Notes on the Classification of Parallel Implementations of Linearly Recursive Programs

Christoph Wedler and Christian Lengauer
Fakultät für Mathematik und Informatik
Universität Passau, Germany

{wedler,lengauer}@fmi.uni-passau.de
http://brahms.fmi.uni-passau.de/cl/

Abstract

We propose a classification of the best parallel implementations of different kinds of functional linearly recursive programs and present examples for different classes.

Keywords: functional programming, linear recursion, parallelization, skeleton

1 Introduction

Functional programming offers a very high-level way of specifying executable problem solutions. For example, the paradigm of linear recursion can be expressed very concisely as a functional "skeleton" [Col89] which we call the *source skeleton*. However, looking at a specific linear recursion at this level of abstraction, it is far from clear how it can be implemented efficiently on a given processor network. In this paper, we propose a method of selecting an appropriate parallel implementation.

We proceed as follows. We transform the source specification such that it matches a different skeleton, the *derived skeleton*, which exposes the possible parallelism more clearly. In general, the programmer will have to propose the transformation and check with a simple proof that it preserves the meaning of the linearly recursive function. The derived skeleton can be specialized in a number of ways; each specialization has a different "best" parallel implementation (where the notion of "goodness" also depends on the measure according to which we want to optimize). We provide a classification of these special cases, discuss their individual implementations (many of which have appeared before in the literature) and provide examples.

2 The *source skeleton* LR_{src}

We start with the the following program skeleton LR_{src} for linear recursion, given in a syntax similar to Haskell [HF92], whose meaning should be self-evident:

$$LR_{\mathsf{src}} : (In \to \mathbb{B}) \to (In \to Out) \to (In \to In) \to ((In, Out) \to Out) \to In \to Out$$

$$LR_{\mathsf{src}}\ P\ base\ pre\ post = f\ \textsf{where}$$
$$\quad f\ x = \textsf{if}\ P\ x\ \textsf{then}\ base\ x\ \textsf{else}\ post\,(x, f\,(pre\ x))$$

E.g., the function $map\ g$ with $map\ g\ [x_1, \ldots, x_n] = [g\,x_1, \ldots, g\,x_n]$ can be defined using LR_{src} as follows:

$$map : (\alpha \to \beta) \to [\alpha] \to [\beta]$$
$$map\ g = LR_{\mathsf{src}}\ null\ (\boldsymbol{\lambda}\_.\,[])\ tail\ (\langle::\rangle \circ (g \circ head \times \mathrm{Id}))$$

where $(\boldsymbol{\lambda}\_.\,[])$ returns the empty list for any argument and $\langle::\rangle$ is the list constructor.

3 The *derived skeleton* LR_{drv}

Unfortunately it is very difficult to reason about the parallelism in the implementation of a linearly recursive function if this function is given as an instance of the source skeleton.

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 to data sets of a common type on each processor. We take this type to be the list.

In this section we propose a second skeleton, the "derived" skeleton, which makes the use of lists as the input and output explicit. As we will see later, this allows us to derive different classes of possible parallel implementations, depending on the way we use the list arguments.

The derived skeleton is defined as follows:

$$
\begin{aligned}
&LR_{\mathsf{drv}} : T_{base} \to T_{pre} \to T_{calc} \to T_{post} \to ([In_{\mathsf{l}}], In_{\mathsf{g}}) \to ([Out_{\mathsf{l}}], Out_{\mathsf{g}}) \ \textbf{where} \\
&\quad T_{base} = ([In_{\mathsf{l}}], In_{\mathsf{g}}) \to ([Out_{\mathsf{l}}], Out_{\mathsf{g}}) \\
&\quad T_{pre} \ = (In_{\mathsf{l}}, In_{\mathsf{g}}) \to (In_{\mathsf{c}}, In_{\mathsf{g}}) \\
&\quad T_{calc} = In_{\mathsf{c}} \to Out_{\mathsf{c}} \\
&\quad T_{post} = (Out_{\mathsf{c}}, Out_{\mathsf{g}}) \to (Out_{\mathsf{l}}, Out_{\mathsf{g}})
\end{aligned}
$$

