\, S_1) \odot)\ S_2, \\ &\quad \text{where } S_1 \ = \ scan\,(\odot)\, x\ ,\ S_2 \ = \ scan\,(\odot)\, y. \end{aligned} \tag{15}$$

Evidently, the right-hand side of (15) does not match format (10), so *scan* is not DH. Fortunately, the idea of introducing auxiliary functions and tupling them can be used again. This time, our goal is to find a closed definition in the component-wise format. We illustrate how the method works for the *scan* function, the generalization for arbitrary tuples is obvious.

The left argument of $+\!\!+$ in (15) can be expressed component-wise immediately: $S_1 \ = \ zip\ \pi_1\ (S_1, S_2)$. To express the right argument of $+\!\!+$ component-wise, we replicate element $last(S_1) \ = \ last\,(scan\,(\odot)\, x) \ = \ red\,(\odot)\, x$, which yields $redd\,(\odot)\, x$. Using *redd* as an auxiliary function, we obtain:

$$S_1 \circledast S_2 \ = \ zip\, \pi_1\ (S_1, S_2) \ +\!\!+ \ zip\,(\odot)\,(R_1, S_2) \tag{16}$$

$$R_1 \circledast R_2 \ = \ zip\,(\odot)\,(R_1, R_2) \ +\!\!+ \ zip\,(\odot)\,(R_1, R_2) \tag{17}$$

where $R_1 \ = \ redd\,(\odot)\, x\ ,\ R_2 \ = \ redd\,(\odot)\, y$, and (17) follows directly from (11).

We have thus obtained a closed, component-wise definition for tuple function $\langle scan\,(\odot)\ ,\ redd\,(\odot)\rangle$, so no other auxiliary functions are needed. The expression of *scan* adjusted to the DH format follows from (16) and (17)

$$scan\,(\odot) \;=\; (map\;\pi_1) \;\circ\; (\,\oplus\updownarrow\otimes\,) \;\circ\; (map\;pair)\,, \tag{18}$$

where

$$\begin{aligned} pair\,a &= (a, a) \\ (s_1, r_1) \oplus (s_2, r_2) &= (s_1\;,\; r_1 \odot r_2) \\ (s_1, r_1) \otimes (s_2, r_2) &= (r_1 \odot s_2\;,\; r_1 \odot r_2) \end{aligned} \tag{19}$$

Substituting the implementation schema (14) in (18), we obtain a hypercube program for *scan* on a list of length 2^k:

$$scan\,(\odot) \;=\; (map\;\pi_1) \circ \Big(\; \bigcirc_{d=1}^{k} (swap\;d\;(\oplus, \otimes)) \;\Big) \;\circ\; (map\;pair) \tag{20}$$

where *pair*, $\oplus$, $\otimes$ are defined by (19).

This is the well-known "folklore" implementation [25]. In Figure 3, it is illustrated for the 2-dimensional hypercube which is computing $scan\,(+)\,[1, 2, 3, 4]$.

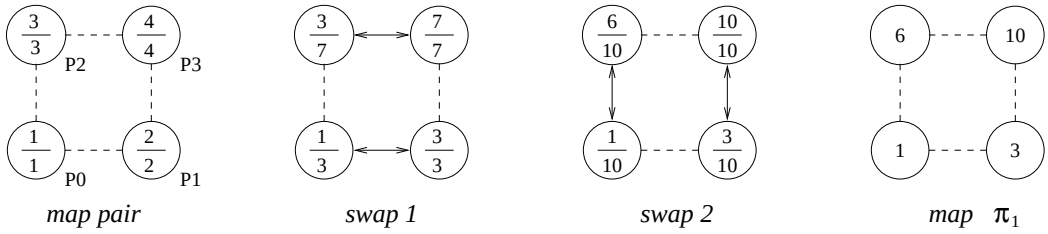


Fig. 3. Computing *scan* on a hypercube

At this phase of program development, where all three stages in (20) are parallel, we can generate the SPMD target program with explicit message passing, which computes $scan\,(\odot)$ on hypercube:

