Parallel Implementations of Combinations of Broadcast, Reduction and Scan

Christoph Wedler and Christian Lengauer
Fakultät für Mathematik und Informatik
Universität Passau, Germany
{wedler,lengauer}@fmi.uni-passau.de
http://www.uni-passau.de/~lengauer/

Abstract

Broadcast, Reduction and Scan are popular functional skeletons which are used in distributed algorithms to distribute and gather data. We derive new parallel implementations of combinations of Broadcast, Reduction and Scan via a tabular classification of linearly recursive functions. The trick in the derivation is to not simply combine the individual parallel implementations of Broadcast, Reduction and Scan, but to transform these combinations to skeletons with a better performance. These skeletons are also linearly recursive.

Keywords: *functional programming, linear recursion, parallelization, skeletons*

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* [3, 5]. 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).

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 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 [4, 9, 13].
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 [14], or are there *patterns* of skeleton combinations which have parallel implementations with a lower cost [7]?

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 (Section 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.

The advantage of a common skeleton is that similarities of representatives of different classes may reveal similarities in their parallel implementations.

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; a comparison with the general form of linear recursion can be found in [16]. As we will see later, this enables us to derive different classes of possible parallel implementations, depending on the way we use the list arguments.

The skeleton of parallel linear recursion (PLR) 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.

Skeleton PLR is defined as follows:

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

The data flow graph of this skeleton (Figure 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 Figure 1) and the vertical data flow, which connects recursion levels, *global* (b and y in Figure 1). In order to predict the costs of the implementations of this skeleton, we assume that the size of the global data be level-independent.

Example: a reduction from the left over a list (*foldl*), informally defined by:

$$foldl\,(\oplus)\,(as, b) = ((b \oplus as_0) \oplus \cdots) \oplus as_{n-1}$$

is formally defined by

$$\begin{array}{ll} foldl\,(\oplus)\,([], b) & = b \\ foldl\,(\oplus)\,(a : as, b) & = foldl\,(\oplus)\,(as, b \oplus a) \end{array}$$

This reduction can be defined as an instance of PLR:

$$\begin{array}{l} foldl\,(\oplus)\,(as, b) = \pi_1\,(PLR \text{ Id } pre \text{ Id } (as, b)) \\ \qquad \text{where} \qquad pre\,(a, b) = ((), b \oplus a) \end{array}$$

All of the computation is done by the recursive application of function pre, functions $base$ and $post$ are the identity. We have only distributed data on the input side (as), no intermediate local data (m always has value $()$), and no distributed data on the output side (zs is a list with elements $()$).

Note that linear recursion without distributed data 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. 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.

A linearly recursive function, which is specified as an instance of the skeleton, consists of two parts: the pre part and the post part (Figure 1). The only difference between the pre part and the post part is that the global data flows downwards in the pre part and upwards in the post part (Figure 1). We consider only the pre part; our classification applies just the same for the post part.

3.1. 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 $2^{2*2} = 16$ 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.

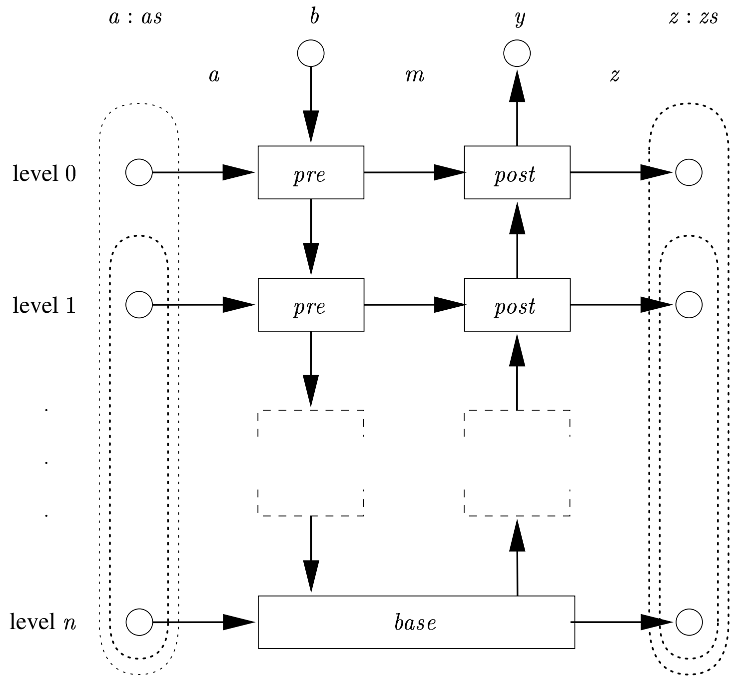
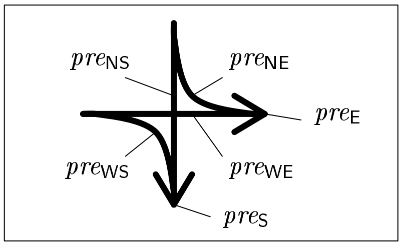