$$
\begin{aligned}
&LR_{\mathsf{drv}}\ base\ pre\ calc\ post = f\ \textbf{where} \\
&\quad f\ ([], in_{\mathsf{g}}) = base\ ([], in_{\mathsf{g}}) \\
&\quad f\ (in_{\mathsf{l}} :: left_{[\mathsf{l}]}, in_{\mathsf{g}}) = \textbf{let}\ (left_{\mathsf{c}}, left_{\mathsf{g}}) = pre\ (in_{\mathsf{l}}, in_{\mathsf{g}}) \\
&\qquad\qquad right_{\mathsf{c}} = calc\ left_{\mathsf{c}} \\
&\qquad\qquad (right_{[\mathsf{l}]}, right_{\mathsf{g}}) = f\ (left_{[\mathsf{l}]}, left_{\mathsf{g}}) \\
&\qquad\qquad (out_{\mathsf{l}}, out_{\mathsf{g}}) = post\ (right_{\mathsf{c}}, right_{\mathsf{g}}) \\
&\qquad\quad \textbf{in}\ (out_{\mathsf{l}} :: right_{[\mathsf{l}]}, out_{\mathsf{g}})
\end{aligned}
$$

Rather than explaining the functions and variables of this skeleton in detail, let us convey the idea by its data flow graph in Figure 1. This graph 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* and the vertical data flow, which connects recursion levels, *global*. In the interest of a realistic cost model, we propose that the size of the global data is level-independent.

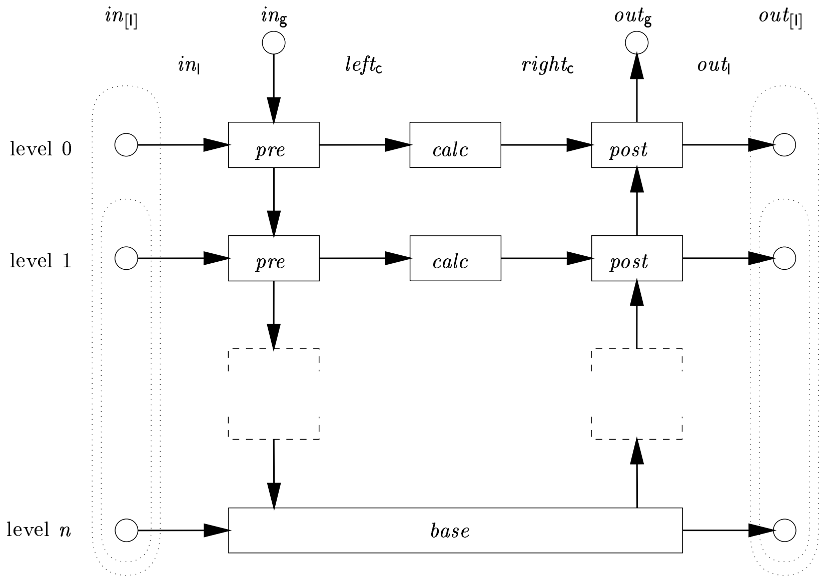


Figure 1: Data flow graph of the derived skeleton

The function *map g*, introduced in the previous section, can be expressed with this skeleton as follows:

$$map\ g\ list = \text{fst}\ (LR_{\mathsf{drv}}\ (\boldsymbol{\lambda}\_ .\,([], ()))\ \text{Id}\ g\ \text{Id}\ (list, ()))$$

where () is the only element of the unit type **1**.

4 Equivalences

In this section we consider the relationship between the source skeleton and the derived skeleton.

After we have defined a linearly recursive function by specifying the customizing functions in the source skeleton LR_{src}, we would like to transform this instance f_1 into a functionally equivalent instance f_2 of the derived skeleton LR_{drv}.

For equivalence there must be two isomorphisms ϕ_{IN} and ϕ_{OUT} which make the following diagram commute:

$$\begin{tikzcd} In_1 \arrow[r, "f_1"] & Out_1 \\ In_2 \arrow[u, "\phi_{\text{IN}}"] \arrow[r, "f_2"] & Out_2 \arrow[u, "\phi_{\text{OUT}}"'] \end{tikzcd} \qquad \begin{array}{rcl} f_1 & = & LR_{\mathsf{src}}\ P_1\ base_1\ pre_1\ post_1 \\ f_2 & = & LR_{\mathsf{drv}}\ base_2\ \langle pre_{\mathsf{l}}, pre_{\mathsf{g}}\rangle\ calc_2\ post_2 \end{array}$$

For our example *map* g we have $In_1 = [\alpha]$, $In_2 = ([\alpha], \mathbf{1})$, $Out_1 = [\beta]$, $Out_2 = ([\beta], \mathbf{1})$ and $\phi_{\text{IN}} = \phi_{\text{OUT}} = \text{fst}$.

