# (De)Composition Rules for Parallel Scan and Reduction

Sergei Gorlatch and Christian Lengauer
University of Passau, D-94030 Passau, Germany
{gorlatch,lengauer}@fmi.uni-passau.de

## Abstract

*We study the use of well-defined building blocks for SPMD programming of machines with distributed memory. Our general framework is based on homomorphisms, functions that capture the idea of data-parallelism and have a close correspondence with collective operations of the MPI standard, e.g., scan and reduction. We prove two composition rules: under certain conditions, a composition of a scan and a reduction can be transformed into one reduction, and a composition of two scans into one scan. As an example of decomposition, we transform a segmented reduction into a composition of partial reduction and all-gather. The performance gain and overhead of the proposed composition and decomposition rules are assessed analytically for the hypercube and compared with the estimates for some other parallel models.*

## 1. Introduction

This paper presents an approach to parallel programming which is based on a set of relatively well-understood primitives for parallelism. The goal is to raise the level of abstraction, and to allow the compiler (possibly aided by the user) to perform semantically sound, performance-guided transformations of a program composed of such primitives.

The approach has its roots in the world of functional programming where programming paradigms are captured as *algorithmic skeletons* [4]. An algorithmic skeleton is a higher-order function (a functional schema) which takes as parameters so-called *customizing functions* needed in a specific application. If the parallel nature of the programming paradigm is understood well enough, the algorithmic skeleton can be associated with a number of *architectural skeletons*, each implementing the paradigm on a specific machine model or architecture.

The strength of the skeleton approach lies in a clear division of responsibilities between application programmers and implementers in the development of parallel programs:

- The details of parallelism and communication are hidden in the architectural skeletons and are invisible to the application programmer who is only supplying the algorithmic skeleton with customizing functions. That is, the application programmer need not be concerned with the correctness of the parallelism in his/her program.

- If one stays within the world of functional programming, skeleton programs can be transformed and optimized via equational reasoning. Moreover, the architectural skeletons can be derived formally from the algorithmic skeleton. That is, the implementer has a reliable means of assuring the correctness of his/her implementation.

The viability of the approach depends heavily on the expressive power of the algorithmic skeletons provided to the user – a fact which the skeletons research community is addressing increasingly [5]. Moreover, the usefulness of the programming with skeletons can be improved dramatically if the user is given sound and practical design methods, and the compiler is enhanced with implementation and optimization techniques. The skeletons considered in the literature range from very simple functions like map and reduce to quite complex patterns of parallelism like general divide-and-conquer [9] or more special-purpose skeletons.

We discuss one particular class of skeletons, called *homomorphisms* [1], which lie in the middle of this range. Due to their simplicity, homomorphism skeletons are well suited for the study of transformations in the quest for better performance.

The usefulness of homomorphisms as program building blocks is determined by the following issues:

- *Expressiveness:* How much ground does the class of homomorphisms cover? Simple divide-and-conquer problems can be expressed directly as homomorphisms. Much larger is the class of *almost-homomorphisms*, which can be turned into homomorphisms when tupled with auxiliary functions. Basically, this increses parallelism at the price of extra computations; a method for finding auxiliary functions is proposed in [6, 8]. More complex problems may require a composition or nest of several (almost-)homomorphisms.

- *Implementation:* How can the homomorphism skeleton be implemented efficiently on parallel computers? For illustration, we use the architectural skeleton *swap* which is, on the one hand, formally derivable from a restricted homomorphic form, called *distributable homomorphism (DH)* and, on the other hand, directly implementable on the hypercube.

- *Composition:* Are certain compositions of standard homomorphisms good candidates for new homomorphism skeletons, and can these be optimized further? We derive formally two composition rules for simple homomorphic skeletons used in practice: reduction and scan. How to use the rules in program design is demonstrated in [9].

- *Decomposition:* Can a more complex homomorphism be decomposed into simpler homomorphisms, with the result of improved performance? We present one such rule for segmented reduction, which is available as `MPI_Allred` in the MPI standard [11].

- *Performance:* How portable are skeleton implementations? Whereas the proposed (de)composition rules are implementation-independent, their impact on the target performance depends on the particular implementation of the skeletons and on the parallel machine used. We provide parametrized analytical estimates for the proposed rules in the hypercube model, and demonstrate that the equilibrium of benefits and overhead changes for other models (e.g., BSP [14]) and implementations.

All these issues are studied within a common transformational functional framework. We aim at SPMD programs on distributed memory, with collective operations of the MPI standard [11] as our target language.

## 2. Homomorphisms and BMF

The skeletons we consider are expressed in the *Bird-Meertens formalism (BMF)*, a notation for functional programs [1]. Our functions are defined on non-empty lists, with list concatenation $+\!\!+$ as constructor. We restrict ourselves to finite lists, which can be viewed as vectors and, ultimately, implemented as arrays.

**Definition 2.1 (Bird [1])** *Function $h$ on lists is a* homomorphism *iff there exists a binary operator $\circledast$ such that, for all lists $x$ and $y$:*

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

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 applied in parallel. Note that $\circledast$ in (1) is necessarily associative, because $+\!\!+$ is associative.

