ELSEVIER

Science of Computer Programming 26 (1996) 59–78

Science of Computer Programming

Massive parallelization of divide-and-conquer algorithms over powerlists

Klaus Achatz, Wolfram Schulte

Fakultät für Informatik, Universität Ulm, D-89069 Ulm, Germany

Abstract

We present transformation rules to parallelize divide-and-conquer (DC) algorithms over powerlists. These rules convert the parallel control structure of DC into a sequential control flow, thereby making the implicit massive data parallelism in a DC scheme explicit. The results given here are illustrated by many examples including Fast Fourier Transform and Batcher's bitonic sort.

1. Introduction

It is well known that the main problem in exploiting the power of modern parallel systems is the development of correct, efficient and portable programs [8,14]. The most promising way to treat these problems seems to be a systematic, formal, top-down development of parallel software.

In this article we choose *transformational programming* to develop parallel programs, which is a methodology for constructing correct and efficient programs from formal specifications by applying meaning-preserving rules [13]. Starting with an operational specification, we derive programs for the *massively data parallel model*, which assumes a large data collection that needs to be processed by a number of processor elements (PEs), one for each member in the collection. The same set of instructions is concurrently applied to multiple data elements (SIMD), i.e., a single control flow guides the computation on all PEs.

As the *problem adequate structure*, we restrict ourselves to *powerlists* [10]. Many important data parallel algorithms, e.g., Fast Fourier Transform, Batcher's bitonic sort, and prefix-sum algorithms, have surprisingly concise recursive descriptions using powerlists. Moreover, simple algebraic properties of powerlists support a formal reasoning about these algorithms. Besides the usual functions over powerlists, certain high-level operations are introduced, which can be interpreted as communication operations on the machine level.

SSDI 0167-6423(95)00022-4

As the starting point of our strategy, we choose a very successful tactic for designing parallel algorithms: *divide-and-conquer* [17]. DC algorithms are particularly suited for parallel implementation, because the emerging subproblems can be solved independently and thus in parallel. Obviously, DC algorithms have explicit control parallelism, i.e., separate independent parts can be processed simultaneously by distinct CPUs. However, the SIMD model of computation does not allow concurrent control flows. Therefore, we aim at exploiting the inherent data parallelism. So, we present a set of semantics preserving *transformation rules*, which make the implicit data parallelism in a DC scheme over powerlists explicit and, by that, introduce topology independent communication operations on powerlists.

The rest of this article is organized as follows. Section 2 briefly presents the concept of powerlists. The new transformation rules towards a massively parallel computation are introduced in Section 3. In Section 4, we show the applicability of our approach with several examples. Finally, some remarks are given in Section 5.

2. Powerlists

Generally, lists can be used to express data parallelism in an abstract way [5]. Each element of the list resides on a particular PE. Data-parallel operations are defined as functions over lists, which are either abstractions of elementary communication-independent computations on all PEs or communication operations, which exchange values using the interconnection network. In this section we explore this approach. In particular, we introduce powerlists as defined by Misra [10], which permit succinct descriptions of DC algorithms.

2.1. Data structure

The basic data structure – suitable for describing DC algorithms – is the powerlist. A powerlist is either a list of one element or constructed by two powerlists of the same type and the same length. This always results in powerlists of length 2^n, for some $n \geqslant 0$, which restricts our theory but is appropriate, since all known massively parallel machines work with 2^n PEs.

Powerlists are joined in two different ways to create longer powerlists. If p, q are powerlists of the same length, then

- $p|q$ is the powerlist concatenating p and q, and
- $p \bowtie q$ is the powerlist formed by successively taking elements from p and q, starting with p.

Using two constructor sets allows us to formulate many algorithms on powerlists in a natural way. Algorithms over powerlists follow the DC paradigm, where each division yields two halves that can be processed in parallel.

In our examples, the sequence of elements of a powerlist is enclosed within angular brackets, thus $\langle 1 \rangle$ is the singleton containing the item 1, and $\langle [1], [2,3] \rangle$ is a powerlist

of length 2 containing two linear lists, which are enclosed in square brackets. The operation | is called tie, and $\bowtie$ is called zip. Examples of their use are:

$$\langle 0\rangle \mid \langle 1\rangle = \langle 0,1\rangle$$
$$\langle 0\rangle \bowtie \langle 1\rangle = \langle 0,1\rangle$$
$$\langle 0,1\rangle \mid \langle 2,3\rangle = \langle 0,1,2,3\rangle$$
$$\langle 0,1\rangle \bowtie \langle 2,3\rangle = \langle 0,2,1,3\rangle$$

Powerlists obey several laws, which are described in detail in [10].

2.2. *Parallel operations over powerlists*

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 $(f\ x)\ y$. The binding power of infix operations is lower than that of function applications. We define functions using pattern matching to decompose the argument list into its component parts. Decomposition uses | and $\bowtie$. If, in addition, assertions on parameters are used, they are given in the surrounding text.

The following functions over powerlists are used to specify programs. They will be removed during program development: the operator # returns the length of a powerlist. The first-order functions *first* and *last* extract the first and last element from a powerlist, respectively. A powerlist of n copies of identical elements is created by the function *copy*. Likewise, the function *dup* creates a powerlist of n identical powerlists.

As elementary data parallel operations, we provide the *parallel application* $*$ and the parallel conditional *join*. The apply-to-all operator $*$ applies a scalar function f to every element of a powerlist independently, and therefore reflects the massively data parallel programming paradigm in the most obvious way. Its definition is

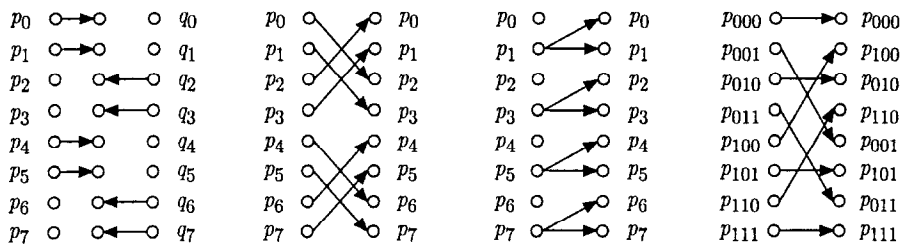
$$\begin{aligned} f * \langle x\rangle &= \langle f\ x\rangle \\ f * (p \mid q) &= f * p \mid f * q \end{aligned} \tag{1}$$

Also, it can be shown that

$$f * (p \bowtie q) = f * p \bowtie f * q \tag{2}$$

To shorten the presentation, the operator $*$ is also used to take an n-tuple of powerlists, having equal length, into a new powerlist in which corresponding elements are combined using any given n-ary scalar function.

