# Diffusion: Calculating Efficient Parallel Programs

Zhenjiang Hu, Masato Takeichi, Hideya Iwasaki

Department of Information Engineering
University of Tokyo
7-3-1 Hongo, Bunkyo, Tokyo 113, Japan
({hu,takeichi,iwasaki}@ipl.t.u-tokyo.ac.jp)

**Abstract**

Parallel primitives (skeletons) intend to encourage programmers to build a parallel program from ready-made components for which efficient implementations are known to exist, making the parallelization process easier. However, programmers often suffer from the difficulty to choose a combination of proper parallel primitives so as to construct efficient parallel programs. To overcome this difficulty, we shall propose a new transformation, called *diffusion*, which can efficiently decompose a recursive definition into several functions such that each function can be described by some parallel primitive. This allows programmers to describe algorithms in a more natural recursive form. We demonstrate our idea with several interesting examples. Our diffusion transformation should be significant not only in development of new parallel algorithms, but also in construction of parallelizing compilers.

**Keywords**: Bird Meertens Formalisms, Data Parallelism, Parallelization, Skeletal Parallel programming

## 1 Introduction

*Data parallelism* is currently one of the most successful models for programming massively parallel computers, compared with *control parallelism* that is explored from the control structures [Pra92]. To support parallel programming, this model basically consists of two parts, namely

- *a parallel data structure* to model a uniform collection of data which can be organized in a way that each of its elements can be manipulated in parallel;
- *a set of parallel primitives* on the parallel data structure to capture parallel skeletons of interest, which can be used as building blocks to write parallel programs.

For instance, in the parallel language Nesl [Ble92], the parallel data structure is sequences, and the most important parallel primitives on sequences are *apply-to-each* and *scan*; and in the BMF parallel model [Bir87, Ski94], the parallel data structure is parallel lists, and the parallel primitives are mainly *map* and *reduce*.

This parallel model not only provides the programmer an easily understandable view of a single execution stream of a parallel program, but also makes the parallelization process easier because of explicit parallelism in the parallel primitives [HS86, Kar87, HL93].

Despite these promising features, the application of current data parallel programming suffers from a problem. Because parallel programming relies on a set of parallel primitives to specify parallelism, programmers often find it hard to choose proper parallel primitives and to integrate them well in order to develop efficient parallel programs. Consider, as an example, that we want to develop an efficient parallel program for the bracket matching problem [Col95, HT99]. This is a kind of language recognition problem, determining whether the brackets of many types in a given string are correctly matched. For example, the string "$g + \{[o + o] * d\}()$" is well matched, whereas "$b\{[a)d]$" is not. The problem itself is so simple, but it is far from being that simple to develop an *efficient* parallel program in terms of the specified set of parallel primitives, say only with map, reduce and scan.

Nevertheless, a simple straightforward sequential algorithm still exists by using a stack. Opening brackets are pushed, and each closing bracket is matched against the current stack top. Failure is indicated by a mismatch, by an empty stack when a match is required, or by a nonempty stack at the end of the scan of the input. Thus we come to the following naive program.

$$
\begin{array}{lcl}
bm\ [\,]\ s & = & isEmpty\ s \\
bm\ (a:x)\ s & = & \textbf{if}\ isOpen\ a\ \textbf{then}\ bm\ x\ (push\ a\ s) \\
 & & \textbf{else if}\ isClose\ a\ \textbf{then} \\
 & & \quad noEmpty\ s\ \wedge \\
 & & \quad match\ a\ (top\ s)\ \wedge \\
 & & \quad bm\ x\ (pop\ s) \\
 & & \textbf{else}\ bm\ x\ s
\end{array}
$$

A possible way to programming with parallel primitives is to use the multi-pass programming method, well-known in the sequential functional programming community [BW88]. We start by a naive specification (without any concern of efficiency) of the problem by a composition of several passes so that each pass can be described in terms of the parallel primitives, and then we optimize it by correctness-preserving program transformation. To be more concrete, an initial naive

In Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99). San Antonio, Texas, January 22-23, 1999.

