

On linear list recursion in parallel^{*}

Christoph Wedler, Christian Lengauer

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

Received: 27 May 1997 / Revised version: 17 March 1998

Abstract. We propose a classification for a set of linearly recursive functions, which can be expressed as instances of a skeleton for parallel linear recursion, and present new parallel implementations for them. This set includes well known higher-order functions, like Broadcast, Reduction and Scan, which we call *basic components*. Many compositions of these basic components are also linearly recursive functions; we present transformation rules from compositions of up to three basic components to instances of our skeleton. The advantage of this approach is that these instances have better parallel implementations than the compositions of the individual implementations of the corresponding basic components.

1 Introduction

Functional programming offers a very high-level approach to specifying executable problem solutions. For example, the scheme of linear recursion can be expressed concisely as a higher-order function. In the data-parallel world, higher-order functions are used which represent classes of parallel algorithms on data structures; these higher-order functions are also called *skeletons* [4, 6]. Well known representatives, which also happen to be linearly recursive, are reduction and scan (parallel prefix) with an associative operator. If one can express one's problem in terms of instances of skeletons with a known parallel implementation, the implementation of the whole problem comes for free (although it is not guaranteed to be optimal).

^{*} Part of this material has been presented at the Second International Workshop on Software Engineering for Parallel and Distributed Systems (PDSE'97) [21].

Current research focuses on the following questions:

1. How can a problem solution be expressed in terms of skeletons? The Bird-Meertens formalism [1] is a well known framework in which a problem, expressed as a functional term, can be *transformed* to a semantically equivalent term. The aim is to obtain, via transformations, a term which uses skeletons whose implementations have a lower cost [2, 8].
2. What are the useful skeletons and their parallel implementations? Often, one obtains a skeleton by means of a *generalization*. E.g., reduction and scan with an associative operator are included in the class of homomorphic functions which match the divide-and-conquer paradigm and, therefore, have a natural parallel implementation [5, 9, 17].
3. Can we optimize combinations, esp. functional composition, of skeleton instances? In other words, is the cost of the parallel implementation of the functional composition of some skeleton instances simply the addition of their individual costs [18], or are there *patterns* of skeleton combinations which have parallel implementations with a lower cost [7]? Two transformation rules with scan and reduction and with two scans can be found in [10].

In this paper, we address (2) and (3) for a set of linearly recursive functions. We generalize some existing skeletons, e.g., we present several new parallel implementations of reduction with a non-associative operator. Additionally, we identify patterns of functional compositions of linearly recursive functions with a better parallel implementation than the naïve composition of the individual implementation.

Quite naturally, points (1) to (3) are not independent from each other: identifying patterns with a better parallel implementation than the naïve composition can be viewed as presenting transformation rules to obtain from these patterns an instance of a skeleton which is specially tailored for this pattern; this may involve defining new skeletons with good parallel implementations (2).

We base the analysis and implementation of linearly recursive functions on one common skeleton (Sect. 3.2) which specifies the call graph of linear recursion. Specializations of this graph lead to different classes of functions. Functions in some classes are divided further into subclasses. Functions in a fixed subclass have common parallel implementations. Some of the simplest subclasses (e.g., reduction) and their parallel implementations are well known; we call them *basic components*. Some subclasses are specially tailored to match simple patterns of functional compositions of basic components; these subclasses require new parallel implementations, which we present.

2 Notation

The notation we use is close to Haskell [19]. Each term t has a *type* T ; we write: $t :: T$. We use also n -tuples $(a_1, \dots, a_n) :: (\alpha_1, \dots, \alpha_n)$. The singleton (a) is the element a itself, the empty tuple $()$ is the only element of the *unit type* $()$. The set of integers from b to t is written as $b..t$.

A list with elements $a_i :: \alpha$, for all $i \in 0..n-1$, is written as $[a_0, \dots, a_{n-1}] :: [\alpha]$. The empty list is written as $[]$. A list whose head is a and whose tail is as is written as $a : as$. The concatenation of two lists as and bs is denoted by $as \# bs$.

A function f with argument type α and result type β , in short $f :: \alpha \rightarrow \beta$, is applied to an argument a via $f a$. Function application has highest precedence, e.g., $f a + g b$ is the same as $(f a) + (g b)$. If the term for the function type has free type variables α_i , these variables are implicitly universally quantified and the function is said to be polymorphic.

Predefined polymorphic functions are the identity function $\text{Id} :: \alpha \rightarrow \alpha$, with $\text{Id } a = a$, and the projection functions $\pi_k :: (\alpha_1, \dots, \alpha_k, \dots, \alpha_n) \rightarrow \alpha_k$, with $\pi_k (a_1, \dots, a_k, \dots, a_n) = a_k$, where $1 \leq k \leq n$ and $1 < n$.¹

If we want to apply a function to two (or more) arguments a_1 and a_2 , it can be defined in two ways. If it is defined to be *uncurried*, we apply function $f :: (\alpha_1, \alpha_2) \rightarrow \beta$ to a pair: $f (a_1, a_2)$. If it is defined to be *curried*, we supply the arguments piecewise to function $f :: \alpha_1 \rightarrow (\alpha_2 \rightarrow \beta)$: written as $(f a_1) a_2$. Since \rightarrow is right-associative and function application is left-associative, this can be written as $f :: \alpha_1 \rightarrow a_2 \rightarrow \beta$ and $f a_1 a_2$.

If function f is defined to be curried, it may be applied to only a part of the arguments; e.g., with $f :: \alpha_1 \rightarrow \alpha_2 \rightarrow \beta$, we may define a function $g :: \alpha_2 \rightarrow \beta$ with $g = f a_1$. In the rest of the paper, functions are normally defined to be curried.

Each infix operation $_ \oplus _$ induces the following functions, the later two are called *sections*:

$$\begin{aligned} (\oplus) &:: \alpha \rightarrow \beta \rightarrow \gamma && \text{where } (\oplus) a b = a \oplus b \\ (a \oplus) &:: \beta \rightarrow \gamma && \text{where } (a \oplus) b = a \oplus b \\ (\oplus b) &:: \alpha \rightarrow \gamma && \text{where } (\oplus b) a = a \oplus b \end{aligned}$$

Functions can be combined via the following functionals:

Functional composition: $(g \circ f) x = g (f x)$, where \circ is associative.

Functional product: $(f_1 \times f_2) (x_1, x_2) = (f_1 x_1, f_2 x_2)$, and similarly for three or more functions.

¹ The projection function is not defined on singleton tuples for the following reason: with $a = (a_1, a_2)$, we would have $\pi_1 (a) = a = (a_1, a_2)$ and also $\pi_1 (a) = \pi_1 a = \pi_1 (a_1, a_2) = a_1$, which is a contradiction. An alternative would be to make the parameter n explicit, i.e., use function symbols π_k^n .

FP's construction: $\langle f_1, f_2 \rangle x = (f_1 x, f_2 x)$, and similarly for three or more functions; $\langle \rangle x = ()$.

3 Skeletons for linear recursion

In this section, we introduce two higher-order functions (skeletons) which express the scheme of linear recursion. Skeleton *LR*, introduced in Sect. 3.1, reflects the general form of linear recursion, but does not reveal any possibilities for its parallelization. Thus, we introduce a second skeleton, skeleton *PLR* for parallel linear recursion in Sect. 3.2. In Sect. 3.3, we relate instances of the two skeletons with each other.

3.1 General form of linear recursion

If a linear recursive function f is applied to an argument x , there are two cases:

- In the *base case*, we apply function *base* to the argument.
- In the *recursive case*, we apply function *pre* to the argument and use the result as the argument for the function f . Function *post* combines the result of this recursive call with the original argument x .

If applying function P to argument x returns true, we have reached the base case, otherwise the recursive case.

The concept of linear recursion can be expressed by the following higher-order function:

$$LR\ P\ base\ pre\ post = f \quad \text{where} \\ f\ x = \text{if } P\ x \text{ then } base\ x \text{ else } post\ (x, f\ (pre\ x))$$

Example: a reduction from the left (*foldl*) over a list $as = [a_0, \dots, a_{n-1}]$, informally defined by:

$$foldl\ (\oplus)\ (as, b) = ((b \oplus a_0) \oplus \dots) \oplus a_{n-1}$$

is formally defined by

$$foldl\ (\oplus)\ ([], b) = b \\ foldl\ (\oplus)\ (a : as, b) = foldl\ (\oplus)\ (as, b \oplus a)$$

This reduction can be defined as an instance of *LR*:

$$foldl\ (\oplus) = LR\ ((= []) \circ \pi_1)\ \pi_2\ pre\ \pi_2 \\ \text{where } pre\ (a : as, b) = (as, b \oplus a)$$

3.2 The skeleton for parallel linear recursion

A key factor in executing a program in parallel is the distribution of data across the processors. In the data-parallel model, the same function is applied by each processor to data sets of a common type. We take this type to be the list.

In this section, we propose a skeleton for parallel linear recursion which makes the use of lists as the input and output explicit: with each recursion step, one element of the input list is processed and one element for the output list produced. Note that it is also allowed to pass a fixed-size argument between the levels of the recursion. As we shall see later, this enables us to derive different classes of possibly parallel implementations, depending on the type of the list elements. See Sect. 3.3 for a comparison between our skeleton for parallel linear recursion and the general form of linear recursion.

The skeleton *PLR* of parallel linear recursion is instantiated by providing a function *base* which is executed in the base case, a function *pre* which computes the input value for the next recursion level and an intermediate local value, and a function *post* which combines this local value with the result of the next recursion level. The base case is reached when there is no distributed data to process, i.e., when the list argument is empty.

The skeleton is defined as follows:

$$\begin{aligned}
 PLR\ base\ pre\ post &= f \quad \text{where} \\
 f(\ [], b) &= base(\ [], b) \\
 f(a : as, b) &= (y, z : zs) \\
 (m, b') &= pre(a, b) \\
 (y', zs) &= f(as, b') \\
 (y, z) &= post(y', m)
 \end{aligned}$$

The data flow graph of this skeleton (Fig. 1) exposes the symmetry between the input and output and the flow of data across and inside the levels of the recursion. We call the horizontal data flow, which stays at a fixed recursion level, *local* (a, m and z in Fig. 1) and the vertical data flow, which connects recursion levels, *carried*² (b and y in Fig. 1). In order to predict the costs of the implementations of this skeleton, we assume that the size of the carried data is level-independent.

For example, the reduction *foldl*, defined in Sect. 3.1, can be defined as an instance of *PLR*:

$$\begin{aligned}
 foldl(\oplus) &= \pi_1 \circ (PLR\ \langle \pi_2, \pi_1 \rangle\ pre_1\ Id) \\
 \text{where } pre_1(a, b) &= ((), b \oplus a)
 \end{aligned}$$

² This data flow was called *global* in [21].