In a data parallel environment, conditionals are different from their sequential counterparts. The action of a *parallel conditional* 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



(a) *join* 2 (b) *corr* 2 (c) *distL* 2 (d) *inv*

Fig. 1. Powerlist Operations: (a) *join* 2 $p\ q$, (b) *corr* 2 p, (c) *distL* 2 p, (d) *inv* p

operation *join* (see Fig. 1(a)). It takes a pair of powerlists p, q, having equal length, into a new powerlist, which consists of alternate slices of p and q each of length $n = 2^i$, $0 \leqslant i < \log_2(\#p)$. We define *join* by

$$\begin{aligned} &join\ n\ (p \mid q)\ (r \mid s) = p \mid s && \text{if } n = \#p \\ &join\ n\ (p \mid q)\ (r \mid s) = join\ n\ p\ r \mid join\ n\ q\ s && \text{if } n < \#p \end{aligned} \tag{3}$$

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. Section 3).

2.3. Communication oriented powerlist operations

A very wide range of scientific problems on powerlists can be computed under the DC scheme using a regular communication pattern. Naturally, some communication patterns are better suited than others for developing parallel algorithms. Essentially, they have structural properties that make it easier to describe the data movement operations necessary for parallel computations. The following communication operations may be the most suitable ones.

Correspondent communication – modeled by function *corr* $n\ p$ – 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 right. An example is depicted in Fig. 1(b). Its definition is straightforward:

$$\begin{aligned} &corr\ n\ (p \mid q) = q \mid p && \text{if } n = \#p \\ &corr\ n\ (p \mid q) = corr\ n\ p \mid corr\ n\ q && \text{if } n < \#p \end{aligned} \tag{4}$$

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, e.g., *distL* $n\ p$ 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:

$$\begin{aligned} distL\ n\ p &= copy\ n\ (last\ p) && \text{if } n = \#p \\ distL\ n\ (p \mid q) &= distL\ n\ p \mid distL\ n\ q && \text{if } n \leqslant \#p \end{aligned} \tag{5}$$

The powerlist operations *corr*, *distL*, *distR* and *join*, mirror the need of our DC scheme to exchange data among PEs and to select different data elements on each PE, respectively. They were defined using $|$. We have decided to take this representation as the natural one, because it reflects the row-major indexing on 2- and 3-dimensional array computers and also the indexing scheme on hypercubes, which are the most important data parallel architectures. Using $|$, construction and decomposition do not change the elements' order and therefore cause no communication costs.

Inversion exhibits a communication pattern induced by the construction or decomposition of a powerlist using $\bowtie$. The effect of inversion *inv* on the indices of p is depicted in Fig. 1(d): an element with index b in p has the reversal of bit string b in *inv* p. We have

$$\begin{aligned} inv\ \langle x \rangle &= \langle x \rangle \\ inv\ (p \mid q) &= inv\ p \bowtie inv\ q \end{aligned} \tag{6}$$

or, alternatively

$$\begin{aligned} inv\ \langle x \rangle &= \langle x \rangle \\ inv\ (p \bowtie q) &= inv\ p \mid inv\ q \end{aligned} \tag{7}$$

Operation *inv* is often used to permute the input or output of a computation that uses $\bowtie$, e.g., in the Fast Fourier Transform (cf. Section 4.6). Note that *inv* is rather expensive on array computers or hypercubes and therefore, should be avoided if possible.

2.4. Properties

Powerlists fulfill many properties, where especially the following two are needed in our parallelization rules (cf. Section 3): Let f denote a function, which maps powerlists to powerlists. The function is said to be *length preserving*, if the length of the output powerlist is equal to the length of the input powerlist: $\#(f\ p) \equiv \#p$. It is said to be *distributive*, if it distributes through concatenation of powerlists: $f\ (p \mid q) \equiv f\ p \mid f\ q$. The generalization to functions taking a tuple of powerlists yielding a single powerlist 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 powerlists to powerlists. If it distributes through a powerlist $p \mid q$, where $n \leqslant \#p$, then the function is said to be *distributive modulo n*, also called *slice-distributive*.

The family of apply-to-all operators are distributive, the family of apply-to-slices operators, *corr*, *distL*, *distR* and *join* are slice-distributive, and all previously mentioned operators and functions, mapping powerlists to powerlists, are length preserving.

3. Parallelization

The recursive structure of powerlists naturally leads to divide-and-conquer algorithms. In this section, we discuss the idea and assumption of our DC parallelization rules followed by their formal account. Finally, we formulate our parallelization strategy.

3.1. Idea

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

(i) If the input is primitive, solve it trivially.
(ii) Otherwise, recursively solve the subproblems, defined by each partition of the input.
(iii) Finally, compose the solutions of the different subproblems into a solution for the overall problem.

A typical instance of this pattern is the merge of two bitonic powerlists. A powerlist 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 powerlists $\langle 5,7,6,4\rangle$ and $\langle 8,3,2,5\rangle$ are both bitonic.

The function $bm\downarrow$ merges two bitonic sequences by applying the minimum and maximum function pointwise to the halves of the given powerlist and then sorting the results recursively.

$$bm\downarrow\ \langle x\rangle = \langle x\rangle$$
$$bm\downarrow(p \mid q) = bm\downarrow(\min * p\ q) \mid bm\downarrow(\max * p\ q)$$

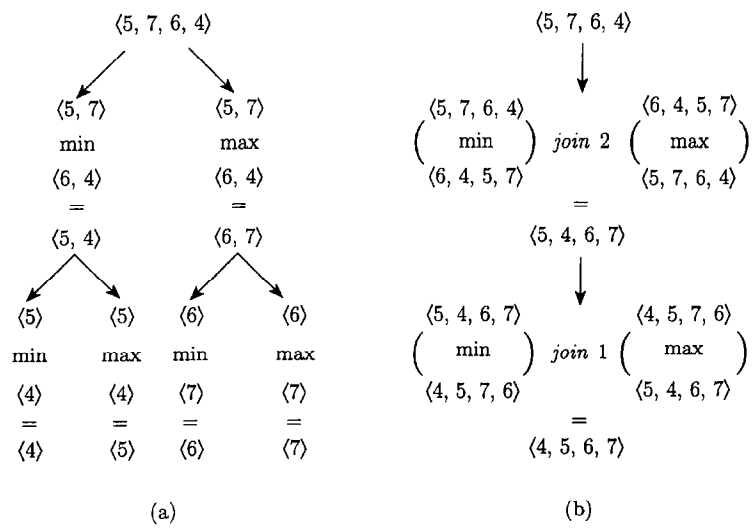


Fig. 2. (a) DC computation of $bm\downarrow\ \langle 5,7,6,4\rangle$, (b) data parallel computation of $bm\Downarrow\ 4\ \langle 5,7,6,4\rangle$.

