From Transformations to Methodology in Parallel Program Development: A Case Study

Sergei Gorlatch

Universität Passau, D–94030 Passau, Germany. E-mail: gorlatch@fmi.uni-passau.de

Abstract

The Bird-Meertens formalism (BMF) of higher-order functions over lists is a mathematical framework supporting formal derivation of algorithms from functional specifications. This paper reports results of a case study on the systematic use of BMF in the process of parallel program development. We develop a parallel program for polynomial multiplication, starting with a straight-forward mathematical specification and arriving at the target processor topology together with a program for each processor of it. The development process is based on formal transformations; design decisions concerning data partitioning, processor interconnections, etc. are governed by formal type analysis and performance estimation rather than made *ad hoc*. The parallel target implementation is parameterized for an arbitrary number of processors; for the particular number, the target program is both time and cost-optimal. We compare our results with systolic solutions to polynomial multiplication.

Keywords: Formal Specification and Design; Parallelization; Program Transformation; Bird-Meertens Formalism.

1 Introduction

This paper deals with the problem of developing correct and efficient programs for parallel architectures. We address this problem using the *Bird-Meertens formalism* (BMF) which is essentially a collection of higher-order functions (*functionals*) over lists together with a set of algebraic identities [3, 21].

Algorithms on lists are *specified* as expressions of BMF, usually functional compositions. Using equational reasoning, a specification can be *transformed* into a form suitable for an efficient implementation. A transformed expression in BMF is viewed as a *program* which, due to the semantic soundness of the transformations, is *provably correct* with respect to the specification. BMF has been used for deriving algorithms in several application domains including: list processing and optimization problems by Bird and de Moor [2], the Fast Fourier Transform (FFT) by G. Jones [17], calculation of recurrences by Cai and Skillicorn [4], tree algorithms by Gibbons [13], parsing by Cole [8], image processing by Harrison and Grant-Duff [15], divide-and-conquer by the author jointly with Lengauer [14], etc.

BMF seems especially promising in the area of parallel algorithms because many of its functionals have a natural parallel implementation. This aspect has been extensively studied by Skillicorn [28], who proposed a cost calculus for parallel functional programs. The higher-order approach consequently leads to algorithmic *skeletons* which encapsulate typical templates of parallelism. Skeletons were introduced by Cole [7] and have been studied by the group of Darlington at Imperial College [9], the group around Pepper [24, 25], Partsch and Geerling [23, 11], etc.

Closely related work is the Ruby system by Jones and Sheeran [18], the research carried out at Belfast [6], the P^3L project at Pisa [1], the KIDS system by Smith [30] and the functional approach by O'Donnell [22].

This paper reports results of a case study on the systematic use of BMF in the process of parallel program development. We are trying "to go all the way" from a mathematical specification of the algorithm to a parallel program using exclusively formal reasoning. The development process has several stages, with *design decisions* to be made on the following issues: recognition and extraction of specification-inherent parallelism, data partitioning and distribution, choice of a suitable processor topology, scheduling parallel computations and communication of processors in the topology, performance analysis and possible optimizations.

Our goal is to make the design decisions systematically, using type analysis, formal transformations and performance considerations within BMF, rather than making the decisions *ad hoc*. We state each decision and the reasons for it explicitly in order to demonstrate how a sequence of decisions can form a development process. This is a first step towards an automatic design methodology for parallel software.

As a case study we use the straight-forward sum accumulation algorithm for polynomial multiplication. This example and its analogue, convolution, resemble structures of parallelism typical for many numerical applications; they have been studied extensively in the polytope theory of loop parallelization [20] and systolic design. We take the opportunity to compare the results obtained by two formal approaches: BMF and the polytope method.

We start with the mathematical specification of the polynomial product, and then go through all development steps within BMF. Our architecture model is SPMD (Single Program Multiple Data) with distributed memory. The development yields a processor topology, a program for each processor and explicit partitioning of the input and output data. The number of processors can be chosen depending on their availability; asymptotically, we achieve both time- and cost-optimality which is not attainable by systolic solutions.

In Section 2, the polynomial multiplication algorithm and its properties are specified. Section 3 presents briefly the Bird-Meertens formalism and describes how the initial specification is expressed in its higher-order notation. In Section 4, we parallelize the specification by formal transformations in BMF. The resulting expression dictates a possible parallel implementation which we describe in Section 5. Section 6 contains a preliminary estimation of complexity of the target program. In Sections 7 and 8, the program is optimized and new complexity estimates are obtained. Section 9 summarizes and discusses the results.

2 Specification

We take the initial specification from [20] in Dijkstra's quantifier format [10].
Let $A(z)$ and $B(z)$ be two polynomials of degree $n-1$, i.e.:

$$\begin{aligned} A(z) &= \langle\, \Sigma\, k : 0 \le k \le n-1 : a_k z^k \,\rangle \\ B(z) &= \langle\, \Sigma\, k : 0 \le k \le n-1 : b_k z^k \,\rangle \end{aligned}$$

Desired is the polynomial $C(z)$ of degree $2(n-1)$ defined by:

$$C(z) = \langle\, \Sigma\, k : 0 \le k \le 2(n-1) : c_k z^k \,\rangle$$

where:

$$\langle\, \forall\, k : 0 \le k \le 2(n-1) : c_k = \langle\, \Sigma\, i,j : 0 \le i,j \le n-1 \wedge i+j=k : a_i * b_j \,\rangle\,\rangle$$

We use $\otimes$ to denote polynomial multiplication, i.e., $C = A \otimes B$. Similarly, we denote polynomial addition by $\oplus$.

To parallelize the computation of $\otimes$, we use a divide-and-conquer approach, i.e., we partition the problem into smaller pieces which can be processed in parallel and then combined to the solution of the original problem. We want to multiply two polynomials of degree $n-1$, i.e., each with n coefficients. For arbitrary p that divides n (i.e. $n = p * k$), we consider partitioning polynomial A into smaller polynomials $A_0, A_1, \cdots, A_{p-1}$, each of degree $k-1$. Polynomial A can be represented as follows:

$$A = A_0 \;\oplus\; A_1 * z^k \;\oplus\; \cdots \;\oplus\; A_{p-1} * z^{(p-1)k} \tag{1}$$

Of ultimate importance for us is how operation $\otimes$ behaves for partitioned polynomials. We use the following so-called partition property for the case of partitioned polynomial A:

$$A \otimes B = (A_0 \otimes B) \;\oplus\; (A_1 \otimes B) * z^k \;\oplus\; \cdots \;\oplus\; (A_{p-1} \otimes B) * z^{(p-1)k} \tag{2}$$

An analogous property holds if we partition the second polynomial:

$$A \otimes B = (A \otimes B_0) \;\oplus\; (A \otimes B_1) * z^k \;\oplus\; \cdots \;\oplus\; (A \otimes B_{p-1}) * z^{(p-1)k} \tag{3}$$

We use these partition properties in the parallelization process.