Here are 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:

$$\begin{array}{rcll} P_1 \circ \phi_{\text{IN}} & = & null \circ \text{fst} & (\text{EQPRED}) \\ base_1 \circ \phi_{\text{IN}} & = & \phi_{\text{OUT}} \circ base_2 & (\text{EQBASE}) \\ pre_1 \circ \phi_{\text{IN}} & = & \phi_{\text{IN}} \circ \langle tail \circ \text{fst}, pre_{\mathsf{g}} \circ in\rangle & (\text{EQPRE}) \\ post_1 \circ (\phi_{\text{IN}} \times \phi_{\text{OUT}}) & = & \phi_{\text{OUT}} \circ out \circ (post_2 \times \text{Id}) \circ mid \circ & (\text{EQPOST}) \\ & & (calc \circ pre_{\mathsf{l}} \times \text{Id}) \circ (in \times \text{Id}) & \end{array}$$

where

$$\begin{array}{rcl} in(in_{[\mathsf{l}]}, in_{\mathsf{g}}) & = & (head\ in_{[\mathsf{l}]}, in_{\mathsf{g}}) \\ mid(val, (out_{[\mathsf{l}]}, out_{\mathsf{g}})) & = & ((val, out_{\mathsf{g}}), out_{[\mathsf{l}]}) \\ out((out_{\mathsf{l}}, out_{\mathsf{g}}), out_{[\mathsf{l}]}) & = & (out_{\mathsf{l}} :: out_{[\mathsf{l}]}, out_{\mathsf{g}}) \end{array}$$

We omit the proofs of the sufficiency of these conditions due to lack of space.

5 Classes of linearly recursive functions

Different linearly recursive functions have different optimal parallel implementations. Our aim is to find classes of derived programs whose optimal parallel implementations have a common structure. In this section we characterize these classes and discuss how a program is assigned to an appropriate class.

A derived program consists basically of three parts: the *pre* part, the *calc* part and the *post* part (Figure 1).

The *calc* part is always the same: *map calc*. *map* can be parallelized trivially with an execution time of $\mathcal{O}(1)$ and a cost of $\mathcal{O}(n)$.

The only difference between the *pre* 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.

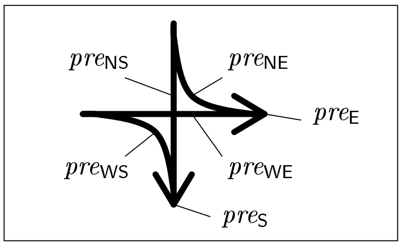
5.1 Classifying the *pre* part

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.

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*:

$$pre\ (in_{\mathsf{l}}, in_{\mathsf{g}}) = (pre_{\mathsf{E}}\ (pre_{\mathsf{WE}}\ in_{\mathsf{l}}, pre_{\mathsf{NE}}\ in_{\mathsf{g}}), pre_{\mathsf{S}}\ (pre_{\mathsf{WS}}\ in_{\mathsf{l}}, pre_{\mathsf{NS}}\ in_{\mathsf{g}}))$$

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

- either an identity function $\mathrm{Id}_{\neq\mathbf{1}}$ on some non-unit type $\alpha \neq \mathbf{1}$ or
- a function $\lozenge$ from some type to the unit type $\mathbf{1}$.

We could always choose $\mathrm{Id}_{\neq\mathbf{1}}$ if the type of the input is not the unit type, but we prefer to choose $\lozenge$ whenever possible, such that function $\mathrm{Id}_{\neq\mathbf{1}}$ 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, since its output is used only by the function *calc*. Therefore, we can rewrite both functions pre_{E} and *calc*:
$$(pre'_{\mathsf{E}}, calc') = (\mathrm{Id}, calc \circ pre_{\mathsf{E}})$$
In the classification of the *post* part, we extend the implementation by an additional *map* $post_{\mathsf{E}}$, applied to the distributed output list, if $post_{\mathsf{E}}$ is not the identity function.

