Architecture Independent Massive Parallelization of Divide-and-Conquer Algorithms

Klaus Achatz and Wolfram Schulte

Fakultät für Informatik, Universität Ulm
E-mail: {`achatz,wolfram`}`@informatik.uni-ulm.de`

Note. This technical report is an extended version of the paper with the same title in the proceedings of *Mathematics of Program Construction, 1995*, Bad Irrsee, which will be published in the Springer series *Lecture Notes in Computer Science.*

Abstract. We present a strategy to develop, in a functional setting, correct, efficient and portable Divide-and-Conquer (DC) programs for massively parallel architectures. Starting from an operational DC program, mapping sequences to sequences, we apply a set of semantics preserving transformation rules, which transform the parallel control structure of DC into a sequential control flow, thereby making the implicit data parallelism in a DC scheme explicit. In the next phase of our strategy, the parallel architecture is fully expressed, where 'architecture dependent' higher-order functions are introduced. Then – due to the rising communication complexities on particular architectures – topology dependent communication patterns are optimized in order to reduce the overall communication costs. The advantages of this approach are manifold and are demonstrated with a set of non-trivial examples.

1 Introduction

It is well-known that the main problems in exploiting the power of modern parallel systems are the development of correct, efficient and portable programs [Pep93, Fox89]. The most promising way to treat these problems in common seems to be a systematic, formal, top-down development of parallel software.

In this paper we choose *transformational programming* to develop parallel programs where transformational programming summarizes a methodology for constructing correct and efficient programs from formal specifications by applying meaning-preserving rules [Par90]. Starting with a functional specification, we derive programs for the *massively data parallel model*, which assumes a large data collection that needs to be processed and that there is a single processor element (PE) for each member in the collection. The same set of instructions is concurrently applied to all data elements, i.e., there is a single control flow which guides the computation on all PEs.

The main characteristics of our strategy, using transformational programming to develop data parallel software, are the following ones: as a *problem* *adequate structure* we restrict ourselves to *sequences*, which are fully satisfactory in the vast majority of situations. The usual data parallel operations, like *apply-to-all* or *reduce*, are provided. In addition, certain high level operations are introduced, which can be interpreted as communication operations on the machine level (cf. Sect. 2).

As the starting point of our strategy, we choose a very popular tactic for designing parallel algorithms: *Divide-and-Conquer* (DC). Batcher's bitonic sort is a well-known example. DC algorithms are particularly suited for parallel implementation because the sub-problems can be solved independently and thus in parallel. Obviously DC algorithms have explicit control parallelism, i.e., there are separate independent parts that can be processed simultaneously by distinct CPUs. However, our model of computation does not allow several control flows. Therefore we aim at exploiting the inherent data parallelism. Hence, we present a set of semantic preserving *transformation rules*, which make the implicit data parallelism in a DC scheme on sequences explicit, thereby introducing architecture independent communication operations on sequences (cf. Sect. 3).

The architecture is fully expressed in the next step of our strategy, where *skeletons* are introduced. Skeletons are higher-order functions to express data parallel operations on specific architectures. The aforementioned sequence operations each have a straightforward implementation in terms of skeletons. In particular it turns out that even the communication oriented sequence operations can be implemented on arrays, meshes and hypercubes equally well. Due to the rising communication complexity on particular architectures, *topology dependent optimizations* become more and more important. We calculate two architecture dependent optimizations (for arrays and meshes) using only the skeleton definitions, where correspondent communications followed by broadcasts can be realized using less communication operations (cf. Sect. 4)

However, aside from answering theoretical questions concerning the correctness of our approach, we want to stress the advantages of our work from a practical and methodological point of view:

- The identification of a transformation rule to exploit the implicit data parallelism of DC and its necessary applicability condition makes the transformation process target directed.
- The developed DC algorithms are efficient and can be ported across several architectures. If, in addition, topology dependent optimizations are applied very efficient algorithms can be derived.
- The presented transformations can be automated using an extended compilation approach, where the user may give hints in the form of laws to the compiler [Fea87].
- Architecture independent data parallelism is distinguished from architecture dependent one. Correspondingly we operate on different levels of abstraction (sequences vs. skeletons) and supply different transformation rules (data parallelization vs. communication transformation).

These aspects are demonstrated with three examples: the parallel prefix computation, Batcher's bitonic sort, and computing the convex hull of a set of points in the plane.

The rest of this paper is organized as follows. Section 2 briefly presents our sequence model, and its relation to the massively data parallel model.

The new DC transformation rules are introduced in Sect. 3. Section 4 defines skeletons, their use and optimizations. We follow in Sect. 5 with two examples, demonstrating the applicability of our approach. Section 6 compares our approach with others. Finally, Sect. 7 draws conclusions and raises issues for further research.

Notation. In notation we follow the standard of lazy functional programming languages, like Haskell or Miranda. For example, we write function application in curried form, as in $f\ x\ y$ which is equivalent to $(f\ x)\ y$, and define functions – whenever possible – using pattern matching. If, in addition, assertions on parameters are used, they are given in the surrounding text.

Addendum. The differences of this technical report wrt. [AS95] are marked as being *addenda* (like this one). Additionally proofs and an implementation of the running example in a real parallel language are presented in the appendices. $\triangleleft$

2 The Balanced Sequence Model

Sequences in general can be used to express data parallelism in an abstract way, where parallelism is achieved exclusively through operations on sequences [Ble92]. In this section we explore this approach, present the traditional operations on sequences and its data parallel view (Sect. 2.1), introduce communication oriented operations (Sect. 2.2), and define some properties (Sect. 2.3) that will be of value in the following exposition.

2.1 Basic Sequence Operations

Our so called *balanced sequence model* is motivated by the underlying parallel program development strategy, viz. divide-and-conquer (see Sect. 3), and by the need to perform the same computation on all data elements of the sequence in parallel. The term "balanced sequence" stems from the fact that our DC scheme always results in balanced computation trees.

The constructors of our balanced sequence model are the following ones: $[]$ is the empty sequence, $[e]$ is the sequence which contains the single element e, and $x +\!\!+\ y$ is the sequence formed by concatenating sequences x and y, but only if both have *equal* length. This always results in sequences of lengths powers of 2, which is appropriate, since all known massively parallel machines work with 2^n PEs.

Addendum. An alternative constructor set replaces concatenation, also called *left-right composition* by the shuffle operator $\bowtie$, also named *odd-even composition.* In a shuffled sequence $x \bowtie y$ the elements with even indices come from x

and the odd ones from y. We will later pick up this constructor set and show that – in a DC scheme – it can be transformed into the former. ◁

The following auxiliary functions are used to specify programs. They will be removed during program development: the operator (#) returns the length of a sequence. The first-order functions *first* and *last* extract the first or last element from a nonempty sequence, respectively. The function *copy* creates a sequence of n copies of identical elements.

It is perfectly well to assume every sequence element corresponds to a data element resting on a particular processor element. Two sequences can be seen as two different storage levels on the parallel machine.

We now start to introduce the set of balanced sequence functions, most of them are commonly used functions [BW88, AJ93]:

– *map*. Applies a function to every element of a sequence independently, and therefore reflects the massively data parallel programming paradigm in the most obvious way.
– *zipWith*/*zipWith*3. Takes a pair/triple of sequences, having equal length, into a new sequence in which corresponding elements are combined using any given binary/ternary operator. The family of *zipWith* functions correspond to the *map* functional working on two or more storage levels.
– *reduce*. Reduces a nonempty sequence using any binary operator. This function can be implemented on a parallel machine in logarithmic time using a 'tree' [Ski93].

In a data parallel environment conditionals are somewhat different to their sequential counterparts. The action of a parallel **if** can be summarized this way: on every PE the condition is evaluated; in components where the condition is true, the *then*-branch is executed, otherwise the *else*-branch.

A specialization of a *parallel conditional* is the operation *join*. It takes a pair of sequences x, y, having equal length, into a new sequence, which consists of alternate slices of x and y each of length n, $n > 0$ (see Fig. 1(a)).
We can define *join* by:

$$\begin{aligned} &\mathit{join}\ n\ (x1 +\!\!+\ x2)\ (y1 +\!\!+\ y2) = x1 +\!\!+\ y2, && \textbf{if}\ n = \#x1 \\ &\mathit{join}\ n\ (x1 +\!\!+\ x2)\ (y1 +\!\!+\ y2) = \mathit{join}\ n\ x1\ y1 +\!\!+ && \\ &\qquad\qquad\qquad\qquad\qquad\qquad \mathit{join}\ n\ x2\ y2, && \textbf{if}\ n < \#x1 \end{aligned} \tag{1}$$

Like the functions defined in the next subsection, *join* is a partial operation. Since these functions are introduced during program development, definedness of the resulting programs must be guaranteed by the appropriate transformation rules (cf. Sect. 3).

2.2 Communication Oriented Sequence Operations

A very wide range of scientific problems can be computed under the DC scheme using a regular communication pattern. Naturally, some communication patterns are better than others for developing parallel algorithms. Essentially, they

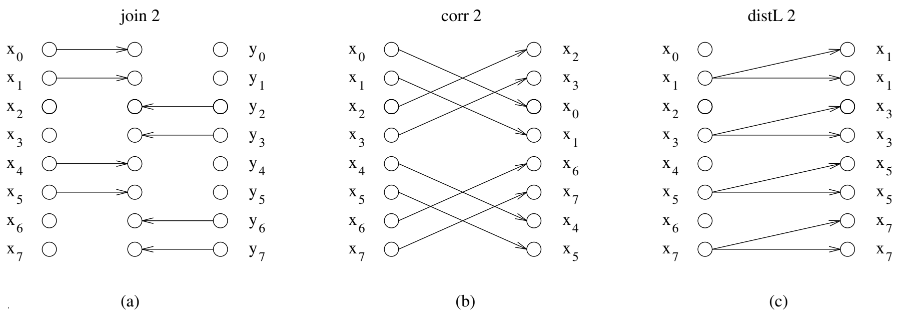

Fig. 1. Sequence operations: (a) *join* 2 x y, (b) *corr* 2 x, (c) *distL* 2 x

have structural properties that make it easier to describe the data movement operations necessary for parallel computations. In the case of our particular DC scheme (see Sect. 3), the following communication operations seem to be the most suitable ones:

Correspondent communication – modeled by function *corr* $n\,x$ – exhibits a butterfly-like communication pattern: for a particular value of n, each PE communicates with each PE whose index differs in the nth bit from the left. An example is depicted in Fig. 1(b). Its definition is straightforward:

$$\begin{array}{ll} \mathit{corr}\ n\ (x +\!\!+\ y) = (y +\!\!+\ x), & \textbf{if } n = \#x \\ \mathit{corr}\ n\ (x +\!\!+\ y) = \mathit{corr}\ n\ x +\!\!+\ \mathit{corr}\ n\ y, & \textbf{if } n < \#x \end{array} \qquad (2)$$

First or last communication can be realized using a correspondent communication followed by a *directed broadcast*. A directed broadcast operates from right to left, where the value of the rightmost element is distributed to the left (*distL*), e.g., *distL* $n\,x$ copies the value of the last element of each slice of length n to its left neighbors (see Fig. 1(c)). The function *distR* operates from left to right. Directed broadcast is related to *copy* by the following definition:

$$\begin{array}{lll} \mathit{distL}\ n\ x & = \mathit{copy}\ n\ (\mathit{last}\ x), & \textbf{if } n = \#x \\ \mathit{distL}\ n\ (x +\!\!+\ y) & = \mathit{distL}\ n\ x +\!\!+\ \mathit{distL}\ n\ y, & \textbf{if } n \leq \#x \end{array} \qquad (3)$$

The introduced sequence operations *corr*, *distL*/*distR* and *join*, mirror the necessity of our DC scheme to exchange data between PEs and to select different data elements on each PE, respectively.

2.3 Properties: Distributivity and Length Preservation

Our balanced sequence model fulfills a number of properties, where especially the following two are needed in our transformation rules given below (cf. Sect. 3).

Let f denote a function, which maps sequences to sequences. The function is said to be *distributive*, if it distributes through concatenation of sequences:

$$f\ (x \mathbin{+\!\!+} y) = f\ x \mathbin{+\!\!+} f\ y$$

It is said to be *length preserving*, if the length of the output sequence is equal to the length of the input sequence:

$$\#(f\ x) = \#x$$

The generalization to functions taking a tuple of sequences yielding a single sequence is straightforward.

Another generalization concerns the distributivity of functions like *corr* or *distL*, which work on slices of length n. This time, let $f\ n$ denote a function, which maps sequences to sequences. If it distributes through a sequence $x \mathbin{+\!\!+} y$, where $n \leq \#x$, then the function is said to be *distributive modulo* n, or – more general spoken – *slice-distributive*.

All (slice-)distributive functions that either map the empty sequence to the empty sequence, or are undefined for empty sequences, are uniquely defined by specifying their effect on 'elementary' sequences (having length n).

It can be shown that functions *map* and *zipWith* are distributive, *corr*, *distL* and *join* are slice-distributive, and *map*, *zipWith*, *corr*, *distL* and *join* are length preserving.

3 Divide and Conquer

First, the idea and assumption of our DC tactic is discussed (Sect. 3.1) followed by its formal account (Sect. 3.2) that aims at transforming the parallel control-structure of DC into a sequential control flow with a parallel data-structure.

3.1 The DC Scheme

DC is a well-known tactic for designing parallel algorithms. It consists of three steps:

1. If the input is not primitive, partition the input.
2. Solve recursively the subproblems, defined by each partition of the input.
3. Compose the solutions of the different subproblems into a solution for the overall problem.

A general DC tactic can be defined as the following higher-order function:

```
$DC\ q\ t\ g\ h\ k\ j = f$
**where** $f\ x \qquad\;\; = t\ x,$                                **if** $q = \#x$
      $f(x \mathbin{+\!\!+} y) = (k\ v\ w) \mathbin{+\!\!+} (j\ v\ w),$              **otherwise**
                  **where** $(v, w) = (f(g\ x\ y), f(h\ x\ y))$
```

In DC, when the input has length q, the problem is solved trivially by t, otherwise the input is split (by pattern matching), the subinputs are preadjusted by g and h, solved in a recursive manner, postadjusted by k and j and then concatenated. Thus the decompose and compose operations consist of two steps: $(g, h) \circ +\!\!+^{-1}$ and $+\!\!+ \circ (k, j)$, respectively. This leads to a computation, where the control flow, expressed by the sequence primitives, is separated from the computation, expressed by the adjust functions. In addition, it is assumed that the trivial, the pre- and the postadjust functions are length preserving.

This DC scheme is perfectly appropriate for data parallelization, since the sequence primitives are independent of the elements in the sequence and hence can be performed in constant time.

The power of this scheme stems from the fact that the pre- and postadjust functions receive the complete input and output sequence, respectively. However, since the adjust functions must be length preserving only "balanced" algorithms can be derived.

These assumptions rule out certain important non-balanced algorithms, as for instance Quicksort. But algorithms that either are not balanced or depend on values are not suitable for massively data parallel computation. They require – in contrast to our adjust functions – irregular communication patterns to get things in the right place, which normally causes high communication costs. Therefore such algorithms are not considered relevant for our current study.

Addendum. Obviously one can choose the alternative constructor set using shuffling instead of concatenation, too. All facts and assumptions that hold for concatenation also hold for shuffling. $\lhd$

3.2 The Rules

The presented DC scheme exhibits cascading recursion and explicit data decomposition. In order to transform this scheme into a corresponding data parallel program, we have to introduce a sequential control flow, i.e., we must transform the cascading recursion into linear, or – even better – tail recursion, and we have to make the explicit data decomposition implicit.

First, we concentrate on simplifying the recursion. The computation proceeds in two phases: in a decompose or 'top-down' phase the preadjust functions g and h are applied to the subsequences, whereas in the compose or 'bottom-up' phase the postadjust functions k and j are applied. For a sequential flow of control, we have to decouple the phases of f, i.e., we introduce two functions one for the top-down computation $f{\downarrow}$ and one for the bottom-up computation $f{\uparrow}$.