3 Casting in the BMF Notation

In this section, we cast our problem in the following functional notation (for brevity, we keep the definitions informal):

$[\alpha]$	the type of lists whose elements are of type α (α-lists);
$[\alpha]_k$	α-lists of length k;
$+\!\!+$	list concatenation;
$\circ$	backward functional composition;
id	identity;
map f	*map* of an unary function f, i.e. $map\, f\, [x_1, \cdots, x_n] \;=\; [f x_1, \cdots, f x_n]$;
$map^2\, f$	"square *map*": $map^2\, f \;=\; map\,(map\, f)$;
red $\odot$	*reduce* over a binary associative operation $\odot \;:\; \alpha \times \alpha \;\rightarrow\; \alpha$, $red\, \odot\, [x_1, \cdots, x_n] \;=\; x_1 \odot x_2 \odot \cdots \odot x_n$;
$< >$	Backus' FP *construction*: $< f_1, \cdots, f_n >\; x \;=\; [f_1\, x, \cdots, f_n\, x]$;
zip	applied to a pair of lists of equal length, yields the list of pairs: $zip\; (\,[x_1, \cdots, x_n]\,,\, [y_1, \cdots, y_n]\,) \;=\; [\,(x_1, y_1), \cdots, (x_n, y_n)\,]$;
zipl $\odot$	"long *zip*": combines elements of two lists with operation $\odot$, returning a list as long as its longest argument, i.e.: $zipl\, \odot\, (\,[\,], x\,) \;=\; x$.

In BMF, function application is expressed by juxtaposition and is assumed to be more binding than any other operator. Interesting algebraic properties of BMF functionals are equalities, e.g.:

$$map\,(f \;\circ\; g) \;=\; map\, f \;\circ\; map\, g \tag{4}$$

We are going to work with lists of polynomial coefficients; the type of coefficients is denoted by τ. The first element of the list is the first coefficient of the polynomial, the term of degree 0. The correspondence between the notation used for specifying the problem and the list notation is obvious: type $[\alpha]_n$ in BMF corresponds to the notation $\langle\; i \;:\; 0 \leq i \leq n-1 \;:\; a_i \;\rangle$ which denotes a list of length n with elements a_i which are of type α. The list length is used explicitly mostly for complexity estimation (similar assumptions were made by Jones [17] and Skillicorn [27]). We omit the length of a list if it is not important in the given context, e.g., lists of arbitrary length whose elements are lists of length k with elements of type α constitute type $[\,[\alpha]_k\,]$.

Lower-case letters are used for lists: a denotes the list of coefficients of polynomial A, b of polynomial B, etc. Operations $\oplus$ and $\otimes$ have the same meaning for lists as for polynomials represented by these lists. The addition of two polynomials can be directly expressed in BMF: $a \oplus b = \mathit{zipl}(+)\,(a,b)$, where $+$ is the addition on type τ of coefficients.

Our objective is to find a parallel implementation of $\otimes$ using the partition properties of the specification presented in Section 2. To enable their use, we partition input lists into lists of p contiguous segments of equal size: exactly how polynomials are partitioned in (2) - (3). Following [27], we express partitioning by operation $\mathit{distr}_p : [\alpha] \;\to\; [\,[\alpha]\,]_p$ with the obvious property $\mathit{red}\,(+\!\!+) \circ \mathit{distr}_p = \mathit{id}$. We call the result of distr_p the *(n-p-k)-partitioning* if a list of length n is partitioned into p parts, each of length k. Thus, our desire to use the partition properties of the specification has led us to the first design decision.

$\boxed{\textbf{First design decision}.\text{ Input lists } a \text{ and } b \text{ must be } (n\text{-}p\text{-}k)\text{-partitioned.}}$

Other partitionings could be chosen, e.g. a cyclical one, but we choose the partitioning into contiguous segments of equal size to enable the use of equalities (1)–(3). Note that n, p and k are parameters.

Let us now look at how the partitioned representation (1) of polynomials can be expressed for lists. We introduce a new operation on lists of lists:

$$\mathit{shift}_k : \; [\,[\alpha]\,] \;\to\; [\,[\alpha]\,]$$

which moves elements of the inner lists to the right by inserting $0,\ k,\ 2k,\ \cdots$ neutral elements at the beginning of the first, second, third, $\cdots$ inner lists correspondingly. The neutral element should be an argument of function shift_k; instead, we just illustrate two cases in which this function is used throughout the paper:

- For an argument of type $[\,[\tau]\,]$, the neutral element is 0, e.g.:

$$\mathit{shift}_k\,[\,[a_1,\cdots,a_n],[b_1,\cdots,b_n],[c_1,\cdots,c_n]\,] =$$
$$[\,[a_1,\cdots,a_n],[\underbrace{0,\cdots,0}_{k},b_1,\cdots,b_n],[\underbrace{0,\cdots,0}_{2k},c_1,\cdots,c_n]\,]$$

- For an argument of type $[\,[\,[\tau]\,]\,]$, the neutral element is the empty list, e.g.:

$$\mathit{shift}_1\,[\,[\,[a_1],\cdots,[a_n]\,],[\,[b_1],\cdots,[b_n]\,],[\,[c_1],\cdots,[c_n]\,]\,] =$$
$$[\,[\,[a_1],\cdots,[a_n]\,],[\,[\,],\ [b_1],\cdots,[b_n]\,],[\,[\,],\ [\,],\ [c_1],\cdots,[c_n]\,]$$

Then, for any list of coefficients $[\tau]_n$ such that $n = p * k$:

$$\mathit{red}\,(\mathit{zipl}+) \;\circ\; \mathit{shift}_k \;\circ\; \mathit{distr}_p \;=\; \mathit{id} \qquad (5)$$

The left-hand side of this equality corresponds directly to representation (1) of an n-polynomial via its segments in the (n-p-k)-partitioning.