Fig. 2(a) shows the computation for the expression $bm\downarrow\langle 5,7,6,4\rangle$, where the arrows represent recursive calls. Obviously, $bm\downarrow$ exhibits cascading recursion and explicit data decomposition using $|$. However, this pattern is not suitable for massively parallel execution, since it has multiple control flows. 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.

To this end, we transform $bm\downarrow\ p$ into the equivalent function $bm\Downarrow\ (\#p)\ p$. Fig. 2(b) shows the data parallel computation of $bm\Downarrow\ 4\ \langle 5,7,6,4\rangle$. The result of each call of $bm\Downarrow$ is the same as the result of the parallel calls of $bm\downarrow$, shown in the same row – modulo concatenation. However, the "way" the result is computed differs.

Contrary to the DC tactic, of which $bm\downarrow$ is a typical instance, the data parallel computation of $bm\Downarrow$ proceeds as follows:

(i) If the termination condition is reached, the problem is trivially solved.
(ii) Otherwise, one minimum and maximum computation on all data elements is performed, where correspondent slices are provided using *corr*, and the solutions of the different computations are composed into a partial solution for the overall problem using *join*.
(iii) Finally, the remaining subproblem is recursively solved.

The result of the transformation, whose rules are described below, yields the following computation:

$$\begin{aligned} bm\Downarrow\ 1\ p &= p \\ bm\Downarrow\ (2n)\ p &= bm\Downarrow\ n\ (join\ n\ (\min *\ p\ q)\ (\max *\ q\ p)) \\ &\quad \textbf{where}\ q = corr\ n\ p \end{aligned}$$

$bm\Downarrow$ includes an additional parameter n, which determines the recursion depth. Thus, it is not necessary anymore to decompose the powerlist p – its length remains constant. On the other hand, the pointwise applications of min and max have to be performed on the appropriate slices of length n. This is achieved by introducing correspondent communication $q = corr\ n\ p$. The scalar operations min and max are applied pointwise to p and q, and the resulting two powerlists are *join*ed by successively taking alternate slices of length n from the minimum and maximum list, starting with the former. A detailed description of the parallelization of Batcher's bitonic sort can be found in [1].

The function $bm\downarrow$ computes the result in a top-down fashion. Alternatively, DC algorithms can compute the result during the bottom-up phase, or even in both phases. In the following subsection, we will explore the different ways in which DC algorithms operate and present a set of transformation rules for their parallelization.

3.2. Rules

The parallelization rules will be represented as conditional equations between higher-order functions.

3.2.1. Input patterns

We define the DC input patterns as the following higher-order functions: $F\downarrow$ exhibits a top-down, and $F\uparrow$ a bottom-up computation. The arguments $\smile$ and $\frown$ stand for the destructor and constructor, respectively, and can either be $|$ or $\bowtie$. $F\downarrow$ has the following form:

$$\begin{aligned} & F\downarrow \ \smile \frown \ \delta\ t\ l\ r = f\downarrow \\ & \textbf{where} \quad f\downarrow\ s\ \langle x\rangle = \langle t\ s\ x\rangle \\ & \qquad f\downarrow\ s\ (p \smile q) = f\downarrow\ (\delta\ s)\ (l\ s\ p\ q) \frown f\downarrow\ (\delta\ s)\ (r\ s\ p\ q) \end{aligned} \tag{8}$$

If the input is a singleton list, the problem is solved trivially by t, otherwise the input is decomposed using $\smile$, the subproblems are *preadjusted* by l and r, solved recursively, and then constructed using $\frown$. Additionally, the scalar s is updated by applying the function δ. It is used as an additional parameter for the trivial and preadjust functions (cf. below).

The function $bm\downarrow$ can be expressed as an instance of $F\downarrow$. However, $bm\downarrow$ does not use the scalar s and consequently does not need the function δ. So, we choose as arbitrary value $\cdot$ for s and use the identity function id for δ:

$$\begin{aligned} & bm\downarrow \equiv F\downarrow \,||\, \mathrm{id}\ t\ l\ r\ \cdot \\ & \qquad \textbf{where} \quad t\ s\ x = x \\ & \qquad\qquad l\ s\ p\ q = \min*\ p\ q \\ & \qquad\qquad r\ s\ p\ q = \max*\ p\ q \end{aligned}$$

The functional $F\uparrow$ has the same parameter list as $F\downarrow$:

$$\begin{aligned} & F\uparrow \ \smile \frown \ \delta\ t\ l\ r = f\uparrow \\ & \textbf{where} \quad f\uparrow\ s\ \langle x\rangle = \langle t\ s\ x\rangle \\ & \qquad f\uparrow\ s\ (p \smile q) = l\ s\ v\ w \frown r\ s\ v\ w \\ & \qquad\qquad\qquad \textbf{where}\ (v,w) = (f\uparrow\ (\delta\ s)\ p, f\uparrow\ (\delta\ s)\ q) \end{aligned} \tag{9}$$

As in the top-down case, if the input is a singleton list, the problem is solved trivially by t. Otherwise the input is decomposed, the scalar s is computed using δ and the subproblems are recursively solved. During the bottom-up phase the subsolutions are *postadjusted* by l and r and the powerlist is (re)constructed.

The update function δ in $F\downarrow$ and $F\uparrow$ is computed before the recursive calls take place and thus cannot be merged with the bottom-up computation of (9) – the additional parameter s and δ increase the power of the input patterns. Additionally, to exploit full parallelism, the same update function δ must be applied to s in both recursive calls.

In $F\downarrow$ and $F\uparrow$ it is assumed that the preadjust and postadjust functions are length preserving. This is a perfectly reasonable assumption, since every element of a powerlist resides on a particular PE and we can neither create nor delete PEs.

These patterns are powerful because the preadjust and postadjust functions receive the complete input and output sequence, respectively. 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. However, algorithms that are either not balanced or depend on particular values of the powerlist 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.

3.2.2. Rules: Replacing zip by tie

Many algorithms can be expressed quite nicely using $\bowtie$ as constructor or destructor. However, we already observed in Section 2.3 that $\bowtie$ is often unnatural when used as a basis for parallelism. Thus, we should transform $\bowtie$ to $|$. This is the task of the communication primitive *inv*.