The simplest homomorphisms are the basic higher-order functions (also called *functionals*) of BMF: map, reduction and scan, which we introduce informally:

**Map** applies a unary function $f$, defined on elements, to each element of a list, i.e.,

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

The computations of $f$ on different elements of the list can be done independently if enough processors are available.

**Reduction** combines the elements of a list using a binary associative operator $\oplus$:

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

Reduction can be computed on a binary tree in logarithmic time.

**Scan (parallel prefix)** computes the list of running totals with associative operator $\oplus$:

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

The following, arguably, non-obvious definition of *scan* reveals it as a homomorphism:

$$\begin{aligned} \mathit{scan}\,(\oplus)\,[a] \;&=\; [a] \\ \mathit{scan}\,(\oplus)\,(x +\!\!+ y) \;&=\; \mathit{scan}\,(\oplus)\,x \;+\!\!+ \\ &\quad \mathit{map}\,(\mathit{last}\,(\mathit{scan}\,(\oplus)\,x)\,\oplus)\,(\mathit{scan}\,(\oplus)\,y) \end{aligned} \tag{2}$$

Here, we have used the notations $(a \oplus)\, b \stackrel{\text{def}}{=} a \oplus b$, and $last\,[x_1, x_2, \ldots, x_n] = x_n$.

In BMF, programs are constructed of functions by means of functional composition $\circ$, defined by the equation $(f \circ g)\, x = f\,(g\, x)$. SPMD parallelism is introduced by functions like *map*, *red*, and *scan*; sequencing is introduced by composition.

The following properties of homomorphisms are useful for program design:

- **Normal form [1]:** Function $h$ is a homomorphism iff it can be factored into the composition:

$$h = red(\circledast) \circ map\, f \qquad (3)$$

  where $f\, a = h\,[a]$ for every element $a$, and $\circledast$ is from (1). Each homomorphism is uniquely determined by $f$ and $\circledast$.

- **Promotion property [1]:** This property expresses the possibility to compute a homomorphism by partitioning data into several pieces:

$$h \circ red(+\!\!+) = red(\circledast) \circ map\, h \qquad (4)$$

  In words, to compute homomorphism $h$ on the flattened list is the same as to compute $h$ in each sublist (block) and then combine the results by $\circledast$.

- **Transformations:** Functional programs consisting of homomorphisms can be transformed using semantics-preserving equational rules; we will see examples later on.

## 3. Composing Homomorphisms

Sequential composition provides a point of synchronization between the individual parallel implementations of the composed skeletons. This synchronization may be unnecessary. A parallel implementation of the composition may have better performance than the combination of the individual parallel implementations. To ascertain this, specific combinations can be merged through equational transformation and their performance assessed.

We demonstrate that systematic looking for a homomorphism format of a composition can help to find its new parallel implementation.

### 3.1. Scan-Reduce Composition

We are interested in finding a good parallel implementation of a composition of a scan with a subsequent reduction:

$$scanred\,(\oplus, \otimes) \stackrel{\text{def}}{=} red\,(\oplus) \circ scan\,(\otimes) \qquad (5)$$

Following definition (1), we try to express *scanred* on a concatenation of two lists via the value of *scanred* on each of these lists:

$$\begin{array}{l}
\quad scanred\,(\oplus,\otimes)\,(x +\!\!+ y) \\
= \qquad \{\text{ Eq. (5),(2) }\} \\
red(\oplus)(scan(\otimes)x +\!\!+ map(last(scan(\otimes)x)\otimes)(scan(\otimes)y)) \\
= \qquad \{\text{ Def. of } red \text{, and } last(scan(\otimes)) = red(\otimes) \ \} \\
red(\oplus)(scan(\otimes)x) \oplus red(\oplus)(map((red(\otimes)x)\otimes)(scan(\otimes)y)) \\
= \qquad \{\text{ Eq. (5) }\} \\
scanred(\oplus,\otimes)x \oplus red(\oplus)(map((red(\otimes)x)\otimes)(scan(\otimes)y))
\end{array}$$

The obtained expression does not yet fit format (1), because functions different from *scanred* are applied to $x$ and $y$. We could go on looking for a direct formulation of *scanred* as a homomorphism, but instead we use the concept of an almost-homomorphism – a function which becomes a homomorphism if tupled with one or more auxiliary functions.

We pair up *scanred* with auxiliary function *red*:

$$scanred'\,(\oplus,\otimes)\,x \stackrel{\text{def}}{=} (scanred\,(\oplus,\otimes)\,x\,,\ red\,(\otimes)\,x) \qquad (6)$$

Function *scanred'* yields a pair, whose first component can be extracted by projection $\pi_1$:

$$scanred\,(\oplus,\otimes) = \pi_1 \circ scanred'\,(\oplus,\otimes) \qquad (7)$$

Now we aim at a homomorphic format for the augmented function *scanred'*, abbreviating its components with $s = scanred\,(\oplus,\otimes)$ and $r = red\,(\otimes)$:

$$\begin{array}{l}
\quad scanred'\,(\oplus,\otimes)\,(x +\!\!+ y) \\
= \qquad \{\text{ Eq. (6) and abbreviations }\} \\
\quad (s\,(x +\!\!+ y)\,,\ r\,(x +\!\!+ y)) \\
= \qquad \{\text{ Expression for } s\,(x +\!\!+ y)\text{, and def. of } r\ \} \\
(\ s\,x \oplus red\,(\oplus)\,(map\,(r\,x\,\otimes)\,(scan\,(\otimes)\,y))\,,\ r\,x \otimes r\,y\ )
\end{array}$$