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

5.2 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 superimposition of the icons of both legends.

	—	*map*	—type error—	
	Sequential	Sequential $\|\|$ *map*	Reduce	Reduce $\|\|$ *map*
	—type error—		(Shift)	(Shift $\|\|$ *map*)
	Broadcast	Broadcast $\|\|$ *map*	Scan	Scan $\|\|$ *map*

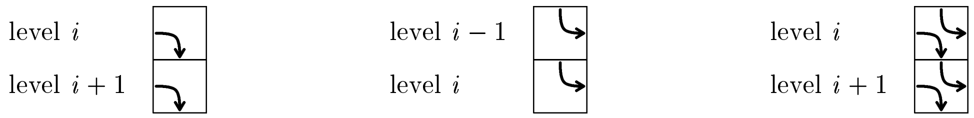
Table 1: Classes of the *pre* part

The only difference between the first and the second row and, similarly, between the third and the fourth row, is the additional data dependence from the local input to the local output in the second row, expressed as an additional independent "computation" of *map* Id.

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

5.2.1 Type error, *map* and Shift

First we look at the entries representing, type errors, *map* and Shift. Their data flows between two successive levels are depicted again in Figure 2.



(a) type error (b) type error (c) Shift

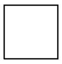
Figure 2: Specific data flows (type errors and shift)

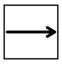
Type errors. We start with an informal explanation of the two type errors; they signify mismatches of the global data flow:

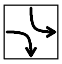
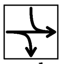
, **:** This case covers the upper left of Table 1; see Figure 2(a). There is a global output of *pre* at level i but no according input at level $i+1$. Since the global output is only used by *pre* of the next level (and the base) case, there is no "real" data dependence to the global output which contradicts the data dependences for this class.

, **:** This case covers the type error 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:

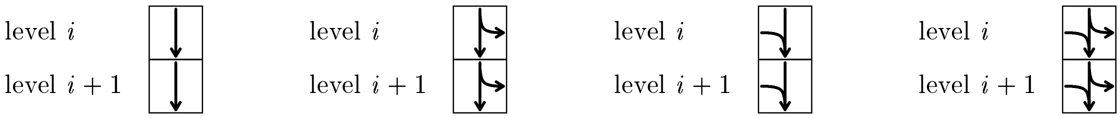
 **:** This case covers the upper left of the Table 1. Functions of this class return no result.

 **:** This is the second entry of the first row. These functions are simply *map* Id, i.e., there is nothing to do.

,  **:** The functions of this class perform a shift; see Figure 2(c). 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 $\mathcal{O}(1)$.

5.2.2 Sequential, Broadcast, Reduce and Scan

The remaining classes are the most interesting ones. Their data flows between two successive levels are depicted again in Figure 3.



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

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

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

6 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 at each processor. For our implementations, one operation is performed on each processor per computation. This fact gives us a choice of two possibilities:

1. we can either stick to this implementation,
2. or we can aggregate several operations on one processor—following Brent's Theorem [Qui92] without an asymptotic penalty in time but with 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.

6.1 Reduce and Scan

For the classes Reduce and Scan, the following results are well-known provided function pre_{S} is associative:

- A reduction can be performed with $time(n) = \mathcal{O}(\log n)$, $cost_{\mathsf{Brent}}(n) = \mathcal{O}(n)$ and $pipe(n) = \mathcal{O}(1)$ [Qui92].
- A two-phase scan (up-sweep and down-sweep) can be performed with $time(n) = \mathcal{O}(\log n)$, $cost_{\mathsf{Brent}}(n) = \mathcal{O}(n)$ and $pipe(n) = \mathcal{O}(1)$ if there is the capability to store $\mathcal{O}(n)$ elements in the communication channel, the sender or receiver; otherwise we have $time(n) = \mathcal{O}(\log n)$, $cost_{\mathsf{Brent}}(n) = \mathcal{O}(n)$ and $pipe(n) = \mathcal{O}(\log n)$ or $time(n) = \mathcal{O}(\log n)$, $cost_{\mathsf{Brent}}(n) = \mathcal{O}(n \log n)$ and $pipe(n) = \mathcal{O}(1)$ [O'D94].
- A single-phase scan can be performed with $time(n) = \mathcal{O}(\log n)$, $cost_{\mathsf{Brent}}(n) = \mathcal{O}(n \log n)$ and $pipe(n) = \mathcal{O}(1)$. This implementation is appropriate for butterfly architectures [CM91].