Figure 1. Data flow graph of the skeleton for parallel linear recursion

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*:

$$\begin{aligned} pre\ (a, b) = (&pre_{\mathsf{E}}\ (pre_{\mathsf{WE}}\ a, pre_{\mathsf{NE}}\ b), \\ &pre_{\mathsf{S}}\ (pre_{\mathsf{WS}}\ a, pre_{\mathsf{NS}}\ b)) \end{aligned}$$

Each of pre_{WE}, pre_{NE}, pre_{WS} and pre_{NS} is

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

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

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 $map\ pre_{\mathsf{E}}$, applied to the distributed output list; this can be parallelized trivially with an execution time of $\mathcal{O}(1)$ and a cost of $\mathcal{O}(n)$ (or no time and cost at all if $pre_{\mathsf{E}} = \mathrm{Id}$).

- Function pre_{S} is the function which determines the subclassification. Therefore, we examine this function for each class individually in Section 3.3.

3.2. Classes of the pre part

Table 1 classifies the pre part according to the choices of data dependence patterns. The horizontal legend lists the possible data dependences of the local input, the vertical legend those of the global 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 $map\ \mathrm{Id}$.

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

	—	Map	—Mismatch—	
	Sequential	Sequential ‖ Map	Reduction	Reduction ‖ Map
	—Mismatch—		Shift	Shift ‖ Map
	Broadcast	Broadcast ‖ Map	Scan	Scan ‖ Map

Table 1. Classes of the pre part

3.2.1. 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 Figure 2.

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

, **:** This case covers the upper right of Table 1; see Figure 2(a). There is a global output of pre at level i but no according input at level $i+1$. Since the global 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 global output which contradicts the data dependences for this class.

, **:** This case covers the mismatch on the left in the table; see Figure 2(b). There is a global input of pre at level i but no according global 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 Figure 2(c). These functions are simply map Id, i.e., there is nothing to do (or just the $map\ pre_{\mathsf{E}}$, mentioned above).

, **:** The functions of this class perform a shift; see Figure 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 an execution time of $\mathcal{O}(1)$.

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

A naïve implementation of a function in these classes is the sequential one with a time complexity of $\mathcal{O}(n)$. In Section 4, we present good parallel implementations for subclasses of these classes; the subclasses are determined by algebraic properties of the function pre_{S}.

3.3. Subclasses of the pre part

All functions in the 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_{S} in our skeleton PLR). The other subclasses are tailored to match simple combination patterns of basic components.

In Section 3.3.1 we list these *basic components*, in Section 3.3.2 we present an example of a combination pattern and show how this pattern leads to a new subclass, in Section 3.3.3 we list the subclasses of Reduce and Scan, in Section 3.3.4 we list the subclasses of Sequential and Broadcast. As mentioned in Section 3.2, 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.