Let us assume for the rest of the paper that $\otimes$ *distributes over* $\oplus$, i.e., $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$, or, for an arbitrary number of applications of $\oplus$:

$$(a\,\otimes) \circ red\,(\oplus) = red\,(\oplus) \circ map\,(a\,\otimes) \qquad (8)$$

Then the first component of the pair can be transformed:

$$\begin{array}{ll}
s\,x \oplus (red\,(\oplus) \circ map\,(r\,x\,\otimes))\,(scan\,(\otimes)\,y) & \{\text{Eq. (8)}\} \\
= s\,x \oplus ((r\,x\,\otimes) \circ red\,(\oplus))\,(scan\,(\otimes)\,y) & \{\text{Eq. (5)}\} \\
= s\,x \oplus (r\,x \otimes s\,y) &
\end{array}$$

Therefore, *scanred'* is a homomorphism whose normal form (3) is: $scanred' = red(\langle\oplus,\otimes\rangle) \circ map\,pair$, where:

$$pair\ a \overset{\text{def}}{=} (a, a), \tag{9}$$

$$(s_1, r_1)\langle\oplus,\otimes\rangle(s_2, r_2) \overset{\text{def}}{=} (s_1 \oplus (r_1 \otimes s_2), r_1 \otimes r_2) \tag{10}$$

Here, two new denotations have been introduced: the name *pair* for function which creates a pair of an element, and $\langle\oplus,\otimes\rangle$ for the operator on pairs constructed from $\oplus$ and $\otimes$ according to (10).

The equational transformations we have just presented prove that a composition of scan and reduction can be transformed into a single reduction, with pairing applied beforehand and projection applied afterwards:

**Theorem 1 (Scan-Reduce Composition)** *For arbitrary binary, associative operators $\oplus$ and $\otimes$, such that $\otimes$ distributes over $\oplus$,*

$$red(\oplus) \circ scan(\otimes) = \pi_1 \circ red(\langle\oplus,\otimes\rangle) \circ map\,pair \tag{11}$$

*where function pair and operator $\langle\oplus,\otimes\rangle$ are defined by* (9) *and* (10), *respectively.*

There is also a version of Theorem 1 for a dual version of scan, called suffix, which is used in the formal derivation and optimization of a parallel implementation for the maximum segment sum problem [9].

In Subsection 5, we analyse the impact of the scan-reduce transformation on performance.

## 3.2. Scan-Scan Composition

In this subsection, we study another important composition, namely of two scans. Let us introduce function *inits*, which yields all initial segments of a list:

$$inits\,[x_1, x_2, \ldots, x_n] = [[x_1], [x_1, x_2], \ldots, [x_1, x_2, \ldots, x_n]]$$

and a few standard BMF transformations [17]:

$$\begin{aligned} map\,(f \circ g) &= map\,f \circ map\,g & (12)\\ scan\,(\oplus) &= map\,(red\,(\oplus)) \circ inits & (13)\\ inits \circ map\,f &= map\,(map\,f) \circ inits & (14)\\ inits \circ scan\,(\oplus) &= map\,(scan\,(\oplus)) \circ inits & (15) \end{aligned}$$

The composition $scan(\oplus) \circ scan(\otimes)$ is transformed using these rules and Theorem 1:

$$\begin{aligned} & scan\,(\oplus) \circ scan\,(\otimes) \\ = &\quad \{\ \text{Eq. } (13)\ \} \\ & map\,(red\,(\oplus)) \circ inits \circ scan\,(\otimes) \\ = &\quad \{\ \text{Eq. } (15),(12)\ \} \\ & map\,(red\,(\oplus) \circ scan\,(\otimes)) \circ inits \\ = &\quad \{\ \text{Theorem 1}\ \} \\ & map\,(\pi_1 \circ red\,(\langle\oplus,\otimes\rangle) \circ map\,pair) \circ inits \\ = &\quad \{\ \text{Eq. } (12),(14)\ \} \\ & map\,\pi_1 \circ map\,(red\,(\langle\oplus,\otimes\rangle)) \circ inits \circ map\,pair \\ = &\quad \{\ \text{Eq. } (13)\ \} \\ & map\,\pi_1 \circ scan\,(\langle\oplus,\otimes\rangle) \circ map\,pair \end{aligned}$$

Since we make use of Theorem 1, its assumptions about the distributivity of the involved operators have to be passed on to the new theorem:

**Theorem 2 (Scan-Scan Composition)** *For associative operators $\oplus$ and $\otimes$, where $\otimes$ distributes over $\oplus$,*

$$scan\,(\oplus) \circ scan\,(\otimes) = map\,\pi_1 \circ scan\,(\langle\oplus,\otimes\rangle) \circ map\,pair \tag{16}$$

Again, a version for two suffixes can be proved [9].