6.2 Sequential and Broadcast

Neither class Sequential nor class Broadcast has local input. Thus, the function which determines the subclassification is:

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

While we have assumed the associativity of pre_{S} in the classes Reduce and Scan to ensure a time complexity of $\log n$ for its implementations, there is no similar property for a unary function like g. But we would like to obtain a logarithmic time complexity for implementations of subclasses of Sequential and Scan. Table 2 lists different functions g with their corresponding subclasses of Sequential and Broadcast which ensure a time complexity of $\log n$. Remember that k-ary functions can be viewed as unary functions with a k-tuple as the argument (since g is of type $In_{\mathsf{g}} \to In_{\mathsf{g}}$, the result is also a k-tuple).

Classes:	Sequential	Broadcast
$g(a) = a$	Identity: time & cost: $\mathcal{O}(1)$	Copy
$g(a, b) = (a, b \oplus a)$	Simple Accumulation	Simple Accumulating Broadcast
$g(a, b, c) = (a, b \oplus a, c \oplus b)$	Commutative Accumulation	Commutative Accumulating Broadcast
$g(a, b, c) = (a, b \odot a, c \oplus b)$	Distributive Accumulation	Distributive Accumulating Broadcast
$time(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
$cost_{\mathsf{Brent}}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
$pipe(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$

Conditions: $\oplus$ and $\odot$ are associative, $\odot$ distributes over $\oplus$, for 'Commutative ... ': $\oplus$ is commutative

Table 2: Subclasses of Sequential and Broadcast

In the following we propose implementations for the subclasses shown in Table 2. Remember that n, the depth of the recursion, is still determined by the list of local inputs $in_{[\mathsf{l}]} = [(), \overset{n \text{ times}}{\dots}, ()]$.

6.2.1 Sequential

In this class we apply function g n times to the global input. For all the subclasses shown in Table 2 we use a processor network with $\log n$ processors in a row 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 the class Broadcast).

Identity. If g is the identity function, there is no computation and nothing to consider.

Simple Accumulation. If $g(a, b) = (a, b \oplus a)$ and $\oplus$ is associative then

$$g^n(a, b) = (a, b \oplus double^{\log n}(a)) \qquad \text{where} \qquad double(s) = s \oplus s$$

This method is the one used for the efficient evaluation of powers, which is quite ancient, see Knuth [Knu80, Sect. 4.6.3] for details.

Commutative Accumulation. If $g(a, b, c) = (a, b \oplus a, c \oplus b)$, $\oplus$ is associative and $\oplus$ is commutative (instead of commutativity it suffices that $b = a \oplus \cdots \oplus a$ or $b = 0_{\oplus}$ or $a = b \oplus \cdots \oplus b$), then

$$g^n(a, b, c) = (a, b \oplus double^{\log n}(a), c \oplus (\mathrm{fst} \circ asqr^{\log n})(b, a)) \quad \text{where} \quad asqr(s, t) = (s \oplus s \oplus t, \underbrace{t \oplus t \oplus t \oplus t}_{4})$$

A computation for a function of this subclass is $\sum_{i=0}^{7}(3 + i)$ with result 52. Calculate: $g^n(1, 3, 0) = (1, 8, 52)$ with $\oplus = +$ and $n = 8$.

$\log n, n$	$0, 1$	$1, 2$	$2, 4$	$3, 8$
$double$	(1)	(2)	(4)	(8)
$asqr$	$(3, 1)$	$(7, 4)$	$(18, 16)$	$(52, 64)$

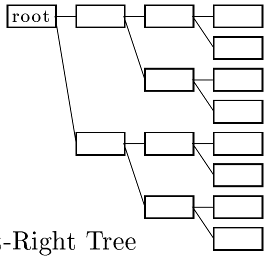
Distributive Accumulation. If $g(a,b,c) = (a, b \odot a, c \oplus b)$, $\oplus$ is associative, $\odot$ is associative and $\odot$ distributes over $\oplus$, then

$$g^n(a,b,c) = (a, b \odot (\mathrm{snd} \circ \mathit{mult}^{\log n})(b,a), c \oplus (\mathrm{fst} \circ \mathit{mult}^{\log n})(b,a)) \quad \text{where} \quad \mathit{mult}(s,t) = (s \oplus (s \odot t), t \odot t)$$