3.3.1. 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 de-

(a) Mismatch (b) Mismatch (c) Map (d) Shift

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

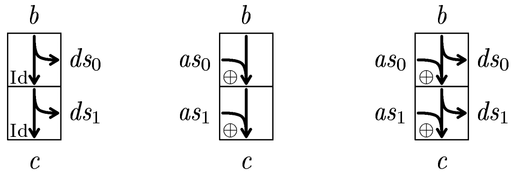
(a) Sequential (b) Broadcast (c) Reduce (d) Scan

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

fined by the following equations, where $\oplus$ is associative:

$$\begin{aligned}
\mathit{copy}_n\ b &= (b, [\overbrace{b, \dots, b}^{n}]) =: (c, \mathit{ds}) \\
\mathit{red}\ (\oplus)\ (\mathit{as}, b) &= b \oplus \mathit{as}_0 \oplus \cdots \oplus \mathit{as}_{n-1} =: c \\
\mathit{scan}\ (\oplus)\ (\mathit{as}, b) &= (\mathit{red}\ (\oplus)\ (\mathit{as}, b), \\
&\qquad [b, b \oplus \mathit{as}_0, \dots, \\
&\qquad\ b \oplus \mathit{as}_0 \oplus \cdots \oplus \mathit{as}_{n-2}]) =: (c, \mathit{ds})
\end{aligned}$$

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.



(a) copy_2 (b) $\mathit{red}\ (\oplus)$ (c) $\mathit{scan}\ (\oplus)$

Figure 4. Basic components

Like any other functional term, basic components can be composed via the following combinators:

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.

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

In order to make life easier, we introduce a (generic) function which modifies the nesting structure of its argument. We denote the regrouping operator by $\overset{()}{\underset{()}{\downdownarrows}}$. Above the operator we write the input structure, below the modified structure. Rather than providing a formal definition, we illustrate the usage of $\overset{()}{\underset{()}{\downdownarrows}}$ by two examples:

$$\overset{(2,1)}{\underset{(1,2)}{\downdownarrows}} ((a,b),c) = (a,(b,c)) \qquad \text{and}$$

$$\overset{(2,3)}{\underset{(1,2,2)}{\downdownarrows}} ((a,b),(c,d,e)) = (a,(b,c),(d,e))$$

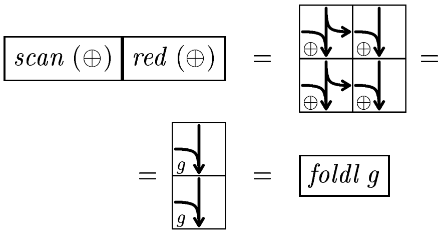
3.3.2. Example of a combination pattern: *scan* and *red*. In this section we look at a typical example of a functional composition of skeleton instances: applying scan with an associative operator $\oplus$ to a list, then applying red with the same operator to the result:

$$(\mathrm{Id} \times \mathit{red}\ (\oplus)) \circ \overset{(2,1)}{\underset{(1,2)}{\downdownarrows}} \circ (\mathit{scan}\ (\oplus) \times \mathrm{Id}) \circ \overset{(1,2)}{\underset{(2,1)}{\downdownarrows}}$$

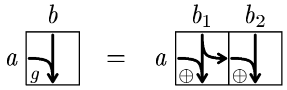
It turns out that this composition can be expressed in terms of a single reduction and an efficient parallel implementation can be found if $\oplus$ is commutative; see Section 4.

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 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:



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



where $\quad b = (b_1, b_2)$

and $\quad g\,(a, (b_1, b_2)) = (b_1 \oplus a, b_2 \oplus b_1)$

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.

3.3.3. Subclasses of Reduction and Scan. Functions in the classes Reduction and Scan have a local and a global input. Thus, the function g which determines the subclassification is exactly the binary function pre_{S}:

$$g\,(a, b) = pre_{\mathsf{S}}\,(a, b)$$

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

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

$$g\,b = pre_{\mathsf{S}}\,((), b)$$

Despite the fact that this function is not binary, associativity again plays a key rôle 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.

4. 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 global 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 (EREW-PRAM). Since the implementation primitives of Appendix A only use restricted communication patters, this assumption holds, e.g., for hypercubes.

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 [12], 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).
- $cost_{\mathsf{Brent}}(n)$: 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.

4.1. Reduction

For all functions in the subclasses of Reduction which are listed in Table 2, we use a tree-like processor network with n processors which computes the result with $time(n) = \mathcal{O}(\log n)$, $cost_{\mathsf{Brent}}(n) = \mathcal{O}(n)$ and $pipe(n) = \mathcal{O}(1)$.