A useful shorthand, called *sectioning*, can be used for the curried operator $\bar{\otimes}$:

$$\otimes\,(a,b) \;=\; (\bar{\otimes})\,a\,b \;=\; (a\,\bar{\otimes})\,b \;=\; (\bar{\otimes}\,b)\,a$$

Using sectioning, the partition properties (2)–(3) of $\otimes$ can be reformulated in BMF for the case of (n-p-k)-partitioning:

$$\otimes(a,b) \;=\; (\,red\,(zipl+) \circ shift_k \circ map\,(\bar{\otimes}\,b) \circ distr_p\,)\;a \tag{6}$$
$$\otimes(a,b) \;=\; (\,red\,(zipl+) \circ shift_k \circ map\,(a\,\bar{\otimes}) \circ distr_p)\;b \tag{7}$$

We want to use equalities (6)-(7) in transforming the expression of $\otimes(a,b)$, so we need an equality describing the relation between curried and uncurried versions of $\otimes$. Let $distr_p\, b \;=\; [b_1, \cdots, b_p]$, then

$$(map\,\bar{\otimes} \circ distr_p)\;b \;=\; [\bar{\otimes}\,b_1, \cdots, \bar{\otimes}\,b_p] \tag{8}$$

The resulting list of functions can be applied per construction, with the following equality relating $\otimes$ and $\bar{\otimes}$ in case of (n-p-k)-partitioning for $x \in [\tau]_n$ and $y \in [\tau]_k$:

$$<(map\,\bar{\otimes} \circ distr_p)\,x>\; y \;=\; (map\;\otimes \circ zip)\;(distr_p\,x\;,\;copy_p\,y) \tag{9}$$

where function $copy_p : \alpha \;\rightarrow\; [\alpha]_p$ yields, for an object of an arbitrary type, a list consisting of p such objects.

4 Transformation in BMF

We can now parallelize $\otimes(a,b)$ by formal transformation:

$$
\begin{array}{ll}
 & \otimes(a,b) \\
= & \quad \{ \text{ equality (6) } \} \\
 & (red\,(zipl+) \circ shift_k \circ map\,(\bar{\otimes} b) \circ distr_p)\,a \\
= & \quad \{ \text{ definitions of } map \text{ and } distr_p \; \} \\
 & (red\,(zipl+) \circ shift_k)\,[\bar{\otimes}\,a_1 b, \cdots, \bar{\otimes}\,a_p b\,] \\
= & \quad \{ \text{ equality (7) } \} \\
 & (red\,(zipl+) \circ shift_k) \\
 & [\,(red\,(zipl+) \circ shift_k \circ map\,(a_1\,\bar{\otimes}) \circ distr_p)\,b, \\
 & \;\cdots, (red\,(zipl+) \circ shift_k \circ map\,(a_p\,\bar{\otimes}) \circ distr_p)\,b\,] \\
= & \quad \{ \text{ definition of } map\text{, equality (8) } \} \\
 & (\,red\,(zipl+) \circ shift_k \circ \\
 & map\,(red\,(zipl+) \circ shift_k \circ <(map\,\bar{\otimes} \circ distr_p)\,a>) \circ distr_p\,)\,b \\
= & \quad \{ \text{ equality (4) } \} \\
 & (\,red\,(zipl+) \circ shift_k \circ map\,(red\,(zipl+) \circ shift_k) \circ \\
 & map\,(<(map\,\bar{\otimes} \circ distr_p)\,a>) \circ distr_p\,)\,b
\end{array}
$$

Further, we transform the second line of this intermediate form, working towards an expression which takes a pair (a,b) as input, like the original expression $\otimes(a,b)$:

$$
\begin{array}{ll}
 & (map\,(<(map\,\bar{\otimes} \circ distr_p)\,a>) \circ distr_p\,)\,b \\
= & \quad \{ \text{ definitions of } map \text{ and } distr_p \; \} \\
 & [<(map\,\bar{\otimes} \circ distr_p)\,a>\,b_1, \cdots, <(map\,\bar{\otimes} \circ distr_p)\,a>\,b_p] \\
= & \quad \{ \text{ equality(9) } \} \\
 & [map\,\otimes\,(zip\,(distr_p\,a, copy_p\,b_1)), \cdots, map\,\otimes\,(zip\,(distr_p\,a, copy_p\,b_p))] \\
= & \quad \{ \text{ definitions of } map^2 \text{ and } zip \; \} \\
 & (\,map^2 \otimes \circ\, map\,(zip)\,)\;(zip\,([distr_p\,a, \cdots, distr_p\,a]\,, [copy_p\,b_1, \cdots, copy_p\,b_p])) \\
= & \quad \{ \text{ definitions of } copy_p \text{ and construction } \} \\
 & (\,map^2 \otimes \circ\, map\,(zip) \circ zip \,\circ \\
 & <copy_p \circ distr_p \circ fst\,,\; map\,(copy_p) \circ distr_p \circ snd>\,)\;(a,b)
\end{array}
$$

Here, *fst* and *snd* yield the first and the second component of a pair of arguments.

Therefore, operation $\otimes$ which accepts a pair of lists (a, b) of coefficients of two polynomials and produces the list of coefficients of their product, has the following BMF representation:

$$\begin{aligned}\otimes = & \ red\,(zipl+) \circ shift_k \circ map\,(red\,(zipl+) \circ shift_k\,) \circ \\ & map^2\,(\otimes) \circ \\ & map\,(zip) \circ zip \circ <copy_p \circ distr_p \circ fst\,,\ map\,(copy_p) \circ distr_p \circ snd> \end{aligned} \tag{10}$$

We separate out the map^2-part of the obtained expression and the parts before and after it; these parts are already shown as separate lines in (10):

- *compute*: this is $map^2\,(\otimes)$, the central part of (10);
- *distribute*: this is the part applied in (10) before *compute*; it prepares the multiplications by *distr*ibuting, *copy*ing and *zip*ping the input;
- *combine*: this is the final part of (10) which combines the partial results of function *compute* using *red*uction, *shift*ing and *zipl*ing.

If our program is to take a pair of n-lists $(\,a\,,\,b\,)$ as input then the types of the program parts read as follows:

· $distribute:\ (\,[\tau]_n\,,\,[\tau]_n\,) \rightarrow [\,[\,(\,[\tau]_k\,,\,[\tau]_k\,)\,]_p]_p$

· $compute:\ [\,[\,([\tau]_k, [\tau]_k)\,]_p]_p \rightarrow [\,[\,[\tau]_{2k-1}\,]_p]_p$

· $combine:\ [\,[\,[\tau]_{2k-1}\,]_p]_p \rightarrow [\tau]_{2pk-1}$

We use $\overset{n}{\otimes}_{pk}$ to denote the parallel version of $\otimes$ which multiplies n-polynomials using the (n-p-k)-partition. Operation $\otimes$ under map^2 is applied to pairs of segments of length k; we denote it by $\overset{k}{\otimes}$. Thus, our first design decision to use the (n-p-k)-partitioning of a and b has paid off: to multiply the original polynomials, it is sufficient, due to map^2, to multiply (independently) the segments of the partitioning.