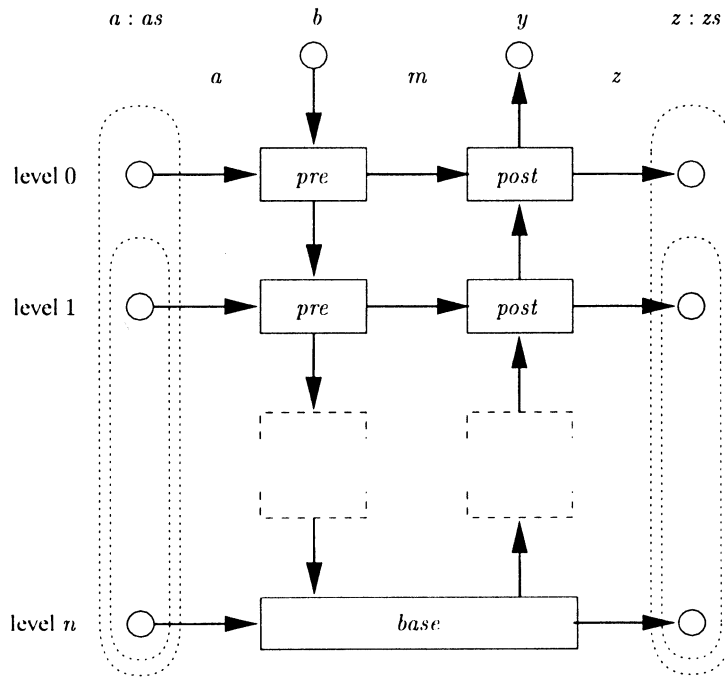


Fig. 1. Data flow graph of the skeleton for parallel linear recursion

All of the computation is done by the recursive application of function *pre*, functions *base* and *post* are essentially the identity. We have only distributed data on the input side³ (*as*), no intermediate local data ($m = ()$), and no distributed data on the output side ($zs = [(), \dots, ()]$).

Note that linear recursion without distributed input is included in this skeleton if the recursion depth is given as an argument. The base case is reached when this argument is 0. This is due to the fact that the natural numbers (including 0) are isomorphic to the lists with elements of unit type.

3.3 Equivalences

In this section, we consider the equivalence between an instance of the skeleton for the general form of linear recursion and an instance of the skeleton for parallel linear recursion.

For equivalence, there must be an isomorphism ϕ_{IN} and a homomorphism ϕ_{OUT} which make the following diagram commute ($f_1 \circ \phi_{\text{IN}} =$

³ Without distributing the list argument, the best time complexity we can obtain is linear, except in trivial cases.

$\phi_{\text{OUT}} \circ f_2$ or $f_1 = \phi_{\text{OUT}} \circ f_2 \circ \phi_{\text{IN}}^{-1}$:

$$\begin{array}{ccc}
 In_1 & \xrightarrow{f_1} & Out_1 \\
 \phi_{\text{IN}} \uparrow & & \uparrow \phi_{\text{OUT}} \\
 In_2 & \xrightarrow{f_2} & Out_2
 \end{array}
 \quad
 \begin{array}{l}
 f_1 = LR\ P_1\ base_1\ pre_1\ post_1 \\
 f_2 = PLR\ base_2\ pre_2\ post_2
 \end{array}$$

Here are sufficient conditions on the relationship between the customizing functions passed to the source skeleton and the functions passed to the derived skeleton, which make the diagram commute:

$$\begin{aligned}
 P_1 \circ \phi_{\text{IN}} &= (= []) \circ \pi_1 & (\text{EQPRED}) \\
 base_1 \circ \phi_{\text{IN}} &= \phi_{\text{OUT}} \circ base_2 & (\text{EQBASE}) \\
 (pre_1 \circ \phi_{\text{IN}})(a : as, b) &= \phi_{\text{IN}}(as, (\pi_2 \circ pre_2)(a, b)) & (\text{EQPRE}) \\
 post_1(\phi_{\text{IN}}(a : as, b), &= (\phi_{\text{OUT}} \circ (\text{Id} \times (: zs))) \circ post_2) & (\text{EQPOST}) \\
 \phi_{\text{OUT}}(y', zs)) &= (y', (\pi_1 \circ pre_2)(a, b))
 \end{aligned}$$

For our example *foldl*, we have $In_1 = In_2 = ([\alpha], \beta)$, $Out_1 = \beta$, $Out_2 = (\beta, [()])$. The homomorphisms are defined by $\phi_{\text{IN}} = \text{Id}$ and $\phi_{\text{OUT}} = \pi_1$. Note that ϕ_{OUT} is no isomorphism, i.e., f_2 also returns the length of the input list (via a list $[(), \dots, ()]$).

$$\begin{aligned}
 \text{EQPRED:} \quad & (= []) \circ \pi_1 \circ \text{Id} = (= []) \circ \pi_1 \\
 \text{EQBASE:} \quad & \pi_2 \circ \text{Id} = \pi_1 \circ \langle \pi_2, \pi_1 \rangle \\
 \text{EQPRE:} \quad & (pre_1 \circ \text{Id})(a : as, b) = (as, b \oplus a) \\
 & = \text{Id}(as, \pi_2((), b \oplus a)) \\
 \text{EQPOST:} \quad & \pi_2(\text{Id}(a : as, b), \pi_1(y', zs)) = y' \\
 & = (\pi_1 \circ (\text{Id} \times (: zs))) \circ \text{Id}(y', \pi_1((), b \oplus a))
 \end{aligned}$$

The main restriction in skeleton *PLR* is that the recursion depth is determined by the length of the input list. This is expressed by *EQPRED*.

4 Classes of linearly recursive functions

Different linearly recursive functions have (parallel) implementations of different quality. Our aim is to find classes of functions which have good (parallel) implementations in common. In this section, we characterize these classes and discuss how a function is assigned to its appropriate class. From now on, we shall use solely skeleton *PLR*.

A linearly recursive function, which is specified as an instance of the skeleton, consists of two parts: the *pre* part and the *post* part (Fig. 1). The only difference between the *pre* part and the *post* part is that the carried

data flows downwards in the **pre** part and upwards in the **post** part (Fig. 1). We consider only the **pre** part; our classification applies just the same for the **post** part.

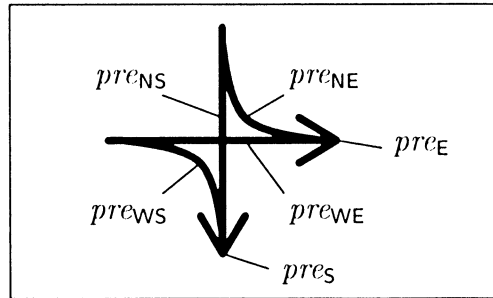
4.1 Classifying the **pre** part

We classify the **pre** part by examining the properties of function *pre* in isolation.

Data flow: *pre* has two inputs and two outputs. A first classification results from checking whether there is a data dependence from a specific input to a specific output. There are $2^{2 \times 2} = 16$ combinations. This classification can probably be performed automatically.

Algebraic properties: We classify the derived programs in some classes further by looking at algebraic properties of the functions involved. These properties are listed for each class separately. The most important property is associativity. In general, this classification cannot be done automatically.

The following diagram depicts the choices of data flow for the **pre** part:



A textual representation of this diagram is given by a formula for *pre*:

$$pre(a, b) = (pre_E(pre_{WE} a, pre_{NE} b), pre_S(pre_{WS} a, pre_{NS} b))$$

Any function *pre* with the given signature could be decomposed this way by choosing *Id* for *pre_{WE}*, *pre_{NE}*, *pre_{WS}* and *pre_{NS}*. Instead, we require these functions to be

- either an identity function $Id_{\neq ()}$ on some non-unit type $\alpha \neq ()$ or
- a function \diamond from some type to the unit type $()$.

We could always choose $Id_{\neq ()}$ if the type of the input is not the unit type, but we prefer to choose \diamond whenever possible, such that function $Id_{\neq ()}$ indicates the presence and function \diamond the absence of a data dependence.

As an example, let us consider $pre(a, b) = (b, a)$ with $a :: \alpha$ and $b :: \beta$. In this case, $pre_E = \pi_2$, $pre_S = \pi_1$. Instead of choosing *Id* for

Table 1. Classes of the *pre* part

				→		↘			↘	
	—		Map		—Mismatch—					
↓	Sequential		Sequential Map		Reduction		Reduction Map			
↘	—Mismatch—				Shift		Shift Map			
↘	Broadcast		Broadcast Map		Scan		Scan Map			

pre_{WE} , pre_{NE} , pre_{WS} and pre_{NS} and having $pre_E :: (\alpha, \beta) \rightarrow \beta$ and $pre_S :: (\alpha, \beta) \rightarrow \alpha$, we choose $pre_{WE} = \diamond$, $pre_{NE} = Id_\beta$, $pre_{WS} = Id_\alpha$ and $pre_{NS} = \diamond$ and have $pre_E :: ((\), \beta) \rightarrow \beta$ and $pre_S :: (\alpha, (\)) \rightarrow \alpha$.

Two functions in our formula for *pre* remain to be explained:

- Function pre_E is of no interest for the classification, since its output is not used by any input of the *pre* part. Thus, the computation inside the *pre* part which is induced by this function is just a simple $map\ pre_E$, applied to the distributed output list; this can be parallelized trivially with a constant execution time and a cost linear to the length of the input list (or no time and cost at all if $pre_E = Id$).
- Function pre_S is the function which determines the subclassification. Therefore, we examine this function for each class individually in Sect. 4.3.

4.2 Classes of the *pre* part

Table 1 classifies the *pre* part according to the alternative data dependence patterns. The horizontal legend lists the possible data dependences of the local input, the vertical legend those of the carried input (by icon). Each table entry corresponds to one class, the one whose data dependence is determined by superposition of the icons of both legends. From now on, we work with these superpositions.

The only difference between the first and the second column and, similarly, between the third and the fourth column, is the additional data dependence from the local input to the local output in the second column, expressed as an additional independent “computation” of $map\ Id$.

Let us now comment on the individual entries of Table 1.

4.2.1 Mismatch, Map and Shift. First we look at the entries representing Mismatch, Map and Shift. Their data flow between two successive levels is depicted again in Fig. 2.

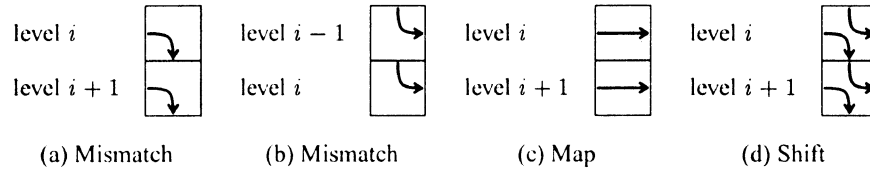
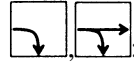
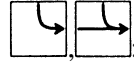



Fig. 2. Specific data flows (Mismatch, Map and Shift)


Mismatch. We start with an informal explanation of the two cases of mismatch of the carried data flow:

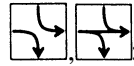
: This case covers the upper right of Table 1; see Fig. 2(a). There is a carried output of *pre* at level i but no according input at level $i + 1$. Since the carried output is only used by *pre* of the next level (and the base) case and this input is not used, there is no “real” data dependence to the carried output which contradicts the data dependences for this class.