Theorem 1 (Generalized divide-and-conquer rule). *Assume g, h, j, k, $t{\uparrow}$, $t{\downarrow}$ and t are length preserving functions and $t = t{\uparrow} \circ t{\downarrow}$. Let f be a general DC algorithm of the form:*

$$\begin{array}{lll} f\ x & = t\ x, & \textbf{if } q = \#x \\ f\ (x +\!\!+ y) & = (k\ v\ w) +\!\!+ (j\ v\ w), & \textbf{otherwise} \\ & \quad \textbf{where } (v, w) = (f\ (g\ x\ y), f\ (h\ x\ y)) & \end{array}$$

Then, f can be decomposed into an equivalent function $f{\uparrow} \circ f{\downarrow}$:

$$\begin{array}{lll}
f\ x = f{\uparrow}\ (f{\downarrow}\ x) & & \\
\textbf{where}\ f{\downarrow}\ x & = t{\downarrow}\ x, & \textbf{if}\ q = \#x \\
\quad f{\downarrow}(x +\!\!+\ y) & = f{\downarrow}(g\ x\ y) +\!\!+\ f{\downarrow}(h\ x\ y), & \textbf{otherwise} \\
\quad f{\uparrow}\ x & = t{\uparrow}\ x, & \textbf{if}\ q = \#x \\
\quad f{\uparrow}(x +\!\!+\ y) & = (k\ v\ w) +\!\!+\ (j\ v\ w), & \textbf{otherwise} \\
 & \quad \textbf{where}\ (v, w) = (f{\uparrow}\ x, f{\uparrow}\ y) &
\end{array}$$

Proof. See Appendix A.1. □

The resulting functions $f{\uparrow}$ and $f{\downarrow}$ still have cascading recursion. But now pre- and postadjust functions are decoupled. Additionally, we know the number of iterations 'beforehand', since the recursive computation only uses split and concatenation on balanced sequences.

We rewrite the functions $f{\downarrow}$ and $f{\uparrow}$ to include an additional parameter, which determines the recursion depth. Thus, it is not necessary anymore to use the sequence to determine the recursion depth – its length becomes constant. On the other hand, the trivial, pre- and postadjust functions have to be performed on the appropriate slices. This is possible, if they are length preserving. Then it is easy to define their slice-distributive generalizations, which work on the whole sequence and not only on the subsequences as in the case of cascading recursion. In order to supply the appropriate slices to the pre- and postadjust functions, we must explicitly introduce correspondent communication followed by a join of the different solutions of the subproblems.

The following two transformation rules enable us to derive tail-recursive and therefore data parallel versions of $f{\downarrow}$ and $f{\uparrow}$.

Theorem 2 (Top-down with pre-adjustment). *Assume functions g, h and t are length preserving. Let $f{\downarrow}$ be a cascading top-down algorithm of the form:*

$$\begin{array}{lll}
f{\downarrow}\ x & = t\ x, & \textbf{if}\ q = \#x \\
f{\downarrow}(x +\!\!+\ y) & = f{\downarrow}\ (g\ x\ y) +\!\!+\ f{\downarrow}(h\ x\ y), & \textbf{otherwise}
\end{array}$$

Then, $f{\downarrow}$ is transformed into an equivalent function $f{\Downarrow}$, which is a tail-recursive top-down computation with pre-adjustment. As an assertion on the parameters of $f{\Downarrow}$ we require $\#x \geq n$:

$$\begin{array}{l}
f{\downarrow}\ x = f{\Downarrow}\ (\#x)\ x \\
\textbf{where} \\
\quad \begin{array}{ll}
f{\Downarrow}\ n\ x = t'\ q\ x, & \textbf{if}\ q = n \\
f{\Downarrow}\ n\ x = f{\Downarrow}\ \frac{n}{2}\ (join\ \frac{n}{2}\ (g'\ \frac{n}{2}\ x\ x')\ (h'\ \frac{n}{2}\ x'\ x)), & \textbf{otherwise}
\end{array} \\
\quad \textbf{where} \\
\quad\quad \begin{array}{lll}
x' & = corr\ \frac{n}{2}\ x & \\
t'\ n\ x & = t\ x, & \textbf{if}\ n = \#x \\
t'\ n\ (x +\!\!+\ y) & = (t'\ n\ x) +\!\!+\ (t'\ n\ y), & \textbf{if}\ n \leq \#x \\
g'\ n\ x\ y & = g\ x\ y, & \textbf{if}\ n = \#x \\
g'\ n\ (x1 +\!\!+\ x2)(y1 +\!\!+\ y2) & = (g'\ n\ x1\ y1) +\!\!+\ (g'\ n\ x2\ y2), & \textbf{if}\ n \leq \#x1 \\
h'\ n\ x\ y & = h\ x\ y, & \textbf{if}\ n = \#x \\
h'\ n\ (x1 +\!\!+\ x2)(y1 +\!\!+\ y2) & = (h'\ n\ x1\ y1) +\!\!+\ (h'\ n\ x2\ y2), & \textbf{if}\ n \leq \#x1
\end{array}
\end{array}$$

Proof. See Appendix A.2. □

Theorem 3 (Bottom-up with post-adjustment). *Assume functions k, j and t are length preserving. Let $f{\uparrow}$ be a cascading bottom-up algorithm of the form:*

$$\begin{array}{lll} f{\uparrow}\ x & = t\ x, & \textbf{if } q = \#x \\ f{\uparrow}\ (x +\!\!+\ y) & = (k\ v\ w) +\!\!+ (j\ v\ w), & \textbf{otherwise} \\ & \quad \textbf{where } (v, w) = (f{\uparrow}\ x, f{\uparrow}\ y) & \end{array}$$

Then $f{\uparrow}$ is transformed into an equivalent function $f{\Uparrow}$, which is a tail-recursive bottom-up computation with post-adjustment. As an assertion on the parameters of $f{\Uparrow}$ we require $\#x \geq n$:

$$\begin{array}{l} f{\uparrow}\ x = f{\Uparrow}\ (\#x)\ q\ (t'\ q\ x) \\ \textbf{where} \\ \quad \begin{array}{ll} f{\Uparrow}\ m\ n\ x = \ x, & \textbf{if } n = m \\ f{\Uparrow}\ m\ n\ x = f{\Uparrow}\ m\ (2n)\ (join\ n\ (k'\ n\ x\ x')\ (j'\ n\ x'\ x)), & \textbf{otherwise} \end{array} \\ \quad \textbf{where} \\ \qquad x' = corr\ n\ x \\ \\ \qquad t' \text{ as defined in Theorem 2} \\ \qquad k', j' \text{ are renamings of } g' \text{ and } h' \text{ of Theorem 2} \end{array}$$

Proof. Analogous to A.2. □

Example Parallel prefix. One of the simplest and most useful building blocks for parallel algorithms is the *parallelprefix* function [Ble93, Bir89], which takes a binary operator $\oplus$, a sequence of 2^i elements

$$[e_1, e_2, \ldots, e_{2^i}]$$

and returns

$$[e_1, (e_1 \oplus e_2), \ldots, (e_1 \oplus e_2 \ldots \oplus e_{2^i})]$$

If $\oplus$ denotes addition, then a possible initial specification for this function (shortly coined *psum*) is:

$$\begin{array}{lll} psum_0\ x & = x, & \textbf{if } \#x = 1 \\ psum_0\ (x +\!\!+\ y) & = v +\!\!+ (map\ ((+)\ (last\ v))\ w), & \textbf{otherwise} \\ & \quad \textbf{where } (v, w) = (psum_0\ x, psum_0\ y) & \end{array}$$

This specification immediately leads to a DC computation, which can be done in $\mathcal{O}(\log\ n)$ time on n PEs – ignoring the communication costs – since each addition can be computed in one timestep, and the depth of the computation is $\mathcal{O}(\log\ n)$.

Applying our strategy, first, we derive a data parallel version for *psum*. Obviously, $psum_0$ matches the input pattern of the bottom-up computation rule. An appropriate instantiation is:

$$
\begin{aligned}
&t\ x \quad = x\\
&k\ x\ y = x\\
&j\ x\ y = \mathit{map}\ ((+)(\mathit{last}\ x))\ y
\end{aligned}
$$

We immediately obtain an iterative data parallel version of $psum_0$. The new functions t', k' and j', however, are still recursive. Although they can be implemented using DC too, it is much better to circumvent the recursion. Therefore, we carry out some precomputations to determine their closed forms:

Derivation. Let $n = (\#x1)$ and $x = x1 +\!\!+\ x2$ and $x' = x2 +\!\!+\ x1$:

$$
\begin{aligned}
&k'\ n\ x\ x'\\
&\quad = [\text{ def. of } x \text{ and } x', \text{ slice-distrib. of } k', \text{ unfold } k']\\
&\qquad (k\ x1\ x2) +\!\!+\ (k\ x2\ x1)\\
&\quad = [\text{ unfold } k]\\
&\qquad x1 +\!\!+\ x2\\
&\quad = [\text{ assumption: } x = x1 +\!\!+\ x2\ ]\\
&\qquad x
\end{aligned}
$$

$$
\begin{aligned}
&j'\ n\ x\ x'\\
&\quad = [\text{ def. } x \text{ and } x', \text{ slice-distrib. } j', \text{ unfold } j'\ ]\\
&\qquad (j\ x1\ x2) +\!\!+\ (j\ x2\ x1)\\
&\quad = [\text{ unfold } j]\\
&\qquad (\mathit{map}\ ((+)\ (\mathit{last}\ x1))\ x2) +\!\!+\ (\mathit{map}\ ((+)\ (\mathit{last}\ x2))\ x1)\\
&\quad = [\text{ property of } \mathit{map} \text{ wrt. } \mathit{zipWith}\ ]\\
&\qquad (\mathit{zipWith}\ (+)\ (\mathit{copy}\ n\ (\mathit{last}\ x1))\ x2) +\!\!+\\
&\qquad (\mathit{zipWith}\ (+)\ (\mathit{copy}\ n\ (\mathit{last}\ x2))\ x1)\\
&\quad = [\text{ fold } \mathit{distL}\ ]\\
&\qquad \mathit{zipWith}\ (+)\ (\mathit{distL}\ n\ x1)\ x2 +\!\!+\ \mathit{zipWith}\ (+)\ (\mathit{distL}\ n\ x2)\ x1\\
&\quad = [\text{ distrib. of } \mathit{zipWith}\ ]\\
&\qquad \mathit{zipWith}\ (+)\ ((\mathit{distL}\ n\ x1) +\!\!+\ (\mathit{distL}\ n\ x2))\ (x2 +\!\!+\ x1)\\
&\quad = [\text{ slice-distrib. of } \mathit{distL}, \text{ assumption on } x \text{ and } x']\\
&\qquad \mathit{zipWith}\ (+)\ (\mathit{distL}\ n\ x)\ x'
\end{aligned}
$$

Due to the slice-distributivity of k' and j', definitions of k' and j' hold for all $n \leq \#x1$. In a similar way, t' can be shown to be equivalent to the identity function.

By means of these definitions, we apply Theorem 3 to $psum_0$ and result in:

```
$psum_0\ x = psum_1\ \#x\ 1\ x$
where
    $psum_1\ m\ n\ x = x,$    if $n = m$
    $psum_1\ m\ n\ (x1 +\!\!+\ x2) = p,$    otherwise
    where  $x' = corr\ n\ x$
           $p\ = psum_1\ m\ (2n)\ (join\ n\ x\ (zipWith\ (+)\ (distL\ n\ x')\ x))$
```

In the following section, we will pick up $psum_1$, and will systematically derive architecture specific array, mesh and hypercube algorithms, respectively. □

Addendum. On closer inspection of the different constructor sets and their use in the DC scheme, we can observe that – under certain conditions – a top-down computation based on split and concatenation is equivalent to a bottom up computation based on unshuffle and shuffle, where the post-adjust function of the latter is the pre-adjust function of the former. However this only holds, if on termination of DC the input has length 1 ($q = 1$) and is then trivially solved by the identity function ($t = id$). This fact was already observed by [CM91].

Theorem 4 (Odd-even vs. left-right). *Let $f\!\downarrow$ be a top-down algorithm with pre-adjustment of the form:*

$$\begin{array}{lll} f\!\downarrow x & = x, & \textbf{if } \#x = 1 \\ f\!\downarrow (x \bowtie y) & = f\!\downarrow (g\ x\ y) \bowtie f\!\downarrow (h\ x\ y), & \textbf{otherwise} \end{array}$$

Then $f\!\downarrow$ is transformed into an equivalent function $f\!\uparrow$, which is a bottom-up computation with post-adjustment.

$$\begin{array}{lll} f\!\uparrow x & = x, & \textbf{if } \#x = 1 \\ f\!\uparrow (x +\!\!+\ y) & = (g\ v\ w) +\!\!+\ (h\ v\ w), & \textbf{otherwise} \\ & \quad \textbf{where } (v, w) = (f\!\uparrow x, f\!\uparrow y) & \end{array}$$

Proof. By computational induction. □

Theorem 4 holds even if the computation ordering is changed that is, if the roles of the pre-and postadjust functions are inverted.

This result justifies our approach, to present the former rules for only one constructor set – whether it is the one which is based on concatenation, the one we have chosen, or the other one, does not really matter. ◁

4 Skeletons and Skeleton Transformations

In this section, the basis for the derivation of architecture specific programs is given, i.e., topology independent skeletons are introduced (Sect. 4.1), followed by topology dependent ones (Sect. 4.2), then the derived sequence skeletons are calculated (Sect. 4.3), and finally communication transformations are presented (Sect. 4.4).

4.1 Basic Skeletons

The skeleton idea is fairly simple. The *data components on all processors are modeled as a data field* [YC92], i.e., as a function over some index domain D, which describes the PE's indices, into some codomain V of problem related values. Then, *data parallel operations can be defined as higher-order functions* (called *skeletons*), which are either abstractions of

– elementary communication-independent computations on all PEs or
– communication operations, which pass values along the network connections.

For instance, the most typical elementary operation on data parallel architectures is a single function operating on multiple PEs. This computation is expressed by the *MAP* skeleton:

$$MAP\ f\ \ a = \lambda\, i.f\,(a\ \ i) \tag{4}$$

The higher-order function *MAP* takes an operator f and a data field a, and returns a data field in which each element is the result of operation f applied to the corresponding element of a.

The skeleton *ZIPWITH* generalizes the *MAP* skeleton in the sense that *ZIPWITH* takes a pair of data fields a and b, and combines them using a dyadic operator $\oplus$.

$$ZIPWITH\ \oplus\ \ a\ \ b = \lambda\, i.(a\ \ i) \oplus (b\ \ i) \tag{5}$$

The introduced skeletons can be applied to every data parallel architecture, because no data exchange between two processors takes place. All data parallel architectures share these *topology independent skeletons*.

Individual types of architectures differ in their topology and thus, in their possible patterns of communication. Communication patterns for linear arrays, meshes and hypercubes will be given in the next subsection.

4.2 Communication Skeletons

This section formally defines three important static processor organizations: linear arrays, meshes and hypercubes.

Linear arrays. Linear arrays have a very simple interconnection network. Every PE is linked to its left and right neighbor, if they exist. An abstraction of a linear array with N PEs, where N in general is a power of 2, will be written as a parameterized type:

$array(\alpha) = index \rightarrow \alpha$
where $index = \{\ i \mid 0 \leq i < N\ \}$

Arrays can have wrap-around connections (then called rings), i.e., PE 0 is connected to PE $N-1$. Here, we only consider arrays without wrap-around connections.[*]

We identify two basic data parallel exchange operations: shifting all elements one position to the left or to the right. The next two skeletons allow communication of k steps at a time, although only one step at a time is an elementary computation on these architectures:

$$\begin{array}{lll} \mathit{SHL}_A\ k\ a = \lambda\, i.\ & a(N-1), & \textbf{if } i \geq N-k \\ & a(i+k), & \textbf{otherwise} \\ \mathit{SHR}_A\ k\ a = \lambda\, i.\ & a(0), & \textbf{if } i < k \\ & a(i-k), & \textbf{otherwise} \end{array}$$

Note. The above communication skeletons are modeled in such a way that PEs, which do not receive a valid data element, yield the appropriate value of a boundary PE. Other patterns could be chosen too.