## 4. Hypercube Implementation

To apply the composition rules of the previous section and other transformations in the process of parallelization, one should be able to estimate their impact on the performance of the target program. The homomorphism approach provides a relatively small set of standard building blocks such as *map*, *red*, *scan*, etc. Thus, the problem of performance predictability is reduced to studying the implementation of these standard primitives on particular parallel architectures.

We choose here one particular network topology: the hypercube with cut-through routing. As argued in [13], for a large class of problems, hypercube algorithms are asymptotically as fast as the optimal PRAM algorithms, and they can be adapted easily to other topologies, e.g., meshes and multistage networks.

### 4.1. Distributable Homomorphisms (DH)

A specialized subclass of homomorphisms, the class of *distributable homomorphisms (DH)* [7], is particularly suited for the hypercube. A generic, architecture-independent implementation is derived in [10]. In this section, we first introduce briefly some necessary notation and then consider the use of DH in the design of parallel programs. DH is defined on *powerlists* [15] of length $2^k$, $k = 0, 1, \ldots$, with balanced concatenation. The definition makes use of function *zip*, which combines elements of two lists of equal length with operator $\odot$:

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

$$\begin{aligned} \mathit{redd\text{-}seg}\,(+)\,[\,[x_1,y_1,z_1,u_1],[x_2,y_2,z_2,u_2],[x_3,y_3,z_3,u_3],[x_4,y_4,z_4,u_4]\,] &= \\ &[\,[X,Y,Z,U\,],\ [X,Y,Z,U\,],\ [X,Y,Z,U\,],\ [X,Y,Z,U\,]] \\ \text{where } X = \textstyle\sum_{i=1}^{4} x_i,\ Y = \sum_{i=1}^{4} y_i, Z = \sum_{i=1}^{4} z_i,\ U = \sum_{i=1}^{4} u_i. \end{aligned}$$

**Figure 1. Illustration of function redd-seg for m=4, p=4.**

**Definition 4.1** *For binary operators* $\oplus$ *and* $\otimes$, *the* concatenating distributable homomorphism (CDH) *on powerlists, denoted* $\oplus\updownarrow\otimes$, *is defined as follows:*

$$(\oplus\updownarrow\otimes)\,[a] \;=\; [a] \qquad (17)$$

$$(\oplus\updownarrow\otimes)\,(x +\!\!+\, y) \;=\; zip\,(\oplus)\,(u,v) +\!\!+\, zip\,(\otimes)\,(u,v),$$
$$\textit{where } u \;=\; (\oplus\updownarrow\otimes)\,x\,,\ v \;=\; (\oplus\updownarrow\otimes)\,y\,.$$

So, CDH is a parametric function with the customizing operators $\oplus$ and $\otimes$. The specialization of CDH in comparison to the general homomorphism (Definition 2.1) is in the format of the combine operator: elementwise computations on the halves, followed by concatenation.

We introduce also a dual version, *interleaving DH (IDH)*, denoted $\oplus \Updownarrow \otimes$. The definition of IDH is obtained by exploiting operation $\bowtie$ [15] on the right-hand side of (17) instead of concatenation:

$$[x_1, x_2, \ldots, x_n] \bowtie [y_1, y_2, \ldots, y_n] \overset{\text{def}}{=} [x_1, y_1, x_2, y_2, \ldots, x_n, y_n]$$

Thus, there are two versions of DH, concatenating DH (CDH) and interleaving DH (IDH), denoted $\oplus\updownarrow\otimes$ and $\oplus\Updownarrow\otimes$, respectively. We simply speak of a DH in the case of a CDH or IDH, if there is no ambiguity.

As a simple example of DH, 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 see that *redd* is a CDH:

$$redd\,(\odot) \;=\; \odot\updownarrow\odot \qquad (18)$$

This function is provided in MPI under the name of `MPI_Allreduce`.

In [7], a hypercube implementation of DH is developed by introducing an architectural skeleton, *swap*, which describes one characteristic behavior of the hypercube: $swap\,d\,(\oplus,\otimes)$ consists of pairwise, bidirectional communication in dimension $d$, followed by an application of $\oplus$ by the processor with the lower rank and of $\otimes$ by the processor with the higher rank.

We abbreviate multiple compositions of *swaps*, e.g.,:

$$\underset{d=1}{\overset{k}{\bigcirc}} swap\,d\,(\oplus,\otimes) \overset{\text{def}}{=} swap\,k\,(\oplus,\otimes) \circ \ldots \circ swap\,1\,(\oplus,\otimes)$$

**Theorem 3 (DH on a Hypercube [7])** *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 for CDH, and from k to 1 for IDH:*

$$\oplus\updownarrow\otimes \;=\; \underset{d=1}{\overset{k}{\bigcirc}}(swap\,d\,(\oplus,\otimes)) \qquad (19)$$

$$\oplus\Updownarrow\otimes \;=\; \underset{d=k}{\overset{1}{\bigcirc}}(swap\,d\,(\oplus,\otimes)) \qquad (20)$$

Note that the CDH and IDH can be viewed as formal descriptions of the classes of so-called ascending and descending algorithms [16].