: This case covers the mismatch on the left in the table; see Fig. 2(b). There is a carried input of *pre* at level i but no according carried output at level $i - 1$.

Map and Shift. Next we discuss the classes which we do not consider any further:

: This case covers the upper left of the Table 1. Functions of this class return the result $([(), \dots, ()], ())$, which conveys no information.

: This is the second entry of the first row; see also Fig. 2(c). These functions are simply *mapId*, i.e., there is nothing to do (or just the *mappre_E*, mentioned above).

: The functions of this class perform a shift; see Fig. 2(d). Data flows from the local input at level i to the local output at level $i + 1$. These functions have trivial parallel implementations with a constant execution time.

4.2.2 Sequential, Broadcast, Reduce and Scan. The remaining classes are the most interesting ones. Their data flow between two successive levels is depicted again in Fig. 3.

A naïve implementation of a function in these classes is the sequential one with a time complexity of $\mathcal{O}(n)$. In Sect. 5, we present good parallel im-

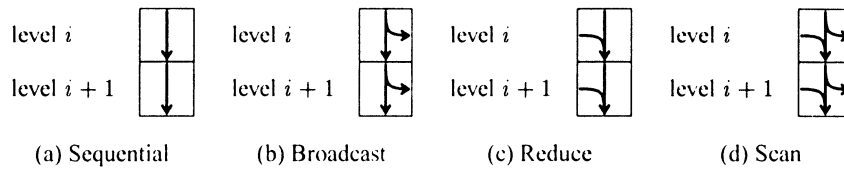


Fig. 3. Specific data flows (Sequential, Broadcast, Reduce and Scan)

plementations for subclasses of these classes; the subclasses are determined by algebraic properties of the function pre_S .

4.3 Subclasses of the pre part

All functions in classes Map and Shift have trivial parallel implementations, so there is no need for a subclassification. Sequential, Broadcast, Reduction and Scan are the remaining classes to be considered. As mentioned in the introduction, parallel implementations of simple representatives in these classes are well known. They are characterized by algebraic properties of the functions given as an argument to the skeleton (pre in our skeleton PLR , or more specifically, pre_S , see Sect. 4.1). The other subclasses are tailored to match simple combination patterns of basic components.

In Sect. 4.3.1 we list these *basic components*, in Sect. 4.3.2 we present an example of a combination pattern and show how this pattern leads to a new subclass, in Sect. 4.3.3 we list the subclasses of Reduce and Scan, in Sect. 4.3.4 we list the subclasses of Sequential and Broadcast. As mentioned in Sect. 4.2, function pre_S determines the subclassification. All classes have an additional subclass which we do not discuss any further: functions in these subclasses are instantiated by function pre_S with none of the listed properties. For these functions we simply take the naïve (sequential) implementation.

We have identified—and are describing here—optimized subclasses for composition patterns of up to three basic components in the classes Sequential and Broadcast and up to two components in the classes Reduce and Scan.

4.3.1 Basic Components. The simple representative of class Sequential is the identity function, which we will not discuss. The simplest representatives of the three remaining interesting classes are called *basic components* and are defined by the following equations, where \oplus is associative:

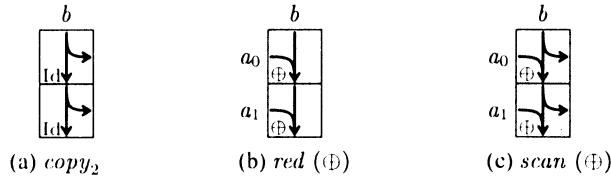


Fig. 4. Basic components

$$\begin{aligned}
 copy_n b &= (b, \overbrace{[b, \dots, b]}^n) \\
 red (\oplus) (as, b) &= b \oplus a_0 \oplus \dots \oplus a_{n-1} \\
 scan (\oplus) (as, b) &= (red (\oplus) (as, b), [b, b \oplus a_0, \dots, b \oplus a_0 \oplus \dots \oplus a_{n-2}])
 \end{aligned}$$

Figure 4 lists the three basic components in our box notation. A box which is adorned with a function symbol or name represents a special instance of *pre*, which uses this function symbol or name.

In order to get the input values of the basic components in the composition to the right places, we introduce a (generic) function which modifies the nesting structure of its argument. We denote the regrouping operator by \Downarrow . Above the operator we write the input structure, below the modified structure. Rather than providing a formal definition, we illustrate the usage of \Downarrow by two examples:

$$\begin{aligned}
 &\begin{matrix} (2,1) \\ \Downarrow \\ (1,2) \end{matrix} ((a, b), c) = (a, (b, c)) \quad \text{and} \\
 &\begin{matrix} (2,3) \\ \Downarrow \\ (1,2,2) \end{matrix} ((a, b), (c, d, e)) = (a, (b, c), (d, e))
 \end{aligned}$$

4.3.2 Example of a combination pattern: scan and red. In this section we look at a typical example of a functional composition of skeleton instances: applying *scan* with an associative operator \oplus to a list, then applying *red* with the same operator to the result:

$$(\text{Id} \times red (\oplus)) \circ \begin{matrix} (2,1) \\ \Downarrow \\ (1,2) \end{matrix} \circ (scan (\oplus) \times \text{Id}) \circ \begin{matrix} (1,2) \\ \Downarrow \\ (2,1) \end{matrix}$$

It turns out that this composition can be expressed in terms of a single reduction and an efficient parallel implementation can be found if \oplus is commutative; see Sect. 5.

It is quite clear that, on most parallel machines, a single reduction is much faster than a scan followed by a reduction—even if the function which is applied at each node of the reduction tree is slightly more complicated in

Table 2. Subclasses of Reduction and Scan

Function g	Conditions	Reduction	Scan
$g(a, b)$ $= b \oplus a$	\oplus is associative	Undirected Red. $red(\oplus)$	Simple Scan $scan(\oplus)$
$g(a, (b_1, b_2))$ $= (b_1 \oplus a, b_2 \oplus b_1)$	\oplus is associative and commutative	SR-Reduction $foldl\ g$	SS-Scan $scanl\ g$
$g(a, (b_1, b_2))$ $= (b_1 \otimes a, b_2 \otimes b_1)$	\oplus and \otimes are assoc. \otimes distributes over \oplus	SR2-Reduction $foldl\ g$	SS2-Scan $scanl\ g$

former case than in the latter. The execution time and the cost of both the specialized and the naïve implementation belong to the same complexity class.

Let us now see, with the help of our box notation, which kind of reduction we end up with: combining *scan* and *red* in the way described above is depicted by two columns of boxes—the left column represents *scan*, the right *red*. In contrast, the functional description above uses backward composition (operator \circ). If we combine the two boxes in each row, we get a data dependence from the left and top to the bottom. Our classification table reveals that this data dependence is characteristic for a reduction:

$$scan(\oplus) \mid red(\oplus) = \begin{array}{|c|c|} \hline \downarrow \oplus & \downarrow \oplus \\ \hline \downarrow \oplus & \downarrow \oplus \\ \hline \downarrow \oplus & \downarrow \oplus \\ \hline \end{array} = \begin{array}{|c|} \hline \downarrow g \\ \hline \downarrow g \\ \hline \end{array} = reduce\ g$$

Now, we have to find the operator g which is subject to the special reduction:

$$a \begin{array}{|c|} \hline b \\ \hline \downarrow g \\ \hline \end{array} = a \begin{array}{|c|c|} \hline b_1 & b_2 \\ \hline \downarrow \oplus & \downarrow \oplus \\ \hline \end{array} \quad \text{where} \quad \begin{array}{l} b = (b_1, b_2) \\ g(a, (b_1, b_2)) = (b_1 \oplus a, b_2 \oplus b_1) \end{array}$$

Note that this function g is not associative—thus, *red* with its parallel implementation cannot be used! For this pattern, we have to introduce a special subclass of Reduction: SR-Reduction; see Table 2. All names of the subclasses presented in the next two sections reflect the patterns which they represent: ‘C’ stands for Copy, ‘R’ for Reduction and ‘S’ for Scan.

4.3.3 Subclasses of Reduction and Scan. Functions in the classes Reduction and Scan have a local and a carried input. Thus, the function g which determines the subclassification is exactly the binary function $pres$:

$$g(a, b) = pres(a, b)$$

Table 2 lists different forms of function g with their corresponding subclasses of Reduction and Scan.

Table 3. Subclasses of Sequential and Broadcast

Function g	Conditions	Sequential	Broadcast
$g\ b = b$		Identity Id	Copy $copy_n$
$g\ (b_0, b_1)$ $= (b_0, b_1 \oplus b_0)$	\oplus is associative	CR-Accum. g^n	CS-Broadcast $broadcast_n\ g$
$g\ (b_0, b_1, b_2)$ $= (b_0, b_1 \oplus b_0, b_2 \oplus b_1)$	\oplus is associative and commutative	CSR-Accum. g^n	CSS-Broadc. $broadcast_n\ g$
$g\ (b_0, b_1, b_2)$ $= (b_0, b_1 \otimes b_0, b_2 \oplus b_1)$	\oplus and \otimes are assoc. \otimes distributes over \oplus	CSR2-Accum. g^n	CSS2-Broadc. $broadcast_n\ g$

4.3.4 Subclasses of Sequential and Broadcast. Neither class Sequential nor class Broadcast has a local input. Thus, the function which determines the subclassification is:

$$g\ b = pres\ ((), b)$$

Despite the fact that this function is not binary, associativity again plays a key role in the subclassification of the classes Sequential and Broadcast. Table 3 lists different forms of function g with their corresponding subclasses of Sequential and Broadcast.

5 Implementations

We specify a parallel implementation by describing its processor network, the flow of data through the network and the operation (function) which is executed on each processor. Here we describe the primitives for these networks informally. Appendix A contains their Haskell definitions with their time and cost complexities.

The time and cost complexities we state are based on the following assumptions:

- The size of the carried and local data is constant.
- The execution time of \oplus and \otimes is constant.
- Sending a datum of constant size from one processor to any other processor takes constant time and is independent from communications between *other* processors (this applies for EREW-PRAM machines). The communication patterns in the implementation primitives of Appendix A are restricted such that this assumption also applies for, e.g., a hypercube.