Meshes. In a mesh network, the nodes are arranged in a q-dimensional lattice. Communication is allowed only between neighboring nodes. Two-dimensional meshes, for instance, have $N \times N$ identical PEs, which are positioned according to an $N \times N$ matrix. Each PE $P(i,j)$ is connected to its neighbor PEs $P(i+1,j), P(i-1,j), P(i,j+1)$, and $P(i,j-1)$, if they exist. The abstraction of two-dimensional meshes reads:

$$\begin{array}{l} \mathit{mesh}(\alpha) = \mathit{index} \to \alpha \\ \textbf{where } \mathit{index} = \{\ (i,j) \mid 0 \leq i,j < N\ \} \end{array}$$

Meshes also can have wrap-around connections, where each column and each row of the mesh is connected like a ring. Again, we only consider meshes without wrap-around connections.

According to these interconnections, we distinguish four different exchange operations: data is sent to its left(*SHL*), to its right (*SHR*) to its upper (*SHU*) or lower neighbors (*SHD*). The skeletons have the form:

$$\begin{array}{lll} \mathit{SHL}_M\ k\ m = \lambda(i,j).\ & m(i,N-1), & \textbf{if } j \geq N-k \\ & m(i,j+k), & \textbf{otherwise} \\ \mathit{SHR}_M\ k\ m = \lambda(i,j).\ & m(i,0), & \textbf{if } j < k \\ & m(i,j-k), & \textbf{otherwise} \\ \mathit{SHU}_M\ k\ m = \lambda(i,j).\ & m(N-1,j), & \textbf{if } i \geq N-k \\ & m(i+k,j), & \textbf{otherwise} \\ \mathit{SHD}_M\ k\ m = \lambda(i,j).\ & m(0,j), & \textbf{if } i < k \\ & m(i-k,j), & \textbf{otherwise} \end{array} \tag{6}$$

[*] Wrap-around connections do not add further functionality to the system, but make communication patterns more efficiently implementable.

Hypercubes. In an n-dimensional hypercube, which has 2^n nodes, each PE has n neighbors, which it can reach in one time step. Its abstraction looks like the one for arrays, i.e., we have:

$$\begin{array}{l} hyper(\alpha) = index \to \alpha \\ \textbf{where } index = \{\ i \mid 0 \le i < 2^n\ \} \end{array}$$

A PE in an n-dimensional hypercube can communicate with n of its neighbors, where nodes are adjacent to each other when their indices differ in exactly one bit position. This bit can be set on or off – correspondingly, we can communicate 'up' or 'down'. Once again we generalize this communication, by specifying communication in dimension d, which has to be a power of 2:

$$\begin{array}{rll} COMMU\ \ d\ h = \lambda\, i.\ h(i-d), & \textbf{if } i \ge (i \operatorname{div} (2d)) \cdot 2d + d \\ h(i), & \textbf{otherwise} \\ COMMD\ \ d\ h = \lambda\, i.\ h(i+d), & \textbf{if } i < (i \operatorname{div} (2d)) \cdot 2d + d \\ h(i), & \textbf{otherwise} \end{array} \tag{7}$$

Note. The integer parameter for shifting elements on the array or mesh describes *the number of elementary communication steps*, whereas the first parameter of *COMMU* and *COMMD* specifies the dimension in which a communication takes place – thus the elementary hypercube communication is performed in *a single step.*

4.3 Derived Skeletons

Now that on the one side, we have derived data parallel functions on sequences, and on the other have specified architecture specific skeletons, it remains to close the gap, i.e., to implement the sequence primitives in terms of skeletons.

We state without proof the correspondence of *map* with *MAP* and *zipWith* with *ZIPWITH*. This can easily be seen, if we recognize that each operation (by means of *map* or *MAP* and *zipWith* or *ZIPWITH*, respectively) is applied *independently* to each data element. Therefore, it makes no difference whether the data component is an element of a sequence or an element of a data field. The communication oriented sequence operations, however, have to be defined in the context of the architecture the algorithm is aimed at.

Arrays. Sequences of length N and linear arrays defined as data fields have a one-to-one correspondence:

$$g : \begin{cases} [\alpha] \to array(\alpha) \\ \quad x \mapsto \lambda\, i.x_i \end{cases}$$

where x_i is the selection of the ith element of the sequence. The inverse of g is:

$$g^{-1} : \begin{cases} array(\alpha) \to [\alpha] \\ \qquad\quad x \mapsto [x(0), \ldots, x(N-1)] \end{cases}$$

We derive the skeleton functions, operating on a linear array from the communication oriented sequence operations. We start with the following definition:

$$\begin{aligned} g(\mathit{join}\ n\ x\ y) &= \mathit{JOIN}_A\ n\ (g\ x)\ (g\ y) \\ g(\mathit{corr}\ n\ x) &= \mathit{CORR}_A\ n\ (g\ x) \\ g(\mathit{distL}\ n\ x) &= \mathit{DISTL}_A\ n\ (g\ x) \\ g(\mathit{distR}\ n\ x) &= \mathit{DISTR}_A\ n\ (g\ x) \end{aligned} \tag{8}$$

After eliminating the bijection g, we get the following direct definitions:

Corollary 5.

$$\begin{aligned} \mathit{JOIN}_A\ n\ a\ b &= \lambda\, i.\, a\ i, \textbf{ if } \mathit{even}(i \operatorname{div} n) \\ &\qquad\quad\ b\ i, \textbf{ otherwise} \\ \mathit{CORR}_A\ n\ a &= \mathit{JOIN}_A\ n\ (\mathit{SHL}_A\ n\ a)\ (\mathit{SHR}_A\ n\ a) \\ \mathit{DISTR}_A\ n\ a &= \lambda\ i.\ a(l \cdot n) \quad \textbf{where} \quad l = i \operatorname{div} n \\ \mathit{DISTL}_A\ n\ a &= \lambda\ i.\ a((l+1) \cdot n - 1) \quad \textbf{where} \quad l = i \operatorname{div} n \end{aligned} \tag{9}$$

Proof. See Appendix A.3. □

In order to obtain an array specific program, we replace the sequence operations by operations on data fields.

Example Parallel prefix cont'd. Unfolding the skeleton operations for arrays in psum_1, results in the following architecture specific psum_2 program:

$$\begin{aligned} &\mathit{psum}_0\ x = \mathit{psum}_2\ \#x\ 1\ x \\ &\textbf{where} \\ &\qquad \mathit{psum}_2\ m\ n\ x = x, \qquad\qquad\qquad\qquad\qquad \textbf{if } m = n \\ &\qquad \mathit{psum}_2\ m\ n\ x = \mathit{psum}_2\ m\ (2n)\ (\mathit{JOIN}_A\ n\ x\ x'), \textbf{otherwise} \\ &\qquad \textbf{where } x' = (\mathit{ZIPWITH}(+)\ (\mathit{DISTL}_A\ n\ (\mathit{CORR}_A\ n\ x))\ x) \end{aligned}$$

Note that the resulting program suffers from a lot of redundant communication operations. Due to our architecture independent transformation rules 2 and 3, we always introduce a correspondent communication. But in the particular case of the above example, we only have to distribute data in one direction, which leads to many superfluous shifts. Below, we will present communication transformations to remove redundant communication operations. □

Index Translations. In order to define the derived skeletons for meshes and hypercubes, we could proceed as already done for arrays. However, having defined arrays as data fields, it is much simpler to map only the index domain of the array to the hypercube or mesh domain instead of mapping the whole data structure.

Let D and E be two index domains. A bijective mapping $g : D \to E$, with inverse $g^{-1} : E \to D$ is called an *index translation.*

In fact, the application of an index translation results in a change of the underlying coordinate system, given by the source index domain D.

Meshes. Linear arrays of length N^2 are mapped onto a mesh with N columns and N rows, using the following index translation:

$$g : \begin{cases} \{0,\ldots,N^2-1\} \to \{0,\ldots,N-1\} \times \{0,\ldots,N-1\} \\ \qquad\qquad k \mapsto (k \operatorname{div} N, k \bmod N) \end{cases}$$

where it is assumed that the indices are in row-major-order. The inverse mapping reads:

$$g^{-1} : \begin{cases} \{0,\ldots,N-1\} \times \{0,\ldots,N-1\} \to \{0,\ldots,N^2-1\} \\ \qquad\qquad (i,j) \mapsto i \cdot N + j \end{cases}$$

The mesh oriented skeletons $\mathit{JOIN}_M, \mathit{CORR}_M, \mathit{DISTR}_M$ and DISTL_M can be derived starting from the corresponding array skeletons, this time using index translations:

$$\begin{aligned} \mathit{JOIN}_M\ n\ x\ y &= (\mathit{JOIN}_A\ n(x \circ g)(y \circ g)) \circ g^{-1} \\ \mathit{CORR}_M\ n\ x &= (\mathit{CORR}_A\ n(x \circ g)) \circ g^{-1} \\ \mathit{DISTL}_M\ n\ x &= (\mathit{DISTL}_A\ n(x \circ g)) \circ g^{-1} \\ \mathit{DISTR}_M\ n\ x &= (\mathit{DISTR}_A\ n(x \circ g)) \circ g^{-1} \end{aligned} \tag{10}$$

Eliminating the index mapping, we obtain the following direct definitions:

Corollary 6.

$$\begin{aligned} \mathit{JOIN}_M\ n\ x\ y &= \lambda(i,j).\, x(i,j),\ \textbf{if}\ \ \mathit{even}((i \cdot N + j) \operatorname{div} n) \\ &\qquad\qquad\ \ y(i,j),\ \textbf{otherwise} \\ \mathit{CORR}_M\ n\ x &= \lambda(i,j).\mathit{JOIN}_M\ n\ x1\ x2 \\ &\quad\ \textbf{where}\ \ x1 = \mathit{SHL}_M\ (n \bmod N)\ (\mathit{SHU}_M\ (n \operatorname{div} N)\ x) \\ &\qquad\qquad\quad x2 = \mathit{SHR}_M\ (n \bmod N)\ (\mathit{SHD}_M\ (n \operatorname{div} N)\ x) \\ \mathit{DISTL}_M\ n\ x &= \lambda(i,j).x(((l+1)n-1) \operatorname{div} N, ((l+1)n-1) \bmod N) \\ &\quad\ \textbf{where}\ l = (i \cdot N + j) \operatorname{div} N \\ \mathit{DISTR}_M\ n\ x &= \lambda(i,j).x((l \cdot n) \operatorname{div} N, (l \cdot n) \bmod N) \\ &\quad\ \textbf{where}\ l = (i \cdot N + j) \operatorname{div} n \end{aligned}$$

Proof. See Appendix A.4. □

Hypercubes. Derived skeletons for the hypercube architecture are defined by choosing the identity function as an index translation ($g = id$). From (10) by replacing the subscript M with H, we obtain:

Corollary 7.

$$\begin{aligned} \mathit{JOIN}_H\ n\ x\ y &= \lambda\ \ i\ .x\ i,\ \textbf{if}\ \ \mathit{even}(i \operatorname{div} n) \\ &\qquad\quad\ y\ i,\ \textbf{otherwise} \\ \mathit{CORR}_H\ n\ x &= \lambda\ \ i\ .\mathit{JOIN}_H\ n\ (\mathit{COMMD}_H\ n\ x)\ (\mathit{COMMU}_H\ n\ x) \\ \mathit{DISTL}_H\ n\ x &= \lambda\ \ i\ .x((l+1) \cdot n - 1)\quad \textbf{where}\ \ l = i \operatorname{div} n \\ \mathit{DISTR}_H\ n\ x &= \lambda\ \ i\ .x(l \cdot n)\quad \textbf{where}\ \ l = i \operatorname{div} n \end{aligned}$$

Proof. See Appendix A.5. □

4.4 Communication Transformations for Array and Mesh

The result of our derivation leads to communication patterns, which probably are not the most efficient ones on a particular architecture. This is caused by the fact that for reasons of architecture independence, we always introduce correspondent communication. Sometimes first or last communication would be perfectly sufficient. Whereas correspondent communication is cheap on the hypercube – it can be performed in one step – it is more expensive on the mesh and rather expensive on the array. Thus it is obvious to specialize first or last communications on these architectures by eliminating correspondent communication. This can be achieved by *partial evaluation of the communication pattern*. As an example, we give two lemmas for arrays and meshes:

Lemma 8 (Communication transformation for linear arrays). *Let the following compound communication pattern for linear arrays be given:*

$$\mathit{JOIN}_A\ n\ x\ (\mathit{ZIPWITH} \oplus\ (\mathit{DISTL}_A\ n\ (\mathit{CORR}_A\ n\ x))\ x)$$

This pattern is partially evaluated into:

$$\mathit{JOIN}_A\ n\ x\ (\mathit{ZIPWITH} \oplus\ (\mathit{DISTR}_A\ n\ (\mathit{SHR}_A\ 1\ x))\ x)$$

Proof. See Appendix A.6. □

Note. The expression $\mathit{DISTL}_A\ n\ (\mathit{CORR}_A\ n\ x)$ is slice-distributive, whereas the substituted expression $\mathit{DISTR}_A\ n\ (\mathit{SHR}_A\ 1\ x)$ is not. However both expressions are at least equal on every second slice of length n. Therefore the expression must be embedded as the second parameter in a $\mathit{JOIN}_A\ n$. The use of *ZIPWITH* generalizes the communication transformation.

While the communication pattern with the correspondent communication needs $3n - 1$ elementary shifts, the improved pattern can do with n shifts.

In a similar way, we can derive a communication improvement for mesh connected computers.

Lemma 9 (Communication transformation for meshes). *Let the following compound communication pattern for meshes be given:*

$$\mathit{JOIN}_M\ n\ x\ (\mathit{ZIPWITH} \oplus (\mathit{DISTL}_M\ n\ (\mathit{CORR}_M\ n\ x))\ x)$$

This pattern is partially evaluated into:

$$\begin{array}{l} \mathit{JOIN}_M\ n\ x\ (\mathit{ZIPWITH} \oplus\ x\ x') \\ \textbf{where}\ \ x' = \mathit{DISTR}_M\ n\ (\mathit{SHR}_M\ 1\ x),\ \ \textbf{if}\ \ n < N \\ \qquad\qquad\quad \mathit{DISTL}_M\ n\ (\mathit{SHD}_M\ \frac{n}{N}\ x),\ \textbf{otherwise} \end{array}$$

Proof. Analogous to the proof of Lemma 8. □

In the worst case ($n > N$), the improved pattern requires $N + \frac{n}{N} - 1$ elementary shifts on meshes, while the original communication with correspondent shifts needs $N + 3\frac{n}{N} - 2$. Since communication costs are crucial for the efficiency of real parallel programs, a reduction of elementary shifts by a factor of about 3 seems worth the work.

Example Parallel prefix cont'd. Applying the communication transformation for arrays to $psum_2$ results in:

```
$psum_0\ x = psum_3\ \#x\ 1\ x$
**where**
    $psum_3\ m\ n\ x = x,$                                  **if** $m = n$
    $psum_3\ m\ n\ x = psum_3\ m\ (2n)\ (JOIN_A\ n\ x\ x')$, **otherwise**
  **where** $x' = ZIPWITH\ (+)\ (DISTR_A\ n\ (SHR_A\ 1\ x)\ x)$
```

An implementation of $psum_3$ in a real data parallel language is now straightforward and presented in Appendix B. □

5 Applications

In order to demonstrate the usefulness of the presented approach, we work out two somewhat more complex examples. In Sect. 5.1, we treat one of the most popular sorting algorithms for data parallel computers viz. Batcher's bitonic sort. Section 5.2 deals with a problem in computational geometry, namely the construction of a convex hull.

5.1 Bitonic Sort

The well-known bitonic sort algorithm was proposed by K. E. Batcher in 1968 for so called sorting networks [Bat68] and later adapted to parallel computers [NS79].

Preliminaries and Operational Specifications

The bitonic sort algorithm is based on the central notion of the *bitonic sequence.* A sequence s is said to be bitonic if it either monotonically increases and then monotonically decreases, or else monotonically decreases and then monotonically increases. For example, the sequences $[1, 4, 6, 8, 3, 2]$ and $[9, 8, 3, 2, 4, 6]$ are both bitonic.

The fundamental idea behind the bitonic sort algorithm rests on the following observation: let $s = x +\!\!+\ y$ be a bitonic sequence and let $d = zipWith\ min\ x\ y$ and $e = zipWith\ max\ x\ y$, where min computes the minimum and max the maximum of two ordered values. Then we have:

(i) d and e are each bitonic and
(ii) $reduce\ max\ d \leq reduce\ min\ e$.