The final expression for $\overset{n}{\otimes}_{pk}$ reads as follows:

$$\overset{n}{\otimes}_{pk} = \ combine \circ compute \circ distribute \tag{11}$$

where

$$\begin{aligned} distribute \ = & \ map\,(zip) \circ zip \circ \\ & <copy_p \circ distr_p \circ fst\,,\ map\,(copy_p) \circ distr_p \circ snd> \end{aligned} \tag{12}$$

$$compute \ = \ map^2\,(\overset{k}{\otimes}) \tag{13}$$

$$combine \ = \ red\,(zipl+) \circ shift_k \circ map\,(red\,(zipl+) \circ shift_k\,) \tag{14}$$

5 Parallel Implementation

In this section, we consider a parallel implementation of expression (11). It is a composition of three functions, *distribute*, *compute* and *combine* that must be applied in sequence, due to the semantics of functional composition.

Second design decision. The parallel program for polynomial multiplication, $P(\overset{n}{\otimes}_{pk})$, consists of three parts: $P(distribute)$, $P(compute)$ and $P(combine)$ which are composed in sequence.

The type of the *compute*-part shows that function $map^2\,(\overset{k}{\otimes})$ applies to $p * p$ pairs of k-segments, so we have p^2 potentially parallel processes, each multiplying two polynomials of length k. This dictates the following design decision.

Third design decision. We pick p^2 processors.

Note that this decision is not very restrictive: we can choose p arbitrarily. The assumption that p divides n is made exclusively for convenience.

In the following subsections we look at the parts of the expression for $\overset{n}{\otimes}_{pk}$.

5.1 Function *distribute*

This function takes a pair of n-lists (a,b) and yields a "square" $p * p$ list of lists, each element of which is a pair of k-segments of a and b. Expression (12) consists of two parts (lines): we call them the *construction* part and the *zip* part.

Both components of the construction part have the same type:

. $(copy_p \circ distr_p \circ fst) \;:\; ([\tau]_n\,,\,[\tau]_n) \;\rightarrow\; [\,[\,[\tau]_k\,]_p]_p$

. $(map\,(copy_p) \circ distr_p \circ snd) \;:\; ([\tau]_n\,,\,[\tau]_n) \;\rightarrow\; [\,[\,[\tau]_k\,]_p]_p$

To illustrate the difference between them, we present our square lists of lists $[\,[\alpha]_p\,]_p$ in a two-dimensional setting for the following example which we will use throughout the paper. Let us multiply two polynomials: $a = [1,3,5,7]$ and $b = [2,4,6,8]$. For (n-p-k)-partitioning with $n = 4$, $p = 2$, $k = 2$, the construction part yields the following pair of "matrices":

$$(a,b) \;\rightarrow\; \left(\begin{array}{c} [\,[\,[1,3],\;\;[5,7]\,], \\ \;\;[\,[1,3],\;\;[5,7]\,]\,] \end{array} ,\; \begin{array}{c} [\,[\,[2,4],\;\;[2,4]\,], \\ \;\;[\,[6,8],\;\;[6,8]\,]\,] \end{array} \right),$$

i.e. the first component of the construction part broadcasts segments of a along the columns, the second broadcasts segments of b along the rows of the matrix. We introduce broadcast functions which can be directly expressed, e.g., in the MPI standard [32]:

. $bcast\text{-}row_p \;=\; map\,(copy_p)$ — for broadcasting a list of segments along the rows of the processor matrix;

. $bcast\text{-}col_p \;=\; copy_p$ — for broadcasting along the columns of the matrix.

Thus, our parallelizing transformations lead us to the following design decision.

Fourth design decision. The input lists are broadcast: the segments of the first list along the column processors, the segments of the second along the rows.

The *zip* part of *distribute* takes two lists of lists of segments (we view them as matrices of segments) and composes their corresponding elements in pairs, in other words, its role is to put both matrices into the $p * p$ matrix of processors.

5.2 Function *compute*

The central part of our program is function *compute*. It performs the multiplication of segments which were placed on each processor at the *distribute* step. We introduce the notation $P(\overset{k}{\otimes})$ for a (sequential) implementation of $\overset{k}{\otimes}$ on one processor. Now, the *compute* part of our program is: $P(compute) \;=\; map^2\,P(\overset{k}{\otimes})$.

5.3 Function *combine*

Let us now concentrate on the last part of our program — function *combine*. As usual, we start with the type analysis. Function *combine* gets its argument of type $[\,[\,[\tau]_{2k-1}\,]_p\,]_p$, the $p*p$ matrix of segment products from function *compute*.

Introducing, for brevity, function $\textit{red-zipl-shift} \;=\; \textit{red}\,(\textit{zipl}\,+) \,\circ\, \textit{shift}_k$, we have

$$\begin{aligned} \textit{combine} \;&=\; \textit{red-zipl-shift} \circ \textit{map}\,(\textit{red-zipl-shift}), \\ \text{where}\;\; \textit{map}\,(\textit{red-zipl-shift}) \;&:\; [\,[\,[\tau]_{2k-1}\,]_p\,]_p \;\rightarrow\; [\,[\tau]_{(p+1)k-1}\,]_p, \\ \textit{red-zipl-shift} \;&:\; [\,[\tau]_{(p+1)k-1}\,]_p \;\rightarrow\; [\tau]_{2pk-1}\,. \end{aligned}$$

So, the first step of *combine* performs reductions in parallel along the rows of the matrix. The results are placed in p processors (the last one in each row). The second step performs the reduction on these processors and yields the result in the processor in the south-east corner of the matrix.

Three phases of our program are illustrated in Figure 1.