In our implementations, one operation is performed on each processor per computation. This fact gives us a choice of two options:

1. we can either stick to the implementation we propose,
2. or we can aggregate several operations on one processor, following Brent's Theorem [16], without an asymptotic penalty in time but with a reduced cost.

We analyze the optimal asymptotic complexity with respect to each of the following objective functions:

$time(n)$: execution time (choice 1 or 2).

$cost_{Brent}(n)$: product of time and number of processors (choice 2). Brent's Theorem implies that $p(n) = \#operations(n)/time(n)$ processors can execute the same algorithm in time $\mathcal{O}(time(n))$. Brent's cost is therefore $\mathcal{O}(time(n) * p(n)) = \mathcal{O}(\#operations(n))$.

$pipe(n)$: lag time between the outputs of two successive computations. For choice 1 this time is of $\mathcal{O}(1)$; then we say that the implementation allows pipelining.

In the following examples, our illustrations always depict choice 1 ($pipe(n) = 1$), but we state the cost of choice 2.

In the rest of this section, we present only new parallel implementations. For each subclass, we state also the combination patterns which can be transformed into an instance of the current subclass.

5.1 Reduction

For all functions in the subclasses of Reduction which are listed in Table 2, we have $time(n) = \mathcal{O}(\log n)$, $cost_{Brent}(n) = \mathcal{O}(n)$ and $pipe(n) = \mathcal{O}(1)$. This is done by using a tree-like processor network and applying some function *node* (different ones for different subclasses) on each node which receives the input from its children on the left and provides the output to its parent on the right. The distributed input (*as* with a function applied to each element) consists of the values at the leaves, the output is the value at the root.

The trees used in this subsection can mapped easily onto a hypercube without a time penalty. Thus, our given complexities also apply to hypercube architectures.

5.1.1 Undirected Reduction. Functions in this subclass are well known (and often simply called “reduction”). They are given by:

$$foldl\ g\ (as, b) \quad \text{where} \quad g(a, b) = (b \oplus a), \quad \oplus \text{ is associative}$$

Functions in this subclass are instances of the basic component “R”. A parallel implementation is well known and given by the higher-order

function $reduceR$, which is instantiated with the associative operator \oplus . The computation traverses a left-right tree ($lrtreeUp$) with the values of its leaves given by the distributed input as :

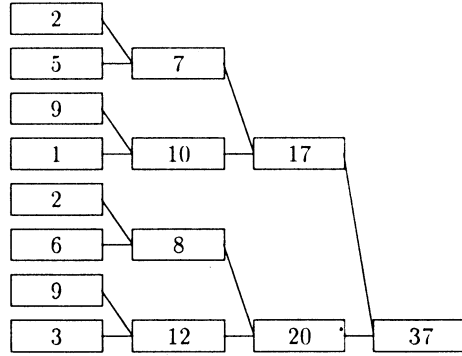
$$reduceR(\oplus)(as, b) = b \oplus lrtreeUp\ node\ as$$

where $node(s_1, s_2) = (s_1 \oplus s_2)$

If the operator \oplus is also commutative, we can traverse an odd-even tree instead of a left-right tree (with same function $node$).

An example computation of a function of this subclass is the sum of all elements in a list. With $\oplus = +$, $as = [2, 5, 9, 1, 2, 6, 9, 3]$, $b = 0$, the result is 37.

$$red(+)([2, 5, 9, 1, 2, 6, 9, 3], 0) = 37$$

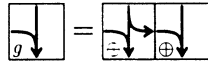


5.1.2 SR-Reduction. Functions in this subclass are given by:

$$foldl\ g\ (as, (b_1, b_2)) \quad \text{where} \quad g(a, (b_1, b_2)) = (b_1 \oplus a, b_2 \oplus b_1),$$

\oplus is associative and commutative

This subclass is a target subclass for a pattern of *scan* and *red*:



$$foldl\ g = (Id \times red(\oplus)) \circ \begin{smallmatrix} (2,1) \\ \Downarrow \\ (1,2) \end{smallmatrix} \circ (scan(\oplus) \times Id) \circ \begin{smallmatrix} (1,2) \\ \Downarrow \\ (2,1) \end{smallmatrix}$$

A parallel implementation is given by the higher-order function $reduceSR$, which is instantiated with the associative and commutative operator \oplus and its neutral element 1_{\oplus} . The computation traverses an odd-even

tree (`oetreeUp`) after the initialization of its leaves with a special initialization for the first leaf (`mapHdTl`); see Appendix A for the primitives `oetreeUp` and `mapHdTl`:

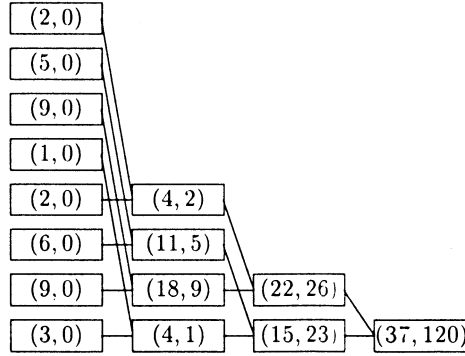
$$\begin{aligned}
 & \text{reduceSR}(\oplus) 1_{\oplus}(as, (b_1, b_2)) = (b_1 \oplus y_1, b_2 \oplus y_2) \\
 & \text{where } (y_1, y_2) = \text{oetreeUp node } as' \\
 & \text{and } as' = \text{mapHdTl leaf_hd leaf_tl } as \\
 & \text{and leaf_hd } s = (s, b_1) \\
 & \text{and leaf_tl } s = (s, 1_{\oplus}) \\
 & \text{and node}((s_1, t_1), (s_2, t_2)) = (s_1 \oplus s_2, t_1 \oplus t_1 \oplus t_2 \oplus t_2 \oplus s_1)
 \end{aligned}$$

An example computation of a function of this subclass is the sum of the prefix sum of a list. With $\oplus = +$, $as = [2, 5, 9, 1, 2, 6, 9, 3]$, $b_0 = 0$ and $b_1 = 0$, the result is 120.

$$((\text{Id} \times \text{red}(+)) \circ \begin{smallmatrix} (2,1) \\ \Downarrow \\ (1,2) \end{smallmatrix}) \circ (\text{scan}(+) \times \text{Id}) \circ \begin{smallmatrix} (1,2) \\ \Downarrow \\ (2,1) \end{smallmatrix}) (as, (0, 0))$$

with its constituents

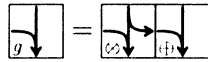
$$\begin{aligned}
 \text{scan}(+)([2, 5, 9, 1, 2, 6, 9, 3], 0) &= (37, [0, 2, 7, 16, 17, 19, 25, 34]) \\
 \text{red}(+)([0, 2, 7, 16, 17, 19, 25, 34], 0) &= 120
 \end{aligned}$$



5.1.3 SR2-Reduction. Functions in this subclass are given by:

$$\begin{aligned}
 & \text{foldl } g(as, (b_1, b_2)) \\
 & \text{where } g(a, (b_1, b_2)) = (b_1 \otimes a, b_2 \oplus b_1), \\
 & \quad \oplus \text{ and } \otimes \text{ are associative, } \otimes \text{ distributes over } \oplus
 \end{aligned}$$

This subclass is a target subclass for a pattern of *scan* and *red*:



$$\text{foldl } g = (\text{Id} \times \text{red}(\oplus)) \circ \begin{smallmatrix} (2,1) \\ \Downarrow \\ (1,2) \end{smallmatrix} \circ (\text{scan}(\otimes) \times \text{Id}) \circ \begin{smallmatrix} (1,2) \\ \Downarrow \\ (2,1) \end{smallmatrix}$$

A parallel implementation is given by the higher-order function *reduceSR2*, which is instantiated with the associative operators \oplus and \otimes and the neutral element 1_\otimes of \otimes . The computation traverses a left-right tree (*lrTreeUp*) after the initialization of its leaves (*map*):

$$\begin{aligned} \text{reduceSR2}(\oplus)(\otimes) 1_\otimes(a, (b_1, b_2)) &= (b_1 \otimes y_1, b_2 \oplus (b_1 \otimes y_2)) \\ \text{where } (y_1, y_2) &= \text{lrTreeUp node}(\text{map leaf } as) \\ \text{and leaf } s &= (s, 1_\otimes) \\ \text{and node}((s_1, t_1), (s_2, t_2)) &= (s_1 \otimes s_2, t_1 \oplus (s_1 \otimes t_2)) \end{aligned}$$

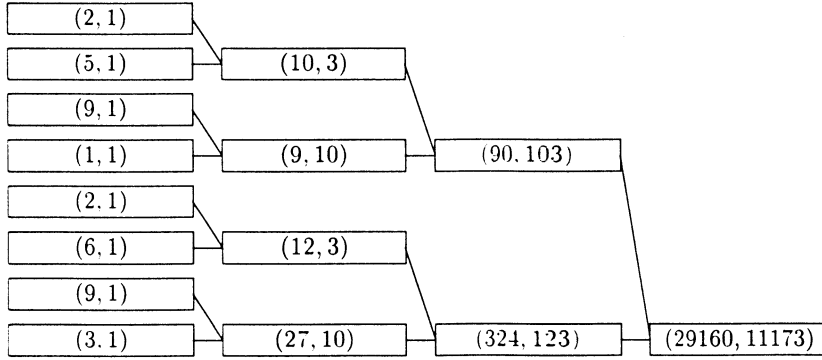
Function *reduceSR2* is similar to Cai's and Skillicorn's implementation of function *recur-reduce* [2] which is used to compute linear recurrences.

An example computation of a function of this subclass is the sum of the prefix sum of a list. With $\oplus = +$, $\otimes = *$, $as = [2, 5, 9, 1, 2, 6, 9, 3]$, $b_0 = 1$ and $b_1 = 0$, the result is 11173.

$$((\text{Id} \times \text{red}(+)) \circ \begin{smallmatrix} (2,1) \\ \Downarrow \\ (1,2) \end{smallmatrix}) \circ (\text{scan}(*) \times \text{Id}) \circ \begin{smallmatrix} (1,2) \\ \Downarrow \\ (2,1) \end{smallmatrix})(as, (1, 0))$$

with its constituents

$$\begin{aligned} \text{scan}(*)([2, 5, 9, 1, 2, 6, 9, 3], 1) &= (37, [1, 2, 10, \dots], 29160) \\ \text{red}(+)([1, 2, 10, 90, 90, 180, 1080, 9720], 0) &= 11173 \end{aligned}$$



5.2 Scan

For all functions in the subclasses of Scan which are listed in Table 2, we have $\text{time}(n) = \mathcal{O}(\log n)$, $\text{cost}_{\text{Brent}}(n) = \mathcal{O}(n)$ and $\text{pipe}(n) = \mathcal{O}(1)$. Note that we do not provide extra implementations for SS-Scan and SS2-Scan. Implementations for functions in these subclasses are constructed from the implementations of functions in the subclass Simple Scan.