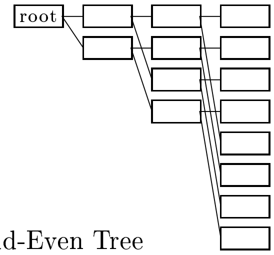
A computation for a function of this subclass is $\sum_{i=0}^{7} 2 * 3^i$ with the result of 6560. Calculate: $g^n(3,2,0) = (3, 13122, 6560)$ with $\oplus = +$, $\odot = *$ and $n = 8$.

6.2.2 Broadcast

In this class we apply the function g n times to the global input and return a list of intermediate results. For all the subclasses shown in Table 2 we use a tree-like processor network with n processors which computes the result with $\mathit{time}(n) = \mathcal{O}(\log n)$, $\mathit{cost}_{\mathsf{Brent}}(n) = \mathcal{O}(n)$ and $\mathit{pipe}(n) = \mathcal{O}(1)$. We have two different kinds of trees:



(a) Left-Right Tree



(b) Odd-Even Tree

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 in_{g} is the input for the root, the list of intermediate results are the value at the leaves.

Copy. If we copy the same value to n places ($g(a) = a$) then

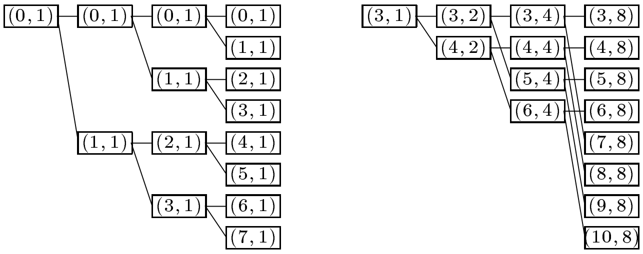
$$[g^i(a) \mid i \in 0..n{-}1] = \mathit{mapLeftRightTree}\ \mathrm{Id}_{\mathrm{LR}}\ a = \mathit{mapOddEvenTree}\ \mathrm{Id}_{\mathrm{OE}}\ a$$
$$\text{where} \qquad \mathrm{Id}_{\mathrm{LR}}(s) = \mathrm{Id}_{\mathrm{OE}}(s) = (s,s)$$

Simple Accumulating Broadcast. If $g(a,b) = (a, b \oplus a)$ and $\oplus$ is associative then

$$\begin{aligned}[\mathrm{snd} \circ g^i(a) \mid i \in 0..n{-}1] &= \mathit{map}\ ((b\oplus) \circ \mathrm{fst})\ (\mathit{mapLeftRightTree}\ \mathit{double}_{\mathrm{LR}}\ (0,a)) \\ &= \mathit{map}\ \mathrm{fst}\ (\mathit{mapOddEvenTree}\ \mathit{double}_{\mathrm{OE}}\ (b,a))\end{aligned}$$
$$\text{where} \qquad \mathit{double}_{\mathrm{LR}}(s,t) = ((s \oplus s, t), (s \oplus s \oplus t, t)) \quad \text{and} \quad \mathit{double}_{\mathrm{OE}}(s,t) = ((s, t \oplus t), (s \oplus t, t \oplus t))$$

The method of distributing the results of $\mathrm{fst} \circ g^i$ is the same as in the subclass Copy.

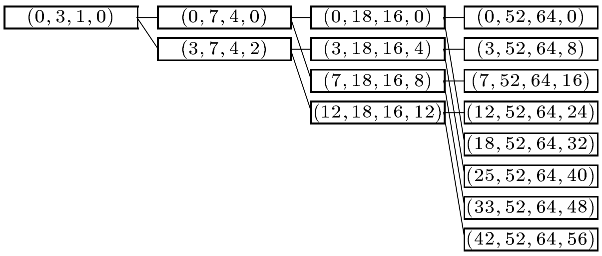
A computation for a function of this subclass is $[3 + i \mid i \in 0..7]$ with the result of $[3, 4, \ldots 10]$. We have $a = 1$, $b = 3$, $\oplus = +$.