$map\;\;pair \quad \rightarrow$

```
[ (s, r) := x;
```

$\bigcirc_{d=1}^{k} (swap\;d\;(\oplus, \otimes)) \quad \rightarrow$

```
[ For d := 1 To k Do
   partner := xor (my-id, 2**(d-1))
   If (my-id < partner)
    Then SEND (s,r) TO partner;
         RECV (t,u) FROM partner;
         (s,r) := (s, r ⊙ u);
    Else RECV (t,u) FROM partner;
         SEND (s,r) TO partner;
         (s,r) := (r ⊙ t, r ⊙ u);
   End-If;
  End-For;
```

$map\;\;\pi_1 \quad \rightarrow$

```
[ y := s;
```

The clear stage structure of (20), preserved in the imperative target program, helps to analyze the program performance. There are three stages: pairing, repeating swaps and projecting. For a list of length $n = 2^k$, we use n processors. Both the pairing and the projection stage require constant time. The central stage is the sequential loop with k swaps. In each swap, pairs of elements are communicated, and computations are performed on pairs as well, so every swap requires constant time. Hence the parallel time is $O(k) = O(\log n)$ and our implementation is *time optimal*. The cost (time-processor product [25]) is $O(n \log n)$, whereas the cost of the sequential computation is $O(n)$, so the implementation is *not cost optimal*. To improve it, we must use fewer processors, with each processor holding a segment rather than one element of the input list.

4.4 Bounded Number of Processors

Let us now consider a more practical situation in which the number of processors, p, is arbitrary but fixed, i.e. $p < n$, where p divides n. We introduce the type $[\alpha]_p$ of lists of length p and use the notation map_p, zip_p, etc. for functions defined on such lists.

Processors work on partitioned lists. Partitioning over 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:

$$red(+\!\!+) \circ dist^{(p)} = id \tag{21}$$

Homomorphisms have the following important property.

Theorem 14 (Promotion [5]) *For a $\circledast$-homomorphism h,*

$$h \circ red(+\!\!+) = red(\circledast) \circ map\, h \tag{22}$$

We apply the promotion theorem to $(\oplus \updownarrow \otimes)$ which, according to definition (10), has combine operator $\circledast = (+\!\!+) \circ \langle zip(\oplus), zip(\otimes)\rangle$, defined on powerlists of equal length. The transformation proceeds as follows:

$$\begin{array}{ll} & \oplus\updownarrow\otimes \\ = & \quad \{ \text{ equality (21), two times } \} \\ & (red(+\!\!+) \circ dist^{(p)}) \circ \oplus\updownarrow\otimes \circ (red(+\!\!+) \circ dist^{(p)}) \\ = & \quad \{ \text{ associativity of } \circ, \text{ promotion law (22) } \} \\ & red(+\!\!+) \circ (dist^{(p)} \circ red((+\!\!+) \circ \langle zip(\oplus), zip(\otimes)\rangle) \circ map_p(\oplus\updownarrow\otimes)) \circ dist^{(p)} \end{array}$$

We separate the first, distributing and the last, flattening stage of the result expression. The remaining, middle part, both accepts and yields distributed data of type $[[\alpha]]_p$. For an arbitrary function $h : [\alpha] \to [\alpha]$, we call such an expression the *p-distributed version* of h and use notation $(\widetilde{h})_p$ for it, such that

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

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

The p-distributed version of DH is thus:

$$\left(\widetilde{\oplus\updownarrow\otimes}\right)_p \;=\; dist^{(p)}\;\circ\;red\,((+\!\!+)\circ\langle zip\,(\oplus), zip\,(\otimes)\rangle)\;\circ\;map_p\,(\oplus\updownarrow\otimes) \qquad (23)$$

Program (23) suffers twice from linear communication costs: it concatenates lists by applying operator $((+\!\!+)\circ\langle zip\,(\oplus), zip\,(\otimes)\rangle)$ and then redistributes the result again. Our goal is to cut these costs down. We begin by introducing an analog of the promotion property for function *zip*.