If in $F\downarrow$ and $F\uparrow$ the constructor and destructor, respectively, is $\bowtie$, we replace it by $|$ and permute the output and input, respectively, into the right order using *inv*.

$$\begin{aligned} F\downarrow \ \frown \bowtie \delta\ t\ l\ r\ s\ p &\equiv inv\ (F\downarrow \ \frown \mid \delta\ t\ l\ r\ s\ p) \\ F\uparrow \bowtie \ \frown \ \delta\ t\ l\ r\ s\ p &\equiv F\uparrow \mid \ \frown \ \delta\ t\ l\ r\ s\ (inv\ p) \end{aligned} \tag{10}$$

Moreover, if the adjust functions l and r are distributive, similar equations hold for $F\downarrow$ and $F\uparrow$ using $\bowtie$ as their destructor and constructor, respectively,

$$\begin{aligned} F\downarrow \bowtie \ \frown \ \delta\ t\ l\ r\ s\ p &\equiv F\downarrow \mid \ \frown \ \delta\ t\ l\ r\ s\ (inv\ p) \\ F\uparrow \ \frown \bowtie \delta\ t\ l\ r\ s\ p &\equiv inv\ (F\uparrow \ \frown \mid \delta\ t\ l\ r\ s\ p) \end{aligned} \tag{11}$$

If constructor and destructor are both formulated using $\bowtie$, we can apply the rules from above. However, we can specialize the derived rules under the assumption, that the adjust functions are distributive. If this condition holds, the adjust functions do not exchange values and thus can be computed using either $|$ or $\bowtie$, (cf. (1) and (2)). If, in addition, we reverse the computation order, i.e., from top-down to bottom-up or vice versa, we can eliminate the pre- and postprocessing by *inv*, since $inv \circ inv \equiv \mathrm{id}$. Thus, a top-down (bottom-up) computation using $\bowtie$ is equivalent to a bottom-up (top-down) computation using $|$, provided δ is bijective. We have for a powerlist p of length 2^n with $n > 0$:

$$\begin{aligned} F\downarrow \bowtie \bowtie \delta\ t\ l\ r\ s\ p &\equiv (t(\delta^n s)) * (F\uparrow \mid \mid \delta^{-1}\ t'\ l\ r(\delta^{n-1}s)\ p) \\ F\uparrow \bowtie \bowtie \delta\ t\ l\ r\ s\ p &\equiv F\downarrow \mid \mid \delta^{-1}\ t'\ l\ r(\delta^{n-1}s)((t(\delta^n s)) * p) \end{aligned} \tag{12}$$

where $t'\ s = \mathrm{id}$. When the computation changes from $F\downarrow$ to $F\uparrow$, we have to apply the trivial function t, which is performed in the base case of $F\downarrow$, to the result of $F\uparrow$. The actual parameter of t must be the value of s at termination of $F\downarrow$, i.e., $\delta^n\ s$. $F\uparrow$ is started with the actual value $\delta^{n-1}\ s$, which is the last value of s in $F\downarrow$ before termination. Since the direction of the computation is swapped, the sequence of actual values of s is reversed by applying δ^{-1} to s. The corresponding fact holds for the dual case, changing the bottom-up computation into a top-down one.

This equivalence is used quite often. For instance, the bitonic merge algorithm is typically presented as a bottom-up computation using $\bowtie$, see e.g. [3].

The rules (10) and (11) can be proved using structural induction on the argument powerlist, whereas (12) is proved by computational induction (cf. [2]).

3.2.3. Output patterns

After having eliminated $\bowtie$, we now present the output patterns of our parallelization rules, which implicitly use $|$ as their basic constructor.

The function $F\Downarrow$ describes the tail-recursive top-down computation with pre-adjustment, while $F\Uparrow$ is the tail-recursive bottom-up computation with post-adjustment. $F\Downarrow$ is defined by

$$
\begin{aligned}
&F\Downarrow\ \delta\ t\ l\ r\ s\ p = f\Downarrow\ \#p\ s\ p\\
&\textbf{where}\\
&\quad f\Downarrow\ 1\ s\ p\ \ = (t\ s) * p\\
&\quad f\Downarrow\ (2n)\ s\ p = f\Downarrow\ n\ (\delta s)\ (join\ n\ ((l\ s) *_n\ p\ q)\ ((r\ s) *_n\ q\ p)\\
&\qquad\qquad\textbf{where}\ q = corr\ n\ p
\end{aligned}
\tag{13}
$$

In the base case, where $n = 1$, the trivial function t is mapped over p. In the recursive case, where $n \geqslant 2$, the preadjust functions l and r are applied over slices of length n of the argument powerlist p and the permuted powerlist $q = corr\ n\ p$. Then the different subsolutions are combined using *join n*.

Mapping the adjust functions over slices is done by the auxiliary apply-to-slices operator $*_n$. It is a variation of the apply-to-all operator and takes a powerlist, in which corresponding slices of fixed length are combined using any length preserving function, into a new powerlist, i.e.:

$$
\begin{aligned}
f *_n\ p &= f\ p && \text{if } n = \#p\\
f *_n\ (p \mid q) &= f *_n\ p \mid f *_n\ q && \text{if } n \leqslant \#p
\end{aligned}
\tag{14}
$$

This operator is only used during intermediate steps of the development and is eliminated, if the adjust functions are slice-distributive (cf. Section 3.3). A generalization to operators taking n-tuples of powerlists is straightforward.

The function $bm\Downarrow$ can be expressed as an instance of $F\Downarrow$, namely,

$$
\begin{aligned}
bm\Downarrow\ \#p\ p \equiv F\Downarrow\ \mathrm{id}\ t\ l\ r\ \cdot\ p\\
\textbf{where}\quad t\ s\ x = x\\
l\ s\ p\ q = \min *\ p\ q\\
r\ s\ p\ q = \max *\ p\ q
\end{aligned}
$$

The functional $F\Uparrow$ has the following form:

$$
\begin{aligned}
&F\Uparrow\ \delta\ l\ r\ s\ p = f\Uparrow\ \#p\ 1\ s\ p\\
&\textbf{where}\\
&\quad f\Uparrow\ 1\ n\ s\ p\ =\ p\\
&\quad f\Uparrow\ (2m)\ n\ s\ p = f\Uparrow\ m\ (2n)\ (\delta s)\ (join\ n\ \ ((l\ s) *_n\ \ p\ q)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad ((r\ s) *_n\ \ q\ \ p))\\
&\qquad\qquad\textbf{where}\ q = corr\ n\ p
\end{aligned}
\tag{15}
$$

Contrary to $F\Downarrow$, $F\Uparrow$ starts the iteration with slice length 1 and duplicates the slice length n in each step. The additional argument m solely controls termination.

3.2.4. Rules: Parallelization

As already apparent from the running example, we have the following correspondence between $F\downarrow$ and $F\Downarrow$:

$$F\downarrow\ |\ | \equiv F\Downarrow \tag{16}$$

The cascading recursion of $F\downarrow\ ||$ is equivalent to the tail-recursive computation of $F\Downarrow$ suitable for massively parallel processing.