## 4.2. Segmented Reduction and Scan

Theorem 3 provides a parallel implementation schema for all DHs. In a particular case, we just need to customize operators $\oplus$ and $\otimes$; for instance, the hypercube program for the distributed reduction (18) is:

$$redd\,(\odot) \;=\; \underset{d=1}{\overset{k}{\bigcirc}}(swap\;d\,(\odot,\odot))$$

The actual `MPI-Allreduce` is a bit more general than our function *redd*: it works also in the case that there is a block (segment) of equal length in each processor. This version is usually called *segmented*: reduction is computed elementwise on all processors. Let us introduce the segmented version, *redd-seg*, which accepts a list of sublists (segments) of equal length, say $m$, and applies operation $\odot$ elementwise on them. Its type is $\mathit{redd\text{-}seg}\,(\odot) : [[\alpha]_m]_p \to [[\alpha]_m]_p$; see Figure 1.

Function *redd-seg* is a DH which works on a list of lists, with the customizing operators which are functions on lists, namely *zips*:

$$\mathit{redd\text{-}seg}\,(\odot) \;=\; (zip\,(\odot))\updownarrow(zip\,(\odot)) \qquad (21)$$

A hypercube implementation for *redd-seg* follows immediately from Theorem 3:

$$\mathit{redd\text{-}seg}\,(\odot) \;=\; \underset{d=1}{\overset{k}{\bigcirc}}(swap\,d\,(zip(\odot), zip(\odot))) \qquad (22)$$

The segmented version of scan, *scan-seg*, is similar to *redd-seg*. It is available in MPI as `MPI_Scan`. Interestingly enough, scan is a homomorphism, but not a DH. However, it can be adjusted to an almost-DH, resulting in the following hypercube implementation for *scan-seg* [7]:

$$\begin{aligned} \mathit{scan\text{-}seg}(\odot) &= \mathit{map}^2\,\pi_1 \circ \\ & \bigcirc_{d=1}^{k} (\mathit{swap}\ d\ (\mathit{zip}(\oplus), \mathit{zip}(\otimes))) \circ \mathit{map}^2\ \mathit{pair} \end{aligned} \tag{23}$$

where $\mathit{map}^2 f \overset{\text{def}}{=} \mathit{map}\,(\mathit{map} f)$, function *pair* is defined by (9), and $\oplus$ and $\otimes$ are defined as follows:

$$\begin{aligned} (s_1, r_1) \oplus (s_2, r_2) &= (s_1\ ,\ r_1 \odot r_2) \\ (s_1, r_1) \otimes (s_2, r_2) &= (r_1 \odot s_2\ ,\ r_1 \odot r_2) \end{aligned} \tag{24}$$

## 5. Performance of Compositions

In this subsection, we estimate the performance of the hypercube implementations for scan and reduction, and then use these estimates to assess the composition rules of the previous section. We have proved the rules for the case of one element per processor, but they are also true for the segmented versions of both reduction and scan: the segments can be viewed as elements.

We assume a hypercube network with the following properties. An elementary operation takes one unit of computation time. Communication links are bidirectional: two neighboring processors can send messages of size $m$ to each other simultaneously in time $t_s + m \cdot t_w$, where $t_s$ is the start-up time and $t_w$ is the per-word transfer time. A processor is allowed to send/receive messages on only one of its links at a time. We ignore the computation time it takes to split or concatenate vectors within a processor.

### 5.1. Scan-Scan Performance

As an example, let us start with estimating the time complexity of scan, according to the DH implementation (23). We ignore the costs of pairing and projecting, since they form just a small additive constant. There are $\log p$ swaps, with $m$ elements communicated and one or two operations performed in each processor according to (24). This yields a time of

$$\log p \cdot (t_s + m \cdot (t_w + 2))$$

The composition of two scans takes time

$$2 \cdot \log p \cdot (t_s + m \cdot (t_w + 2))$$

which is the time complexity of the left-hand side of composition rule (16).

The right-hand side of rule (16) performs pairing at the beginning and projection at the end, whose time we ignore as well. The rest is a usual scan but, this time, on a list of pairs, with the base operation defined by (10). Thus, the right-hand side of (16) requires a time of

$$\log p \cdot (t_s + m \cdot (2 \cdot t_w + 6))$$

Now, it is easy to calculate when optimization (16) pays off:

$$t_s + 2 \cdot m \cdot t_w + 6 \cdot m \ < \ 2 \cdot t_s + 2 \cdot m \cdot t_w + 4 \cdot m$$

which simplifies to

$$t_s \ > \ 2 \cdot m$$

Thus, in the hypercube model, the optimization pays off if the machine has a relatively high start-up cost and/or if the length of the blocks held in the processors is comparatively small. This result is in line with intuition, since we trade synchronization costs (expressed by the start-up) for additional computations (which grow with the vector length). For very long vectors, the optimization may be impractical anyway because of the additional memory consumption.

### 5.2. Scan-Reduce Performance

The time required by the distributed reduction implemented by program (22) is:

$$\log p \cdot (t_s + m \cdot (t_w + 1)) \tag{25}$$

Together with the time for scan from the previous subsection, this gives us the time for the left-hand side of composition rule (11):

$$\log p \cdot (2 \cdot t_s + m \cdot (2 \cdot t_w + 3))$$

The time for the right-hand side of rule (11) is:

$$\log p \cdot (t_s + m \cdot (2 \cdot t_w + 3))$$

Therefore, optimization (11) always pays off on a hypercube. The speed-up approaches 2 for relatively small blocks and high start-up cost. The main overhead here is the double memory requirement.