The distributed input (as in Figure 1) determines the values at the leaves; at each inner node, a function computes the value from the values received from the children; the result of the reduction is extracted from the value at the root.

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 $scanl\ g$
$g\,(a,(b_1,b_2)) = (b_1 \otimes a, b_2 \oplus b_1)$	$\oplus$ and $\otimes$ are assoc. $\otimes$ distributes over $\oplus$	SR2-Reduction $foldl\ g$	SS2-Scan $scanl\ g$

Table 2. Subclasses of Reduction and Scan

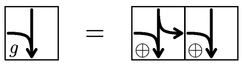
Function g	Conditions	Sequential	Broadcast
$g\ b = b$		Identity Id	Copy $copy_n$
$g\,(b_0,b_1) = (b_0, b_1 \oplus b_0)$	$\oplus$ is associative	CR-Accumulation g^n	CS-Broadcast $broadcast_n\ g$
$g\,(b_0,b_1,b_2) = (b_0, b_1 \oplus b_0, b_2 \oplus b_1)$	$\oplus$ is associative and commutative	CSR-Accumulation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

Table 3. Subclasses of Sequential and Broadcast

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

$$\begin{aligned}&foldl\ g\,(as,(b_1,b_2))\\ &\quad\text{where}\quad g\,(a,(b_1,b_2)) = (b_1 \oplus a, b_2 \oplus b_1),\\ &\qquad\qquad \oplus \text{ is associative and commutative}\end{aligned}$$

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



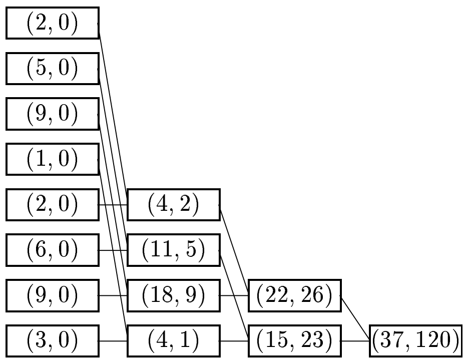
$$\begin{aligned} foldl\ g &= (\mathrm{Id} \times red\ (\oplus)) \circ \\ &\quad \overset{(2,1)}{\underset{(1,2)}{\downarrow\downarrow}} \circ (scan\ (\oplus) \times \mathrm{Id}) \circ \overset{(1,2)}{\underset{(2,1)}{\downarrow\downarrow}}\end{aligned}$$

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`:

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

An example computation of a function of this subclass is $7 * as_0 + 6 * as_1 + \cdots + 0 * as_7,\ as = [2,5,9,1,2,6,9,3]$, with result 120. Calculate

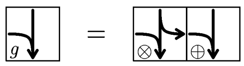
$$reduceSR\,(+)\,0\,([2,5,9,1,2,6,9,3],(0,0)) = (37,120)$$



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

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

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



$$\begin{aligned} foldl\ g &= (\mathrm{Id} \times red\ (\oplus)) \circ \\ &\quad \overset{(2,1)}{\underset{(1,2)}{\downarrow\downarrow}} \circ (scan\ (\otimes) \times \mathrm{Id}) \circ \overset{(1,2)}{\underset{(2,1)}{\downarrow\downarrow}}\end{aligned}$$

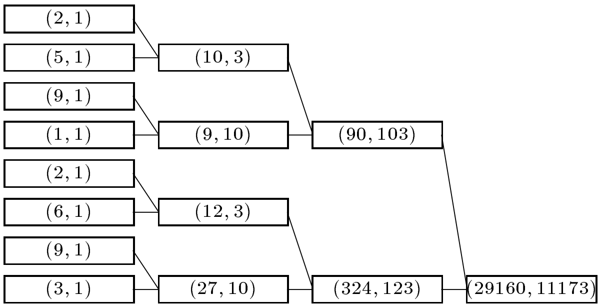
A parallel implementation is given by the higher-order function $reduceSR2$, which is instantiated with the associative operators $\oplus$ and $\otimes$ and the neutral element $1_\otimes$ of $\otimes$. The computation traverses a left-right tree (`lrtreeUp`) after the initialization of its leaves (`map`):