The correlation of the bottom-up computation $F\uparrow$ and the tail-recursive computation $F\Uparrow$ is more complicated due to the scalar function δ. In $F\uparrow$ the scalar s is computed during the top-down phase, but used during the bottom-up phase. However, in $F\Uparrow$ there is only a single pass. This problem can be overcome, by calling $F\Uparrow$ with the value $\delta^{n-1}s$, where $\#p = 2^n$, which is the last value of s in $F\uparrow$ before termination, and by computing the actual parameter s in reverse order, i.e., by applying δ^{-1} to s. In addition, p is preprocessed by t, using as its actual parameter $\delta^n\ s$, i.e., the value of s at termination of $F\uparrow$. We have

$$F\uparrow\ ||\ \delta\ t\ l\ r\ s\ p \equiv F\Uparrow\ \delta^{-1}\ l\ r\ (\delta^{n-1}\ s)\ ((t\ (\delta^n\ s)) * p) \tag{17}$$

Note that the above transformation rules can considerably be simplified for particular instances of the parameters. For example, if the parameter s is not needed, function δ and all applications of s can be eliminated.

The rules of this subsection can be proved by computational induction, provided that the adjust functions are length preserving (cf. [2]).

3.3. Strategy

The overall goal of our approach is to transform a given DC algorithm on powerlists to an efficient tail-recursive computation suitable for massively parallel processing. Let f be a given DC algorithm on powerlists, $f\updownarrow$ and $f\Updownarrow$ the according DC functions of the input and output patterns for our transformation rules, where $\updownarrow$ and $\Updownarrow$ indicate either a top-down or bottom-up computation. The strategy consists of five steps, which are discussed in detail below:

(i) Modify f, to match the input $f\updownarrow$.
(ii) Rewrite $f\updownarrow$, to replace $\bowtie$ by $|$.

(iii) Parallelize $f\updownarrow$ to $f\Updownarrow$.
(iv) Specialize $f\Updownarrow$, to eliminate $*_n$.
(v) Optimize $f\Updownarrow$, to increase parallelism.

Ad (i). Many algorithms on powerlists do not a priori match the input pattern $f\updownarrow$, either because they only compute a scalar instead of a powerlist, or they take only scalar arguments instead of scalars and a powerlist. These algorithms can simply be reformulated. If the result of f is a scalar, distribute the result to all PEs, i.e., $f\updownarrow\ p \equiv$ *copy* $\#p\ (f\ p)$. If the input of f is only a scalar s instead of a powerlist, two cases can be distinguished. First, if s only controls termination, it can be replaced by a powerlist of appropriate length, i.e., $f\updownarrow\ (copy\ \#(f\ s)\ \cdot) \equiv f\ s$, where $\cdot$ denotes an arbitrary value. Second, if s contributes to the computation, an additional powerlist parameter must be introduced, i.e., $f\updownarrow\ s\ (copy\ \#(f\ s)\ \cdot) \equiv f\ s$. Finally, note that matching requires the adjust functions to be length preserving.
Ad (ii). Powerlist algorithms can nicely be expressed using either $|$ or $\bowtie$. However, we decided to take the former as the primitive one (cf. Section 2.3). Therefore, the $\bowtie$ operations of $f\updownarrow$ are transformed into $|$ using the rules of Section 3.2.2.
Ad (iii). Having eliminated $\bowtie$, we can apply our parallelization rules (16) and (17), which make the implicit parallelism in $f\updownarrow$ explicit, by transforming the cascading recursion into tail-recursion and by introducing topology independent communication operations in $f\Updownarrow$.
Ad (iv). To apply the adjust function concurrently on all PEs in $f\Updownarrow$, the apply-to-slices operator $*_n$ should be eliminated. Let f be a length preserving adjust function. Two cases can be distinguished: if f is distributive, then $f *_n p \equiv f * p$ holds. If f has a slice-distributive generalization f', with property $f'\ \#p\ p \equiv f\ p$, then we have: $f *_n p \equiv f'\ n\ p$. The generalization to functions taking a tuple of powerlists and yielding a single powerlist is straightforward. Note that by definition the derived functions $f\Updownarrow$ of $f\updownarrow$ are slice-distributive. We have $f\Downarrow\ \#p\ p \equiv f\downarrow\ p$ and $f\Uparrow\ \#p\ 1\ p \equiv f\uparrow\ p$. Sometimes the adjust functions have to be rewritten, so that they become either distributive or slice-distributive. If the adjust function f uses first or last communication, e.g., is defined by $f\ p\ q = ((last\ p)\oplus) * q$, then f has the following slice-distributive generalization using broadcast: $f'\ n\ p\ q = \oplus * (distL\ n\ p)\ q$. If f with body e only depends on the length of its argument powerlists, then we define a new function $\hat{f}$, where $\hat{f}\ n = e[n \text{ for } \#p]$ such that $\hat{f}\ \#p \equiv f\ p$. The slice-distributive generalization f' of f then reads: $f'\ n\ p = dup\ (\frac{\#p}{n})\ (\hat{f}\ n)$.
Ad (v). Finally, the adjust functions l and r of the output pattern of $f\Updownarrow$ may share computations that will be processed sequentially, although they could be merged and processed concurrently. This is the aim of our horizontal fusion tactic. Using the "forget" property of $join\ n\ p\ q$, which only takes half the elements of p and q, respectively, we can transform the output pattern of $f\Updownarrow$, to increase parallelism further. If p and q are equal then *join* has no effect, i.e., $join\ n\ p\ p \equiv p$. If p and q are preprocessed by the same slice-distributive function h, then these two preprocessing steps can be merged into one: $join\ n\ (h\ n\ p)\ (h\ n\ q) \equiv h\ n\ (join\ n\ p\ q)$. If p and q are processed by

distinct slice-distributive functions g and h, then by applying $join\ n\ (g\ n\ p)\ (h\ n\ q) \equiv join\ n\ (g\ n\ (join\ n\ p\ q))\ \ (h\ n\ (join\ n\ p\ q))$, we can try to merge the computations of p and q. Again, the last two equations can be generalized to adjust functions taking tuples of powerlists. Note that horizontal fusion leads to an optimization only if the shared computations are more expensive than the operation *join*.

This strategy is powerful, although it is not complete, i.e., we cannot prove that it always succeeds. However, we did not find a useful function where the strategy has failed.

4. Examples

We show the usefulness of the presented strategy by several algorithms on powerlists, which will be parallelized to work on massively parallel computers. As input, we mainly choose the examples given in [10]. These include such well-known examples as the Parallel Prefix and the Fast Fourier Transform. A more complex example, namely the construction of a convex hull is treated in [1].