### 5.3. Performance in BSP and Other Models

The estimates presented in the previous subsection apply only to the DH-induced hypercube implementation based on a logarithmic sequence of swaps along the hypercube dimensions. Other implementations of scan and reduction may exemplify completely different communication patterns and, thus, must be assessed separately.

Let us assess the scan-scan transformation in the BSP model. As shown in [12], the optimal, "transpose scan" algorithm has BSP costs:

$$2 \cdot (m \cdot g + l) + m$$

where $l$ is the cost of the barrier synchronization, and $g$ is the single-word delivery cost, both normalized by the instruction rate of the processors. The condition under which optimization (16) pays off is as follows:

$$4 \cdot (m \cdot g + l) + 2 \cdot m \ > \ 2 \cdot (2 \cdot m \cdot g + l) + 3 \cdot m$$

which simplifies to

$$2 \cdot l \ > \ m$$

which, interestingly enough, is quite similar to the estimate obtained in Subsection 5.1 for the hypercube algorithm. Since the value of $l$ for big configurations ranges from 500 on a Cray T3D to $3 \cdot 10^5$ on a Parsytec GCel, rule (16) does improve performance for a broad range of segment lengths.

Yet quite different assessment applies for a mesh-connected transputer network. In the MPI implementation available to us on this machine (MPICH 1.0), scan is implemented by a linear shift across the processors, with buffered send-receive message exchange. Obviously, this linear-time implementation is suboptimal and works well only for small processor configurations. Regarding the composition rules, the shift implementation behaves quite differently from the optimal (logarithmic-time) implementation. The scan-reduce optimization becomes extremely advantageous for large configurations, since it simply eliminates the inefficient scan (the speed-up rises to 2 on approximately 20 processors, and grows further for larger configurations). In contrast, two shift scans can be executed very quickly in sequence, due to the pipeline effect. Fusing them into one, more complicated shift scan reduces performance: we observe a slow-down of up to 2 on the transputer network.

When using the transformation rules for MPI programs, one has to take into account whether special hardware for performing particular collective MPI operations, e.g., reduction or scan, is available on the machine in question [13]. Thus, transformations must be assessed individually for a given implementation of the communication primitives on a given machine.

## 6. Decomposing Homomorphisms

In this section, we look more closely at the implementation of the segmented reduction, `MPI_Allred`. In the time estimate (25), even if we ignore the communication cost by assuming $t_s = t_w = 0$, the required time is $m \cdot \log p$, which means that the parallel speed-up is not better than $p/\log p$ on $p$ processors. Thus, as a monolithic homomorphism, reduction has suboptimal performance. The reason is easy to see: since the same result segments are computed on all processors, we perform many redundant computations.

### 6.1. Decomposition Rule for Reduction

In this section, we optimize the segmented reduction by decomposing it into two homomorphisms. The idea is, first, to compute different parts of the result in different processors and, second, to combine them in each processor. For simplicity, this version is explained under the assumption that arguments are of type $[[\alpha]_m]_p$, where both $m$ and $p$ are powers of 2, and $p \leq m$.

Let us introduce function *redd-parts* (for *reduction-in-parts*) which does the first part of the job. It has type

$$\textit{redd-parts}\,(\odot) \ : \ [[\alpha]_m]_p \ \rightarrow \ [[\alpha]_{m/p}]_p$$

We illustrate it on a list of two sublists, each with four elements, i.e., $m = 4, p = 2$:

$$\textit{redd-parts}\,(+)\,[\,[x_1, y_1, z_1, u_1], [x_2, y_2, z_2, u_2]\,] \ = \ [\,[x_1 + x_2\,,\ y_1 + y_2]\,,\,[z_1 + z_2\,,\ u_1 + u_2]\,]$$

This function can be defined as an IDH, if we assume that $\odot$ is both associative and commutative:

$$\textit{redd-parts}\,(\odot) \ = \ (\textit{left} \circ \textit{zip}(\odot)) \Updownarrow (\textit{right} \circ \textit{zip}(\odot)) \quad (26)$$

Here, functions *left* and *right* yield the left and the right halves of a list, respectively. Thus, the result is always of half the length of the arguments, for example:

$$(\textit{left} \ \circ \ \textit{zip}\,(+))\,([x_1, y_1, z_1, u_1], [x_2, y_2, z_2, u_2]) \ = \ [x_1 + x_2\,,\ y_1 + y_2\,]$$
$$(\textit{right} \ \circ \ \textit{zip}\,(+))\,([x_1, y_1, z_1, u_1], [x_2, y_2, z_2, u_2]) \ = \ [z_1 + z_2\,,\ u_1 + u_2\,]$$

After computing *redd-parts*$(\odot)$, each processor keeps a different part of the result, and it remains to combine those parts in each processor. This can be accomplished in MPI by the primitive `MPI_Allgather`. We model it by function *allgather*, which accepts a list