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

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

An example computation of a function of this subclass is $1 + as_0 * (1 + as_1 * \cdots * (1 + as_6)\ldots)$, $as = [2,5,9,1,2,6,9,3]$, with result 11173. Calculate

$$\begin{aligned}
&reduceSR2\,(+)\,(*)\,1\,([2,5,9,1,2,6,9,3],(1,0))\\
&\quad = (29160, 11173)
\end{aligned}$$



4.2. Sequential

In this class, we apply function g to the global input n times. Remember that n, the depth of the recursion, is still determined by the list of local inputs $as = [(),\overset{n\ \text{times}}{\cdots},()]$. 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 $time(n) = \mathcal{O}(\log n)$, $cost_{\mathsf{Brent}}(n) = \mathcal{O}(\log n)$ and $pipe(n) = \mathcal{O}(1)$.

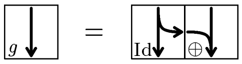
We assume n to be a power of 2. For other values, we have to use more complex computations (which are similar to the computations used in class Broadcast).

4.2.1. CR-Accumulation.

Functions in this subclass are given by:

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

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



$$g^n = (\mathrm{Id} \times red\,(\oplus)) \circ \underset{(1,2)}{\overset{(2,1)}{\downdownarrows}} \circ (copy_n \times \mathrm{Id})$$

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

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

This method is the one used for the efficient evaluation of powers [11, Sect. 4.6.3].

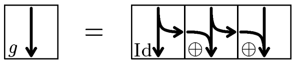
4.2.2. CSR-Accumulation.

Functions in this subclass are given by:

$$\begin{aligned}
&g^n\,(b_0,b_1,b_2)\\
&\text{where}\quad g\,(b_0,b_1,b_2) = (b_0, b_1 \oplus b_0, b_2 \oplus b_1),\\
&\qquad\quad \oplus \text{ is associative and commutative}
\end{aligned}$$

Instead of commutativity it suffices that $b = a \oplus \cdots \oplus a$ or $b = 1_\oplus$.

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



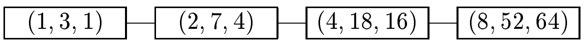
$$\begin{aligned}
g^n = \;& (\mathrm{Id} \times \mathrm{Id} \times red\,(\oplus)) \circ\\
& \underset{(1,1,2)}{\overset{(1,2,1)}{\downdownarrows}} \circ (\mathrm{Id} \times scan\,(\oplus) \times \mathrm{Id}) \circ\\
& \underset{(1,2,1)}{\overset{(2,1,1)}{\downdownarrows}} \circ (copy_n \times \mathrm{Id} \times \mathrm{Id})
\end{aligned}$$

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

$$\begin{aligned}
&accCSR\,(\oplus)\,(n,(b_0,b_1,b_2)) = (b_0, b_1 \oplus y_1, b_2 \oplus y_2)\\
&\text{where}\quad (y_1,y_2,_) = \texttt{repeat}\,node\,(\log_2 n)\,(b_0,b_1,b_0)\\
&\text{and}\quad node\,(s,t,u)\\
&\qquad = (s \oplus s, t \oplus t \oplus u, \underbrace{u \oplus u \oplus u \oplus u}_{4})
\end{aligned}$$

An example computation of a function of this subclass is $\sum_{i=0}^{7}(3+i)$ with result 52. Calculate:

$$accCSR\,(+)\,(8,(1,3,0)) = (1,8,52)$$



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

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

Instead of commutativity it suffices that $b = a \oplus \cdots \oplus a$ or $b = 1_\oplus$.

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 $\otimes$:

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

4.3. Broadcast

In this class we apply function g to the global 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 $time(n) = \mathcal{O}(\log_2 n)$, $cost_{\mathsf{Brent}}(n) = \mathcal{O}(n)$ 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 global input b is the input for the root; the list of intermediate results is composed of the values at the leaves.