5.2.1 Simple Scan. Functions in this subclass are well known (and often simply called “scan” or “pre-scan”). They are given by:

$$\text{scanl } g \text{ } (as, b) \quad \text{where} \quad g(a, b) = (b \oplus a), \quad \oplus \text{ is associative}$$

Functions in this subclass are instances of the basic component “S”. There are two well-known ways to implement a simple scan:

- A two-phase scan (up-sweep and down-sweep) is appropriate for tree architectures [15].
- A single-phase scan is appropriate for hypercube and butterfly architectures [3].

5.2.2 SS-Scan. Functions in this subclass are given by:

$$\text{scanl } g \text{ } (as, (b_1, b_2)) \quad \text{where} \quad g(a, (b_1, b_2)) = (b_1 \oplus a, b_2 \oplus b_1), \\ \oplus \text{ is associative and commutative}$$

The new part of the elements in the distributed output (vs. the result of a simple Scan) can be computed by:

$$\begin{array}{c} \boxed{\begin{array}{c} \downarrow \\ g \end{array} \rightarrow \boxed{\begin{array}{c} \downarrow \\ \pi_2 \end{array}} = \boxed{\begin{array}{c} \downarrow \\ \oplus \end{array}} \rightarrow \boxed{\begin{array}{c} \downarrow \\ \vdots \end{array}} \rightarrow \boxed{\begin{array}{c} \downarrow \\ \oplus \end{array}} \\ (\text{Id} \times \text{map } \pi_2) \circ (\text{scanl } g) = \begin{array}{c} (1,2) \\ \Downarrow \\ (2,1) \end{array} \circ (\text{Id} \times \text{scan } (\oplus)) \circ \begin{array}{c} (2,1) \\ \Downarrow \\ (1,2) \end{array} \circ \\ (\text{scan } (\oplus) \times \text{Id}) \circ \begin{array}{c} (1,2) \\ \Downarrow \\ (2,1) \end{array} \end{array}$$

This pattern uses two implementations of a scan in a pipeline.

5.2.3 SS2-Scan. Functions in this subclass are given by:

$$\text{scanl } g \text{ } (as, (b_1, b_2)) \\ \text{where} \quad g(a, (b_1, b_2)) = (b_1 \otimes a, b_2 \oplus b_1), \\ \oplus \text{ and } \otimes \text{ are associative, } \otimes \text{ distributes over } \oplus$$

The new part of the elements in the distributed output (vs. the result of a simple Scan) can be computed by:

$$\begin{array}{c} \boxed{\begin{array}{c} \downarrow \\ g \end{array} \rightarrow \boxed{\begin{array}{c} \downarrow \\ \pi_2 \end{array}} = \boxed{\begin{array}{c} \downarrow \\ \otimes \end{array}} \rightarrow \boxed{\begin{array}{c} \downarrow \\ \oplus \end{array}} \rightarrow \boxed{\begin{array}{c} \downarrow \\ \otimes \end{array}} \\ (\text{Id} \times \text{map } \pi_2) \circ \text{scanl } g = \begin{array}{c} (1,2) \\ \Downarrow \\ (2,1) \end{array} \circ (\text{Id} \times \text{scan } (\oplus)) \circ \begin{array}{c} (2,1) \\ \Downarrow \\ (1,2) \end{array} \circ \\ (\text{scan } (\otimes) \times \text{Id}) \circ \begin{array}{c} (1,2) \\ \Downarrow \\ (2,1) \end{array} \end{array}$$

This pattern uses two implementations of a scan in a pipeline.

5.3 Sequential

In this class, we apply function g to the carried input n times. Remember that n , the depth of the recursion, is still determined by the list of local inputs $as = [(), n \text{ times}, ()]$. For all subclasses of Sequential shown in Table 3, except Identity, we use a processor network with $\log n$ processors in a row (`repeat`) which computes the result with $time(n) = \mathcal{O}(\log n)$, $cost_{Brent}(n) = \mathcal{O}(\log n)$ and $pipe(n) = \mathcal{O}(1)$.

A row of processors can be mapped easily onto any processor network without a time penalty. Thus, our given complexities also apply to other architectures.

We assume n to be a power of 2. For other values, we have to use more complex computations (which are similar to the computations used in class Broadcast).

5.3.1 CR-Accumulation. Functions in this subclass are given by:

$$g^n(b_0, b_1) \quad \text{where} \quad g(b_0, b_1) = (b_0, b_1 \oplus b_0), \quad \oplus \text{ is associative}$$

This subclass is the target subclass for a pattern of *copy* and *red*:

$$g^n = (\text{Id} \times \text{red}(\oplus)) \circ \begin{matrix} (2,1) \\ \Downarrow \\ (1,2) \end{matrix} \circ (\text{copy}_n \times \text{Id})$$

A parallel implementation is given by the higher-order function *accCR* which is instantiated by the associative operator \oplus :

$$\begin{aligned} \text{accCR}(\oplus)(n, (b_0, b_1)) &= (b_0, b_1 \oplus \text{repeat node}(\log_2 n) b_0) \\ \text{where } \text{node } s &= s \oplus s \end{aligned}$$

This method is the one used for the efficient evaluation of powers [13, Sect. 4.6.3].

5.3.2 CSR-Accumulation. Functions in this subclass are given by:

$$g^n(b_0, b_1, b_2) \quad \text{where} \quad g(b_0, b_1, b_2) = (b_0, b_1 \oplus b_0, b_2 \oplus b_1),$$

\oplus is associative and commutative

Instead of commutativity it suffices that $\exists e. b_0 = e \oplus \dots \oplus e, b_1 = e \oplus \dots \oplus e$ or $b_1 = 1_{\oplus}$.

This subclass is a target subclass for a pattern of *copy*, *scan* and *red*:

$$\begin{array}{c}
 \boxed{g \downarrow} = \boxed{\text{Id} \downarrow \oplus \downarrow \oplus \downarrow \oplus \downarrow} \\
 g^n = (\text{Id} \times \text{Id} \times \text{red}(\oplus)) \circ \begin{array}{c} (1,2,1) \\ \downarrow\downarrow \\ (1,1,2) \end{array} \circ (\text{Id} \times \text{scan}(\oplus) \times \text{Id}) \circ \\
 \begin{array}{c} (2,1,1) \\ \downarrow\downarrow \\ (1,2,1) \end{array} \circ (\text{copy}_n \times \text{Id} \times \text{Id})
 \end{array}$$

A parallel implementation is given by the higher-order function *accCSR*, which is instantiated with the associative and commutative operator \oplus :

$$\begin{aligned}
 \text{accCSR}(\oplus)(n, (b_0, b_1, b_2)) &= (b_0, b_1 \oplus y_1, b_2 \oplus y_2) \\
 \text{where } (y_1, y_2, _) &= \text{repeat node}(\log_2 n)(b_0, b_1, b_0) \\
 \text{and } \text{node}(s, t, u) &= (s \oplus s, t \oplus t \oplus u, \underbrace{u \oplus u \oplus u \oplus u}_4)
 \end{aligned}$$

An example computation of a function of this subclass is $\sum_{i=0}^7 (3 + i)$ with result 52. We have $\oplus = +$, $n = 8$, $b_0 = 1$, $b_1 = 3$ and $b_2 = 0$.

$$\begin{array}{c}
 ((\text{Id} \times \text{Id} \times \text{red}(+)) \circ \begin{array}{c} (1,2,1) \\ \downarrow\downarrow \\ (1,1,2) \end{array} \circ (\text{Id} \times \text{scan}(+) \times \text{Id}) \circ \\
 \begin{array}{c} (2,1,1) \\ \downarrow\downarrow \\ (1,2,1) \end{array} \circ (\text{copy}_n \times \text{Id} \times \text{Id}))(1, 3, 0)
 \end{array}$$

with its constituents

$$\begin{aligned}
 \text{scan}(+)([1, 1, 1, 1, 1, 1, 1, 1], 3) &= (11, [3, 4, 5, 6, 7, 8, 9, 10]) \\
 \text{red}(+)([3, 4, 5, 6, 7, 8, 9, 10], 0) &= 52
 \end{aligned}$$

$$\boxed{(1, 3, 1)} \text{---} \boxed{(2, 7, 4)} \text{---} \boxed{(4, 18, 16)} \text{---} \boxed{(8, 52, 64)}$$

5.3.3 CSR2-Accumulation. Functions in this subclass are given by:

$$g^n(b_0, b_1, b_2) \quad \text{where} \quad g(b_0, b_1, b_2) = (b_0, b_1 \otimes b_0, b_2 \oplus b_1), \\
 \oplus \text{ and } \otimes \text{ are assoc., } \otimes \text{ distributes over } \oplus$$

This subclass is a target subclass for a pattern of *copy*, *scan* and *red*, similar to CSR-Accumulation.

A parallel implementation is given by the higher-order function *accCSR2* which is instantiated by the associative operators \oplus and \otimes :

$$\begin{aligned}
 \text{accCSR2}(\oplus)(\otimes)(n, (b_0, b_1, b_2)) &= (b_0, b_1 \otimes y_1, b_2 \oplus y_2) \\
 \text{where } (y_1, y_2) &= \text{repeat node}(\log_2 n)(b_0, b_1) \\
 \text{and } \text{node}(s, t) &= (s \otimes s, t \oplus (t \otimes s))
 \end{aligned}$$

5.4 Broadcast

In this class, we apply function g to the carried input n times and return a list of intermediate results and the final result whose implementation is known from the previous section. For all subclasses shown in Table 3, we use a tree-like processor network with n processors which computes the result with $time(n) = \mathcal{O}(\log_2 n)$, $cost_{\text{Brent}}(n) = \mathcal{O}(n)$ and $pipe(n) = \mathcal{O}(1)$. At each node, a function is applied which receives the input from its parent on the left side and provides two outputs to its children on the right. The carried input b is the input for the root; the list of intermediate results is composed of the values at the leaves.

The odd-even tree used in this subsection can be mapped easily onto a hypercube without a time penalty. Thus, our given complexities also apply to hypercube architectures.