The proof of this proposition can be found in [Bat68].

Bitonic Sorter. This fact, merging two bitonic sequences gives an ascending sequence, immediately gives us an operational specification according to the DC paradigm. As a precondition, we require the input sequence to be nonempty and bitonic.

$$\begin{aligned} &\mathit{bimerge}\ [e] &&= [e] \\ &\mathit{bimerge}(x +\!\!+\ y) &&= \mathit{bimerge}(\mathit{zipWith}\ \mathit{min}\ x\ y) +\!\!+\ \mathit{bimerge}(\mathit{zipWith}\ \mathit{max}\ x\ y) \end{aligned}$$

Arbitrary Sorter. A sorter for arbitrary sequences (implemented by function *bisort*) can be constructed from bitonic sorters using a sorting-by-merging scheme: decompose a sequence of length n into separate intervals of length 2. Trivially, these intervals are bitonic so that we can use the algorithm for bitonic sequences. In this way, we obtain $\frac{n}{2}$ pairs of sorted elements.

Unfortunately, two adjacent subsequences in ascending order cannot be put together to form a single bitonic sequence. To achieve this, the intervals have to be sorted alternately in ascending and descending order, or every second interval has to be reversed. Doing so, we get $\frac{n}{4}$ intervals of length 4, all of them are bitonic so that again the above algorithm for bitonic sequences can be applied. This process is repeated until we get a single bitonic interval, which eventually will be sorted by function *bimerge*.

Again, we can summarize this informal description into an operational specification using the DC strategy:

$$\begin{aligned} &\mathit{sort}\ s = \mathit{bimerge}(\mathit{bisort}\ s) \\ &\textbf{where} \\ &\qquad \mathit{bisort}\ [e] = [e] \\ &\qquad \mathit{bisort}(x +\!\!+\ y) = \mathit{bimerge}\ (\mathit{bisort}\ x) +\!\!+\ \mathit{reverse}\ (\mathit{bimerge}\ (\mathit{bisort}\ y)) \end{aligned}$$

Note. Algorithm *bisort* explicitly reverses every second interval, putting an ascending sequence into a descending one by means of the auxiliary function *reverse*. The same effect can be achieved by inverting the comparisons, i.e., instead of *min* in function *bimerge* we use *max* and vice versa. Function $\mathit{bimerge}' = \mathit{reverse} \circ \mathit{bimerge}$ uses inverted comparisons in order to return sequences in descending order.

We redefine function *sort* by explicitly using function $\mathit{bimerge}'$:

$$\begin{aligned} &\mathit{sort}\ s = \mathit{bimerge}(\mathit{bisort}'\ s) \\ &\textbf{where} \\ &\qquad \mathit{bisort}'\ [e] = [e] \\ &\qquad \mathit{bisort}'(x +\!\!+\ y) = \mathit{bimerge}\ (\mathit{bisort}'\ x) +\!\!+\ \mathit{bimerge}'\ (\mathit{bisort}'\ y) \end{aligned}$$

Parallelization

A closer inspection of the operational specifications shows that they both fit the patterns provided by the transformation rules given in Sect. 3.

Transformation of function *bimerge*. In order to apply the rule *Top-down with pre-adjustment* to function *bimerge*, we have to instantiate the input scheme given by Theorem 2:

$$\begin{aligned}&t\ x = x\\&g\ x\ y = \mathit{zipWith}\ \mathit{min}\ x\ y\\&h\ x\ y = \mathit{zipWith}\ \mathit{max}\ x\ y\end{aligned}$$

In the next step, we want to rewrite the cascading recursive definitions of t', g' and h' given in Theorem 2. Remember that we aim at a data-parallel computation scheme, where we can apply a single instruction to multiple data elements.

Derivation. Let $n = \#x1$ and $x = x1 +\!\!+\ x2$ and $x' = x2 +\!\!+\ x1$:

$$\begin{aligned}&g'\ n\ x\ x'\\&= [\text{definition of } x \text{ and } x'\text{, slice-distributivity of } g'\text{, unfold } g'\text{, unfold } g]\\&\qquad (\mathit{zipWith}\ \mathit{min}\ x1\ x2) +\!\!+ (\mathit{zipWith}\ \mathit{min}\ x2\ x1)\\&= [\text{distributivity of } \mathit{zipWith}\text{, assumption: } x = x1 +\!\!+\ x2 \text{ and } x' = x2 +\!\!+\ x1]\\&\qquad \mathit{zipWith}\ \mathit{min}\ x\ x'\end{aligned}$$

In a similar way, we derive simplified definitions for functions t' and h':

$$\begin{aligned}&t'\ n\ x \qquad = x\\&h'\ n\ x\ x' = \mathit{zipWith}\ \mathit{max}\ x\ x'\end{aligned}$$

Due to the slice-distributivity of t', g' and h', their definitions hold for all $n \leq \#x1$. □

Under the assumption $\#x \geq 1$, application of the transformation rule (see Theorem 2) results in:

$$\begin{aligned}&\mathit{bimerge}\ x = \mathit{bimerge}{\downarrow}\ (\#x)\ x\\&\textbf{where}\\&\quad \mathit{bimerge}{\downarrow}\ n\ x = x, && \textbf{if } n = 1\\&\quad \mathit{bimerge}{\downarrow}\ n\ x = \mathit{bimerge}{\downarrow}\ \tfrac{n}{2}\ (\mathit{join}\ \tfrac{n}{2}\ v\ w), && \textbf{otherwise}\\&\quad \textbf{where}\ \ x' = \mathit{corr}\ \tfrac{n}{2}\ x\\&\qquad\qquad (v, w) = (\mathit{zipWith}\ \mathit{min}\ x\ x', \mathit{zipWith}\ \mathit{max}\ x'\ x)\end{aligned}$$

Analogously, we can develop a top-down version of function $\mathit{bimerge}'$:

$$\begin{aligned}&\mathit{bimerge}'{\downarrow}\ n\ x = x, && \textbf{if } n = 1\\&\mathit{bimerge}'{\downarrow}\ n\ x = \mathit{bimerge}'{\downarrow}\ \tfrac{n}{2}\ (\mathit{join}\ \tfrac{n}{2}\ v\ w), && \textbf{otherwise}\\&\textbf{where}\ \ x' = \mathit{corr}\ \tfrac{n}{2}\ x\\&\qquad (v, w) = (\mathit{zipWith}\ \mathit{max}\ x\ x', \mathit{zipWith}\ \mathit{min}\ x'\ x)\end{aligned}$$

Transformation of function *bisort'*. We start with an instantiation of the transformation rule *Bottom-up with post-adjustment* (see Theorem 3):

$$
\begin{aligned}
&t\ x = \ x\\
&k\ x\ y = \mathit{bimerge}\ x\\
&j\ x\ y = \mathit{bimerge}'\ x
\end{aligned}
$$

Again, we replace the (recursive) definitions of t', k' and j' by appropriate data parallel (non-recursive) versions:

Derivation. Let $n = \#x1$ and $x = x1 +\!\!+\, x2$ and $y = y1 +\!\!+\, y2$:

$$
\begin{aligned}
&k'\ n\ x\ y\\
&= [\text{definition of } x \text{ and } y,\ \text{slice-distributivity of } k',\ \text{unfold } k',\ \text{unfold } k]\\
&\qquad (\mathit{bimerge}\ x1) +\!\!+\, (\mathit{bimerge}\ x2)\\
&= [\text{property of } \mathit{bimerge}{\downarrow},\ \text{assumption: } n = \#x1 \text{ and } x = x1 +\!\!+\, x2]\\
&\qquad \mathit{bimerge}{\downarrow}\ n\ x
\end{aligned}
$$

In exactly the same way, we compute instantiations for t' and j':

$$
\begin{aligned}
&t'\ n\ x \quad = x\\
&j'\ n\ x\ y = \mathit{bimerge}'{\downarrow}\ n\ x
\end{aligned}
$$

Due to the slice-distributivity of t', k' and j', their definitions hold for all $n \leq \#x1$. □

Under the assumption $\#x \geq 1$, the application of the transformation rule *Bottom-up with post-adjustment* yields:

```
$bisort'\ x = bisort{\Uparrow}\ (\#x)\ 1\ x$
where
    $bisort{\Uparrow}\ m\ n\ x = x,$                                  if  $m = n$
    $bisort{\Uparrow}\ m\ n\ x = bisort{\Uparrow}\ m\ (2n)\ (join\ n\ v\ w),$ otherwise
    where  $x' = corr\ n\ x$
           $(v, w) = (bimerge{\downarrow}\ n\ x, bimerge'{\downarrow}\ n\ x)$
```

An obvious simplification (since x' does not occur in the body of *bisort*$\Uparrow$) results in:

```
$bisort'\ x = bisort{\Uparrow}\ (\#x)\ 1\ x$
where
    $bisort{\Uparrow}\ m\ n\ x = x,$                                  if  $m = n$
    $bisort{\Uparrow}\ m\ n\ x = bisort{\Uparrow}\ m\ (2n)\ (join\ n\ v\ w),$ otherwise
    where $(v, w) = (bimerge{\downarrow}\ n\ x, bimerge'{\downarrow}\ n\ x)$
```

The final result of our transformational derivation of bitonic sort is summarized in the following program:

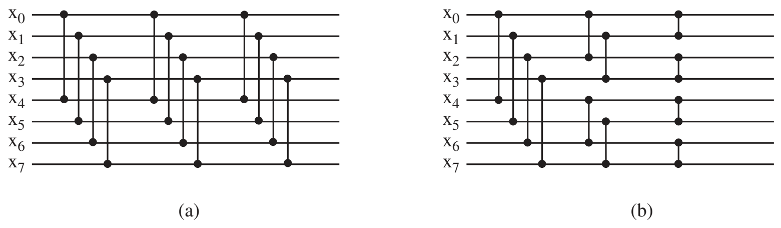

Fig. 2. Sorting a bitonic sequence of 8 elements using: (a) $\mathit{bimerge}_{+\!\!+}$ (b) $\mathit{bimerge}_{\bowtie}$

```
$sort\ s = bimerge{\downarrow}\ (\#x)\ (bisort{\Uparrow}\ (\#x)\ 1\ x)$
where
    $bisort{\Uparrow}\ m\ n\ x = x,$                                  if $m = n$
    $bisort{\Uparrow}\ m\ n\ x = bisort{\Uparrow}\ m\ (2n)\ (join\ n\ v\ w),$ otherwise
    where
        $(v, w) = (bimerge{\downarrow}\ n\ x, bimerge'{\downarrow}\ n\ x)$
        $x'\quad = corr\ \frac{n}{2}\ x$

        $bimerge{\downarrow}\ n\ x = x,$                               if $n = 1$
        $bimerge{\downarrow}\ n\ x = bimerge{\downarrow}\ \frac{n}{2}\ (join\ \frac{n}{2}\ v\ w),$ otherwise
        where $(v, w) = (zipWith\ min\ x\ x', zipWith\ max\ x'\ x)$

        $bimerge'{\downarrow}\ n\ x = x,$                               if $n = 1$
        $bimerge'{\downarrow}\ n\ x = bimerge'{\downarrow}\ \frac{n}{2}\ (join\ \frac{n}{2}\ v\ w),$ otherwise
        where $(v, w) = (zipWith\ max\ x\ x', zipWith\ min\ x'\ x)$
```

It can be efficiently executed on massively parallel computers with such diverse topologies as linear array, mesh connected computer or hypercube.

Addendum. The bitonic merge algorithm is often presented with the alternative constructor set based on odd-even division.

The difference can nicely be illustrated using a comparison network, which is comprised solely of wires and comparators. We draw wires as horizontal lines, its inputs appear on the left, its outputs on the right and draw the comparator, which receives two inputs x and y and generates the two outputs $x' = min\ x\ y$ and $y' = max\ x\ y$ as vertical lines.

We immediately observe that in the derived *bimerge* function, henceforth called $\mathit{bimerge}_{+\!\!+}$, the connections between comparators varies from stage to stage (see Fig. 2(a)), whereas the connections between comparators is constant using a shuffle network. This was already realized by Stone [Sto72]. His *bimerge* variant, here called $\mathit{bimerge}_{\bowtie}$ (see Fig. 2(b)), is a bottom-up computation with an odd-even division instead of the left-right one:

$$\begin{aligned} &\mathit{bimerge}_{\bowtie}\ [e] &&= [e] \\ &\mathit{bimerge}_{\bowtie}(x \bowtie y) &&= \mathit{zipWith}\ \mathit{min}\ v\ w \bowtie \mathit{zipWith}\ \mathit{max}\ v\ w \\ & &&\quad \textbf{where}\ (v, w) = (\mathit{bimerge}_{\bowtie} x, \mathit{bimerge}_{\bowtie} y) \end{aligned}$$

Obviously, $\mathit{bimerge}_{\bowtie}$ matches the input pattern of the odd-even to left-right division rule. We apply theorem 4 to $\mathit{bimerge}_{\bowtie}$ and result in $\mathit{bimerge}_{+\!\!+}$. Thus both versions are equivalent; the data parallelization of $\mathit{bimerge}_{\bowtie}$ needs only one initial transformation step. $\lhd$

5.2 Convex Hull

This section considers the problem of constructing the convex hull from a finite set S of points in the two-dimensional real space $\mathbb{R} \times \mathbb{R}$. The algorithm given here is mainly an adaptation of a sequential one presented in [PH77] with major changes to fit the massively parallel paradigm.

Preliminaries and Operational Specifications

Given a set $S = \{s_1, s_2, \ldots, s_{2n}\}$ of points in the plane, the convex hull of S is the smallest convex polygon P, for which each point in S is either on the boundary of P or in its interior. The following analogy given in [Akl89] might be useful: Assume that the points of S are nails driven halfway into a wooden board. A rubber band is now stretched around the set of nails and then released. When the band settles, it has the shape of a polygon. Those nails touching the band at the corners of that polygon are the vertices of the convex hull.

It simplifies the exposition, if we divide the problem into two sub-problems. First, we calculate the upper hull $UH(S)$ of set S. This is that part of its boundary traced by a clockwise path from the leftmost to rightmost points in S. In a second phase, we compute the according lower hull $LH(S)$. Since the computation of $UH(S)$ is analogous to the computation of $LH(S)$, we omit the latter. In a preprocessing step, a sequence is created containing the elements of S sorted by x-coordinate (e.g., by applying the bitonic sort algorithm given above).

To start with, we consider an algebraic type that defines the points in the plane in addition with suitable operations on it. Suppose Point denotes a pair of real numbers on which the following operations are defined:

$$\begin{aligned} &.x, .y && :: \mathit{Point} \to \mathit{Real} \\ &. = . && :: \mathit{Point} \to \mathit{Point} \to \mathit{Bool} \\ &\mathit{max}_x, \mathit{max}_y, \mathit{min}_x, \mathit{min}_y && :: \mathit{Point} \to \mathit{Point} \to \mathit{Point} \end{aligned}$$

The interpretation of these operations is as follows:

$$\begin{aligned} &(a, b).x = a && (a, b).y = b \\ &\mathit{max}_x\ p\ q = q,\ \textbf{if}\ p.x < q.x && \mathit{max}_y\ p\ q = q,\ \textbf{if}\ p.y < q.y \\ &\qquad\qquad\ \ p,\ \textbf{otherwise} && \qquad\qquad\ \ p,\ \textbf{otherwise} \\ &\mathit{min}_x\ p\ q = p,\ \textbf{if}\ p.x < q.x && \mathit{min}_y\ p\ q = p,\ \textbf{if}\ p.y < q.y \\ &\qquad\qquad\ \ q,\ \textbf{otherwise} && \qquad\qquad\ \ q,\ \textbf{otherwise} \\ &(p = q) = (p.x = q.x)\ \wedge\ (p.y = q.y) \end{aligned}$$

The DC method of constructing $UH(S)$ given in [PH77] is as follows: Let S be a sequence of $2n$ points in the plane such that $s_1.x \leq s_2.x \leq \ldots \leq s_{2n}.x$ where n is a power of 2. If $n \leq 1$, then S itself is an upper hull of S (primitive case). Otherwise, we subdivide S into two subsequences $S_1 = [s_1, s_2, \ldots, s_n]$ and $S_2 = [s_{n+1}, \ldots, s_{2n}]$. Then, we recursively compute $UH(S_1)$ and $UH(S_2)$ in parallel. As the final step, we must find the upper common tangent between $UH(S_1)$ and $UH(S_2)$, and deduce the upper hull of S.