To shorten the presentation, superfluous parameters of the input and output patterns, given in Section 3.2, are omitted and input and output instances are presented in a simplified form. Only in the first example, the derivation is carried out in detail. In later subsections, the strategy is applied without explicitly being mentioned.

4.1. Prefix sum

One of the simplest and most useful building blocks for parallel algorithms is the *parallel prefix* function [4,6], which takes a binary, associative operator $\oplus$, a powerlist p of n scalar elements $\langle p_0, \ldots, p_{n-1}\rangle$ and returns the powerlist $\langle p_0, p_0 \oplus p_1, \ldots, p_0 \oplus p_1 \oplus \ldots \oplus p_{n-1}\rangle$.

The following function $ps\uparrow$ defines a scheme for computing the prefix sum of a powerlist p:

$$\begin{aligned} ps\uparrow \oplus \langle x\rangle &= \langle x\rangle \\ ps\uparrow \oplus (p \mid q) &= v \mid ((last\ v)\oplus) * w \\ &\quad \textbf{where } (v, w) = (ps\uparrow \oplus p, ps\uparrow \oplus q) \end{aligned}$$

We follow the five steps of our strategy (cf. Section 3.3) to derive a massively parallel prefix sum computation.

Step (i): Apart from the missing parameter s, $ps\uparrow\ \oplus$ matches the input pattern (9) with the following instantiation:

$$\begin{aligned} \smile &\equiv \mid \\ \frown &\equiv \mid \\ t\ x &\equiv x \\ l\ p\ q &\equiv p \\ r\ p\ q &\equiv ((last\ p)\oplus) * q \end{aligned}$$

Step (ii): Since there is no $\bowtie$-constructor or -destructor in $ps\uparrow$, we can immediately proceed with the next step.
Step (iii): The cascading bottom-up computation of $ps\uparrow \oplus$ is replaced by a tail-recursion using (17), which results in:

$$
\begin{aligned}
&ps\uparrow \oplus p \equiv ps\Uparrow \#p\ 1\ p\\
&\textbf{where}\quad ps\Uparrow\ 1\ n\ p = p\\
&\qquad ps\Uparrow\ (2m)\ n\ p = ps\Uparrow\ m\ (2n)\ (join\ n\ (l *_n p\ q)\ (r *_n q\ p))\\
&\qquad\qquad \textbf{where } q = corr\ n\ p
\end{aligned}
$$

Step (iv): The post-adjust function l as the projection on the first argument is obviously distributive. Thus, we obtain $l *_n p\ q \equiv l * p\ q \equiv p$. A closer inspection of r reveals that it is neither distributive nor slice-distributive. According to our strategy, a slice-distributive generalization of r is: $r'\ n\ p\ q = \oplus * (distL\ n\ p)\ q$. Thus, we have $r *_n p\ q \equiv r'\ n\ p\ q$.
Step (v): l and r have no computation in common. Therefore, a further optimization using horizontal fusion has no effect. The final version reads:

$$
\begin{aligned}
&ps\Uparrow\ 1\ n\ p = p\\
&ps\Uparrow\ (2m)\ n\ p = ps\Uparrow\ m\ (2n)\ (join\ n\ p\ (\oplus * (distL\ n\ q)\ p))\\
&\qquad\qquad \textbf{where } q = corr\ n\ p
\end{aligned}
$$

4.2. Reversal

We continue with a most simple function $rev\downarrow$, which reverses the order of elements of the argument powerlist. The input scheme

$$
\begin{aligned}
&rev\downarrow\ \langle x\rangle = \langle x\rangle\\
&rev\downarrow\ (p \mid q) = rev\downarrow\ q \mid rev\downarrow\ p
\end{aligned}
$$

is transformed to a parallel version by applying (16), which, after horizontal fusion, results in:

$$
\begin{aligned}
&rev\downarrow\ p = rev\Downarrow\ \#p\ p\\
&\textbf{where}\quad rev\Downarrow\ 1\ p = p\\
&\qquad rev\Downarrow\ (2n)\ p = rev\Downarrow\ n\ (corr\ n\ p)
\end{aligned}
$$

4.3. Reduction

Reduction is a higher-order function that takes a binary, associative operator $\oplus$ and a powerlist $p = \langle p_0, \ldots, p_{n-1}\rangle$ yielding $(p_0 \oplus \ldots \oplus p_{n-1})$. Its definition over powerlists is

$$
\begin{aligned}
&red \oplus \langle x\rangle = x\\
&red \oplus (p \mid q) = (red \oplus p)\ \oplus\ (red \oplus q)
\end{aligned}
$$

This scheme does not quite fit our pattern, because the result is a single element instead of a whole powerlist. According to our strategy, we start with $red\uparrow \oplus p = copy\ \#p\ (red \oplus p)$, which easily can be shown to have the following recursive definition:

$$\begin{aligned} red\uparrow \oplus \langle x\rangle &= \langle x\rangle \\ red\uparrow \oplus (p \mid q) &= u \mid u \\ &\quad \textbf{where } u = \oplus * (red\uparrow \oplus p)(red\uparrow \oplus q) \end{aligned}$$

Applying (17) in addition with horizontal fusion, we obtain the parallel version

$$\begin{aligned} &red\uparrow \oplus\ p = red\Uparrow \oplus \#p\ 1\ p \\ &\textbf{where} \\ &\quad red\Uparrow \oplus 1\ n\ p = p \\ &\quad red\Uparrow \oplus (2m)\ n\ p = red\Uparrow \oplus m\ (2n)\ (\oplus *\ (join\ n\ p\ q)\ (join\ n\ q\ p)) \\ &\qquad\qquad \textbf{where } q = corr\ n\ p \end{aligned}$$

which is a useful building block for many parallel algorithms.

Alternatively, reduction can be implemented using $ps\uparrow$ followed by a directed broadcast: $red\uparrow \oplus p \equiv distL\ \#p\ (ps\uparrow \oplus p)$.