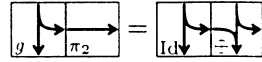
5.4.1 Copy. The only function in this subclass is $ga = a$. Two parallel implementations are given by:

$$\begin{aligned} broadcastC(n, b) &= \text{lr treeDn node}(\log_2 n) b \\ &= \text{oetreeDn node}(\log_2 n) b \\ \text{where } node\ s &= (s, s) \end{aligned}$$

5.4.2 CS-Broadcast. Functions in this subclass are given by:

$$broadcast_n g(b_0, b_1) \quad \text{where } g(b_0, b_1) = (b_0, b_1 \oplus b_0), \oplus \text{ is associative}$$

This subclass is a target subclass for a pattern of *copy* and *scan*:



$$(\text{Id} \times \text{map } \pi_2) \circ broadcast_n g = (\text{Id} \times \text{scan}(\oplus)) \circ \begin{matrix} (2,1) \\ \Downarrow \\ (1,2) \end{matrix} \circ (\text{copy}_n \times \text{Id})$$

A parallel implementation is given by the higher-order function $broadcastCS$, which is instantiated with the associative operator \oplus . The computation traverses an odd-even tree (oetreeDn), the result is extracted from the values of its leaves (map):

$$\begin{aligned} broadcastCS(\oplus)(n, (b_0, b_1)) &= \text{map leaf } zs' \\ \text{where } zs' &= \text{oetreeDn node}(\log_2 n)(b_0, b_1, b_0) \\ \text{and } node(s, t, u) &= ((s, t, u \oplus u), (s, t \oplus u, u \oplus u)) \\ \text{and } leaf(s, t, _) &= (s, t) \end{aligned}$$

The carried result is the result of a CR-Accumulation.

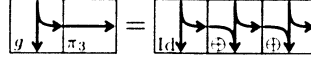
5.4.3 *CSS-Broadcast*. Functions in this subclass are given by:

$$\text{broadcast}_n g(b_0, b_1, b_2) \quad \text{where} \quad g(b_0, b_1, b_2) = (b_0, b_1 \oplus b_0, b_2 \oplus b_1),$$

\oplus is associative and commutative

Instead of commutativity it suffices that $\exists e. b_0 = e \oplus \dots \oplus e, b_1 = e \oplus \dots \oplus e$ or $b_1 = 1_{\oplus}$.

This subclass is a target subclass for a pattern of *copy*, *scan* and *scan*:



$$\begin{aligned} (\text{Id} \times \text{Id} \times \text{map } \pi_3) \circ \text{broadcast}_n g &= (\text{Id} \times \text{Id} \times \text{scan}(\oplus)) \circ \begin{matrix} (1,2,1) \\ \Downarrow \\ (1,1,2) \end{matrix} \circ \\ &= (\text{Id} \times \text{scan}(\oplus) \times \text{Id}) \circ \begin{matrix} (2,1,1) \\ \Downarrow \\ (1,2,1) \end{matrix} \circ \\ &= (\text{copy}_n \times \text{Id} \times \text{Id}) \end{aligned}$$

A parallel implementation is given by the higher-order function *broadcCSS* which is instantiated by the associative and commutative operator \oplus and its neutral element 1_{\oplus} . The computation traverses an odd-even tree (*oetreeDn*), the result is extracted from the values of its leaves. Part of the result is already known from the CS-Broadcast and combined with the new one (*zip*):

$$\begin{aligned} \text{broadcCSS}(\oplus) 1_{\oplus}(n, (b_0, b_1, b_2)) &= \text{zip join}(old, new) \\ \text{where } \text{join}((y_0, y_1), (s, -, -, w)) &= (y_0, y_1, w \oplus s) \\ \text{and } old &= \text{broadcCS}(\oplus)(n, (b_0, b_1)) \\ \text{and } new &= \text{oetreeDn node}(\log_2 n)(1_{\oplus}, b_1, b_0, 1_{\oplus}, b_2) \\ \text{and } \text{node}(s, t, u, v, w) &= ((s, t \oplus t \oplus u, u \oplus u \oplus u \oplus u, v \oplus v, w), \\ &\quad (s \oplus t \oplus v, \underbrace{t \oplus t \oplus u}_2, \underbrace{u \oplus u \oplus u \oplus u}_4, \underbrace{u \oplus u}_2, \underbrace{v \oplus v}_2, w)) \end{aligned}$$

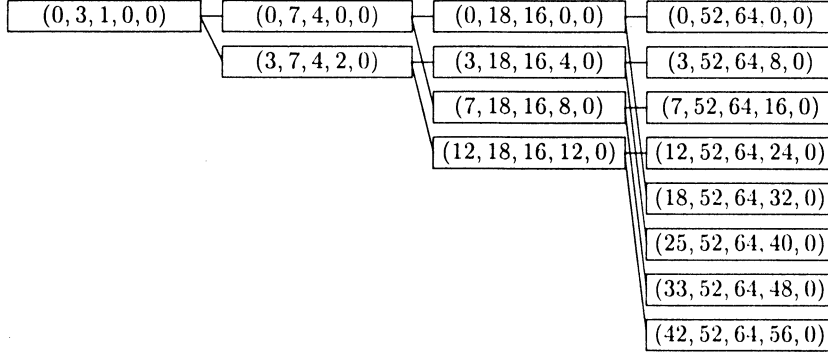
The carried result is the result of a CSR-Accumulation.

An example computation of a function of this subclass is $\left[\sum_{j=0}^{i-1} (3+j) \mid i \in 0..7 \right]$ with result $[0, 3, 7, 12, 18, 25, 33, 42]$. We have $\oplus = +, n = 8, b_0 = 1, b_1 = 3$ and $b_2 = 0$.

$$\begin{aligned} ((\text{Id} \times \text{Id} \times \text{scan}(+)) \circ \begin{matrix} (1,2,1) \\ \Downarrow \\ (1,1,2) \end{matrix} \circ (\text{Id} \times \text{scan}(+) \times \text{Id}) \circ \\ \begin{matrix} (2,1,1) \\ \Downarrow \\ (1,2,1) \end{matrix} \circ (\text{copy}_n \times \text{Id} \times \text{Id}))(1, 3, 0) \end{aligned}$$

with its constituents

$$\begin{aligned} \text{scan}(+)([1, 1, 1, 1, 1, 1, 1], 3) &= (11, [3, 4, 5, 6, 7, 8, 9, 10]) \\ \text{scan}(+)([3, 4, 5, 6, 7, 8, 9, 10], 0) &= (52, [0, 3, 7, 12, 18, 25, 33, 42]) \end{aligned}$$



5.4.4 *CSS2-Broadcast*. Functions in this subclass are given by:

$$\text{broadcast}_n g(b_0, b_1, b_2) \quad \text{where} \quad g(b_0, b_1, b_2) = (b_0, b_1 \otimes b_0, b_2 \oplus b_1),$$

\oplus and \otimes are associative, \otimes distributes over \oplus

This subclass is a target subclass for a pattern of *copy*, *scan* and *scan*, similar to *CSS-Broadcast*.

A parallel implementation is given by the higher-order function *broadcCSS2*, which is instantiated by the associative operators \oplus and \otimes and the neutral element 1_{\oplus} of \oplus . The computation traverses an odd-even tree (*oetreeDn*), the result is extracted from the values of its leaves, part of the result is already known from the *CS-Broadcast* and combined with the new one (*zip*):

$$\begin{aligned} \text{broadcCSS2}(\oplus) 1_{\oplus}(\otimes)(n, (b_0, b_1, b_2)) &= \text{zip join}(old, new) \\ \text{where } \text{join}(y_0, y_1)(s, -, -, v) &= (y_0, y_1, v \oplus s) \\ \text{and } old &= \text{broadcCS}(\oplus)(n, (b_0, b_1)) \\ \text{and } new &= \text{oetreeDn node}(\log_2 n)(1_{\oplus}, b_1, b_0, b_2) \\ \text{and } \text{node}(s, t, u, v) &= ((s, t \oplus (t \otimes u), u \otimes u, v), \\ &\quad (t \oplus (s \otimes u), t \oplus (t \otimes u), u \otimes u, v)) \end{aligned}$$

The carried result is the result of a *CSR2-Accumulation*.

6 Combinations of pre and post parts

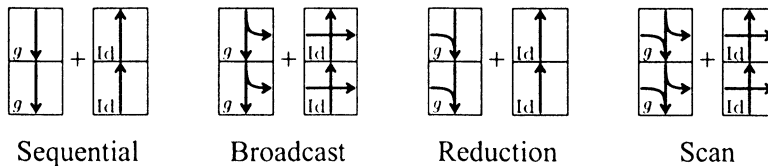
In Sect. 4, we have classified the **pre** part whose carried data flows downwards and mentioned that the same could be done for the **post** part whose carried data flows upwards. In this section we look at combinations of the **pre** and the **post** part.

Some combinations are trivial (Sect. 6.1), whereas more complicated ones (which are shown in Sect. 6.5) use the decomposition of both parts into basic components (Sect. 6.2).

6.1 Trivial combinations of the pre and post part

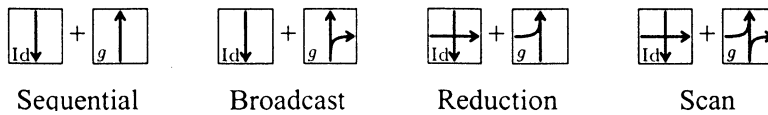
An instance of skeleton *PLR* for parallel linear recursion is a trivial combination of a **pre** and a **post** part if the computation is just performed in one part.

With a trivial **post** part, i.e., if the computation is just performed in the **pre** part, the functional argument *post* for the skeleton *PLR* is the identity function. For these combinations, we can use the implementations presented in Sect. 5. Here are the boxes for classes Sequential, Broadcast, Reduction, and Scan with a trivial **post** part:



Next, we depict the **pre** and the **post** part by one row instead of two.

With a trivial **pre** part, i.e., if the computation is just performed in the **post** part, the functional argument *pre* for skeleton *PLR* is the identity function. For these combinations, we can use the reverse of the implementations presented in Sect. 5. In a *reverse* implementation the input and/or output list has to be provided in the reverse order. Here are the boxes for classes Sequential, Broadcast, Reduction, and Scan with a trivial **pre** part:



6.2 Decomposition into basic components

There are two equivalent points of view: (1) we can just combine the **pre** and **post** part, with data flowing downwards in the **pre** part and upwards

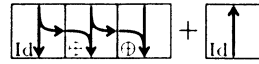
in the **post** part, or (2) we can decompose both parts into basic components and combine those, with data flowing, possibly, in both directions. Implementations that we develop for (2) are also implementations for (1).

Our method in the rest of this section is as follows:

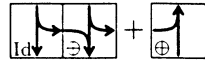
1. We decompose the subclasses into basic components, i.e., we remember that specific subclasses are targets for combinations of basic components.
2. We note that the implementations of Sect. 5 can be used for trivial combinations of the **pre** and **post** part.
3. We show that the carried data flow of some basic components can be reversed from downward to upward without affecting the result. We obtain a combination of basic components with a downward carried data flow followed by basic components with an upward carried data flow.
4. We combine the basic components with a downward carried data flow in the **pre** part and the basic components with an upward carried data flow in the **post** part. This combination can use the implementation of the original subclass of item 1.