The informal description given above can immediately be formulated as an operational specification on non-empty sequences of points:

$$\begin{array}{lll} UH :: [Point] \to [Point] & & \\ UH\ s & = s, & \textbf{if}\ \#s \leq 2 \\ UH\ (s1 +\!\!\!+\ s2) & = UCT\ (UH\ s1)\ (UH\ s2), & \textbf{otherwise} \end{array}$$

Function UCT combines two nonintersecting upper hulls $UH(S_1)$ and $UH(S_2)$ by means of the *upper common tangent*, which is the unique line touching both $UH(S_1) = [p_1, \ldots, p_M]$ and $UH(S_2) = [q_1, \ldots, q_N]$ at unique corners p and q (see Fig. 3(b)).

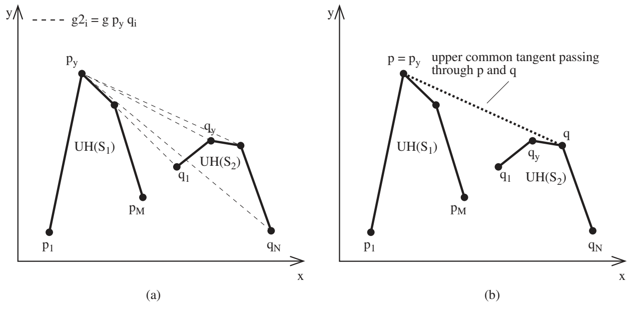

Fig. 3. Upper common tangent of $UH(S_1)$ and $UH(S_2)$

The upper common tangent can be computed by first determining those points p_y and q_y of $UH(S_1)$ and $UH(S_2)$, respectively, with the maximal y-coordinates. To compute a point s_y with the maximal y-coordinate in a sequence of points s, we use the reduce operation: $s_y =_{def} reduce\ max_y\ s$.

Then, p is defined as the rightmost point in $UH(S_1)$ with the minimal slope wrt. q_y. Its formal definition is: $p =_{def} p_i,\ i \in \{1, \ldots, M\}$ such that

1. $g\ q_y\ p_i < g\ q_y\ p_j$, for all $j \in \{i+1, \ldots, M\}$ and
2. $g\ q_y\ p_i \leq g\ q_y\ p_j$ for all $j \in \{1, \ldots, i-1\}$

where g determines the slope of the line passing through the points a and b:

$$
\begin{array}{ll}
\multicolumn{2}{l}{g :: \mathit{Point} \rightarrow \mathit{Point} \rightarrow \mathit{Real}} \\
g\ a\ b = \bot, & \textbf{if } (a = \bot) \vee (b = \bot) \\
\qquad\quad (b.y - a.y)/(b.x - a.x), & \textbf{otherwise}
\end{array}
$$

Henceforth, $\bot$ denotes an undefined value, which remains unchanged during computation.

The second corner q in $UH(S_2)$ is specified in a similar way, where only the signs of the slopes are inverted.

Figure 3(a) depicts two upper hulls $UH(S_1)$ and $UH(S_2)$. The dashed lines are the tangents passing through p_y. The tangent with the minimal slope (modulo sign) determines the right corner q. Figure 3(b) pictures the result of computing the upper common tangent. The new upper hull now consists of points $[p_1, p, q, q_N]$.

An operational specification of the above description reads as follows:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{UCT} :: [\mathit{Point}] \rightarrow [\mathit{Point}] \rightarrow [\mathit{Point}]} \\
\multicolumn{3}{l}{\mathit{UCT}\ s1\ s2 = s1' +\!\!+\ s2'} \\
\textbf{where} & (p_y, q_y) & = (\mathit{reduce}\ \mathit{max}_y\ s1, \mathit{reduce}\ \mathit{max}_y\ s2) \\
 & (g1, g2) & = (\mathit{map}\ (g\ q_y)\ s1, \mathit{map}\ (\mathit{neg} \circ (g\ p_y))\ s2) \\
 & (m1, m2) & = (\mathit{reduce}\ \mathit{min}\ g1, \mathit{reduce}\ \mathit{min}\ g2) \\
 & (f1, f2) & = (\mathit{find}\ m1\ g1\ s1, \mathit{find}\ m2\ g2\ s2) \\
 & (p, q) & = (\mathit{reduce}\ \mathit{max}_x\ f1, \mathit{reduce}\ \mathit{min}_x\ f2) \\
 & (s1', s2') & = (\mathit{map}\ (\mathit{upd}\ (<)\ p)\ s1, \mathit{map}\ (\mathit{upd}\ (>)\ q)\ s2)
\end{array}
$$

In UCT, first the maximal points in $s1$ and $s2$ wrt. the y-coordinate are determined, resulting in the pair (p_y, q_y). Then, in every subsequence $s1$ and $s2$, respectively, the slopes are computed by means of the auxiliary function g. In $s2$, function neg additionally negates the slopes, where

$$
\begin{array}{ll}
\mathit{neg}\ x = \bot, & \textbf{if } x = \bot \\
\qquad\quad -x, & \textbf{otherwise}
\end{array}
$$

The pair $(m1, m2)$ denotes the minimal slope in each subsequence $s1$ and $s2$. Points, whose tangents wrt. p_y and q_y have a slope equal to $m1$ and $m2$ are assembled in the pair of sequences $(f1, f2)$ by means of function *find*:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{find} :: \mathit{Real} \rightarrow [\mathit{Real}] \rightarrow [\mathit{Point}] \rightarrow [\mathit{Point}]} \\
\multicolumn{2}{l}{\mathit{find}\ m\ gs\ s = \mathit{zipWith}\ (\mathit{is}_m\ m)\ gs\ s} \\
\textbf{where}\ \mathit{is}_m\ m\ m'\ x = x, & \textbf{if } m = m' \\
\qquad\qquad\qquad\qquad\quad \bot, & \textbf{otherwise}
\end{array}
$$

Then, the unique corners p and q of $s1$ and $s2$ are the rightmost and leftmost points in the according subsequences. Finally those elements in $s1$ and $s2$, resp., which do not belong to the upper hull, are replaced by dummy elements, according to the definition of function *upd*:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{upd} :: (\mathit{Point} \rightarrow \mathit{Point} \rightarrow \mathit{Bool}) \rightarrow \mathit{Point} \rightarrow \mathit{Point} \rightarrow \mathit{Point}} \\
\mathit{upd}\ \oplus\ a\ b = \bot, & \textbf{if } a.x\ \oplus\ b.x \\
\qquad\qquad\quad b, & \textbf{otherwise}
\end{array}
$$

Unfolding function UCT in the body of UH leads to a version, which fits the input scheme of transformation rule *Bottom-up with post-adjustment*:

$$
\begin{array}{lll}
UH\ s & = s, & \textbf{if } \#s \leq 2 \\
UH\ (s1 +\!\!+\ s2) & = k\ v\ w +\!\!+\ j\ v\ w, & \textbf{otherwise}
\end{array}
$$

where

$$
\begin{array}{llll}
\multicolumn{4}{l}{(v, w) = (UH\ s1, UH\ s2)} \\
k\ v\ w & = map\ (upd\ (<)\ (p\ v\ w))\ v & j\ v\ w & = map\ (upd\ (>)\ (q\ v\ w))\ w \\
p\ v\ w & = reduce\ max_x\ (f1\ v\ w) & q\ v\ w & = reduce\ min_x\ (f2\ v\ w) \\
f1\ v\ w & = find\ (m1\ v\ w)\ (g1\ v\ w)\ v & f2\ v\ w & = find\ (m2\ v\ w)\ (g2\ v\ w)\ w \\
m1\ v\ w & = reduce\ min\ (g1\ v\ w) & m2\ v\ w & = reduce\ min\ (g2\ v\ w) \\
g1\ v\ w & = map\ (g\ (q_y\ v\ w))\ v & g2\ v\ w & = map\ (neg \circ (g\ (p_y\ v\ w)))\ w \\
p_y\ v\ w & = reducemax_y\ v & q_y\ v\ w & = reducemax_y\ w
\end{array}
$$

Note. In order to ease the following parallelization we lifted the object declarations of UCT to functions in UH.

Parallelization

As in the previous subsection, we carry out some precomputations in order to derive instantiations of t', k' and j' without using recursion:

Derivation. Let $n = \#s1$ and $s = s1 +\!\!+\ s2$ and $s' = s2 +\!\!+\ s1$.

$k'\ n\ s\ s'$

$=$ [definition of s and s', slice-distributivity of k' unfold k', unfold k]

$\quad map\ (upd\ (<)\ (p\ s1\ s2))\ s1 +\!\!+\ map\ (upd\ (<)\ (p\ s2\ s1))\ s2$

$=$ [property of map wrt. $zipWith$, distributivity of $zipWith$]

$\quad zipWith\ (upd\ (<))\ (copy\ n\ (p\ s1\ s2) +\!\!+\ copy\ n\ (p\ s2\ s1))\ (s1 +\!\!+\ s2)$

$=$ [$s = s1 +\!\!+\ s2$, $p'\ n\ s\ s' =_{def} copy\ n\ (p\ s1\ s2) +\!\!+\ copy\ n\ (p\ s2\ s1)$]

$\quad zipWith\ (upd\ (<))\ (p'\ n\ s\ s')\ s$

$p'\ n\ s\ s' =_{def} copy\ n\ (p\ s1\ s2) +\!\!+\ copy\ n\ (p\ s2\ s1)$

$=$ [unfold p]

$\quad copy\ n\ (reduce\ max_x\ (f1\ s1\ s2)) +\!\!+\ copy\ n\ (reduce\ max_x\ (f1\ s2\ s1))$

$=$ [$reduce{\uparrow}\ \oplus\ s =_{def} copy\ (\#s)(reduce\ \oplus\ s)$]

$\quad reduce{\uparrow}\ max_x\ (f1\ s1\ s2) +\!\!+\ reduce{\uparrow}\ max_x\ (f1\ s2\ s1)$

$=$ [$reduce{\Uparrow}^{\star\star}\ \oplus\ (\#s1)\ 1\ (s1 +\!\!+\ s2) = reduce{\uparrow}\ \oplus\ s1 +\!\!+\ reduce{\uparrow}\ \oplus\ s2$]

$\quad reduce{\Uparrow}\ max_x\ n\ 1\ (f1\ s1\ s2 +\!\!+\ f1\ s2\ s1)$

$=$ [$f1'\ n\ s\ s' =_{def} f1\ s1\ s2 +\!\!+\ f1\ s2\ s1$]

$\quad reduce{\Uparrow}\ max_x\ n\ 1\ (f1'\ n\ s\ s')$

** The function $reduce{\Uparrow}$ is a parallel version of $reduce{\uparrow}$. Its derivation is analogous to the given ones.

In an analogous way, we can find generalizations for $f1, m1, g1$ and q_y:

$$\begin{aligned}
&f1'\ n\ s\ s' &&= \mathit{zipWith3}\ is_m\ (m1'\ n\ s\ s')\ (g1'\ n\ s\ s')\ s\\
&m1'\ n\ s\ s' &&= \mathit{reduce}{\Uparrow}\ \mathit{min}\ n\ 1\ (g1'\ n\ s\ s')\\
&g1'\ n\ s\ s' &&= \mathit{zipWith}\ g\ (q'_y\ n\ s\ s')\ s\\
&q'_y\ n\ s\ s' &&= \mathit{reduce}{\Uparrow}\ \mathit{max}_y\ n\ 1\ s'
\end{aligned}$$

Due to the slice-distributivity of k', definition of k' holds for all $n \leq \#s1$. Analogously, we can derive:

$$\begin{aligned}
&t'\ n\ s = s\\
&j'\ n\ s\ s' = \mathit{zipWith}\ (\mathit{upd}\ (>))\ (q'\ n\ s\ s')\ s'\\
&\textbf{where}\\
&\qquad q'\ n\ s\ s' = \mathit{reduce}{\Uparrow}\ \mathit{min}_x\ n\ 1\ (f2'\ n\ s\ s')\\
&\qquad f2'\ n\ s\ s' = \mathit{zipWith3}\ is_m\ (m2'\ n\ s\ s')\ (g2'\ n\ s\ s')\ s'\\
&\qquad m2'\ n\ s\ s' = \mathit{reduce}{\Uparrow}\ \mathit{min}\ n\ 1\ (g2'\ n\ s\ s')\\
&\qquad g2'\ n\ s\ s' = \mathit{map}\ \mathit{neg}\ (\mathit{zipWith}\ g\ (p'_y\ n\ s\ s')\ s')\\
&\qquad p'_y\ n\ s\ s' = \mathit{reduce}{\Uparrow}\ \mathit{max}_y\ n\ 1\ s
\end{aligned}$$

□

The application of Theorem 3 results in:

$$\begin{aligned}
&\mathit{UH}\ s = \mathit{UH}'\ (\#s)\ 2\ s\\
&\textbf{where}\\
&\quad \mathit{UH}'\ m\ n\ s = s, && \textbf{if } m = n\\
&\qquad\qquad \mathit{UH}'\ m\ (2n)\ (\mathit{join}\ n\ (k'\ n\ s\ s')\ (j'\ n\ s'\ s)), && \textbf{otherwise}\\
&\quad \textbf{where}\\
&\qquad s' = \mathit{corr}\ n\ s\\
&\qquad k' \text{ and } j' \text{ as defined above}
\end{aligned}$$

which, after several unfolding steps and consistent renaming, leads to a data parallel version of UH':

$$\begin{aligned}
&\mathit{UH}'\ m\ n\ s = s, && \textbf{if } m = n\\
&\qquad\qquad \mathit{UH}'\ m\ (2n)\ (\mathit{join}\ n\ \overline{k}\ \overline{j}), && \textbf{otherwise}\\
&\textbf{where}\\
&\quad s' = \mathit{corr}\ n\ s\\
&\quad \overline{k} = \mathit{zipWith}\ (\mathit{upd}\ (<))\ \overline{p}\ s && \overline{j} = \mathit{zipWith}\ (\mathit{upd}\ (>))\ \overline{q}\ s\\
&\quad \overline{p} = \mathit{reduce}{\Uparrow}\ \mathit{max}_x\ n\ 1\ \overline{f1} && \overline{q} = \mathit{reduce}{\Uparrow}\ \mathit{min}_x\ n\ 1\ \overline{f2}\\
&\quad \overline{f1} = \mathit{zipWith3}\ is_m\ \overline{m1}\ \overline{g1}\ s && \overline{f2} = \mathit{zipWith3}\ is_m\ \overline{m2}\ \overline{g2}\ s\\
&\quad \overline{m1} = \mathit{reduce}{\Uparrow}\ \mathit{min}\ n\ 1\ \overline{g1} && \overline{m2} = \mathit{reduce}{\Uparrow}\ \mathit{min}\ n\ 1\ \overline{g2}\\
&\quad \overline{g1} = \mathit{zipWith}\ g\ \overline{p_y}\ s && \overline{g2} = \mathit{map}\ \mathit{neg}\ (\mathit{zipWith}\ g\ \overline{q_y}\ s)\\
&\quad \overline{q_y} = \mathit{reduce}{\Uparrow}\ \mathit{max}_y\ n\ 1\ s' && \overline{p_y} = \mathit{reduce}{\Uparrow}\ \mathit{max}_y\ n\ 1\ s'
\end{aligned}$$

A closer inspection of this version of UH' shows that due to the generality of our transformation rules we wasted a lot of parallelism. Since *join* only takes half of the elements of its argument sequences, we compute some data values sequentially instead of parallel. Thus, we continue our derivation by applying an adapted horizontal fusion strategy [Par90], which amounts to "merging" different computations into a single one.