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

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

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

$$(\mathrm{Id} \times map\;\pi_2) \circ broadcast_n\; g \;=\; (\mathrm{Id} \times scan\,(\oplus)) \circ \underset{(1,2)}{\overset{(2,1)}{\downarrow\downarrow}} \circ (copy_n \times \mathrm{Id})$$

A parallel implementation is given by the higher-order function $broadcCS$, 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`):

$$broadcCS\,(\oplus)\,(n,(b_0,b_1)) = \texttt{map}\; leaf\; zs'$$
$$\text{where}\quad zs' = \texttt{oetreeDn}\; node\,(\log_2 n)\,(b_0,b_1,b_0)$$
$$\text{and}\quad node\,(s,t,u) = ((s,t,u\oplus u),(s,t\oplus u,u\oplus u))$$
$$\text{and}\quad leaf\,(s,t,\_) = (s,t)$$

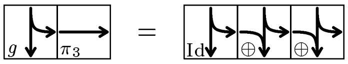
The global result is the result of a CR-Accumulation.

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

$$broadcast_n\; g\,(b_0,b_1,b_2)$$
$$\text{where}\quad g\,(b_0,b_1,b_2) = (b_0,\, b_1 \oplus b_0,\, b_2 \oplus b_1),$$
$$\oplus \text{ is associative and commutative}$$

Instead of commutativity it suffices that $b = a \oplus \cdots \oplus a$ or $b = 1_\oplus$ or $a = b \oplus \cdots \oplus b$.

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



$$\begin{aligned}(\mathrm{Id} \times \mathrm{Id} \times map\;\pi_3)&\\ \circ\, broadcast_n\; g \;&=\; (\mathrm{Id} \times \mathrm{Id} \times scan\,(\oplus)) \circ \underset{(1,1,2)}{\overset{(1,2,1)}{\downarrow\downarrow}} \circ\\ &\quad (\mathrm{Id} \times scan\,(\oplus) \times \mathrm{Id}) \circ \underset{(1,2,1)}{\overset{(2,1,1)}{\downarrow\downarrow}} \circ\\ &\quad (copy_n \times \mathrm{Id} \times \mathrm{Id})\end{aligned}$$

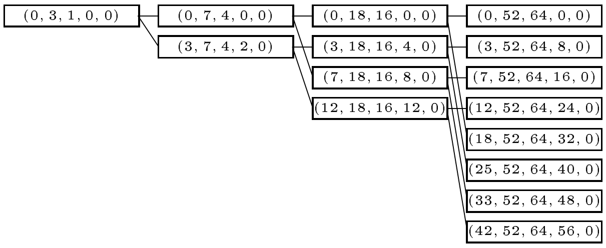
A parallel implementation is given by the higher-order function $broadcCSS$ which is instantiated by the associative and commutative operator $\oplus$ and its neutral element $1_\oplus$. The computation traverses an odd-even tree (`oetreeDn`), the result is extracted from the values of its leaves. Part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

$$broadcCSS\,(\oplus)\,1_\oplus\,(n,(b_0,b_1,b_2)) = \texttt{zip}\; join\,(old,new)$$
$$\text{where}\quad join\,((y_0,y_1),(s,\_,\_,\_,w)) = (y_0,y_1,w\oplus s)$$
$$\text{and}\quad old = broadcCS\,(\oplus)\,(n,(b_0,b_1))$$
$$\text{and}\quad new = \texttt{oetreeDn}\; node\,(\log_2 n)\,(1_\oplus,b_1,b_0,1_\oplus,b_2)$$
$$\begin{aligned}\text{and}\quad & node\,(s,t,u,v,w)\\ &= ((s,\, t\oplus t\oplus u,\, u\oplus u\oplus u\oplus u,\, v\oplus v,\, w),\\ &\quad (s\oplus t\oplus v,\, \underbrace{t\oplus t}_{2}\oplus u,\, \underbrace{u\oplus u\oplus u\oplus u}_{4},\\ &\quad \underbrace{u\oplus u}_{2}\oplus \underbrace{v\oplus v}_{2},\, w))\end{aligned}$$

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

An example computation of a function of this subclass is $[\sum_{j=0}^{i-1}(3+j) \mid i \in 0..7]$:

$$(map\;\pi_3 \circ broadcCCS)\,(+)\,0\,(8,(1,3,0))$$
$$= [0,3,7,12,18,25,33,42]$$



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

$$\begin{aligned}&broadcast_n\ g\ (b_0, b_1, b_2)\\ &\quad \text{where}\quad g\ (b_0, b_1, b_2) = (b_0, b_1 \otimes b_0, b_2 \oplus b_1),\\ &\qquad\qquad \oplus \text{ and } \otimes \text{ are associative, } \otimes \text{ distributes over } \oplus\end{aligned}$$

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 $broadcCCS2$, 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`):

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

The global result is the result of a CSR2-Accumulation.