Theorem 15 (Promotion of Zip) *For partitioned powerlists, x and y, of equal length and equal block size,*

$$zip\,(\oplus)\,(red\,(+\!\!+)\,x\;,\;red\,(+\!\!+)\,y) \;=\; red\,(+\!\!+)\,(zip\,(zip\,(\oplus))\,(x,y)) \qquad (24)$$

This theorem will be directly used in proving the following major fact.

Theorem 16 (Distributed DH) *For an argument of type $[\,[\alpha]\,]_p$,*

$$\left(\widetilde{\oplus\updownarrow\otimes}\right)_p \;=\; \left((zip\,\oplus)\updownarrow(zip\,\otimes)\right)_p\;\circ\;map_p\,(\oplus\updownarrow\otimes) \qquad (25)$$

PROOF. Applying (21) to (23) and noting that, for $x \in [\alpha]_p$, the following holds: $dist^{(p)}\,x \;=\; map_p\,[.]\,x$, we obtain:

$$\begin{aligned}\left(\widetilde{\oplus\updownarrow\otimes}\right)_p \;=\; dist^{(p)}\;\circ\;red\,((+\!\!+)\circ\langle zip\,(\oplus), zip\,(\otimes)\rangle)\;\circ\;\\ red\,(+\!\!+)\;\circ\;map_p\,[.]\;\circ\;map_p\,(\oplus\updownarrow\otimes)\end{aligned} \qquad (26)$$

For an argument of type $[\,[\,[\alpha]\,]_1]_p$, which is the output of stage $map_p\,[.]$ in (26), we prove, by induction on $p = 2^k$, that:

$$red\,(+\!\!+ \circ \langle zip\,(\oplus), zip\,(\otimes)\rangle) \;\circ\; red\,(+\!\!+) \;= \qquad (27)$$
$$red\,(+\!\!+) \;\circ\; red\,((+\!\!+) \circ \langle zip\,(zip\,(\oplus)), zip\,(zip\,(\otimes))\rangle)$$

Induction base: $p = 1$. The argument x of (27) is then of type $[\,[\,[\alpha]\,]_1]_1$, so $x = [\,[z]\,]$, where $z \in [\alpha]$. Since reduction on a singleton list yields the element of the list both LHS and RHS of (27) are equal to z.

Induction hypothesis: (27) holds for $p = 2^k$.

Induction step: an argument powerlist of length $p = 2^{k+1}$ can be represented as $x +\!\!+ y$, where $x, y \in [\,[\,[\alpha]\,]_1]_{2^k}$.

By calculation:

$$\begin{array}{ll}
 & \Big(red\,((+\!\!+) \circ \langle zip\,(\oplus), zip\,(\otimes)\rangle) \;\circ\; red\,(+\!\!+)\Big)\,(x \;+\!\!+\; y) \\
= & \quad \{\; red\,(\circledast) \text{ is a homomorphism with combine operator } \circledast \;\} \\
 & red\,((+\!\!+) \circ \langle zip\,(\oplus), zip\,(\otimes)\rangle)\;(red\,(+\!\!+)\,x \;+\!\!+\; red\,(+\!\!+)\,y) \\
= & \quad \{\; \text{homomorhic property of reduction again} \;\} \\
 & zip\,(\oplus)\;\Big(\;(red\,((+\!\!+) \circ \langle zip\,(\oplus), zip\,(\otimes)\rangle) \;\circ\; red\,(+\!\!+))\,x\,, \\
 & \qquad\qquad (red\,((+\!\!+) \circ \langle zip\,(\oplus), zip\,(\otimes)\rangle) \;\circ\; red\,(+\!\!+))\,y\;\Big) \;+\!\!+ \\
 & zip\,(\otimes)\;\Big(\;(red\,((+\!\!+) \circ \langle zip\,(\oplus), zip\,(\otimes)\rangle) \;\circ\; red\,(+\!\!+))\,x\,, \\
 & \qquad\qquad (red\,((+\!\!+) \circ \langle zip\,(\oplus), zip\,(\otimes)\rangle) \;\circ\; red\,(+\!\!+))\,y\;\Big) \\
= & \quad \{\; \text{induction hypothesis for } x \text{ and } y \;\} \\
 & zip\,(\oplus)\;\Big(\;red\,(+\!\!+)\,(red\,((+\!\!+) \circ \langle zip\,(zip\,(\oplus)), zip\,(zip\,(\otimes))\rangle)\,x)\,, \\
 & \qquad\qquad red\,(+\!\!+)\,(red\,((+\!\!+) \circ \langle zip\,(zip\,(\oplus)), zip\,(zip\,(\otimes))\rangle)\,y)\;\Big) \;+\!\!+ \\
 & zip\,(\otimes)\;\Big(\;red\,(+\!\!+)\,(red\,((+\!\!+) \circ \langle zip\,(zip\,(\oplus)), zip\,(zip\,(\otimes))\rangle)\,x)\,, \\
 & \qquad\qquad red\,(+\!\!+)\,(red\,((+\!\!+) \circ \langle zip\,(zip\,(\oplus)), zip\,(zip\,(\otimes))\rangle)\,y)\;\Big) \\
= & \quad \{\; \text{promotion (24) for } zip\,(\oplus) \text{ and } zip\,(\otimes) \;\} \\
 & red\,(+\!\!+)\;\Big(\;zip\,(zip\,(\oplus))\,(red\,((+\!\!+) \circ \langle zip\,(zip\,(\oplus)), zip\,(zip\,(\otimes))\rangle)\,x\,, \\
 & \qquad\qquad red\,((+\!\!+) \circ \langle zip\,(zip\,(\oplus)), zip\,(zip\,(\otimes))\rangle)\,y)\;\Big) \;+\!\!+ \\
 & red\,(+\!\!+)\;\Big(\;zip\,(zip\,(\otimes))\,(red\,((+\!\!+) \circ \langle zip\,(zip\,(\oplus)), zip\,(zip\,(\otimes))\rangle)\,x\,, \\
 & \qquad\qquad red\,((+\!\!+) \circ \langle zip\,(zip\,(\oplus)), zip\,(zip\,(\otimes))\rangle)\,y)\;\Big) \\
= & \quad \{\; \text{property of } red\,(+\!\!+) \;\} \\
 & \Big(\;red\,(+\!\!+) \;\circ\; ((+\!\!+) \circ \langle zip\,(zip\,(\oplus)), zip\,(zip\,(\otimes))\rangle)\;\Big)\;\Big(\;red\,(+\!\!+ \circ \\
 & \langle zip\,(zip\,(\oplus)), zip\,(zip\,(\otimes))\rangle\,x\,,\; red\,((+\!\!+) \circ \langle zip\,(zip\,(\oplus)), zip\,(zip\,(\otimes))\rangle\;y) \\
= & \quad \{\; \text{definition of reduction} \;\} \\
 & \Big(\;red\,(+\!\!+) \;\circ\; red\,((+\!\!+) \circ \langle zip\,(zip\,(\oplus)), zip\,(zip\,(\otimes))\rangle)\;\Big)\;(x \;+\!\!+\; y)
\end{array}$$

Applying (27) (just proved), together with (21), to (26) and noting that expression $\Big(\; red\,((+\!\!+) \circ \langle zip\,(zip\,(\oplus)), zip\,(zip\,(\otimes))\rangle) \;\circ\; map_p\,[.]\;\Big)$ represents the

homomorphism $(zip\,\oplus)\updownarrow(zip\,\otimes)$, we obtain the desired property (25).