Derivation. Without loss of generality, we assume $n = \frac{\#s}{2}$. The auxiliary functions *left* and *right* take the first and the second half of a sequence, respectively: $\mathit{left}\ (s1 +\!\!+\ s2) = s1$ and $\mathit{right}\ (s1 +\!\!+\ s2) = s2$.

$$
\begin{aligned}
& \mathit{join}\ n\ \overline{k}\ \overline{j} \\
=\ & [\ \text{unfold } \overline{k} \text{ and } \overline{j}\ ] \\
& \qquad \mathit{join}\ n\ (\mathit{zipWith}\ (\mathit{upd}\ (<))\ \overline{p}\ s)\ (\mathit{zipWith}\ (\mathit{upd}\ (>))\ \overline{q}\ s) \\
=\ & [\ \text{distributivity of } \mathit{zipWith}, \text{ unfold } \mathit{join}\ ] \\
& \qquad \mathit{zipWith}\ (\mathit{upd}\ (<))\ (\mathit{left}\ \overline{p})\ s1 +\!\!+\ \mathit{zipWith}\ (\mathit{upd}\ (>))\ (\mathit{right}\ \overline{q})\ s2 \\
=\ & [\ \overline{pq} =_{def} \mathit{left}\ \overline{p} +\!\!+\ \mathit{right}\ \overline{q}\ \ ] \\
& \qquad \mathit{join}\ n\ (\mathit{zipWith}\ (\mathit{upd}\ (<))\ \overline{pq}\ s)\ (\mathit{zipWith}\ (\mathit{upd}\ (>))\ \overline{pq}\ s)
\end{aligned}
$$

$$
\begin{aligned}
& \overline{pq} =_{def} \mathit{left}\ \overline{p} +\!\!+\ \mathit{right}\ \overline{q} \\
=\ & [\ \text{unfold } \overline{p} \text{ and } \overline{q}\ ] \\
& \qquad \mathit{left}\ (\mathit{reduce}\Uparrow\ \mathit{max}_x\ n\ 1\ \overline{f1}) +\!\!+\ \mathit{right}\ (\mathit{reduce}\Uparrow\ \mathit{min}_x\ n\ 1\ \overline{f2}) \\
=\ & [\ \text{property of } \mathit{reduce}\Uparrow \text{ under the assumption } n = \#\overline{f1} = \#\overline{f2}\ ] \\
& \qquad \mathit{reduce}\Uparrow\ \mathit{max}_x\ n\ 1\ (\mathit{left}\ \overline{f1}) +\!\!+\ \mathit{reduce}\Uparrow\ \mathit{min}_x\ n\ 1\ (\mathit{right}\ \overline{f2}) \\
=\ & [\ \overline{f} =_{def} \mathit{left}\ \overline{f1} +\!\!+\ \mathit{right}\ \overline{f2}\ ] \\
& \qquad \mathit{join}\ n\ (\mathit{reduce}\Uparrow\ \mathit{max}_x\ n\ 1\ \overline{f})\ (\mathit{reduce}\Uparrow\ \mathit{min}_x\ n\ 1\ \overline{f})
\end{aligned}
$$

Similar derivations lead to appropriate equations for $\overline{f}, \overline{m}, \overline{g}, \overline{a}$ and $\overline{pq_y}$ (see below). □

Our final version of the convex hull algorithm is summarized in the following program:

$$
\begin{aligned}
& \mathit{UH}\ s = \mathit{UH}'\ (\#s)\ 2\ s \\
& \textbf{where} \\
& \qquad \mathit{UH}'\ m\ n\ s = s, \qquad\qquad\qquad\qquad\quad \textbf{if } m = n \\
& \qquad\qquad\qquad\quad \mathit{UH}'\ m\ (2n)\ (\mathit{join}\ n\ \overline{k}\ \overline{j}), \quad \textbf{otherwise} \\
& \qquad\quad \textbf{where}\ \begin{array}{ll}
s' & = \mathit{corr}\ n\ s \\
\overline{k} & = \mathit{zipWith}\ (\mathit{upd}\ (<))\ \overline{pq}\ s \\
\overline{j} & = \mathit{zipWith}\ (\mathit{upd}\ (>))\ \overline{pq}\ s \\
\overline{pq} & = \mathit{join}\ n\ (\mathit{reduce}\Uparrow\ \mathit{max}_x\ n\ 1\ \overline{f})\ (\mathit{reduce}\Uparrow\ \mathit{min}_x\ n\ 1\ \overline{f}) \\
\overline{f} & = \mathit{zipWith3}\ \mathit{is}_m\ \overline{m}\ \overline{g}\ s \\
\overline{m} & = \mathit{reduce}\Uparrow\ \mathit{min}\ n\ 1\ \overline{g} \\
\overline{g} & = \mathit{join}\ n\ \overline{a}\ (\mathit{map}\ \mathit{neg}\ \overline{a}) \\
\overline{a} & = \mathit{zipWith}\ g\ \overline{pq_y}\ s \\
\overline{pq_y} & = \mathit{reduce}\Uparrow\ \mathit{max}_y\ n\ 1\ s'
\end{array}
\end{aligned}
$$

This algorithm uses all those higher order functions on sequences, which can immediately be rewritten as skeletons for a particular massively parallel architecture.

The algorithm we have derived here differs from those in the parallel literature (cf. [JáJ92, Akl89]). Especially, it does not need unrealistic assumptions like a concurrent read access to shared memory variables as e.g. given by the PRAM model, but is well suited for massively parallel computation on distributed memory architectures by making efficiently use of the underlying interconnection network to exchange data.

6 Related Work

Much attention has been paid to the formal parallelization of DC algorithms. Smith develops a DC theory [Smi85, Smi93], e.g., DC can be treated as a morphism from a decomposition algebra on the input domain to a composition algebra on its output domain. His emphasis is on the development of a DC algorithm, whereas we are interested in its data parallelization on a particular architecture. Thus, our work can be seen as a completion of Smith's work towards data parallel execution.

Mou and Houdak describe DC in a algebraic model called Divacon [MH88]. They recognize that the original DC model is too restrictive with respect to decomposition and communication. For the latter, they introduce so called pre- and postmorphims, which correspond with our 'adjustment' functions g, h, k and j. They illustrate the expressive power of this generalized DC, with a broad range of examples. However, they only sketch the mapping of the model on parallel computers.

This algebraic model was later picked up by Carpentiery and Mou, who study communication issues in the model [CM91]. They present hypercube specific rules to optimize communication by introducing new storage levels. These rules are expressed in Divacon, whereas our approach takes the architecture explicitly into account. However, their approach is neither calculated nor transparent.

Axford and Joy [Axf92, AJ93] have proposed to use DC as a fundamental design principle, and have either proposed arrays or sequences as suitable data structures. In fact, the balanced sequence primitives that we use, were proposed by Axford and Joy. Aside from this, no calculation nor interesting distributed implementation is presented.

Among the first, who used the skeleton approach in a functional setting, initiated by Cole [Col89], was a group at Imperial College [DFH$^+$93]. Their skeletons are rather highlevel, e.g., they distinguish farming, pipelining, DC and other high level skeletons, but do not tackle massive parallelism, as it is understood by us.

Still more abstract is the work on investigating parallelism within the Bird-Meertens formalism, which recently has gained much attention (cf. e.g. [Col93]). However, all these different approaches have in common that they stop on the level of DC algorithms or homomorphisms, whereas our approach proceeds down to an architecture specific target program.

An exception to these works is presented by Gorlatch and Lengauer [GL93]. They develop a DC function, using mainly the control parallelism. In particular, they do not require that there is a single PE for each member in the sequence, but assume that there is a single PE for a group of members in the sequence. As before, the step to a working imperative implementation is still left open.

Work that is closely related to ours is done at the University of Nijmegen [Gee92, Gee93, Par93, BGP93, Gee94]. In fact, the skeletons which we propose were adapted from their work. Opposite to our goals, their research aims at introducing data parallelism out of a parallel control structure, which can be achieved by means of partial inversion. Recently, Geerling also considers data type transformations in order to adapt algorithms to different hardware. We start, however, with a problem dependent data structure, which enables right from the start implicit data parallelism.

In contrast to our approach, a group in Yale introduces data fields right from the beginning of the derivation process [CC90, YC92]. They make extensive use of so called domain morphisms in order to specify parallel-program optimizations. Their approach seems to work well for numerical problems, where the problem domain is given by matrices. The main problems lie in the absence of a strategy for deriving programs and in difficulties to find appropriate index domain morphisms, which lead to optimizations.

The important problem of how to cope with the usual situation that the number of processors is smaller than the size of the input domain is ignored in our work. We believe that this is perfectly reasonable, since either the hardware of massively parallel computers (e.g. Connection Machine CM-2), or the software (e.g. Fortran on the MASPAR) abstracts from the number of real processors. However, not all massively parallel machines support virtual processors. Therefore, data distribution is still a major problem, which is tackled by a group around Pepper [PES93].

7 Conclusion and Future Research

In this paper, we have presented a transformation strategy to develop correct, efficient, data parallel DC algorithms, and showed how such derivation is guided. The main advantage of making the strategy explicit lies in its reuseability. A similar problem can be solved in a similar fashion, which is demonstrated by the examples.

We distinguish data parallelism in the problem domain (here: sequences) from data parallelism on the level of the architecture (here: skeletons). This distinction gives rise to develop portable parallel programs, since data parallelism on the problem domain must be mapped differently on existing hardware, if the diversity in architectures is exploited in full.

In addition, we claim that the transformational approach taken here is rather crucial to the presented development: The calculational properties of functional programs, in particular skeletons, give a basis for a solid understanding and a formal treatment for the derivation of massive parallel algorithms from a high-level specification down to the low-level hardware.

More research is necessary for the development of further strategies. In this context, our ultimate goal is the development of a methodology for transformational data parallel program development.

Acknowledgements. We would like to thank Helmuth A. Partsch and Ton Vullinghs for their helpful comments.

References

[AJ93] T. Axford and M. Joy. List processing primitives for parallel computation. *Computer Languages*, 19(1):1–12, 1993.

[Akl89] S. G. Akl. *The Design and Analysis of Parallel Algorithms.* Prentice-Hall, 1989.

[AS95] K. Achatz and W. Schulte. Architecture independent massive parallelization of devide-and-conquer algorithms. In B. Möller, editor, *Proceedings of Mathematics of Program Construction, Bad Irsee, 1995*, volume forthcoming of *Lecture Notes in Computer Science*, 1995.

[Axf92] T. Axford. Crystal: The divide-and conquer paradigm as a basis for parallel language design. In L. Kronsjo and D. Shumsheruddin, editors, *Advances in Parallel Algorithms*, chapter 2. Blackwell, 1992.

[Bat68] K. E. Batcher. Sorting networks and their applications. *AFIPS Spring Joint Computer Conference*, pages 307–314, 1968.

[BBES92] I. Barth, T. Bräunl, S. Engelhardt, and F. Sembach. Parallaxis version 2 user manual. Technical Report 2/92, Fakultät Informatik, Universität Stuttgart, September 1992.

[BGP93] E. A. Boiten, A. M. Geerling, and H. A. Partsch. Transformational derivation of (parallel) programs using skeletons. Technical Report 93-20, Katholieke Universiteit Nijmegen, September 1993. Also: Proceedings of Computer Science in the Netherlands 1993, Utrecht.

[Bir89] R. Bird. Lectures on constructive functional programming. In M. Broy, editor, *Constructive methods in computing science. NATO ASI Series. Series F: Computer and systems sciences 55*, pages 151–216, Berlin, 1989. Springer-Verlag.

[Ble92] G. E. Blelloch. NESL: A nested data-parallel language (version 2.0). Technical Report CMU-CS-93-129, School of Computer Science, Carnegie Mellon University, April 1992.

[Ble93] G. E. Blelloch. Prefix sums and their applications. In J Reif, editor, *Synthesis of Parallel Algorithms*, chapter 1, pages 35–60. Morgan Kaufmann Publishers, 1993.

[BW88] R. Bird and Ph. Wadler. *An Introduction to Functional Programming.* Prentice-Hall, 1988.

[CC90] M. Chen and Y. Choo. Domain morphisms: A new construct for parallel programming and formalizing program optimization. Technical Report DCS/TR-817, Department of Computer Science, Yale University, August 1990.

[CM91] B. Carpentieri and G. Mou. Compile-time transformations and optimizations of parallel divide-and conquer algorithms. *ACM SIGPLAN Notices*, 20(10):19–28, 1991.

[Col89] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press, 1989.

[Col93] M. Cole. List homomorphic parallel algorithms for bracket matching. Technical Report CSR-29-93, Department of Computer Science, University of Edinburgh, August 1993.

[DFH+93] J. Darlington, A. Field, P. Harrison, P. Kelly, D. Sharp, Q. Wu, and R. White. Parallel programming using skeleton functions. In A. Bode, M. Reeve, and G. Wolf, editors, *PARLE'93 Parallel Architectures and Languages Europe*, volume 694 of *Lecture Notes in Computer Science*, pages 146–160, 1993.

[Fea87] M. S. Feather. A survey and classification of some program transformation approaches and techniques. In L.G.L.T. Meertens, editor, *Program Specification and Transformation.* North-Holland, 1987.

[Fox89] G.C. Fox. Parallel computing comes of age: Supercomputer level parallel computations at caltech. *Concurrency: Practice and Experience*, 1(1):63–103, 1989.

[Gee92] A. M. Geerling. Two examples of parallel-program derivation: Parallel-prefix and matrix multiplication. Technical Report DoC 92/33, Imperial College London, November 1992.

[Gee93] A. M. Geerling. Formal derivation of SIMD parallelism from non-linear recursive specifications. Technical Report CSI-R9324, Katholieke Universiteit Nijmegen, September 1993.

[Gee94] A.M. Geerling. Formal derivation of SIMD parallelism from non-linear recursive specifications. In B. Buchberger and J. Volkert, editors, *CONPAR'94 VAPP VI International Conference on Parallel and Vector Processing*, pages 136–147. Springer-Verlag, 1994.

[GL93] S. Gorlatch and C. Lengauer. Parallelization of divide-and conquer in the Bird- Meertens formalism. Technical Report 12/93, Fakultät für Mathematik und Informatik, Universität Passau, Dezember 1993.

[JáJ92] J. JáJá. *An introduction to parallel algorithms.* Addison-Wesley, 1992.

[MH88] Z.G. Mou and M. Hudak. An algebraic model for divide-and-conquer algorithms and its parallelism. *Journal of Supercomputing*, 2(3):257–278, 1988.

[NS79] D. Nassimi and S. Sahni. Bitonic sort on a mesh-connected parallel computer. *IEEE Transactions on Computers*, 27(1):2–7, 1979.

[Par90] H. A. Partsch. *Specification and Transformation of Programs.* Springer-Verlag, 1990.

[Par93] H. Partsch. Some experiments in transforming towards parallel executability. In R. Paige, J. Reif, and R. Wachter, editors, *Parallel Algorithm Derivation and Program Transformation.* Kluwer Academic Publisher, 1993.

[Pep93] P. Pepper. Deductive derivation of parallel programs. In R. Paige, J. Reif, and R. Wachter, editors, *Parallel Algorithm Derivation and Program Transformation.* Kluwer Academic Publishers, 1993. Also: Technical Report 92-23, Technische Universität Berlin, July 1992.

[PES93] P. Pepper, J. Exner, and M. Südholt. Functional development of massively parallel programs. In D. Bjorner, M. Broy, and I.V. Pottosin, editors, *Formal Methods in Programming and Their Applications. Proceedings In-*

ternational Conference Novosibirsk, June/July 1993., volume 735 of *Lecture Notes in Computer Science*, pages 217–238, Berlin, 1993. Springer-Verlag.

[PH77] F. P. Preparata and S. J. Hong. Convex hulls of finite sets of points in two and three dimensions. *Communications of The ACM*, 20:88–93, 1977.

[Ski93] D.B. Skillicorn. A cost calculus for parallel functional programming. Technical Report ISSN-0836-0227-93-348, Department of Computing and Information Science, Queen's University, March 1993.

[Smi85] D.R. Smith. The design of divide-and-conquer algorithms. *Science of Computer Programming*, 5:37–58, 1985.

[Smi93] D. R. Smith. Derivation of paralel sorting algorithms. In R. Paige, J. Reif, and R. Wachter, editors, *Parallel Algorithm Derivation and Program Transformation.* Kluwer Academic Publisher, 1993.

[Sto72] H. S. Stone. Parallel processing with perfect-shuffle. *IEEE Computer*, pages 153–161, February 1972.

[YC92] J. A. Yang and Y. Choo. Data fields as parallel programs. Technical Report CT 06520–2158, Department of Computer Science, Yale University, March 1992.