program in terms of the parallel primitives may be something like

$$\begin{array}{rcl} prog & = & \cdots\ scan\ (\otimes)\ \circ\ reduce\ (\oplus)\ \cdots \\ & & scan\ (\odot)\ \circ\ map\ f\ \cdots \end{array}$$

This could be quite inefficient if expensive operations of $f$, $\otimes$, $\oplus$ and $\odot$ are used in the parallel primitives. To make it efficient, in the sequential programming we can adopt the fusion (deforestation) transformation [Wad88, Chi92] to merge several passes into a single pass resulting in a compact recursive program; but in the parallel programming we cannot do so easily. The major difficulty lies in the primitive-closed requirement that the fusion of two primitives should give a primitive again. In fact, it is known to be hard to establish a set of efficient calculational rules for such kind of fusion transformation [Bir87], and to construct a suitable cost model to guide derivation of efficient parallel programs [Ski94].

In this paper, we shall propose a new transformation, called *diffusion*, to calculate efficient parallel programs in terms of a fixed set of parallel primitives. In contrast to the well-known *fusion* transformation, diffusion efficiently decomposes a recursive definition into several functions such that each function can be described using a parallel primitive, allowing programmers to define their algorithms in a natural recursive form. We shall adopt Bird Meertens Formalisms as our abstract parallel computation model [Bir87, Ski94]. Our main contributions are as follows.

- We propose a novel theorem for diffusion transformation (Section 3.1) in a calculational form [THT98]. Our diffusion theorem integrates the existing parallelization technique [HTC98], and generalizes the well-known homomorphism lemma [Bir87, Col95], with a nice use of scan to deal with accumulating parameters in recursive definitions.
- Our diffusion transformation can be applied to a wide class of recursive definitions. In fact, the recursive form in the diffusion theorem is very natural in its own right. Moreover, as illustrated in derivation of an efficient parallel program for bracket matching in Section 3.2, if combined with the normalization algorithm [HTC98] which is based on fusion and tupling calculation, a wider class of recursive definitions can be turned into the form to which our diffusion theorem can be applied.
- We highlight how to generalize our idea from recursive definitions on lists to those on other data structures like trees. This indicates a new way to do parallel programming efficiently over trees or more general data structures, which has been argued to be important but difficult in data parallel programming [NO94, KC98].

In summary, our diffusion theorem provides a significant guide for both parallel programming by hand and automatic parallelization by machine. If a recursive definition is in the required form to which our diffusion theorem can be applied, then it can be automatically parallelized into a set of efficient parallel primitives.

The organization of this paper is as follows. In Section 2, we review the notational conventions and some basic concepts used in this paper, and explain the existing problem of parallel programming in Bird Meertens Formalisms. We propose our idea of diffusion transformation in Section 3, and generalize the idea to be applicable to functions on data structures other than lists in Section 4. Related work and discussions are given in Section 5.

## 2 BMF and Parallel Computation

In this section, we briefly review the notational conventions and some basic concepts in Bird Meertens Formalisms (BMF for short) [Bir87, Ski94], and point out some related results which will be used in the rest of this paper.

*Function application* is denoted by a space and the argument which may be written without brackets. Thus $f\,a$ means $f\,(a)$. Functions are curried, and application associates to the left. Thus $f\,a\,b$ means $(f\,a)\,b$. Function application binds stronger than any other operator, so $f\,a \oplus b$ means $(f\,a) \oplus b$, but not $f\,(a \oplus b)$. *Function composition* is denoted by a centralized circle $\circ$. By definition, we have $(f \circ g)\,a = f\,(g\,a)$. Function composition is an associative operator, and the identity function is denoted by $id$. Infix binary operators will often be denoted by $\oplus, \otimes$ and can be *sectioned*; an infix binary operator like $\oplus$ can be turned into unary functions by

$$(a\oplus)\,b = a \oplus b = (\oplus\,b)\,a.$$

Besides, function $zip$ which will be used later is informally defined by:

$$zip\ [x_1, x_2, \ldots, x_n]\ [y_1, \ldots, y_n] = [(x_1, y_1), \ldots, (x_n, y_n)].$$

Lists are finite sequences of values of the same type. A list is either empty, a singleton, or the concatenation of two other lists. We write $[\,]$ for the empty list, $[a]$ for the singleton list with element $a$ (and $[\cdot]$ for the function taking $a$ to $[a]$), and $x +\!\!+ y$ for the concatenation of two lists $x$ and $y$. Concatenation is associative, and $[\,]$ is its unit. For example, the term $[1] +\!\!+ [2] +\!\!+ [3]$ denotes a list with three elements, often abbreviated to $[1, 2, 3]$. We also write $a : xs$ for $[a] +\!\!+ xs$.