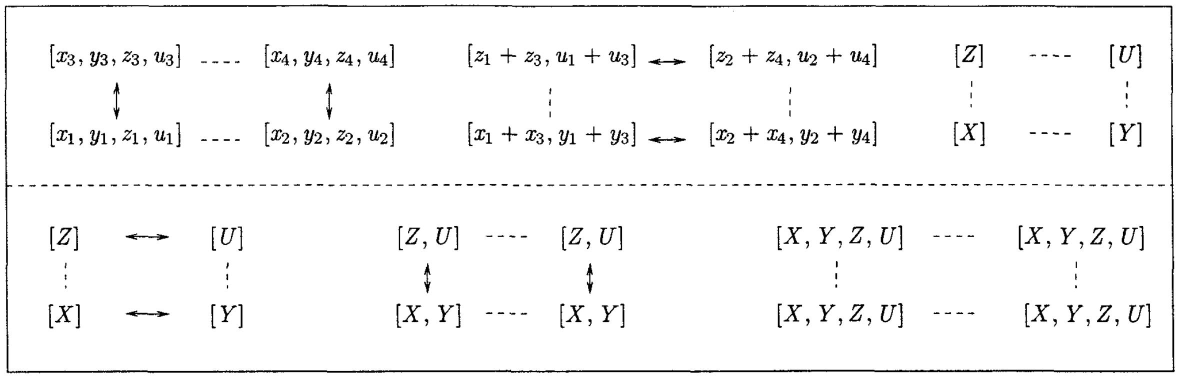

**Figure 2. Decomposed reduction on a hypercube:** $X = \sum_{i=1}^{4} x_i$, $Y = \sum_{i=1}^{4} y_i$, $Z = \sum_{i=1}^{4} z_i$, $U = \sum_{i=1}^{4} u_i$.

of segments and returns the list whose elements are all equal to the concatenation of the segments:

$$\begin{aligned} &\mathit{allgather} : [[\alpha]_l]_p \to [[\alpha]_{l*p}]_p \\ &\mathit{allgather}\,[[x_1, x_2], [x_3, x_4]] = [[x_1, x_2, x_3, x_4], [x_1, x_2, x_3, x_4]] \end{aligned}$$

Function *allgather* can be captured in our DH-framework as an unsegmented reduction with ++ as basic operator, i.e., a CDH:

$$\mathit{allgather} = (+\!\!+) \updownarrow (+\!\!+) \tag{27}$$

Thus, we can use for *allgather* the generic implementation schema (19), where the segment length communicated between processors doubles at each of $\log p$ steps, from $l$ at the beginning to $l \cdot p$ at the end.

Thereby, we have obtained a *decomposition rule* which splits segmented reduction over a commutative operator $\odot$, into a sequence of two DHs: the IDH *redd-parts* followed by the CDH *allgather*:

$$\mathit{redd\text{-}seg}\,(\odot) = \mathit{allgather} \circ \mathit{redd\text{-}parts}\,(\odot) \tag{28}$$

Recollecting how a DH can be implemented according to Theorem 3, we obtain the following hypercube program for the result of the decomposition:

$$\begin{aligned} \mathit{redd\text{-}seg}\,(\odot) = &\bigcirc_{d=1}^{k} (\mathit{swap}\ d\ (+\!\!+, +\!\!+)) \ \circ \\ &\bigcirc_{d=k}^{1} (\mathit{swap}\ d\ (\mathit{left} \circ \mathit{zip}(\odot)\,,\ \mathit{right} \circ \mathit{zip}(\odot))) \end{aligned} \tag{29}$$

This program consists of two stages, composed sequentially. Figure 2 illustrates program (29) on the hypercube of four processors, each of which keeps a segment of length 4. Each of the two stages, the first depicted on the top, the second on the bottom, consists of two swaps, with decreasing and increasing order of the dimensions, respectively.

Another possible decomposition, which does not require the commutativity of $\odot$, is to define *redd-parts* as CDH, which yields an interleaved output, and then to use `MPI_Allgather` with a suitable re-enumeration of the processors.

## 6.2. Performance of the Decomposition

Let us analyze the improvement gained by applying decomposition rule (28).

Observe that the length of communicated segments first decreases in *redd-parts* from $m/2$ to $m/2^p$ in $\log p$ steps, giving a time of

$$\sum_{i=1}^{\log p} (t_s + m \cdot (t_w + 1)/2^i)$$

After the first stage, each of the $p$ processors keeps a segment of length $m/p$; these segments are gathered at the second stage to the result segment of length $m$ in each processor, which requires a time of

$$\sum_{i=1}^{\log p} (t_s + m \cdot t_w/2^i)$$

The total time complexity is:

$$2 \cdot t_s \cdot \log p + m \cdot (2 \cdot t_w + 1)(p - 1)/p$$

For comparison, the time of the reduction from Subsection 5.2 is:

$$\log p \cdot (t_s + m \cdot (t_w + 1))$$

Thus, performance is improved if

$$t_s \cdot \log p \; < \; m \cdot ((t_w + 1) \cdot \log p - (2 \cdot t_w + 1) \cdot (p-1)/p)$$

For a large number of processors, the condition becomes:

$$t_s \; < \; m \cdot t_w$$

Rough estimates of the communication parameters for, say, the hypercubic Intel 860*i* are $t_s = 10^3$ and $t_w = 10$. Thus, for a sufficiently large configuration of the machine, composition scan-reduce always pays off, scan-scan pays off if $m < 500$, and the reduction decomposition pays off if $m > 100$.