Theorem 16 is proved.

To map the architecture-independent solution (25) onto the hypercube of p processors, we use (14) with $k = \log p$, which yields:

$$\left(\widetilde{\oplus\updownarrow\otimes}\right)_p \;=\; \left(\bigcirc_{d=1}^{\log p} swap_p\ d\ (zip\,(\oplus)\,, zip\,(\otimes)\,)\right) \circ\ map_p\,(\oplus\updownarrow\otimes) \qquad (28)$$

Program (28) provides a generic, provably correct implementation of a DH function 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 (28), for an input list of length n and p processors, 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 component-wise computations on them at each step; its time complexity is thus $O\,((n/p)\cdot\log p)$. For functions whose sequential time complexity is $O\,(n\log n)$, e.g., Fast Fourier transform, the first stage dominates, so program (28) is both time and cost optimal.

Therefore, cutting down the linear communication costs for class DH pays off for many practical functions: the generic program (28) provides the optimal solution for them. However, for functions of linear time complexity, like *scan*, the second stage in (28) still dominates the total time, and thus the generic program is not optimal. We address this difficulty in the next subsection.

4.5 Localization Schema and Scan

For some concatenating homomorphisms, the DH format (10) is still too general. For instance, the combine operator of *scan*, defined by (15), does not make use of list v on the left of $+\!\!+$ and uses only one, the last, element of u on the right of concatenation! This enables so-called *localization* of the reduction stage: communication and computations are performed on a local portion, possibly on one element, of each block, rather than on the whole blocks.

Definition 17 *A localization schema of a homomorphism with combine operator* $\circledast\ =\ (+\!\!+)\circ\langle f, g\rangle$ *is the representation of the reduction stage:*

$$red\,(\circledast)\ =\ red\,(+\!\!+)\ \circ\ zip\,join\ \circ\ \langle compute\ \circ\ map\,select\,,\ id\rangle\,, \qquad (29)$$

with parameters $select : [\alpha]\to\alpha$, $compute : [\alpha]\to[\alpha]$, $join : (\alpha,[\alpha])\to[\alpha]$.

Intuitively, schema (29) picks one element of each block by *select* and computes a new value for each block by function *compute*, using selected elements of all blocks. This value is then used, together with the initial block preserved by *id*, in the concluding computation by function *join*.

If we can find a suitable localization schema for a homomorphism h with combine operator $\circledast$, then its p-distributed version can be derived as follows:

$$
\begin{aligned}
& (\widetilde{h})_p \;=\; dist^{(p)} \circ h \circ red(+\!\!+) \\
= \quad & \{ \text{ promotion (22) } \} \\
& dist^{(p)} \circ red(\circledast) \circ map_p\, h \\
= \quad & \{ \text{ localization (29) } \} \\
& dist^{(p)} \circ red\,(+\!\!+) \circ zip_p\; join \circ \langle compute_p \circ map_p\; select\,,\; id \rangle \circ map_p\, h \\
= \quad & \{ \text{ for an argument of type } [\alpha]_p,\; dist^{(p)} \circ red\,(+\!\!+) \;=\; id \; \} \\
& zip_p\; join \circ \langle compute_p \circ map_p\; select\,,\; id \rangle \circ map_p\, h
\end{aligned}
$$

Thus the generic program after localization reads as follows:

$$(\widetilde{h})_p \;=\; zip_p\; join \circ \langle compute_p \circ map_p\; select\,,\; id \rangle \circ map_p\, h \qquad (30)$$