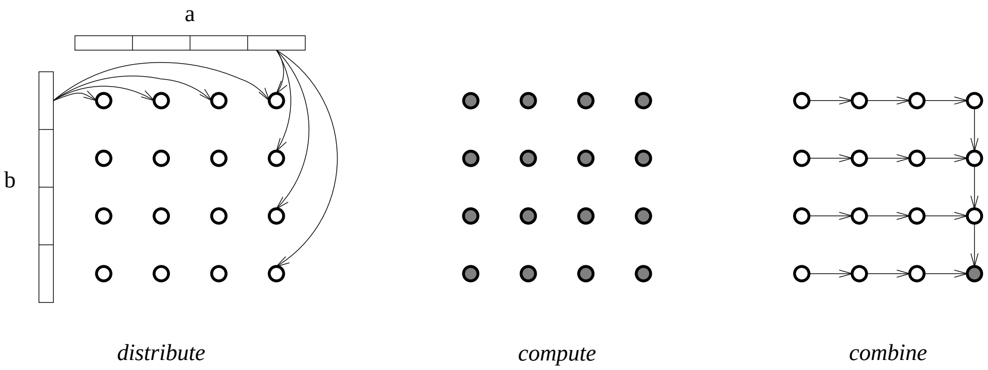

Figure 1: Three phases of the program

We illustrate how the phases of $P(\overset{n}{\otimes}_{pk})$ work on our concrete example with $p=2$. The processor positions in the $2*2$ matrix are represented by * or by a piece of data in the processor (a list, a pair of lists, etc.). Arrows within the matrix show communications between processors in the step which has this matrix as initial state.

$$\begin{array}{rcccc} & a = [\,[1,3], & & [5,7]\,] & \\ & \downarrow & & \downarrow & \\ b = [\,[2,4], \rightarrow & * & \rightarrow & * & \\ & \downarrow & & \downarrow & \\ [6,8]\,] \rightarrow & * & \rightarrow & * & \end{array} \quad \overset{\textit{distribute}}{\Longrightarrow} \quad \begin{pmatrix} (\,[1,3],[2,4]\,) & (\,[5,7],[2,4]\,) \\ & \\ (\,[1,3],[6,8]\,) & (\,[5,7],[6,8]\,) \end{pmatrix}$$

$$\Downarrow\; \textit{compute}$$

$$\begin{pmatrix} * & [2,10,22,34,28] \\ & \downarrow \\ * & [6,26,54,82,56] \end{pmatrix} \quad \overset{\textit{map}\;(\,\textit{red-zipl-shift}\,)}{\Longleftarrow} \quad \begin{pmatrix} [2,10,12] & \rightarrow & [10,34,28] \\ & & \\ [6,26,24] & \rightarrow & [30,82,56] \end{pmatrix}$$

$$\Downarrow\; \textit{red-zipl-shift}$$

$$\begin{pmatrix} * & * \\ & \\ * & [2,10,28,60,82,82,56] \end{pmatrix}$$

6 First Estimation of Complexity.

In this section, we estimate the complexity of our parallel program. We will see that complexity considerations may lead to new design decisions and require additional program transformations.

The sequential complexity of the straight-forward multiplication of polynomials of degree n is $O(n^2)$. This algorithm can be improved to $O(n^{\log 3})$ by saving some multiplications or to $O(n \log n)$ by using FFT, but we do not consider these opportunities in our case study. The systolic solution for our straight-forward polynomial multiplication has time complexity $O(n)$ on $O(n)$ processors [20]. We would like to improve the asymptotic parallel time complexity to $O(\log n)$.

We are interested not only in the *time* complexity but also in the *number of processors* and in the *cost* of the implementation which is the product of time and processors; the cost is $O(n^2)$ in the sequential case. In practice, we have to work on an arbitrary fixed number of processors. Due to the *third design decision*, we require p^2 processors; therefore, the lower time complexity bound for our problem is $O\left((n/p)^2\right)$. The complexity must be estimated for each part of the program (11)-(14), i.e. for *distribute*, *compute* and *combine*.

The *distribute* part has two steps: first, partitioning each of input lists in p segments, placing the segments of a in the processors of the first row and b in the first column and second, broadcasting along the columns and along the rows of the processor matrix. We can organize the partitioning in one of two ways: 1) pipelining requires p steps, with time $O(n/p)$ at each step; 2) using a tree structure we need $\log p$ steps with the varying length of lists to be transmitted at each step: $n/2$, $n/4$, $\cdots$, n/p. Both implementations have complexity $O(n)$. To achieve logarithmic time, we make an assumption which is usual in the systolic design:

Fifth design decision. Lists a and b are partitioned in advance, parallel program $P(\overset{n}{\otimes}_{pk})$ accepts a pair $(distr_p\, a, distr_p\, b)$ as input.

As noted in [27], the cost of partitioning can be ignored: first, it is often hidden in the cost of loading the program, second, with disk arrays becoming more common, the lists may already be stored in segments local to the processors. By *distribute*, we refer in the sequel to the changed function.

The next action in the *distribute* part is broadcasting whose complexity depends on the underlying topology. If the processors were connected in a 2D-mesh, broadcast would also cost linear time which is not satisfactory. One possible way to enable fast broadcasting, satisfying at the same time the practical requirement of a fixed fan-in topology, is to use a *mesh of trees* [19] where processors in one row (column) are connected in a balanced binary tree.

Sixth design decision. Processors are connected in the $p * p$ mesh of trees.

Note that we do not need usual 2D-mesh connections and that, although the mesh of trees does not belong to usual physical topologies, it can be efficiently embedded in most of them due to the property of fixed fan-in [19].

Under these circumstances, each broadcast requires $O(\log p)$ steps; at each step, segments of length n/p are transmitted. Therefore:

$$t\,(\mathit{distribute}) \;=\; O\,(n \log p/p)$$

In the *compute* part, all processors work independently, each multiplying two polynomials of length n/p. As this multiplication is done sequentially, we have:

$$t\,(compute) \;=\; O\left((n/p)^2\right)$$

In the *combine* part of the program, we must compute function *red-zipl-shift* (see Subsection 5.3), first in parallel for all rows of the processor matrix and then along the last column. The number of time steps needed at each stage is $O(\log p)$ due to the tree connections in the matrix. At each step, we add two lists and transmit the resulting lists between processors. Both actions are linear in the length of computed (transmitted) lists; this length is directly available from the type analysis and is evidently greater at the second stage, *red-zipl-shift* : $[\,[\tau]_{(p+1)k-1}\,]_p \;\to\; [\tau]_{2pk-1}$, i.e. it is $O(n)$, so the lower bound of the complexity is:

$$t\,(combine) \;\geq\; O(n \log p)$$

To obtain the total complexity of the program, we sum the obtained complexities of its parts, which yields $O\left((n/p)^2 + n\log p\right)$. There is no problem with $O\left(n^2/p^2\right)$ – it is the lower bound. But the *combine* part of the program has poor performance, so we need to look more closely at it.

7 Optimization and New Complexity

In this section, we arrive at a new parallel implementation of *combine* as a result of an additional transformation in BMF and two new design decisions. We would like to change the reduction steps in (14) so that the length of lists which are *zipl*ed and transmitted between processors does not increase.