4.4. Gray code

A Gray code sequence for $n \geqslant 0$ is a sequence of 2^n n-bit strings, where consecutive strings in the sequence differ in exactly one bit position. For example, a Gray code sequence for $n = 2$ reads: $\langle[00],[01],[11],[10]\rangle$. The last and the first strings in the sequence are also considered consecutive. The respective powerlist algorithm is computed by function G, for any n:

$$\begin{aligned} G\ 0 &= \langle[\ ]\rangle \\ G\ (n+1) &= (0:) * (G\ n) \mid (1:) * (rev\downarrow\ (G\ n)) \end{aligned}$$

Here, $(0:)$ and $(1:)$ are functions over strings that take a string as argument and append 0 or 1, respectively, as its prefix. Again, this algorithm must be modified to fit the input pattern of our transformation rules (see Section 3.3). Since the parameter n is only used for termination, we substitute n by a powerlist of length 2^n: $G\ n \equiv G\uparrow\ (copy\ 2^n\ \cdot)$. We get the following definition:

$$\begin{aligned} G\uparrow \langle x\rangle &= \langle[\ ]\rangle \\ G\uparrow\ (p \mid q) &= (0:) * (G\uparrow\ p) \mid (1:) * (rev\downarrow\ (G\uparrow\ q)) \end{aligned}$$

By (17), this algorithms is equivalent to:

$$\begin{aligned} &G\uparrow\ p = G\Uparrow\ \#p\ 1\ (copy\ \#p\ [\ ]) \\ &\textbf{where} \quad G\Uparrow\ 1\ n\ p = p, \\ &\qquad G\Uparrow\ (2m)\ n\ p = G\Uparrow\ m\ (2n)\ (join\ n\ ((0:) * p)\ ((1:) * (rev\Downarrow\ n\ p)) \end{aligned}$$

Note that $copy\ \#p\ [\ ]$ initializes all PEs with the empty string $[\ ]$.

4.5. Polynomial

A polynomial $P(w) = \sum_{i=0}^{2^n-1} p_i \times w^i$, where $n \geqslant 0$, can be represented by a powerlist p whose ith element is p_i. For $n > 0$, the evaluation of P at some point w is equivalent to

$$\sum_{i=0}^{2^{n-1}-1} p_{2i} \times w^{2i} + \sum_{i=0}^{2^{n-1}-1} p_{2i+1} \times w^{2i+1}$$

The following function ep uses this strategy to evaluate $P(w)$.

$$\begin{aligned} ep\ w\ \langle x \rangle &= x \\ ep\ w\ (p \bowtie q) &= (ep\ w^2\ p) + w \times (ep\ w^2\ q) \end{aligned}$$

According to our strategy, we distribute the result among all PEs, i.e., $ep{\uparrow}\ w\ p = copy\ \#p\ (ep\ w\ p)$, yielding:

$$\begin{aligned} ep{\uparrow}\ w\ \langle x \rangle &= \langle x \rangle \\ ep{\uparrow}\ w\ (p \bowtie q) &= +*(ep{\uparrow}\ w^2\ p)\ ((w\times)*(ep{\uparrow}\ w^2\ q)) \mid \\ &\quad +*(ep{\uparrow}\ w^2\ p)\ ((w\times)*(ep{\uparrow}\ w^2\ q)) \end{aligned}$$

Different from the parameter n in the previous subsection, w does not guide the flow of control, but is needed for computation. To obtain a parallel version of the above algorithm, we first eliminate $\bowtie$ using (10). The final parallel algorithm is derived with (17) and subsequent horizontal fusion:

$$ep{\uparrow}\ w\ p = ep{\Uparrow}\ \#p\ 1\ w^{2(\log_2(\#p)-1)}\ (inv\ p)$$

where

$$\begin{aligned} ep{\Uparrow}\ 1\ n\ w\ p &= p \\ ep{\Uparrow}\ (2m)\ n\ w\ p &= ep{\Uparrow}\ m\ (2n)\ \sqrt{w}\ (+*\ (join\ n\ p\ q) \\ &\qquad ((w\times)*(join\ n\ q\ p)) \\ &\quad \textbf{where}\ q = corr\ n\ p \end{aligned}$$

The scalar w as part of the postadjust functions is needed to compute the values on all PEs. Thus, it should not be calculated on the control unit but concurrently on all PEs. This can easily be established by embedding w into a powerlist with the same length as p. Note that due to the simplicity of the elementary operations $+$ and $\times$, horizontal fusion may have no optimization effect.

4.6. Fast Fourier transform

A continuous function $f(x)$ is discretized into a sequence $\langle f(x_0), f(x_0 + \Delta x), \ldots, f(x_0 + (N-1)\Delta x) \rangle$ by taking N samples Δx units apart. For $N = 2^n$, this sequence can be represented as a powerlist $p = \langle p_0, \ldots, p_{N-1} \rangle$, where $p_i = f(x_0 + i\Delta x)$,

$i = 0,\ldots,N-1$. The discrete Fourier transform of p is given by the powerlist q, where

$$q_k = \frac{1}{N}\sum_{i=0}^{N-1} p_i \times w_N^{ik} \equiv \frac{1}{N}\sum_{i=0}^{N/2-1} p_{2i} \times w_N^{(2i)k} + \frac{1}{N}\sum_{i=0}^{N/2-1} p_{2i+1} \times w_N^{(2i+1)k}$$

for $k = 0,\ldots,N-1$ and $w_N = \exp(-j\frac{2\pi}{N})$ is the Nth principal root of 1. The Fast Fourier transform algorithm depends on the method of successive doubling, which can be summarized by the following DC algorithm on powerlists:

$$\begin{aligned}
&FFT\ p = \left(\frac{1}{\#p}\times\right) * (FFT\uparrow\ p)\\
&\textbf{where}\\
&\qquad FFT\uparrow\ \langle x\rangle = \langle x\rangle\\
&\qquad FFT\uparrow\ (p \bowtie q) = +*(FFT\uparrow\ p)\ (\times*(powers\ p)\ (FFT\uparrow\ q)) \mid\\
&\qquad\qquad\qquad\qquad -*(FFT\uparrow\ p)\ (\times*(powers\ q)\ (FFT\uparrow\ q))\\
&\qquad powers\ p = \left(\lambda\ k.\exp\left(-j\frac{\pi k}{\#p}\right)\right) * \langle 0,\ldots,\#p-1\rangle
\end{aligned}$$

A detailed description of the Fast Fourier Transform can be found, e.g., in [18].

To get a parallel version of $FFT\uparrow$, we first eliminate the $\bowtie$-destructor by means of (10) and then apply (17). After optimization, the final version is

$$\begin{aligned}
&FFT\uparrow\ p = FFT\Uparrow\ \#p\ 1\ (inv\ p)\\
&\textbf{where}\\
&\qquad FFT\Uparrow\ 1\ n\ p = p\\
&\qquad FFT\Uparrow\ (2m)\ n\ p = FFT\Uparrow\ m\ (2n)\ (join\ n\ (+*c\ d)\ (-*c\ d))\\
&\qquad\textbf{where}\qquad q = corr\ n\ p\\
&\qquad\qquad\qquad\quad c = join\ n\ p\ q\\
&\qquad\qquad\qquad\quad d = \times*w\ (join\ n\ q\ p)\\
&\qquad\qquad\qquad\quad w = dup\ \frac{\#p}{n}\ (powers'\ n)\\
&\qquad\qquad powers'\ n = \left(\left(\lambda\ k.\exp\left(-j\frac{\pi k}{n}\right)\right) * \langle 0,\ldots,n-1\rangle\right)
\end{aligned}$$

Since $dup\ \frac{\#p}{n}\ (powers'\ n) \equiv (\lambda\ k.\exp(-j\frac{\pi(k \bmod n)}{n})) * \langle 0,\ldots,\#p-1\rangle$, powerlist w can be computed locally on every PE using its local processor index k.