5. 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 these 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 the case that the global data flow "downwards", i.e., to deeper levels of the recursion. The same can be done for the case that the global data flow "upwards". Combining these two cases can lead to further specialized parallel implementations.

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

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 [15]. 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 [10].

Acknowledgments

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

References

[1] R. S. Bird. Lectures on constructive functional programming. In M. Broy, editor, *Constructive Methods in Computing Science (1988, Marktoberdorf, Germany)*, volume 55 of *NATO ASI Series F*, pages 150–216. Springer-Verlag, 1989.

[2] W. Cai and D. B. Skillicorn. Calculating recurrences using the Bird-Meertens formalism. *Parallel Processing Letters*, 5(2):179–190, 1995.

[3] M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

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

[5] J. Darlington, A. Field, P. G. Harrison, et al. Parallel programming using skeleton functions. In A. Bode, M. Reeve, and G. Wolf, editors, *Parallel Architectures and Languages Europe (PARLE'93)*, volume 694 of *Lecture Notes in Computer Science*, pages 146–160. Springer-Verlag, 1993.

[6] Gesellschaft für Informatik e.V. *GI/ITG FG PARS'95*, number 14 in PARS Mitteilungen, 1995.

[7] S. Gorlatch. Stages and transformations in parallel programming. In M. Kara, J. Davy, D. Goodeve, and J. Nash, editors, *Abstract Machine Models for Parallel and Distributed Computing (AMW'96, Leeds, U. K.)*, pages 147–162. IOS Press, 1996.

[8] S. Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Second European Conf.*

on Parallel Processing (Euro-Par'96, Lyon, France), Volume 2, volume 1124 of *Lecture Notes in Computer Science*, pages 401–408. Springer-Verlag, 1996.
[9] S. Gorlatch. Systematic extraction and implementation of divide-and-conquer parallelism. In H. Kuchen and S. Doaitse Swierstra, editors, *Eighth Internat. Symposium on Programming Languages: Implementations, Logics, and Programs (PLILP'96, Aachen, Germany)*, volume 1140 of *Lecture Notes in Computer Science*, pages 274–288. Springer-Verlag, 1996.
[10] C. Herrmann and C. Lengauer. Notes on the space-time mapping of divide-and-conquer recursions. In GI [6], pages 132–139.
[11] D. E. Knuth. *The Art of Computer Programming*, volume 2: Seminumeral Algorithms. Addison-Wesley, second edition, 1980.
[12] M. J. Quinn. *Parallel Computing: Theory and Practice*. McGraw-Hill, second edition, 1994.
[13] D. B. Skillicorn. *Foundations of Parallel Programming*. Cambridge University Press, 1994.
[14] D. B. Skillicorn and W. Cai. A cost calculus for parallel functional programming. *J. of Parallel and Distributed Computing*, 28:65–83, 1995.
[15] S. Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, 1996.
[16] C. Wedler and C. Lengauer. Notes on the classification of parallel implementations of linearly recursive programs. In GI [6], pages 140–147.

A. Implementation Primitives

The next page contains the main part of the Haskell module `PrimLR` which defines the primitives used in Section 4. 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`:

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

Functions `mapList`, `mapHdTl`, and `zipPair` can be executed with $time(n) = \mathcal{O}(1)$ 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 Section 4.1.1 and Section 4.3.2. 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 Section 4.1.2.

```haskell
-- 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

-- Apply function(1) to all elements in list(2).
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.
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"

-- 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] -- ParTime = O(log n)
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 -- ParTime = O(log n)
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 -- ParTime = O(log n)
oetreeUp f as = head (repUntil ((zipPair f) . split) isSingleton as)
              where split as = splitAt (div (length as) 2) as
```