6.3 Example: CSR-Accumulation

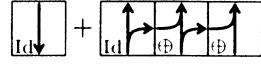
Section 5.3.2 presents an implementation for a CSR-Accumulation in the **pre** part. As mentioned in Sect. 6.1, this implementation can be used for a combination with a trivial **post** part:



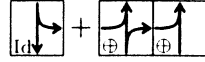
Note that the last basic component in the **pre** part is a reduction with an associative operator \oplus . This means that we can reduce from the right rather than from the left. Thus, we can reverse the carried data flow of the last basic component from downward to upward, which means that this basic component belongs to the **post** part (we have a CS-Broadcast in the **pre** part and an undirected reduction in the **post** part):



According to Sect. 6.1, the reverse implementation can be used for a combination with a trivial **pre** part. Since we have no input and no output list (more precisely, just lists with elements of the unit type), there is nothing to reverse, i.e., we can use the original implementation for a CSR-Accumulation:



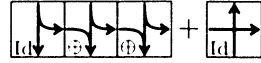
The first basic component in the **post** part distributes the same value to all places of its output. It does not matter whether it is forwarded from the bottom or from the top. Thus, we can reverse the carried data flow of the first basic component from upward to downward, which means that this basic component belongs to the **pre** part (we have a copy in the **pre** part and an SR-Reduction in the **post** part):



It turns out that the implementation of Sect. 5.3.2 can be used for all combinations of the basic components *copy*, *scan*, and *red* with different directions of the carried data flow.

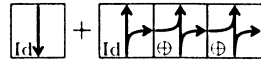
6.4 Example: CSS-Broadcast

Section 5.4.3 presents an implementation for a CSS-Broadcast in the **pre** part. As mentioned in Sect. 6.1, this implementation can be used for a combination with a trivial **post** part:

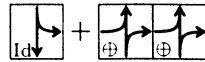


The last basic component in the **pre** part is a scan, whose carried data flow cannot be reversed without changing the result.

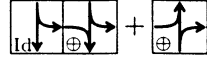
According to Sect. 6.1, the reverse implementation can be used for a combination with a trivial **pre** part:



The first basic component in the **post** part distributes the same value to all places of its output. It does not matter whether it is forwarded from the bottom or from the top. Thus, we can reverse the carried data flow of the first basic component from upward to downward, which means that this basic component belongs to the **pre** part:



The last basic component in the **post** part is a scan whose carried data flow cannot be reversed without changing the result. Thus, no implementation of Sect. 5 can be used to implement the following combination of basic components:



The implementation of this “reflecting” broadcast is given in Sect. 6.6.

6.5 Basic components in the **pre** and **post** part

Table 4 lists combinations of basic components in the **pre** and the **post** part for which we offer efficient implementations.

In Sects. 6.3 and 6.4, we have seen that a specific implementation can be used for different combinations of the **pre** and the **post** part.

In each row of Table 4, the first entry names the implementation we can use for the combinations of basic components in the **pre** and the **post** part, depicted by the following entries in the row.

The implementations we use are either the ones presented in Sect. 5, their reverses (mentioned in Sect. 6.1), or an extra implementation if necessary (motivated in Sect. 6.4, presented in Sect. 6.6).

The first combination in each row is always a trivial combination of a **pre** and a **post** part, except in the row for the extra implementation. The fact that the same implementation can be used for different combinations is indicated by the symbols $\stackrel{\text{ass}}{=}$, $\stackrel{\text{Id}}{=}$, and $\stackrel{\text{seq}}{=}$, see Sect. 6.3 and Sect. 6.4 for details.

6.6 Implementation: CSrS-Broadcast

In this subsection, we propose an implementation for a functional composition of a CS-Broadcast and then a simple Scan from the right (the reverse direction!). This composition can be viewed as a broadcast and is given by:

$$\text{scan}(\oplus)(\pi_2(\text{broadcast}_n g(a, b)), c)$$

where $g(a, b) = (a, b \oplus a)$, \oplus is associative and commutative

Instead of commutativity it suffices that $\exists e. b_0 = e \oplus \dots \oplus e, b_1 = e \oplus \dots \oplus e$ or $b_1 = 1_{\oplus}$. This pseudo subclass is a target subclass for a pattern of *copy*, *scanl* and *scanr*:

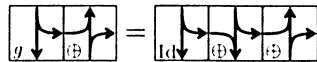
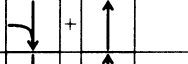
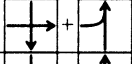
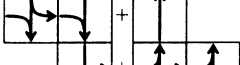
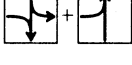

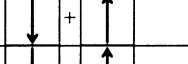
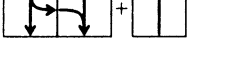
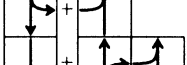
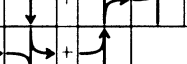
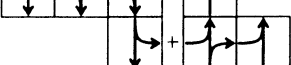
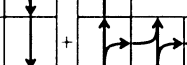
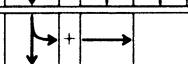
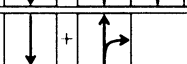
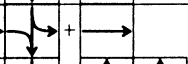
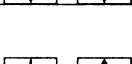
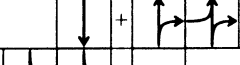
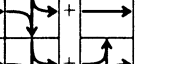

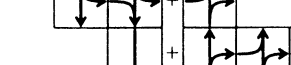
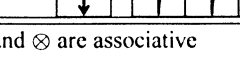
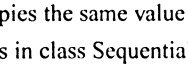
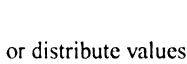


Table 4. Implementations for the pre and post part

may be used by:			
Implementation	pre + post		pre + post
Undirected Red.		$\stackrel{\text{ass}}{=}$	
SR(2)-Reduction		$\stackrel{\text{ass}}{=}$	
... (post part)			
All Scans parts are implemented independently			
Identity			
CR-Accum.		$\stackrel{\text{ass}}{=}$	
		$\stackrel{\text{Id}}{=}$	
CSR(2)-Accum.		$\stackrel{\text{ass}}{=}$	
$\stackrel{\text{Seq}}{=}$		$\stackrel{\text{Id}}{=}$	
Copy		$\stackrel{\text{Id}}{=}$	
CS-Broadcast			
... (post part)		$\stackrel{\text{Id}}{=}$	
CSS(2)-Broadc.			
... (extra impl.)			
... (post part)		$\stackrel{\text{Id}}{=}$	

$\stackrel{\text{ass}}{=}$: equality, since \oplus and \otimes are associative
 $\stackrel{\text{Id}}{=}$: equality, since copy_n copies the same value
 $\stackrel{\text{Seq}}{=}$: equality, since functions in class Sequential do not collect or distribute values

A parallel implementation is given by the higher-order function *broadcCSrS* which is instantiated by the associative and commutative operator \oplus and its neutral element 1_{\oplus} . The “r” in “CSrS” stands for “reflecting”. The computation traverses an odd-even tree (*oetreeDn*), the result is extracted from the values of its leaves. Function *broadcCSrS* combines (with *zip*) one part of the result, which is given by the CS-Broadcast previously described and which we call *old*, with another part, which we call *new*.

The computation is defined by:

$$\begin{aligned}
 \text{broadcCSrS } (\oplus) \text{ } 1_{\oplus} (a, b, c) &= \text{zip join } (old, new) \\
 \text{where } \text{join}((y_0, y_1), (s, -, -, w)) &= (y_0, y_1, w \oplus s) \\
 \text{and } old &= \text{broadcCS } (\oplus) (n, (b_0, b_1)) \\
 \text{and } new &= \text{oetreeDn node } (n, (0_{\oplus}, b, a, 0_{\oplus}, c)) \\
 \text{and } h(s, t, u, v, w) &= ((s \oplus t \oplus u, t \oplus t \oplus u, u \oplus u \oplus u \oplus u, u \oplus u \oplus v \oplus v, w), \\
 &\quad (s \oplus v, \underbrace{t \oplus t \oplus u}_2, \underbrace{u \oplus u \oplus u \oplus u}_4, \underbrace{v \oplus v}_2, w))
 \end{aligned}$$

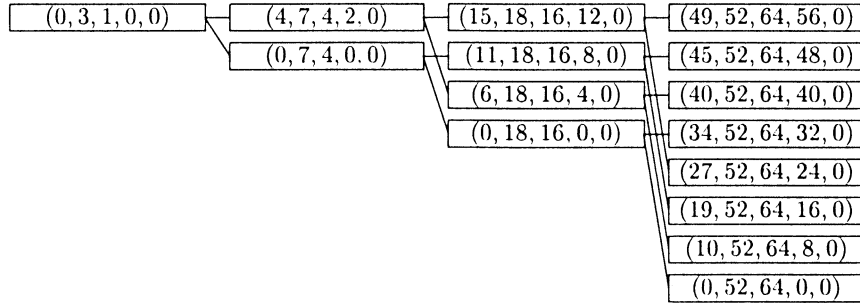
The carried result is the result of a CSR-Accumulation with argument (a, b, c) .

An example computation of a function of this subclass is $[\sum_{j=i}^{8-1} (3+j) \mid i \in 1..8]$ with the result of $[49, 45, 40, 34, 27, 19, 10, 0]$. We have $\oplus = +$, $n = 8$, $b_0 = 1$, $b_1 = 3$ and $b_2 = 0$.

$$\begin{aligned}
 ((\text{Id} \times \text{Id} \times \text{scanr } (+)) \circ \begin{smallmatrix} (1,2,1) \\ \Downarrow \\ (1,1,2) \end{smallmatrix} \circ (\text{Id} \times \text{scanl } (+) \times \text{Id})) \circ \\
 \begin{smallmatrix} (2,1,1) \\ \Downarrow \\ (1,2,1) \end{smallmatrix} \circ (\text{copy}_n \times \text{Id} \times \text{Id})) (1, 3, 0)
 \end{aligned}$$

with its constituents

$$\begin{aligned}
 \text{scanl } (+) ([1, 1, 1, 1, 1, 1, 1, 1], 3) &= (11, [3, 4, 5, 6, 7, 8, 9, 10]) \\
 \text{scanr } (+) ([3, 4, 5, 6, 7, 8, 9, 10], 0) &= (52, [49, 45, 40, 34, 27, 19, 10, 0])
 \end{aligned}$$



7 Conclusions

The skeletal approach aims at a plug-in style of parallel programming. In developing skeletons, one is seeking popular patterns of parallelism and communication which one can offer to programmers as building blocks for

larger parallel programs. Our contribution is to look at combinations of some of the most basic building blocks and optimize them further. We obtain these optimizations by staying in the world of linearly recursive functions and by classifying special cases of linear recursion in a table which we then use in our analysis.

Our targets have been compositions of the skeletons Broadcast, Reduction and Scan. We have looked at three cases: (1) that the carried data flow downwards, i.e., to deeper levels of the recursion, (2) that it flows upwards and (3) the combination of both directions which occurs in linearly recursive functions.

Good parallel implementations require the binary operator subject to a reduction or scan to be associative. When combining these targets, additional algebraic properties, such as commutativity or distributivity are sometimes required.