The inverse of the Fourier transform, IFT, can be defined similarly to the FFT. [10] derives a definition of IFT from that of FFT. The solution of this derivation yields the following algorithm:

$$\begin{aligned}
&IFT\ p = (\#p\times) * (IFT\downarrow\ p)\\
&\textbf{where}\qquad IFT\downarrow\ \langle x\rangle = \langle x\rangle\\
&\qquad IFT\downarrow\ (p \mid q) = IFT\downarrow\ ((0.5\times) * (+*p\ q)) \bowtie\\
&\qquad\qquad\qquad\qquad IFT\downarrow\ ((0.5\times) * (\times*(-*p\ q)\ (powers^{-1}\ p)))\\
&\qquad powers^{-1}\ p = \left(\lambda\ k.\exp\left(j\frac{\pi k}{\#p}\right)\right) * \langle 0,\ldots,\#p-1\rangle
\end{aligned}$$

Similar to the development of a parallel version for *FFT*, we first eliminate the $\bowtie$-constructor with (10). Afterwards, the final massively parallel solution is obtained by applying (16):

$$
\begin{array}{lll}
IFT\ p = inv((\#p\times) * (IFT\Downarrow\ \#p\ p)) & & \\
\textbf{where} & IFT\Downarrow\ 1\ p & = p \\
& IFT\Downarrow\ (2n)\ p & = IFT\Downarrow\ n\ ((0.5\times) * (join\ n\ a\ b)) \\
& \textbf{where} \quad q & = corr\ n\ p \\
& (a,b) & = (+ * p\ q, \times * (- * q\ p)\ w) \\
& w & = \left(\lambda\ k.\exp\left(j\frac{\pi(k \bmod n)}{n}\right)\right) * \langle 0,\ldots,\#p-1\rangle
\end{array}
$$

5. Remarks

5.1. Related work

Massively data parallel algorithms apply the same set of operations to a huge collection of data. Among the first, who recognized this programming paradigm, although in an imperative setting, were Preparata and Vullemin [15]. They show that this paradigm can be used to express many known algorithms and present their implementation on Cube Connected Cycle topologies.

Mou and Houdak describe DC in an algebraic model called Divacon [11]. They recognize that the original DC model is too restrictive with respect to decomposition and communication. For the latter, they introduce the so-called premorphisms and postmorphisms, which correspond to our 'adjust' functions. This algebraic model was later picked up by Carpentiery and Mou, who study communication issues in the model [7]. They present hypercube specific rules to optimize communication by introducing new storage levels.

Still more abstract is the work on investigating parallelism within the Bird–Meertens formalism, which has recently gained much attention (cf. [16]). In this context we picked up the work of Misra on powerlists [10]. His emphasis is on the development of DC algorithms on powerlists, whereas we have presented their massive data parallelization.

Based on [10], Kornerup derives a strategy for mapping most powerlist functions to efficient programs for hypercubic architectures [9].

5.2. Mapping powerlists onto specific architectures

The important problem of how to map powerlists, i.e., the powerlist primitives $*$, *join*, *corr*, etc., onto a specific architecture is not handled in this article. In [1], we propose to use *skeletons*. Skeletons are architecture specific higher-order functions, which are either abstractions of elementary communication-independent computations on all PEs or communication operations, which pass values along the network connec-

tions. Those powerlist primitives each have a straightforward implementation as skeletons. In particular, it turns out that even the communication oriented primitives can be implemented on arrays, meshes and hypercubes equally well. The resulting (skeleton based) programs have a straightforward implementation in an imperative programming language. Additionally, the derived programs are efficient and can – due to the transformational programming paradigm – be ported across several architectures.

5.3. Conclusion

In this article, we have presented a transformation strategy to parallelize powerlist algorithms using transformation rules. The main advantage of making the rules explicit lies in their reuseability. Similar problems can be solved in a similar way, which is demonstrated by the examples. In fact, the presented transformation strategy has been automated using an extended compilation approach, where the user gives hints in the form of laws to the compiler.

Acknowledgements

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

References

[1] K. Achatz and W. Schulte, Architecture independent massive parallelization of divide-and-conquer algorithms, in: B. Möller, ed., *Mathematics of Program Construction*, Lecture Notes in Computer Science, Vol. 947 (Springer, Berlin, 1995) 97–127.

[2] K. Achatz and W. Schulte, Massive parallelization of divide-and-conquer algorithms over powerlists, Tech. Report 95-12, Universität Ulm, Fakultät für Informatik, 1995.

[3] K.E. Batcher, Sorting networks and their applications, *AFIPS Spring Joint Computer Conf.* (1968) 307–314.

[4] R. Bird, Lectures on constructive functional programming, in: M. Broy, ed., *Constructive Methods in Computing Science*, NATO ASI Series. Series F: Computer and Systems Sciences, Vol. 55 (Springer, Berlin, 1989) 151–216.

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

[6] G.E. Blelloch, Prefix sums and their applications, in: J. Reif, ed., *Synthesis of Parallel Algorithms*, (Morgan Kaufmann Los Altos, CA, 1993) Chapter 1, 35–60.

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

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

[9] J. Kornerup, Mapping a functional notation for parallel programs onto hypercubes, *Inform. Process. Lett.* **53** (1995) 153–158.

[10] J. Misra, Powerlist: a structure for parallel recursion, *ACM Trans. Programming Languages Systems* **16** (1994) 1737–1767.

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

[12] R. Paige, J. Reif and R. Wachter, eds., *Parallel Algorithm Derivation and Program Transformation* (Kluwer Academic Publishers, Dordrecht, 1993).
[13] H.A. Partsch. *Specification and Transformation of Programs* (Springer, Berlin, 1990).
[14] P. Pepper, Deductive derivation of parallel programs, in: Paige et al. [12].
[15] F.P. Preparata and J. Vuillemin, The cube-connected cycles: A versatile network for parallel computation, *Commun. ACM* **24** (1981) 300–309.
[16] D.B. Skillicorn, The Bird–Meertens formalism as a parallel model, in: J. S. Kowalik and L. Grandinetti, eds., *Software for Parallel Computation* (Springer, Berlin, 1992).
[17] D.R. Smith, Derivation of parallel sorting algorithms, in: Paige et al. [12].
[18] J.R. Smith, *The Design and Analysis of Parallel Algorithms* (Oxford Univ. Press, Oxford, 1993).