A Proofs

A.1 Proof of the Generalized Divide-and-Conquer Rule

We show

$$f\ x = f{\uparrow}\ (f{\downarrow}\ x)$$

by induction on the length of the argument:

Induction Basis: $\#x = q$

$$\begin{aligned}
& f{\uparrow}\ (f{\downarrow}\ x) \\
= & \ [\text{ unfold } f{\uparrow} \text{ and } f{\downarrow} \text{ under assumption } \#x = q\ ] \\
& t{\uparrow}\ (t{\downarrow}\ x) \\
= & \ [\text{ applicability condition: } t = t{\uparrow} \circ t{\downarrow}\ ] \\
& t\ x \\
= & \ [\text{ fold } f\ ] \\
& f\ x
\end{aligned}$$

Induction Step: $\#(x +\!\!+\ y) > q$

$$\begin{aligned}
& f{\uparrow}(f{\downarrow}\ (x +\!\!+\ y)) \\
= & \ [\text{ unfold } f{\uparrow} \text{ and } f{\downarrow} \text{ under assumption } \#(x +\!\!+\ y) > q\ ] \\
& k\ v\ w +\!\!+\ j\ v\ w \quad \textbf{where}\ (v,w) = (f{\uparrow}\ (f{\downarrow}\ (g\ x\ y)), f{\uparrow}\ (f{\downarrow}\ (h\ x\ y))) \\
= & \ [\text{ induction hypothesis }] \\
& f\ (x +\!\!+\ y)
\end{aligned}$$

□

A.2 Proof of Transformation Rule: Top-down with Pre-adjustment

The proof consists of two steps and makes substantial use of a lemma, which also will be given during this subsection.

Step 1: Embedding. We start our proof by an appropriate embedding of function $f{\downarrow}$ in order to introduce a termination parameter n, which denotes the length of the input sequence of $f{\downarrow}$:

$$\begin{aligned}
& f{\downarrow}\ x = f{\downarrow}'\ (\#x)\ x \\
& \textbf{where}\ f{\downarrow}'\ n\ x =_{def} f{\downarrow}\ x \quad \text{provided } \#x = n \\
= & \ [\text{ unfold } f{\downarrow}\ ] \\
& \begin{array}{lll} f{\downarrow}'\ n\ x & = t\ x, & \textbf{if}\ \#x = q \\ f{\downarrow}'\ n\ (x +\!\!+\ y) & = f{\downarrow}\ (g\ x\ y) +\!\!+\ f{\downarrow}\ (h\ x\ y), & \textbf{otherwise} \end{array} \\
= & \ [\ \#x = q \Rightarrow n = q,\ g \text{ and } h \text{ length preserving, fold with assertion }] \\
& \begin{array}{lll} f{\downarrow}'\ n\ x & = t\ x, & \textbf{if}\ n = q \\ f{\downarrow}'\ n\ (x +\!\!+\ y) & = f{\downarrow}'\ \frac{n}{2}\ (g\ x\ y) +\!\!+\ f{\downarrow}'\ \frac{n}{2}\ (h\ x\ y), & \textbf{otherwise} \end{array}
\end{aligned}$$

Step 2: Computational Induction. In order to proof the equality of $f{\downarrow}'$ and $f{\Downarrow}$, we define two functionals:

$$\begin{array}{lll} \tau[f{\downarrow}']\ n\ x & = t\ x, & \textbf{if } n = q \\ \tau[f{\downarrow}']\ n\ (x +\!\!+\ y) & = f{\downarrow}'\ \frac{n}{2}\ (g\ x\ y) +\!\!+\ f{\downarrow}'\ \frac{n}{2}\ (h\ x\ y), & \textbf{otherwise} \end{array}$$

for which we assume, that the parameter n denotes the length of the input sequence x and $x +\!\!+\ y$, respectively, and

$$\begin{array}{ll} \sigma[f{\Downarrow}]\ n\ x = t\ x, & \textbf{if } n = q \\ \sigma[f{\Downarrow}]\ n\ x = f{\Downarrow}\ \frac{n}{2}\ (join\ \frac{n}{2}\ v\ w), & \textbf{otherwise} \\ \quad \textbf{where } \ x' = corr\ \frac{n}{2}\ x & \\ \qquad\qquad (v, w) = (g'\ \frac{n}{2}\ x\ x', h'\ \frac{n}{2}\ x'\ x) & \end{array}$$

for which we require that $\#x \geq n$.
Now we have to show:

$$\tau[f{\downarrow}']\ (\#x)\ x = \sigma[f{\Downarrow}]\ (\#x)\ x$$

As an abbreviation we define: $s = x +\!\!+\ y$ and $s' = y +\!\!+\ x = corr\ \#x\ s$.

$$\begin{array}{ll}
 & \tau[f{\downarrow}']\ n\ x \\
= & [\text{ unfold } \tau[f{\downarrow}'] \text{]} \\
 & \begin{array}{ll} t\ x, & \textbf{if } n = q \\ f{\downarrow}'\ \frac{n}{2}\ (g\ x\ y) +\!\!+\ f{\downarrow}'\ \frac{n}{2}\ (h\ x\ y), & \textbf{otherwise} \end{array} \\
= & [\text{ induction hypothesis]} \\
 & \begin{array}{ll} t\ x, & \textbf{if } n = q \\ f{\Downarrow}\ \frac{n}{2}\ (g\ x\ y) +\!\!+\ f{\Downarrow}\ \frac{n}{2}\ (h\ x\ y), & \textbf{otherwise} \end{array} \\
= & [\text{ Lemma A.2]} \\
 & \begin{array}{ll} t\ x, & \textbf{if } n = q \\ f{\Downarrow}\ \frac{n}{2}\ (g\ x\ y +\!\!+\ h\ x\ y), & \textbf{otherwise} \end{array} \\
= & [\text{ fold } g' \text{ and } h'\text{, since } \#x = \#y = \tfrac{n}{2}\text{, property of join]} \\
 & \begin{array}{ll} t\ x, & \textbf{if } n = q \\ f{\Downarrow}\ \frac{n}{2}\ (join\ \frac{n}{2}\ (g'\ \frac{n}{2}\ s\ s')\ (h'\ \frac{n}{2}\ s'\ s)), & \textbf{otherwise} \end{array} \\
= & [\text{ fold } t{\downarrow} \text{]} \\
 & \begin{array}{ll} t{\downarrow}\ q\ x, & \textbf{if } n = q \\ f{\Downarrow}\ \frac{n}{2}\ (join\ \frac{n}{2}\ (g'\ \frac{n}{2}\ s\ s')\ (h'\ \frac{n}{2}\ s'\ s)), & \textbf{otherwise} \end{array} \\
= & [\text{ fold } \sigma[f{\Downarrow}] \text{ with assertion]} \\
 & \sigma[f{\Downarrow}]\ n\ x
\end{array}$$

□

In the above proof, we have used the slice-distributivity of function $f{\Downarrow}$:

Lemma (Slice-distributivity of $f{\Downarrow}$). *Let $\#x = \#y \geq n$. Then function $f{\Downarrow}$ (see Theorem 2) fulfills the following property:*

$$f{\Downarrow}\ n\ (x +\!\!+\ y) = f{\Downarrow}\ n\ x +\!\!+\ f{\Downarrow}\ n\ y$$

The proof is made by induction on the length of the argument.

Induction Basis: $n = q$.

$$\begin{aligned}
& f\Downarrow\ n\ (x +\!\!+\ y) = t'\ q\ (x +\!\!+\ y) \\
= {} & [\text{ unfold } f\Downarrow\ ] \\
& t'\ q\ (x +\!\!+\ y) \\
= {} & [\text{ unfold } t'\ ] \\
& t'\ q\ x +\!\!+\ t'\ q\ y \\
= {} & [\text{ fold } f\Downarrow\ ] \\
& f\Downarrow\ q\ x +\!\!+\ f\Downarrow\ q\ y
\end{aligned}$$

Induction Step: $\#x \geq 2n$. As an abbreviation, we define: $(x', y') = (corr\ n\ x, corr\ n\ y)$.

$$\begin{aligned}
& f\Downarrow\ (2n)\ (x +\!\!+\ y) \\
= {} & [\text{ unfold } f\Downarrow\ ] \\
& f\Downarrow\ n\ (join\ n\ (g'\ n\ s\ s')(h'\ n\ s'\ s)) \\
& \textbf{where } (s, s') = (x +\!\!+\ y, corr\ n\ s) \\
= {} & [\text{ property of } corr\ ] \\
& f\Downarrow\ n\ (join\ n\ (g'\ n\ (x +\!\!+\ y)\ (x' +\!\!+\ y'))(h'\ n\ (x' +\!\!+\ y')\ (x +\!\!+\ y))) \\
= {} & [\text{ unfold } g' \text{ and } h'\ ] \\
& f\Downarrow\ n\ (join\ n\ (g'\ n\ x\ x' +\!\!+\ g'\ n\ y\ y')(h'\ n\ x'\ x +\!\!+\ h'\ n\ y'\ y)) \\
= {} & [\text{ property of } join\ ] \\
& f\Downarrow\ n\ (join\ n\ (g'\ n\ x\ x')\ (h'\ n\ x'\ x) +\!\!+\ join\ n\ (g'\ n\ y\ y')\ (h'\ n\ y'\ y)) \\
= {} & [\text{ induction hypothesis }] \\
& f\Downarrow\ n\ (join\ n\ ((g'\ n\ x\ x')\ (h'\ n\ x'\ x)) +\!\!+\ f\Downarrow\ n\ (join\ n\ (g'\ n\ y\ y')\ (h'\ n\ y'\ y)) \\
= {} & [\text{ fold } f\Downarrow \text{ with assertion }] \\
& f\Downarrow\ (2n)\ x +\!\!+\ f\Downarrow\ (2n)\ y
\end{aligned}$$

□

A.3 Proof of Corollary 4

We only pick out the proposition

$$g\ (corr\ n\ x) = CORR_A\ n\ (g\ x)$$

the remaining propositions can be treated similarly.

Induction Basis: $n = \#x$

$$
\begin{array}{ll}
& CORR_A n(g(x ++ y)) \\
= & [\text{ unfold } CORR_A \text{ and } g\ ] \\
& \begin{array}{ll} SHL_A\ n\ (\lambda\ i\ .\ (x ++ y)_i), & \textbf{if } even(i \text{ div } n) \\ SHR_A\ n\ (\lambda\ i\ .\ (x ++ y)_i), & \textbf{if } \neg even(i \text{ div } n) \end{array} \\
= & [\ even(i \text{ div } n) \equiv i < n, \text{ unfold } SHL_A \text{ and } SHR_A\ ] \\
& \lambda\ i\ .\ \begin{array}{ll} (x ++ y)_{i+n}), & \textbf{if } i < n \\ (x ++ y)_{i-n}), & \textbf{if } i \geq n \end{array} \\
= & [\text{ property of } ++\ ] \\
& \lambda\ i\ .\ \begin{array}{ll} y_i, & \textbf{if } i < n \\ x_{i-n}, & \textbf{if } i \geq n \end{array} \\
= & [\text{ property of list concatenation }] \\
& \lambda\ i\ .\ (y ++ x)_i \\
= & [\text{ fold } g \text{ and } corr\ ] \\
& g\ (corr\ n\ (x ++ y))
\end{array}
$$

Induction Step: $n < \#x$. Let $i \in \{0, \ldots, N-1\}$.

$$
\begin{array}{ll}
& (CORR_A\ n\ (g\ (x ++ y)))(i) \\
= & [\text{ unfold } CORR_A\ ] \\
& \begin{array}{ll} (SHL_A\ n\ g(x ++ y))(i), & \textbf{if } even(i \text{ div } n) \\ (SHR_A\ n\ g(x ++ y))(i), & \textbf{if } \neg even(i \text{ div } n) \end{array} \\
= & [\text{ unfold } SHL_A, SHR_A \text{ and } g\ ] \\
& \begin{array}{ll} (x ++ y)_{N-1}, & \textbf{if } i \geq N-n\ \wedge\ even(i \text{ div } n) \\ (x ++ y)_{i+n}, & \textbf{if } i < N-n\ \wedge\ even(i \text{ div } n) \\ (x ++ y)_0, & \textbf{if } i < n\ \wedge\ \neg even(i \text{ div } n) \\ (x ++ y)_{i-n}, & \textbf{if } i \geq n\ \wedge\ \neg even(i \text{ div } n) \end{array} \\
= & [\ even(i \text{ div } n) \Rightarrow i < N-n, \neg even(i \text{ div } n) \Rightarrow i \geq n\ ] \\
& \begin{array}{ll} (x ++ y)_{i+n}, & \textbf{if } i < N-n\ \wedge\ even(i \text{ div } n) \\ (x ++ y)_{i-n}, & \textbf{if } i \geq n\ \wedge\ \neg even(i \text{ div } n) \end{array}
\end{array}
$$

Case 1: $i < \#x$.

$$
\begin{array}{ll}
& \begin{array}{ll} x_{i+n}, & \textbf{if } 0 \leq i < \#x - n\ \wedge\ even(i \text{ div } n) \\ x_{i-n}, & \textbf{if } n \leq i < \#x\ \wedge\ \neg even(i \text{ div } n) \end{array} \\
= & [\text{ fold } g \text{ and } CORR_A\ ] \\
& (CORR_A\ n\ g(x))(i) \\
= & [\text{ induction hypothesis }] \\
& (g\ (corr\ n\ g(x)))(i) \\
= & [\ i < \#x\ ] \\
& (g\ (corr\ n\ g(x ++ y)))(i)
\end{array}
$$

Case 2: $i \geq \#x$.

$$\begin{array}{ll} y_{i-\#x+n}, & \textbf{if } \#x \leq i < N-n \ \wedge\ even(i \text{ div } n) \\ y_{i-\#x-n}, & \textbf{if } i \geq n+\#x \ \wedge\ \neg even(i \text{ div } n) \end{array}$$

$= [$ index translation $]$

$$\begin{array}{ll} y_{i+n}, & \textbf{if } 0 \leq i < \#x-n \ \wedge\ even(i \text{ div } n) \\ y_{i-n}, & \textbf{if } i \geq n \ \wedge\ \neg even(i \text{ div } n) \end{array}$$

$= [$ analog to case 1 $]$

$\ldots$

$(g\ (corr\ n\ g(x +\!\!\!+\ y)))(i)$

□

A.4 Proof of Corollary 5

As a representative of the four propositions, we only proof

$$CORR_A\ n\ (x \circ g)) \circ g^{-1} = CORR_M\ n\ x$$

Let $i, j \in \{0, \ldots, N-1\}$.