Commutative Accumulating Broadcast. If $g(a,b,c) = (a, b \oplus a, c \oplus b)$, $\oplus$ is associative and $\oplus$ is commutative (instead of commutativity it suffices that $b = a \oplus \cdots \oplus a$ or $b = 0_{\oplus}$ or $a = b \oplus \cdots \oplus b$), then

$$[\mathrm{Proj}_3^3 \circ g^i(a) \mid i \in 0..n{-}1] = \mathit{map}\ ((c\oplus) \circ \mathrm{fst})\ (\mathit{mapOddEvenTree}\ \mathit{asqr}_{\mathrm{OE}}\ (0,b,a,0))$$
$$\begin{aligned}\text{where} \qquad \mathit{asqr}_{\mathrm{OE}}(s,t,u,v) = (&(s, t \oplus t \oplus u, u \oplus u \oplus u \oplus u, v \oplus v), \\ &(s \oplus t \oplus v, \underbrace{t \oplus t}_{2} \oplus u, \underbrace{u \oplus u \oplus u \oplus u}_{4}, \underbrace{u \oplus u}_{2} \oplus \underbrace{v \oplus v}_{2}))\end{aligned}$$

A computation for a function of this subclass is $[\sum_{j=0}^{i-1}(3 + j) \mid i \in 0..7]$ with the result of $[0, 3, 7, 12, 18, 25, 33, 42]$. We have $a = 1$, $b = 3$, $c = 0$ $\oplus = +$.



Distributive Accumulating Broadcast. If $g(a, b, c) = (a, b \odot a, c \oplus b)$, $\oplus$ is associative, $\odot$ is associative and $\odot$ distributes over $\oplus$, then

$$[\mathrm{Proj}_3^3 \circ g^i(a) \mid i \in 0..n{-}1] = \mathit{map}\ ((c\oplus) \circ \mathrm{fst})\ (\mathit{mapOddEvenTree}\ \mathit{mult}_{\mathrm{OE}}\ (0, b, a))$$
$$\text{where} \qquad \mathit{mult}_{\mathrm{OE}}(s, t, u) = ((s, t \oplus (t \odot u), u \odot u), (t \oplus (s \odot u), t \oplus (t \odot u), u \odot u))$$

7 Future Work

Currently we are looking at the interaction between the *pre* part and the *post* part. The question is: can we do better than just parallelizing the two parts separately? We have already found an interesting result: if the *pre* part is in class Simple Accumulation Broadcast and the *post* part is a reduction, then the implementation of both parts together is in class Commutative Accumulation (Sequential).

The next step concerns the more general problem: given two functions with their implementations and their costs, what is the optimal implementation for the functional composition of both functions [SC95]?

Further, we plan to look at data partitioning [Pep93] and at nested linear recursion, i.e., at the case where *pre* and/or *post*, which are customizing functions of our skeleton, are linearly recursive.

Acknowledgments

This work is partially supported by an ARC exchange grant from the DAAD. Thanks to D. K. Arvind, M. I. Cole, J. T. O'Donnell, S. Gorlatch and C. Herrmann for helpful discussions.

References

[CM91] B. Carpentieri and G. Mou. Compile-time transformations and optimization of parallel divide-and-conquer algorithms. *ACM SIGPLAN Notices*, 26(10):19–28, 1991.

[Col89] M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

[HF92] P. Hudak and J. H. Fasel. A gentle introduction to Haskell. *ACM SIGPLAN Notices*, 27(5):T1–T53, May 1992.

[Knu80] D. E. Knuth. *The Art of Computer Programming*, volume 2: Seminumeral Algorithms. Addison-Wesley, second edition, 1980.

[O'D94] J. T. O'Donnell. A correctness proof of parallel scan. *Parallel Processing Letters*, 4(3):329–338, 1994.

[Pep93] P. Pepper. Deductive derivation of parallel programs. In R. Paige, J. Reif, and R. Wachter, editors, *Parallel Algorithm Derivation and Program Transformation*, chapter 1, pages 1–53. Kluwer Academic Publishers, 1993.

[Qui92] M. J. Quinn. *Parallel Computing: Theory and Practice*. McGraw-Hill, second edition, 1992.

[SC95] D. B. Skillicorn and W. Cai. A cost calculus for parallel functional programming. *Journal of Parallel and Distributed Computing*, 28:65–83, 1995.