Let us construct a localization schema for *scan*. The natural candidate for the parameter function *select* is *last*. The customizing operator *join* is of the form $\odot\!\!\triangleleft$, such that $a \odot\!\!\triangleleft\; u \;=\; map\;(a \odot)\; u$. The *compute* function for *scan* is function *prescan*, which yields the result of *scan*, "shifted to the right":

$$prescan\,(\odot)\,[x_1, x_2, \cdots, x_n] \;=\; [0_\odot,\; x_1,\; x_1 \odot x_2,\; \cdots,\; x_1 \odot x_2 \odot \cdots \odot x_{n-1}]$$

where $0_\odot$ is the neutral element of $\odot$. Despite its close similarity to *scan*, function *prescan* is not a homomorphism: e.g., $prescan\,(+)\;[x_1, x_2, x_3, x_4]$ which is $[0, x_1, x_1 + x_2, x_1 + x_2 + x_3]$ cannot be expressed via $prescan\,(+)\;[x_1, x_2] \;=\; [0, x_1]$ and $prescan\,(+)\;[x_3, x_4] \;=\; [0, x_3]$, because x_2 is "lost". Fortunately, *prescan* can be adjusted to the DH-format using exactly the same tupling method as for *scan* in Subsection 4.3. The only difference is in the pairing function, which for *prescan* is of the form: $prepair\, a \;=\; (0_\odot, a)$.

Substituting the parameters $select = last, compute = prescan, join = \odot\!\!\triangleleft$ in (30) and using the *prescan* analog of (20), we obtain the following target program for $scan\,(\odot)$ on p processors:

$$\left(\widetilde{scan\,(\odot)} \right)_p \; x \;=\; zip_p \; (\odot\!\!\triangleleft) \; (y \,,\, z)\,, \tag{31}$$

where $z \;=\; map_p \; (scan\,(\odot))\; x$

$$y \;=\; \Big(map_p \; \pi_1 \;\circ\; \bigcirc_{d=1}^{\log p} (swap_p \; d \; (\oplus, \otimes)) \;\circ\; map_p \; (prepair \circ last)\Big)\; z$$

with $\oplus, \otimes$ from (19).

Functions map_p and zip_p in (31) imply that all p processors execute the same program. In case of $swap_p$, the individual processor's program depends on the processor's coordinate in the hypercube. Therefore, BMF-program (31) represents the SPMD-style parallel target program that computes function *scan* on list x, where x is partitioned block-wise between p processors.

The program is a sequence of the following three stages:

- *Compute* z: each processor independently computes the *scan* function 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 (19). The first element of the result pair is assigned to variable y.
- *Compute the result*: each processor computes $y \odot\!\!\triangleleft z$; here, no communication is necessary.

This is exactly the implementation of *scan* used in practice on a hypercube with an arbitrary fixed number of processors [25]. The time complexity of both first and third stages is $O(n/p)$ since both *scan* and $\odot\!\!\triangleleft$ are linear. The second stage consists of $\log p$ steps, with communications and computations on pairs of elements, which yields time $O(\log p)$. Therefore, the total time complexity of program (31) is $O(n/p + \log p)$, the best one can expect on p processors. This is a clear improvement over the generic implementation (28) of DH.

Another application of a localization schema is the so-called *parallel suffix* which yields the list of prefix sums in a list inspected from right to left.

5 Related Work

Our work is inspired by the current research on systematic methods of deriving correct and efficient programs for parallel machines by transformation.

Our approach to the homomorphism extraction can be compared with work by Grant–Duff and Harrison [15] since we consider the same examples. On

simple examples like *length*, we actually do the same. For *scan*, rather involved calculations and intuition are required in [15] to obtain the combine operator; our `cons` & `snoc` method arrives at the result in a systematic way. For almost-homomorphisms, the method of [15] is not suitable at all. In an earlier work by Barnard *et al.* [3] and previously cited papers by J. Gibbons [12,11], the existence of leftwards and rightwards algorithms is used as evidence that a homomorphic algorithm exists; unlike our approach, the authors do not provide a method to derive it.