### 6.3. Composition and Subsequent Decomposition

Both composition and decomposition can be applied together. For example, after applying composition scan-reduce, we can decompose the resulting reduction. The overall transformation is not a pure composition or decomposition anymore; after an additional small transformation which promotes $\pi_1$ through *allgather*, it reads:

$$\begin{aligned} &\textit{redd-seg}(\oplus) \;\circ\; \textit{scan-seg}(\otimes) \;\; = \qquad (30) \\ &\textit{allgather} \;\circ\; \textit{map}^2\, \pi_1 \;\circ\; \textit{redd-parts}(\langle\oplus,\otimes\rangle) \;\circ\; \textit{map}^2\, \textit{pair} \end{aligned}$$

This rule gets us from the time achieved after applying rule (11) in Subsection 5.2:

$$\log p \cdot (t_s + m \cdot (2 \cdot t_w + 3))$$

to a new time of

$$2 \cdot t_s \cdot \log p + m \cdot (4 \cdot t_w + 3) \cdot (p-1)/p$$

which is an improvement for large $p$ if

$$t_s < 2 \cdot m \cdot t_w$$

Thus, for small vectors and/or high start-up times, rule (11) yields better target performance than the more complex transformation (30).

## 7. Related Work

The skeleton approach has recently received much attention. An overview of some current work can be found in [5].

Transformation work related to ours has been carried out in BMF: a version of Theorem 1 in the sequential setting was proved and used in [2, 18]. An analog of our operator *red* $(\langle\oplus,\otimes\rangle)$, called *recur-reduce*, has been used to parallelize linear recurrences [3] and later to tabulate parallel implementations of linearly recursive programs [19]. Our transformation rule (11) differs in that it aims directly at the scan-reduce composition, and is arguably more convenient for use in algorithm design and MPI programming in practice.

To the best of our knowledge, the composition rule for two scans in Theorem 2 is new; the target expression is similar to *recur-prefix* [3].

## 8. Conclusions

(De)composition of skeletons in general and of homomorphisms in particular can be employed in two main areas.

First, (de)composing transformations can be used in the process of program derivation. E.g., for the maximum segment sum problem, application of the scan-reduce composition rule enables a series of further transformations, which eventually leads to an optimal parallel algorithm [9].

The second area is programming with parallel skeletons. One challenge of the skeleton approach to parallel programming is that compositionality is not straightforward: at least, it cannot be guaranteed that the sequential composition of the best parallel implementations of two skeletons yields again optimal parallelism. Instead, one must face the possibility that an individual implementation effort of composed skeletons is called for. Furthermore, this effort may have to be reinvested when one moves to another machine model.

What consequences does this have for the user of the skeletons and the compiler? It would be very helpful if the compiler had enough knowledge about which compositions can be implemented efficiently on which machine models. For instance, MPI implementations should not only provide efficient collective operations but also try and accomplish optimizing (de)compositions for the underlying platforms. This requirement actually does not go much farther than, say, loop fusion implemented in the modern sequential compilers. If portability between machine models is not provided by the compiler then the application programmer should at least be aware of the model he/she is programming for; switching models may mean reprogramming.

## 9. Acknowledgements

Thanks to Christoph Wedler and Christoph A. Herrmann for discussing the manuscript, and to Holger

Bischof for experiments. This work is part of the DFG-funded project *RecuR2*, and is supported by the DAAD exchange programmes ARC and PROCOPE.

## References

[1] R. 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.

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

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

[4] M. Cole. *Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computation.* Pitman, 1989.

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

[6] A. Geser and S. Gorlatch. Parallelizing functional programs by generalization. In M. Hanus, J. Heering, and K. Meinke, editors, *Algebraic and Logic Programming (ALP'97)*, Lecture Notes in Computer Science 1298, pages 46–60. Springer-Verlag, 1997.

[7] S. Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In L. Bougé et al., editors, *Parallel Processing. Euro-Par'96*, Lecture Notes in Computer Science 1124, pages 401–408. Springer-Verlag, 1996.

[8] S. Gorlatch. Systematic extraction and implementation of divide-and-conquer parallelism. In H. Kuchen and D. Swierstra, editors, *Programming Languages: Implementation, Logics and Programs (PLILP'96)*, Lecture Notes in Computer Science 1140, pages 274–288. Springer-Verlag, 1996.

[9] S. Gorlatch. Optimizing compositions of scans and reductions in parallel program derivation. Technical Report MIP-9711, Universität Passau, May 1997. Available at http://www.fmi.uni-passau.de/cl/papers/Gor97b.html.

[10] S. Gorlatch and 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. Available at http://www.fmi.uni-passau.de/cl/papers/GorBi97.html.

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

[12] J. Hill and D. Skillicorn. The BSP Tutorial at Euro-Par'97. University of Passau, 1997.

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

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

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

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

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

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

[19] 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. IEEE Computer Society Press, 1997.