Let us consider the effect of applying function $shift_1$ to a p-list of p-lists, which in our considerations represents the matrix of processors. We have illustrated this case in Section 3 with the empty list as the neutral element. E.g., for a matrix of segments l_{ij} we get:

$$\begin{array}{l} [\,[\,l_{11},\; l_{12},\; l_{13}\,],\\ \;\;[\,l_{21},\; l_{22},\; l_{23}\,],\\ \;\;[\,l_{31},\; l_{32},\; l_{33}\,]\,] \end{array} \;\stackrel{shift_1}{\Longrightarrow}\; \begin{array}{l} [\,[\,l_{11},\; l_{12},\; l_{13}\,],\\ \;\;[\,[\,],\; l_{21},\; l_{22},\; l_{23}\,],\\ \;\;[\,[\,],\; [\,],\; l_{31},\; l_{32},\; l_{33}\,]\,] \end{array}$$

We see that $shift_1$ yields a list of lists (a matrix) of segments where non-empty segments in columns constitute north-east diagonals of the original matrix. This idea is expressed by the following transformation in BMF:

$$\begin{aligned} red\,(zipl\,+) \circ shift_k \circ map\,(red\,(zipl\,+) \circ shift_k\,) \;=&\\ red\,(zipl\,+) \circ shift_k \circ red\,(zipl\,(zipl\,+)) \circ shift_1& \end{aligned} \tag{15}$$

Applying this transformation, we arrive at the following expression for *combine*:

$$combine \;=\; red\,(zipl\,+) \circ shift_k \circ red\,(zipl\,(zipl\;+)) \circ shift_1 \tag{16}$$

where function $shift_1$: $[\,[\,[\tau]_{2k-1}\,]_p\,]_p \;\to\; [\,[\,[\tau]_{0..2k-1}\,]\,]_p$ does not increase the length of the innermost lists, unlike function $shift_k$ in the first version of *combine*. The first part (in order of application) of this expression:

$$combine_1 \;=\; red\,(zipl\,(zipl\;+)) \circ shift_1$$

can be directly implemented on our matrix of processors if we add tree-like connections along the north-east diagonals in the topology.

Seventh design decision. Processors are connected in the *mesh of trees with diagonal trees* topology ([19]).

Now, $combine_1$ is just the reduction of the segments along the diagonals with operation $zipl\,+$, yielding the result of type $[\,[\tau]_{2k-1}\,]_{2p-1}$. We assume that the $2p-1$ processors in which it is located are those on the north-east border.

For our concrete example, we have:

$$\begin{pmatrix} [2,10,12] & & [10,34,28] \\ & \nearrow & \\ [6,26,24] & & [30,82,56] \end{pmatrix} \stackrel{combine_1}{\Longrightarrow} \begin{pmatrix} [2,10,12] & [16,60,52] \\ & \\ * & [30,82,56] \end{pmatrix}$$

It remains to compute

$$combine_2 \;=\; red\,(zipl\,+) \,\circ\, shift_k$$

which has type $[\,[\tau]_{2k-1}\,]_{2p-1} \;\rightarrow\; [\tau]_{2pk-1}$, i.e., it increases the length of the lists to be transmitted, and we again come to the linear time as a lower bound. This is unavoidable if we stick to the original type of *combine*: we start with p^2 lists of length $2k-1$ and arrive at one list of length $2pk-1$.

The only possibility is to make the same assumption as for the *distribute* part: to change the type of the program output, and this is our final design decision.

Eighth design decision. Output list is distributed among the north-east border.

Note that the distribution of the output cannot be done exactly in the same way as the distribution of input lists: in general, the length of the result, $2pk-1$, may be even a prime number, so it does not generally divide into segments of equal size.

The optimized version of *combine* is illustrated in Figure 2.

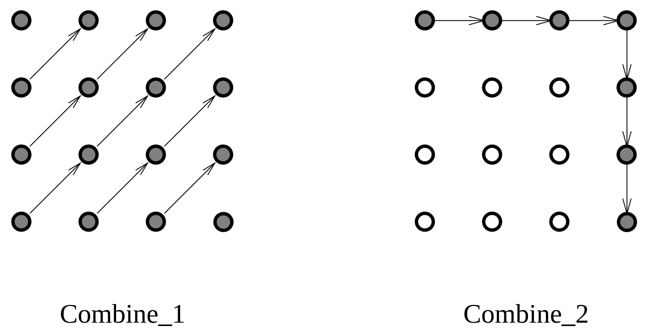

Figure 2: Optimized *combine*.

We sketch a parallel implementation of $combine_2$ without giving a complete proof. The main idea comes once again from the type analysis: for segments of length $2k$–1, the disjoint parts of the result of $combine_2$ can be obtained by pairwise *zipl*ing in the neighbouring processors.

To express this formally, we introduce functions $lhalf^-$, $lhalf^+$, $rhalf^-$ and $rhalf^+$, which for a list $[\tau]_{2k-1}$ yield a list consisting of the first $k-1$, first k, last $k-1$ or last k elements of an argument list, correspondingly.

Let d be a result of $combine_1$, i.e. of the form $d \;=\; [\,d_1, d_2, \cdots, d_{2p-1}\,]$, where each d_i is of type $[\tau]_{2k-1}$. We can compute the list of lists dd:

$$dd \;=\; [\,d_1,\, d_1 \odot d_2,\, d_2 \odot d_3,\, \cdots,\, d_{2p-2} \odot d_{2p-1}\,]\,,$$

where $x \odot y \;=\; (zip\,+\,(rhalf^- x, lhalf^- y)) +\!\!\!+\; rhalf^+ y$.

The interpretation on the matrix of processors is that if d is the distribution on the north-east processors after $combine_1$, then dd is the distributed result of $combine_2$.

The resulting list c can be composed from dd as follows:

$$c \;=\; (\, red\,(+\!\!+) \,\circ\, map\,(select)\,)\; dd$$

where $select \;=\; is.last \rightarrow id\,;\; lhalf^{+}$.

Here we use the conditional from Backus' FP. Predicate *is.last* holds only for the last element of a list. Function *select* picks the first k elements of the segment of dd in each processor, and the whole segment of the last processor (in the east-south corner of the matrix). It can be proven that $c \;=\; combine_2(d)$.