Elsewhere, patterns of recursive functions, also on lists, have been studied in the Dutch STOP project [14]. There, general linear recursion is captured by the notion of a hylomorphism. Similar to our work, the STOP project led to valid transformations for the composition of these patterns. However, the transformations serve not for the identification of better parallel implementations but as a starting point for extending the STOP theory to data types other than lists. We have concentrated on lists, have identified cases in which our transformations improve the quality of the parallel implementation and have derived the improved implementations.

Transformations for optimizing compositions have also been considered elsewhere, mostly by improving the distribution of the data or using pipelines as in To's Ph.D. thesis [20]. Our optimization is based on the algebraic properties of the functions involved.

We have chosen the paradigm of functional programming, but the skeletal approach applies also to imperative programs. The advantage of the functional paradigm is that program transformations can be checked more directly, via equational proofs. We have implemented our solutions in Haskell [19]. Just like, e.g., the Glasgow Haskell compiler compiles Haskell into C, the parallel implementations of our skeletons will have to be in a language like C with MPI calls. Here, we have not addressed this issue, but we are working on it in the domain of divide-and-conquer recursions [11, 12].

Acknowledgements. This work is partially supported by grants from the ARC and PROCOPE exchange programs of the DAAD. Thanks to D. K. Arvind, L. Bougé, M. I. Cole, J. T. O'Donnell, S. Gorlatch and C. Herrmann for helpful discussions. The anonymous referees were helpful in improving the presentation.

References

1. R. S. Bird: Lectures on Constructive Functional Programming. In: Broy (ed) *Constructive Methods in Computing Sciences* (Internat. Summer School 1988, Marktoberdorf Germany), Vol 55 of NATO ASI Series F, pp 150–216. Berlin Heidelberg New York: Springer 1989
2. W. Cai, David B. Skillicorn: Calculating recurrences using the Bird-Meertens formalism. *Parallel Processing Letters*, 5(2):179–190, 1995
3. B. Carpentieri, G. Mou: Compile-time transformations and optimization of parallel divide-and-conquer algorithms. *ACM SIGPLAN Notices*, 26(10):19–28, 1991
4. M. I. Cole: *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. Pitman 1989
5. M. I. Cole: Parallel programming with list homomorphisms. *Parallel Processing Letters*, 5(2):191–204, 1995
6. J. Darlington, A. Field, P. G. Harrison: Parallel programming using skeleton functions. In: A. Bode, M. Reeve, G. Wolf (eds) *Parallel Architectures and Languages Europe (PARLE'93)*, Lecture Notes in Computer Science, Vol. 694, pp 146–160, Berlin Heidelberg New York: Springer 1993
7. S. Gorlatch: Stages and transformations in parallel programming. In: M. Kara, J. Davy, D. Goodeve, J. Nash (eds) *Abstract Machine Models for Parallel and Distributed Computing (AMW'96)*, pp 147–162, IOS Press 1996
8. S. Gorlatch: Systematic efficient parallelization of scan and other list homomorphisms. In: L. Bougé, P. Fraigniaud, A. Mignotte, Y. Robert (eds) *2nd European Conference on Parallel Processing (Euro-Par'96)*, Vol. 2, Lecture Notes in Computer Science, Vol. 1124, pp 401–408, Berlin Heidelberg New York: Springer 1996
9. S. Gorlatch: Systematic extraction and implementation of divide-and-conquer parallelism. In: H. Kuchen, S. Doaitse Swierstra (eds) *8th Int. Symposium on Programming Languages: Implementations, Logics, and Programs (PLILP'96)*, Lecture Notes in Computer Science, Vol. 1140, pp 274–288, Berlin Heidelberg New York: Springer 1996
10. S. Gorlatch. *Optimizing Compositions of Scans and Reductions in Parallel Program Derivation*. Fakultät für Mathematik und Informatik, Universität Passau May 1997
11. C. A. Herrmann, C. Lengauer: On the space-time mapping of a class of divide-and-conquer recursions. *Parallel Processing Letters*, 6(4): 525–537, 1996
12. C. A. Herrmann, C. Lengauer: Transformation of divide & conquer to nested parallel loops. In: H. Glaser, P. Hartel, H. Kuchen, (eds) *9th Int. Symposium on Programming Languages: Implementations, Logics, and Programs (PLILP'96)*, Lecture Notes in Computer Science, Vol. 1292, pp 95–109, Berlin Heidelberg New York: Springer 1997
13. D. E. Knuth: *The Art of Computer Programming*, Vol. 2: *Seminumerical Algorithms*. Addison-Wesley (2nd ed) 1980
14. E. Meijer, M. Fokkinga, R. Paterson: Functional programming with bananas, lenses, envelopes and barbed wire. In: J. Hughes (ed) *5th Conference on Functional Programming Languages and Computer Architecture (FPCA'91)*, Lecture Notes in Computer Science, Vol. 523, pp. 124–144, Berlin Heidelberg New York: Springer 1991
15. J. T. O'Donnell: A correctness proof of parallel scan. *Parallel Processing Letters*. 4(3):329–338, 1994
16. M. J. Quinn: *Parallel Computing: Theory and Practice*. McGraw-Hill (2nd edn.) 1994
17. D. B. Skillicorn: *Foundations of Parallel Programming*. Cambridge: Cambridge University Press 1994

18. D. B. Skillicorn, W. Cai: A cost calculus for parallel functional programming. *J. of Parallel and Distributed Computing*, 28:65–83, 1995
19. S. Thompson: *Haskell: The Craft of Functional Programming*. Addison-Wesley 1996
20. H. W. To: *Optimising the Parallel Behaviour of Combinations of Program Components*. PhD thesis, Imperial College, University of London, Sept. 1995
21. C. Wedler, C. Lengauer: Parallel implementations of combinations of broadcast, reduction and scan. In: G. Agha, S. Russo (eds) *2nd Int Workshop on Software Engineering for Parallel and Distributed Systems (PDSE'97)*, pp 108–119., IEEE Computer Society Press, 1997

Appendix: Implementation Primitives

In this section we list the main part of the Haskell module `PrimLR` which defines the primitives used in Sect. 5. To avoid name clashes with functions defined in the Haskell library `Prelude.hs`, we use `repTimes`, `mapList`, and `zipPair` instead of `repeat`, `map`, and `zip`.

The following functions are used from the Haskell library `Prelude.hs`:

$$\begin{aligned} \text{fst } (a, b) &= a \\ \text{snd } (a, b) &= b \\ \text{head } (a : as) &= a \\ \text{length } [a_1, \dots, a_n] &= n \\ \text{splitAt } k [a_1, \dots, a_k, a_{k+1}, \dots, a_n] \\ &= ([a_1, \dots, a_k], [a_{k+1}, \dots, a_n]) \end{aligned}$$

Functions `mapList`, `mapHdTl`, and `zipPair` can be executed with $\text{time}(n) = \mathcal{O}(1)$ on $\mathcal{O}(n)$ processors with the assumption that the functional argument has a constant execution time.

Functions `oetreeDn` and `oetreeUp` describe computations on an odd-even tree whose structures are depicted in the examples of Sect. 5.1.2 and Sect. 5.4.3. On architectures which provide these communication structures, e.g., on hypercubes, the parallel time is in $\mathcal{O}(\log n)$, where n is the length of the input/output list. The number of operations is in $\mathcal{O}(n)$.

Function `lrtreeUp` describes a similar computation on a left-right tree whose structure is depicted in the example of Sect. 5.1.3.

```

=====
---  "Sequential" Functions
=====

-- Apply function(1) number(2) times, initial argument is (3)
repTimes :: (a->a) -> Int -> a -> a
repTimes f k a
  | k == 0    = a
  | k > 0     = repTimes f (k-1) (f a)
  | otherwise = error "PrimLR.repTimes: negative argument"

-- Repeatedly apply function(1) until predicate(2) returns
-- True, initial argument is (3)
repUntil :: (a->a) -> (a->Bool) -> a -> a
repUntil f pred a
  | pred a    = a
  | otherwise = repUntil f pred (f a)

-- Check whether list(1) has length 1. Using ((=1) . length)
-- instead is slow
isSingleton :: [a] -> Bool
isSingleton [_] = True
isSingleton _   = False

=====
---  "Parallel" Functions with ParTime = O(1)
=====

-- Apply function(1) to all elements in list(2).
-- Prelude.map not only works on lists.
mapList :: (a->b) -> [a] -> [b]
mapList _ []      = []
mapList f (a:as) = f a : mapList f as

-- Apply function(1) to head, function(2) to all elements in
-- tail of list(3).
mapHdTl :: (a->b) -> (a->b) -> [a] -> [b]
mapHdTl fa fas (a:as) = fa a : mapList fas as
mapHdTl _ _ []       = error "PrimLR.mapHdTl: empty list"

-- Apply uncurried binary function(1) to all elements in
-- lists(2), pairing elements at same position.
-- Prelude.zipWith :: (a->b->c) -> [a] -> [b] -> [c]
zipPair :: ((a,b)->c) -> ([a],[b]) -> [c]
zipPair _ ([],[ ]) = []
zipPair f (a:as,b:bs) = f (a,b) : zipPair f (as,bs)
zipPair _ _ = error "PrimLR.zipPair: unequal length of lists"

```

```

=====
---  Tree functions: Left-Right, Odd-Even, ParTime = O(log n)
=====

-- Go down left-right tree of depth(2), starting at root with
-- element(3), apply function(1) at each node to compute
-- the values for the two children.
lrtreeDn :: (a->(a,a)) -> Int -> a -> [a]
lrtreeDn f k a = repTimes (step f) k [a]
                where step _ []      = []
                      step f (a:as) = let (b1,b2) = f a
                                         in b1 : b2 : step f as

-- Go down odd-even tree of depth(2), starting at root with
-- element(3), apply function(1) at each node to compute
-- the values for the two children.
oetreeDn :: (a->(a,a)) -> Int -> a -> [a]
oetreeDn f k a = repTimes (step f) k [a]
                where step f as = mapList (fst . f) as ++
                                   mapList (snd . f) as

-- Go up left-right tree, starting at leaves with list(2),
-- apply function(1) at the children to compute the value for
-- each node.
lrtreeUp :: ((a,a)->a) -> [a] -> a
lrtreeUp f as = head (repUntil (step f) isSingleton as)
                where step _ []      = []
                      step f (a1:a2:as) = f (a1,a2) : step f as
                      step _ [_]      = error
                                "PrimLR.lrtreeUp: length of list is no power of 2"

-- Go up odd-even tree, starting at leaves with list(2),
-- apply function(1) at the children to compute the value for
-- each node.
oetreeUp :: ((a,a)->a) -> [a] -> a
oetreeUp f as = head (repUntil ((zipPair f) . split)
                        isSingleton as)
                where split as = splitAt (div (length as) 2) as

```