$(CORR_M\ n\ x)(i,j)$

$= [$ unfold $CORR_M$ $]$

$$\begin{array}{l} (JOIN_M\ n\ s1\ s2)(i,j) \\ \textbf{where}\ \ s1 = SHL_M(n \bmod N)(SHU_M(n \text{ div } N)x) \\ \qquad\qquad s2 = SHR_M(n \bmod N)(SHD_M(n \text{ div } N)x) \end{array}$$

Case 1: $n < N$. This implies: $n \text{ div } N = 0 \ \wedge\ n \bmod N = n$. Then by simplifying the above expression, we yield:

$(JOIN_M\ n\ s1\ s2)(i,j)$

$\textbf{where}\ (s1 = SHL_M\ n\ x, s2 = SHR_M\ n\ x)$

$= [$ unfold $JOIN_M$ $]$

$$\begin{array}{ll} (SHL_M\ n\ x)(i,j), & \textbf{if } even((i \cdot N + j) \text{ div } n) \\ (SHR_M\ n\ x)(i,j), & \textbf{otherwise} \end{array}$$

$= [$ unfold SHL_M an SHR_M $]$

$$\begin{array}{ll} x(i, N-1), & \textbf{if } j \geq N-n \ \wedge\ even((i \cdot N + j) \text{ div } n) \\ x(i, j+n), & \textbf{if } j < N-n \ \wedge\ even((i \cdot N + j) \text{ div } n) \\ x(i, 0), & \textbf{if } j < n \ \wedge\ \neg\ even((i \cdot N + j) \text{ div } n) \\ x(i, j-n), & \textbf{if } j \geq n \ \wedge\ \neg\ even((i \cdot N + j) \text{ div } n) \end{array}$$

$= [\ n < N \Rightarrow (even((i \cdot N + j) \text{ div } n) = even(j \text{ div } n) \Rightarrow j < N-n) \ \wedge$
$(\neg even(j \text{ div } n) \Rightarrow j \geq n)]$

$$\begin{array}{ll} x(i, j+n), & \textbf{if } j < N-n \ \wedge\ even((i \cdot N + j) \text{ div } n) \\ x(i, j-n), & \textbf{if } j \geq n \ \wedge\ \neg\ even((i \cdot N + j) \text{ div } n) \end{array}$$

$$= [\ g(i \cdot N + j + n) = (i, j+n) \text{ and } g(i \cdot N + j - n) = (i, j-n)\ ]$$

$$\begin{array}{ll} (x \circ g)(i \cdot N + j + n), & \textbf{if } j < N - n \ \wedge\ \mathit{even}((i \cdot N + j) \text{ div } n) \\ (x \circ g)(i \cdot N + j - n), & \textbf{if } j \geq n \ \wedge\ \neg\ \mathit{even}((i \cdot N + j) \text{ div } n) \end{array}$$

$$= [\ \mathit{even}(j \text{ div } n) \Rightarrow i \cdot N + j < N^2 - n,$$
$$\neg \mathit{even}(j \text{ div } n) \Rightarrow i \cdot N + j \geq n,\ j \geq n \Rightarrow i \cdot N + j \geq n]$$

$$\begin{array}{ll} (x \circ g)(N^2 - 1), & \textbf{if } (i \cdot N + j) \geq N^2 - n \ \wedge \\ & \qquad \mathit{even}((i \cdot N + j) \text{ div } n) \\ (x \circ g)(i \cdot N + j + n), & \textbf{if } j < N - n \ \wedge\ \mathit{even}((i \cdot N + j) \text{ div } n) \\ (x \circ g)(0), & \textbf{if } i \cdot N + j < n \ \wedge\ \neg\ \mathit{even}((i \cdot N + j) \text{ div } n) \\ (x \circ g)(i \cdot N + j - n), & \textbf{if } i \cdot N + j \geq n \ \wedge\ \neg\ \mathit{even}((i \cdot N + j) \text{ div } n) \end{array}$$

$$= [\ \text{fold } \mathit{SHL}_A \text{ and } \mathit{SHR}_A\ ]$$

$$\begin{array}{ll} (\mathit{SHL}_A\ n\ (x \circ g))(i \cdot N + j), & \textbf{if } \mathit{even}((i \cdot N + j) \text{ div } n) \\ (\mathit{SHR}_A\ n\ (x \circ g))(i \cdot N + j), & \textbf{otherwise} \end{array}$$

$$= [\ \text{fold } \mathit{JOIN}_A\ ]$$

$$(\mathit{JOIN}_A\ n\ s1\ s2)(i \cdot N + j)$$
$$\textbf{where } (s1, s2) = \ (\mathit{SHL}_A\ n\ (x \circ g), \mathit{SHR}_A\ n\ (x \circ g))$$

$$= [\ \text{fold } \mathit{CORR}_A \text{ and } g^{-1}\ ]$$

$$((\mathit{CORR}_A\ n\ (x \circ g)) \circ g^{-1})\ i\ j$$

Case 2: $n \geq N$. This implies: $n \text{ div } N = \frac{n}{N} \ \wedge\ n \bmod N = 0$, since both n and N must be a power of 2. Then by simplifying the above expression, we yield:

$$(\mathit{JOIN}_M\ n\ s1\ s2)(i, j) \quad \textbf{where } (s1 = \mathit{SHU}_M\ \tfrac{n}{N}\ x, s2 = \mathit{SHD}_M\ \tfrac{n}{N}\ x)$$

$$= [\ \text{unfold } \mathit{JOIN}_M\ ]$$

$$\begin{array}{ll} (\mathit{SHU}_M\ \frac{n}{N}\ x)(i, j), & \textbf{if } \mathit{even}((i \cdot N + j) \text{ div } n) \\ (\mathit{SHD}_M\ \frac{n}{N}\ x)(i, j), & \textbf{otherwise} \end{array}$$

$$= [\ \text{unfold } \mathit{SHU}_M \text{ and } \mathit{SHD}_M\ ]$$

$$\begin{array}{ll} x(N-1, j), & \textbf{if } i \geq N - \frac{n}{N} \ \wedge\ \mathit{even}((i \cdot N + j) \text{ div } n) \\ x(i + \frac{n}{N}, j), & \textbf{if } i < N - \frac{n}{N} \ \wedge\ \mathit{even}((i \cdot N + j) \text{ div } n) \\ x(0, j), & \textbf{if } i < \frac{n}{N} \ \wedge\ \neg\ \mathit{even}((i \cdot N + j) \text{ div } n) \\ x(i - \frac{n}{N}, j), & \textbf{if } i \geq \frac{n}{N} \ \wedge\ \neg\ \mathit{even}((i \cdot N + j) \text{ div } n) \end{array}$$

$$= [\ \mathit{even}((i \cdot N + j) \text{ div } n) \Rightarrow (i < N - \tfrac{n}{N}) \ \wedge\ ((i \cdot N + j) < N^2 - n),$$
$$\neg \mathit{even}((i \cdot N + j) \text{ div } n) \Rightarrow (i \geq \tfrac{n}{N}) \ \wedge\ ((i \cdot N + j) \geq n)]$$

$$\begin{array}{ll} x(N-1, N-1), & \textbf{if } (i \cdot N + j) \geq N^2 - n \ \wedge\ \ \mathit{even}((i \cdot N + j) \text{ div } n) \\ x(i + \frac{n}{N}, j), & \textbf{if } (i \cdot N + j) < N^2 - n \ \wedge\ \ \mathit{even}((i \cdot N + j) \text{ div } n) \\ x(0, 0), & \textbf{if } i \cdot N + j < n \ \wedge\ \neg\ \mathit{even}((i \cdot N + j) \text{ div } n) \\ x(i - \frac{n}{N}, j), & \textbf{if } i \cdot N + j > n \ \wedge\ \neg\ \mathit{even}((i \cdot N + j) \text{ div } n) \end{array}$$

$$= [\ g(i \cdot N + j + n) = (i + \tfrac{n}{N}, j) \text{ and } g(i \cdot N + j - n) = (i - \tfrac{n}{N}, j)\ ]$$

$$\begin{array}{ll} (x \circ g)(N^2 - 1), & \textbf{if } (i \cdot N + j) \geq N^2 - n \ \wedge\ \mathit{even}((i \cdot N + j) \text{ div } n) \\ (x \circ g)(i \cdot N + j + n), & \textbf{if } (i \cdot N + j) < N^2 - n \ \wedge\ \mathit{even}((i \cdot N + j) \text{ div } n) \\ (x \circ g)(0), & \textbf{if } i \cdot N + j < n \ \wedge\ \neg\ \mathit{even}((i \cdot N + j) \text{ div } n) \\ (x \circ g)(i \cdot N + j - n), & \textbf{if } i \cdot N + j \geq n \ \wedge\ \neg\ \mathit{even}((i \cdot N + j) \text{ div } n) \end{array}$$

$$\begin{aligned}
&= [\text{ fold } \mathit{SHL}_A \text{ and } \mathit{SHR}_A \text{]} \\
&\qquad \begin{array}{ll} (\mathit{SHL}_A\ n\ (x \circ g))(i \cdot N + j), & \textbf{if } \mathit{even}((i \cdot N + j) \text{ div } n) \\ (\mathit{SHR}_A\ n\ (x \circ g))(i \cdot N + j), & \textbf{otherwise} \end{array} \\
&= [\text{ fold } \mathit{JOIN}_A \text{]} \\
&\qquad (\mathit{JOIN}_A\ n\ s1\ s2)(i \cdot N + j) \\
&\qquad \textbf{where } (s1, s2) = \ (\mathit{SHL}_A\ n\ (x \circ g), \mathit{SHR}_A\ n\ (x \circ g)) \\
&= [\text{ fold } \mathit{CORR}_A \text{ and } g^{-1} \text{]} \\
&\qquad ((\mathit{CORR}_A\ n\ (x \circ g)) \circ g^{-1})(i, j)
\end{aligned}$$

□

A.5 Proof of Corollary 6

As a representative of the four propositions, we only proof

$$\mathit{CORR}_A\ n\ (x \circ g)) \circ g^{-1} = \mathit{CORR}_H\ n\ x$$

Let $i \in \{0, \ldots, 2^n - 1\}$.

$$\begin{aligned}
&\qquad (\mathit{CORR}_H\ n\ x)\ i \\
&= [\text{ unfold } \mathit{CORR}_H \text{]} \\
&\qquad (\mathit{JOIN}_H\ n\ (\mathit{COMMD}_H\ n\ x)\ (\mathit{COMMU}_H\ n\ x))\ i \\
&= [\text{ unfold } \mathit{JOIN}_H \text{]} \\
&\qquad \begin{array}{ll} (\mathit{COMMD}_H\ n\ x), & \textbf{if } \mathit{even}(i \text{ div } n) \\ (\mathit{COMMU}_H\ n\ x), & \textbf{otherwise} \end{array} \\
&= [\ \mathit{even}(i \text{ div } n) \Rightarrow i < (i \text{ div } 2n) \cdot 2n + n, \text{ unfold } \mathit{COMMD}_H, \mathit{COMMU}_H \text{]} \\
&\qquad \begin{array}{ll} x\ (i + n), & \textbf{if } \mathit{even}(i \text{ div } n) \\ x\ (i - n), & \textbf{otherwise} \end{array} \\
&= [\text{ fold } \mathit{SHR}_A \text{ and } \mathit{SHL}_A \text{]} \\
&\qquad \begin{array}{ll} (\mathit{SHL}\ n\ x)\ i, & \textbf{if } \mathit{even}(i \text{ div } n) \\ (\mathit{SHR}\ n\ x)\ i, & \textbf{otherwise} \end{array} \\
&= [\text{ fold } \mathit{JOIN}_A, \text{ fold } \mathit{CORR}_A \text{]} \\
&\qquad (\mathit{CORR}_A\ n\ x)\ i
\end{aligned}$$

□

A.6 Proof of Lemma 7

Let $i \in \{0, \ldots, N - 1\}$:

$$\begin{aligned}
&\qquad (\mathit{JOIN}_A\ n\ s\ (\mathit{ZIPWITH}\ (\oplus)\ (\mathit{DISTL}_A\ n\ (\mathit{CORR}_A\ n\ s))\ s))(i) \\
&= [\text{ unfold } \mathit{JOIN}_A \text{]} \\
&\qquad \begin{array}{ll} s(i), & \textbf{if } \mathit{even}(i \text{ div } n) \\ (\mathit{ZIPWITH}(\oplus)(\mathit{DISTL}_A\ n(\mathit{CORR}_A\ n\ s))\ s)(i), & \textbf{otherwise} \end{array}
\end{aligned}$$

We concentrate on the case $\neg even(i \text{ div } n)$ and start by unfolding *ZIPWITH*.

$$
\begin{aligned}
& \quad DISTL_A\ n(CORR_A\ n\ s))(i) \oplus s(i) \\
= & \text{[unfold } DISTL_A \text{]} \\
& \quad (CORR_A\ n\ s))((i \text{ div } n)n + n - 1) \oplus s(i) \\
= & \text{[unfold } CORR_A \text{]} \\
& \quad (JOIN_A\ n\ (SHL_A\ n\ s)(SHR_A\ n\ s))((i \text{ div } n)n + n - 1) \oplus s(i) \\
= & \text{[unfold } JOIN_A\text{, distributivity over conditional]} \\
& \quad (SHL_A\ n\ s) \\
& \qquad ((i \text{ div } n)n + n - 1) \oplus\ s(i), \quad \textbf{if } even(((i \text{ div } n)n + n - 1) \text{ div } n) \\
& \quad (SHR_A\ n\ s) \\
& \qquad ((i \text{ div } n)n + n - 1) \oplus s(i), \quad \textbf{otherwise} \\
= & \text{[} ((i \text{ div } n)n + n - 1) \text{ div } n = (i \text{ div } n) \text{]} \\
& \quad (SHR_A\ n\ s)((i \text{ div } n)n + n - 1) \oplus s(i) \\
= & \text{[unfold } SHR_A \text{]} \\
& \quad s(0), \qquad \textbf{if } (i \text{ div } n)n + n - 1 < n \\
& \quad s((i \text{ div } n)n + n - 1 - n) \oplus s(i), \quad \textbf{otherwise} \\
= & \text{[} (i \text{ div } n)n + n - 1 < n = (i \text{ div } n)n < 1 \text{]} \\
& \quad s(0), \qquad \textbf{if } (i \text{ div } n)n < 1 \\
& \quad s((i \text{ div } n)n + n - 1 - n) \oplus s(i), \quad \textbf{otherwise} \\
= & \text{[abstraction]} \\
& \quad (\lambda\ j\ .\ s(0), \qquad \textbf{if } j < 1 \\
& \qquad\quad s(j-1), \quad \textbf{otherwise}\ )((i \text{ div } n)n) \oplus s(i) \\
= & \text{[fold } SHR_A\ 1\text{, fold } DISTR_A \text{ and fold } ZIPWITH \text{]} \\
& \quad (ZIPWITH(\oplus)(DISTR_A\ n\ (SHR_A\ 1\ s))\ s)(i)
\end{aligned}
$$

Now putting the two cases together and folding $JOIN_A$ results in:

$$(JOIN_A\ n\ s(ZIPWITH\ (\oplus)\ (DISTR_A\ n\ (SHR_A\ 1\ s))\ s))(i)$$

□

B Example Implementation of Prefix Sums

We give an implementation of $psum_3$ (see 4.4) by means of an imperative parallel language, viz. Parallaxis [BBES92]. Parallaxis is a Modula-2 like imperative language with explicit parallel control constructs as well as communication operations. It is not dedicated to a particular architecture, but allows the user to specify a concrete one. Parallaxis follows the SIMD computation model, i.e. there is one control unit, which provides a single instruction stream to hundreds or thousands of PEs. According to that, Parallaxis distinguishes two kinds of variables: (a) scalar variables which reside on the control unit and (b) vector variables, which denote data elements spread over all PEs. Communication primitives are *PROPAGATE*, *RECEIVE* and *SEND* which can only be distinguished by their behavior on inactive PEs.

```
SYSTEM Prefix_Sum;
CONST N = 1024;   (* natural number, power of 2 *)
TYPE inat = [1..N];
(**** Architecture specification: linear array with N PEs ****)
CONFIGURATION list[1..N];
CONNECTION left: list[i] -> list[i-1].right;
           right: list[i] -> list[i+1].left;
(****  Definition of extended architecture skeletons: ****)
(****  JOIN, CORR and DISTR                            ****)
PROCEDURE JOIN (SCALAR n:inat; VECTOR s,t:INTEGER):VECTOR INTEGER;
   VECTOR res: INTEGER;
BEGIN IF EVEN((id_no - 1) DIV n) THEN res := s ELSE res := t END;
      RETURN res
END JOIN;
PROCEDURE CORR (SCALAR n:inat; VECTOR s:INTEGER):VECTOR INTEGER;
   VECTOR t,u: INTEGER;
BEGIN PROPAGATE.left^n (s,t); PROPAGATE.right^n (s,u);
      RETURN (JOIN(n,t,u))
END CORR;
PROCEDURE DISTR (SCALAR n:inat;VECTOR s:INTEGER):VECTOR INTEGER;
   SCALAR i: INTEGER;
BEGIN FOR i := 1 TO n DO
         IF (id_no - 1) MOD n # 0 THEN
            RECEIVE list.left(s) FROM list.right(s)
      END END; RETURN s
END DISTR;
(**** Computation of the parallel prefix sum ****)
PROCEDURE psum (SCALAR m,n:inat; VECTOR s:INTEGER):VECTOR INTEGER;
   VECTOR t: INTEGER;
BEGIN WHILE m # n DO PROPAGATE.right(s,t);  t := DISTR(n,t) + s;
                     s := JOIN(n,s,t); n := 2 * n
      END;  RETURN s
END psum;
(**** Main program ****)
BEGIN PARALLEL ...; s := psum(N,1,s); ... ENDPARALLEL END Prefix_Sum.
```