For our concrete example, we show the transition from d to c on the three north-east processors. Lists in d are shown "shifted":

$$\begin{array}{lll} d \;=\; [\,[2,10,12], & & \\ \qquad\quad\;\downarrow & & \\ & [16,60,52], & \\ & \qquad\quad\;\downarrow & \\ & & [30,82,56]\,] \end{array}$$

$$dd \;=\; [\,[2,10,12]\,,\,[28,60,52]\,,\,[82,82,56]\,]$$

$$c \;=\; [2,10] +\!\!+ [28,60] +\!\!+ [82,82,56]$$

Let us now analyze the time complexity of the new *combine* implementation. Reduction along the diagonals in $combine_1$ takes $O\,(\log p)$ steps, with $O\,(n/p)$ computations and $O\,(n/p)$ communications in each of them, i.e., the required time is $O\,(n \log p/p)$. Step $combine_2$ is the computation of dd as explained before. This requires first pairwise communications of size $O\,(n/p)$, then all processors simultaneously perform computations which take sequentially $O\,(n/p)$. Therefore, the complexity of *combine* is $O\,(n \log p/p)$.

The new complexity estimate for *combine* improves the total complexity of the parallel target program to $O\left((n \log p/p) + (n/p)^2\right)$.

The value of parameter p can be chosen between 1 and n. The most practical situation is when p is a number of processors available on a particular machine.

Asymptotically, the following cases are of interest:

- $p = 1$: we get $t = O(n^2)$, so we have not worsened the sequential situation;
- $p = n$: we achieve $t = O(\log n)$ on n^2 processors; this solution gives the best time but is clearly not cost-optimal;
- $p = n/\log n$: a cost-optimal solution with $t = O\left(\log^2 n\right)$ on $(n/\log n)^2$ processors;
- $p = \sqrt{n}$: $t = O(n)$ on n processors: equal to the systolic solution and cost-optimal.

8 Additional Optimization

As we see, our recent version achieves either the time- or the cost-optimality. We would like to have both simultaneosly, which would mean that the implementation is asymptotically the best possible for the given algorithm.

The reason that it is not so yet is clearly the term $O\,(n \log p/p)$ which reflects the cost of both the *distribute* and the *combine* phase. Remember how this term is obtained: it is the complexity of the computation on a binary tree of height $\log p$, the levels of which become active after each other, with segments of length n/p being transferred between levels at each step. In the *combine* phase, there is also a computation of length n/p at each level, namely the componentweise addition of segments, which does not change the asymptotic behaviour.

Let us try to improve these phases of the algorithm by exploiting the well-known pipelining technique. The idea is to partition every segment of length n/p into smaller subsegments and to pipeline them through the tree.

Let $m(n,p)$ be the number of subsegments in each segment, defined by

$$m(n,p) = min\{n/p, (\log p{+}1)\}$$

The length of a subsegment is thus $n/(p \cdot m(n,p))$.

Now, if the subsegments are pipelined into the tree after each other, the required number of steps increases from $\log p$ to $(\log p + m(n,p))$. Each step is linear in the length of the subsegment, so the time of the pipelined version is: $t(\mathit{distribute}) = t(\mathit{combine}_1) = O\left(\frac{n \cdot \log p}{p \cdot m(n,p)} + \frac{n}{p}\right)$. This improves the total complexity of the parallel target program to

$$t = O\left(\frac{n \cdot \log p}{p \cdot m(n,p)} + (n/p)^2\right)$$

The value of parameter p can again be chosen between 1 and n. If we take $p = n/\sqrt{\log n}$ then

$$m(n,p) = min\{\sqrt{\log p}, (\log p{+}1)\} = \sqrt{\log p}$$

which yields $t = O(\log n)$. We have the optimal, logarithmic, time complexity which is achieved on $p^2 = (n^2/\log n)$ processors, so the costs are $O(n^2)$ which is optimal as well. Therefore, the pipelined implementation on $n^2/\log n$ processors is *both time- and cost-optimal* for our straightforward algorithm.

It is not difficult to see that on other processor numbers the pipelined version yields asymptotically the same time complexity as the implementation without pipelining studied in the previous section.

From the practical viewpoint, we are more interested in the situation when the processor number is an arbitary fixed value and the problem size n is relatively large. Then the term $(n/p)^2)$ clearly dominates in the expression of the time complexity which guarantees the so-called *scaled linear speed-up* [26].

9 Conclusions

Our case study demonstrates the usefulness of BMF as a formal framework at various stages of parallel program development. In the course of the development, we have made eight design decisions based on formal transformations, type analysis and performance considerations. Together, these decisions determine the organization of data and control flow in the parallel target program.

Methodologically, the design decisions exemplify the crucial points in the parallel software design process. The experienced parallel programmer would probably go through almost the same points based on his/her expertise. The advantage of the formal approach studied here is that determining the crucial points of the design process and making the appropriate decisions is systematic rather than *ad hoc*. The approach not only supports the programmer in the development but also gives confidence in the correctness and performance of the parallel target program.

The design decisions made in our case study can be summarized as follows:

- We have chosen the input data partitioning and distribution using the properties of the initial specification.

- By formal transformation, we have arrived at a BMF expression which dictates the two-dimensional shape of the processor topology and describes the program structure of each processor.
- In order to exploit the potential parallelism fully, we have decided to create several copies of data segments and to broadcast them between processors.
- A performance analysis of the BMF program has determined the processor topology, a mesh of trees with diagonal links, which provides an envisaged performance of the program.
- A type analysis and performance considerations have pointed to optimizations that have been carried out by transformations together with a suitable choice of the output data partitioning.

We see the novelty of our work in that, in contrast to previous work on the Bird-Meertens formalism [3, 17, 24], we do not restrict the consideration to formal derivation of an BMF expression with apparently good time complexity. In addition, we concentrate on the methodological aspects of the transition from a very high-level specification of an algorithm to a provably correct, predictably efficient multiprocessor implementation of this algorithm.

Our work continues efforts by Skillicorn and Cai [29], Pepper and Südholt [24, 31] and others to incorporate data distribution into the development process. A new aspect is the use of transformations with curried and uncurried functional forms; this leads to a data distribution with copying and increases the implementable parallelism. Our type analysis has common features with the abstract type transformation by Harrison [16]; we use it in a different context, namely for determining the proper output data distribution. The idea to use the list length explicitly in the type expressions is due to Jones [17].

Our target program can be viewed as a skeleton (template) which captures the structure typical for many applications and thus deserves to be studied further. Its three parts, *distribute*, *compute* and *combine*, especially the first two, are themselves often regarded as skeletons – similar structures were analyzed e.g. by Darlington et al. [9], Geerling [12] and others. We believe that the optimized *combine*, the result of the transformation of two-dimensional reduction into a diagonal form with the distributed result, might be useful as a skeleton in many applications.