Our solutions for the maximum segment sum problem are similar to those presented by Smith [29] and Cole [8]. Our contribution is the systematic `cons` & `snoc` method which firstly provides a uniform way of introducing the necessary auxiliary functions and secondly exploits a rigorous generalization procedure for deriving the resultant combine operator on tuples. A similar observation applies to the case of nested almost-homomorphisms when compared to [8]. It has been recently brought to our attention that the idea of tupling resembles the technique called "strengthening the invariant" used in imperative program derivation [17].

Parallelization of the *scan* function has a rich history, starting from the seminal work by Ladner and Fisher [20], who, reworking earlier results by other authors, show that parallel prefix can be computed in logarithmic time on a linear number of processors. Blelloch [6] stresses the wide applicability of scans and proposes their use as parallel language primitives. Meanwhile, parallel algorithms for computing scans are a part of folklore [25], and are usually presented in an *ad hoc* manner. On inspection of these algorithms, one is convinced that they really compute the *scan*-function, but the reasons for the *eureka* decisions and their applicability to functions other than *scan* remain unclear.

There are systematic approaches to *scan* parallelization, which we compare to our work. Mou [22] specifies the *scan*-algorithm within a general algebraic model of divide-and-conquer and presents [7] an optimization, which is similar to our maximally parallel version. However, the tuple structure arises as a result of a non-formal argument and the optimization is not formally proved. An algorithm for an unlimited number of processors has been formally verified by O'Donnell [23] (interestingly enough, the author cites, as one of the reasons for addressing this problem, the fact that some published parallel *scan* algorithms were erroneous), and later formally derived by J. Gibbons [10]. A similar two stage tree algorithm has been mapped to the mesh topology by A. Gibbons and Ziani [9]. Kornerup [18] formally arrives at the algorithm by Ladner and Fisher in the recursive powerlist notation; the algorithm requires a linear number of processors and Gray encoding of the hypercube nodes.

Our approach differs from previous results in that our target implementation:

(1) is a result of a systematic, provably correct adjustment and specialization process applicable to a broader class of algorithms, (2) is obtained in an iterative form, where all stages of computations and communication can be seen explicitly, and (3) is realizable on both linear and bounded number of processors with predictable performance. The *scan* algorithm for a bounded number of processor has, to the best of our knowledge, neither been formally derived nor explained methodologically elsewhere before.

Our implementation restriction to powerlists originates from work by Misra [21]. Unlike him, we get rid of the explicit recursion in the target program by introducing iterative constructs. The target program in the form of a sequential composition of parallel stages has the following advantages: it is easier to understand, is directly transformable into an SPMD program and allows for a simple performance prediction. Our proofs are not more complex, since we use the semantically sound transformations of BMF, where recursion is hidden in the higher-order functions. However, our approach is more restrictive, since we do not consider the list interleaving constructor $\bowtie$, used by Misra.

The ideas of transformations with distributed data are initially due to [27,28], we extend and specialize them by introducing new rules. The construction of function *iter* is a special case of the Compound List Operators of Kumar and Skillicorn [19], which we use here for a different purpose from the original. An approach similar to ours, of deriving an architecture-independent iterative solution and then mapping it onto particular topologies, has been taken by Achatz and Schulte [1]. We consider a more special class DH, which allows us to make use of more powerful transformation rules. Moreover, we aim at MIMD architectures with an arbitrary bounded number of processors, unlike the SIMD model with one element per processor used in [1].

Reid-Miller [26] has implemented an algorithm, similar to ours (31), for the slightly more complex case of data stored as linked lists. This algorithm has reportedly outperformed all other known algorithms for *scan* when $p << n$.

Our implementation schema for DH functions on the hypercube resembles the common structure of *ascending* algorithms studied in the seminal paper by Preparata and Vuillemin [24]. We view this analogy as a promising sign for research towards building a taxonomy of functions with respect to their efficient parallel implementation.

6 Conclusion and Future Work

In this paper, we propose an approach to exploiting homomorphic parallelism in functions on lists, which consists of two steps: first, parallelism extraction

by finding a homomorphic representation of the given function, second, parallelism implementation by adjusting the function to the DH format and using the generic parallel implementation schema.

We claim that our approach is more systematic than previous methods:

- At the *parallelism extraction* step, the user provides two sequential definitions of a given function in a closed form. The requirement of closedness, as we demonstrate by several examples, "guides" the introduction of the necessary auxiliary functions. The rest of the job is done by the generalization procedure.
- At the *parallelism implementation* step, the function must be cast in the DH format, which again serves as a guide for the user. It then remains to customize the generic implementation schema.
- Our approach is applicable to a class of problems: one does not need to start the derivation from scratch for every new specification.

Methodologically, an important feature of the parallelism extraction is that it is based on sequential thinking. Considerations involving data and control dependencies, which are usual in parallelization techniques, are avoided.

The contribution to the implementation methodology is in the definition of the DH class of functions on lists and the formal derivation of a common efficient parallel implementation schema for all functions of the class. The derivation is based on the semantically sound transformation rules of the BMF, which guarantee its correctness. The performance of the target implementation is easily predictable and conforms with the best known estimates.

Both extraction and implementation methods enrich our SAT program development approach [14]. The advantages of the SAT approach, demonstrated in the paper, include: (1) an easy to understand single-threaded program structure which is preserved from the specification phase through to the target program, (2) suitability for program transformations and performance prediction, (3) direct generation of an SPMD program without additional synchronization.

Our current work includes designing a standard generalization procedure for the `cons` & `snoc` method. We have developed an algorithm based on rewriting, which is provably correct and terminating. Although the algorithm is not complete, it works successfully for all homomorphisms known to us, including those presented in this paper. A remaining open question is a systematic way of deriving localization schemata for representative classes of functions. We also work towards extensions of the DH class which still allow for efficient and practically relevant parallel implementations.

Acknowledgement

Special thanks to Murray Cole and Chris Lengauer for proof-reading the complete paper and greatly improving both the contents and the form. I am grateful to Alfons Geser, Jeremy Gibbons and Lambert Meertens for remarks on different parts of the manuscript and to Christoph Wedler for many technical discussions.

The anonymous referees helped a lot to make the manuscript a better paper.

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

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

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

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

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

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

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

[9] A. Gibbons and R. Ziani. The balanced binary tree technique on mesh-connected computers. *Information Processing Letters*, 37:101–109, 1991.

[10] J. Gibbons. Upwards and downwards accumulations on trees. In R. Bird, C. Morgan, and J. Woodcock, editors, *Mathematics of Program Construction*, Lecture Notes in Computer Science 669, pages 122–138, 1992.

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

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

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

[14] S. Gorlatch. 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.

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

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

[17] A. Kaldewaij. *Programming: the derivation of algorithms.* Prentice-Hall, 1990.

[18] J. Kornerup. Mapping a functional notation for parallel programs onto hypercubes. *Information Processing Letters*, 53:153–158, 1995.

[19] K. Kumar and D. Skillicorn. Data parallel geometric operations on lists. *Parallel Computing*, 21(3):447–459, 1995.

[20] R. Ladner and M. Fischer. Parallel prefix computation. *J. ACM*, 27:831–838, 1980.

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

[22] Z. G. Mou. Divacon: A parallel language for scientific computing based on divide and conquer. In *Proc. 3rd Symposium on the Frontiers of Massively Parallel Computation*, pages 451–461, October 1990.

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

[24] F. Preparata and J. Vuillemin. The cube-connected cycles: A versatile network for parallel computation. *Communications of the ACM*, 24(5):300–309, 1981.

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

[26] M. Reid-Miller. List ranking and list scan on the Cray C-90. In *Proceedings SPAA'94*, pages 104–113, 1994.

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

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

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

[30] D. Walker. The design of a standard message passing interface for distributed memory concurrent computers. *Parallel Computing*, 20:657–673, 1994.