An important feature of our target program is that it can be represented directly in a parallel programming language like the MPI standard [32], and is therefore implementable on a real machine. BMF expressions can be interpreted as parallel imperative programs with message passing as demonstrated in [14]. Real implementation requires also tuning to the available processor number, which is greatly simplified in our case since this number is a parameter of the program. The topology found in the course of development must be embedded into the topology at hand; solutions for mapping a mesh of trees with diagonal links to more standard topologies can be found in [19].

Let us briefly compare our results with polytope solutions for the polynomial multiplication problem [20]. The number of processors is a parameter in our design; it can be chosen arbitrarily depending on the availability of processors. In the systolic solution, the number of processors is $O(n)$, i.e., if the problem size exceeds the available processor number, additional problems of mapping and scheduling must be addressed [5] (these problems are already solved in our BMF program). For the asymptotically linear number of processors, both our and the systolic version achieve linear time complexity and are cost-optimal. Our program can also reach the optimal time complexity $O(\log n)$ on $n^2/\sqrt{\log p}$ processors, which is cost-optimal and is not possible in the polytope model.

10 Acknowledgement

Thanks to Chris Lengauer, Johannes Rehmet, Christoph Wedler and Eberhard Zehendner for discussing the manuscript.

References

[1] B. Bacci, M. Danelutto, et al. Efficient compilation of structured parallel programs for distributed memory MIMD machines. In G. R. Joubert et al., editors, *Parallel Computing: Trends and Applications*, pages 565–568. Elsevier Science, 1994.

[2] R. Bird and O. de Moor. Solving optimization problems with catamorphisms. In C. Morgan R. Bird and J. Woodcock, editors, *Mathematics of Program Construction*, Lecture Notes in Computer Science 669, pages 45–66, 1992.

[3] R. S. Bird. Lectures on constructive functional programming. In M. Broy, editor, *Constructive Methods in Computing Science*, NATO ASO Series F: Computer and Systems Sciences. Vol. 55, pages 151–216. Springer Verlag, 1988.

[4] W. Cai and D. Skillicorn. Calculating recurrences using the Bird-Meertens formalism. *Parallel Processing Letters*. To appear.

[5] Z. Chen and W. Shang. Mapping of uniform dependence algorithm onto fixed size processor arrays. In *Proc. 7th Int. Parallel Processing Symposium*, pages 804–809, 1993.

[6] M. Clint, S. Fitzpatrick, T. Harmer, P. Kilpatrick, and J. Boyle. A family of data-parallel derivations. In W. Gentzsch and U. Harms, editors, *High Performance Computing and Networking*, volume II of *Lecture Notes in Computer Science 797*, pages 457–462. Springer-Verlag, 1994.

[7] M. Cole. Algorithmic skeletons: A structured approach to the management of parallel computation. Ph.D. Thesis. Technical Report CST-56-88, Department of Computer Science, University of Edinburgh, 1988.

[8] M. Cole. List homomorphic parallel algorithms for bracket matching. Technical report, Department of Computer Science, University of Edinburgh, 1993.

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

[10] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag, 1990.

[11] M. Geerling. Formal derivation of SIMD parallelism ¿from non-linear recursive specifications. In B. Buchberger and J. Volkert, editors, *Parallel Processing: CONPAR 94 – VAPP VI*, Lecture Notes in Computer Science 854, pages 136–147. 1994.

[12] M. Geerling. Program transformations and skeletons: formal derivation of parallel programs. Technical Report CSI-R9411, Katholike Universiteit Nijmegen, October 1994.

[13] J. Gibbons. Upwards and downwards accumulations on trees. In R. Bird, C. Morgan, and J. Woodcock, editors, *Mathematics of Program Construction*, Lecture Notes in Computer Science 669, pages 122–138, 1992.

[14] S. Gorlatch and C. Lengauer. Systematic development of an SPMD implementation schema for mutually recursive divide-and-conquer specifications. In H. J. Siegel, editor, *Proc. Eighth Int. Paral. Proc. Symp. (IPPS'94)*, pages 368–375. IEEE Computer Society Press, 1994.

[15] Z. Grant-Duff and P. Harrison. Skeletons, list homomorhisms and parallel program transformation. Technical report, Imperial College, 1994.

[16] P. G. Harrison. A higher-order approach to parallel algorithms. *The Computer Journal*, 35(6):555–566, 1992.

[17] G. Jones. Deriving the fast Fourier algorithm by calculation. In *Proc. Conf. on Functional Programming*, pages 80–102, 1989.

[18] G. Jones and M. Sheeran. Designing arithmetic circuits by refinement in Ruby. In C. Morgan R. Bird and J. Woodcock, editors, *Mathematics of Program Construction*, Lecture Notes in Computer Science 669, pages 208–232, 1992.

[19] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann Publ., 1992.

[20] C. Lengauer. Loop parallelization in the polytope model. In E. Best, editor, *CONCUR '93*, Lecture Notes in Computer Science 715, pages 398–416. Springer-Verlag, 1993.

[21] L. Meertens. Constructing a calculus of programs. In J. L. A. van de Snepscheut, editor, *Mathematics of Program Construction*, Lecture Notes in Computer Science 375, pages 66–90, 1989.

[22] J. O'Donnell. A correctness proof of parallel scan. *Parallel Processing Letters*. To appear.

[23] H. Partsch. Some experiments in transforming towards parallel executability. In R. Paige, J. Reif, and R. Wachter, editors, *Parallel Algorithm Derivation and Program Transformation*, pages 71–110. Kluwer Academic Publishers, 1993.

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

[25] P. Pepper, J. Exner, and M. Südholt. Functional development of massively parallel programs. In M. Broy D. Bjorner and I. Pottosin, editors, *Formal Methods in Programming and Their Applications*, Lecture Notes in Computer Science 735, pages 217–238. 1993.

[26] M. J. Quinn. *Parallel Computing*. McGraw-Hill, Inc., 1994.

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

[28] D. B. Skillicorn. Deriving parallel programs from specifications using cost information. *Science of Computer Programming*, 26:205–221, 1993.

[29] D.B. Skillicorn and W. Cai. Equational code generation: Implementing categorical data types for data parallelism. In *Proceedings of TENCON '94*, Singapore, 1994.

[30] D. Smith and M. Lowry. Algorithm theories and design tactics. In J. L. A. van de Snepscheut, editor, *Mathematics of Program Construction*, Lecture Notes in Computer Science 375, pages 379–398, 1989.

[31] M. Südholt. Data distribution algebras — a formal basis for programming using skeletons. In *Programming Concepts, Methods and Calculi (PROCOMET'94)*, pages 38–57, 1994.

[32] D. Walker. The design of a standard message passing interface for distributed memory concurrent computers. *Parallel Computing*, 20:657–673, 1994.