### 2.1 BMF: An Architecture Independent Parallel Model

It has been argued in [Ski90] that BMF [Bir87] is a nice architecture-independent parallel computation model, consisting of a small fixed set of specific higher order functions which can be regarded as parallel primitives suitable for parallel implementation. Three important higher order functions are *map*, *reduce* and *scan*.

Map is the operator which applies a function to every element in a list. It is written as an infix $*$. Informally, we have

$$k * [x_1, x_2, \ldots, x_n] = [k\,x_1, k\,x_2, \ldots, k\,x_n].$$

Reduce is the operator which collapses a list into a single value by repeated application of some associative binary operator. It is written as an infix $/$. Informally, for an associative binary operator $\oplus$, we have

$$\oplus/\ [x_1, x_2, \ldots, x_n] = x_1 \oplus x_2 \oplus \cdots \oplus x_n.$$

Scan is the operator that accumulates all intermediate results for computation of reduce. Informally, for an associative binary operator $\oplus$ with an unit $\iota_\oplus$, we have

$$\begin{aligned}&\oplus\!\!\#\,[x_1, x_2, \ldots, x_n]\\ &\quad = [\iota_\oplus, x_1, x_1 \oplus x_2, \ldots, x_1 \oplus x_2 \oplus \cdots \oplus x_n].\end{aligned}$$

It has been shown that map, reduce and scan have nice massively parallel implementations on many architectures [Ski90, Ble89]. If $k$ and an associative $\oplus$ use $O(1)$ parallel time, then $k*$ can be implemented using $O(1)$ parallel time, and both $\oplus/$ and $\oplus\!\!\#$ can be implemented using $O(\log N)$ parallel time ($N$ denotes the size of the list). For example, $\oplus/$ can be computed in parallel on a tree-like structure with the combining operator $\oplus$ applied in the nodes, while $k*$ is computed in parallel with $k$ applied to each of the leaves. The study on efficient parallel implementation of $\oplus\!\!\#$ can be found in [Ble89], though it is not so obvious.

A similar model to BMF that sounds more practical to use is the parallel functional language called Nesl [Ble92] that includes two principle parallel primitives, namely *apply-to-each* (like map) and *scan*.

### 2.2 Simple Diffusion: the Homomorphism Lemma

*List homomorphisms* (or *homomorphisms* for short) [Bir87] are those *recursive* functions on finite lists that *promote* through list concatenation. A function $h$ satisfying the following equations is called a *list homomorphism*:

$$\begin{array}{lcl} h\ [a] & = & k\,a \\ h\ (x +\!\!+ y) & = & h\,x \oplus h\,y \end{array}$$

where $\oplus$ is an *associative* binary operator. We write $([k, \oplus])$ for the unique function $h$. For example, the function $sum$, for summing up all elements in a list, can be defined as a homomorphism of $([id, +])$.

The relevance of homomorphisms to parallel programming is basically from the *homomorphism lemma* [Bir87]:

$$([k, \oplus]) = (\oplus/) \circ (k*)$$

saying that every list homomorphism can be *diffused* to be the composition of a reduce and a map. It follows that if we can derive list homomorphisms, then we can get corresponding parallel programs. Though being so simple, the homomorphism lemma plays an important role to bridge the gap between programs in recursive form and programs in compositional form, and it has led to surprisingly many good results [Gor96a, Gor96b, HIT97, HTC98]. The major reason is that list homomorphisms provide us a new interface to develop parallel programs.

The importance of using a recursion instead of map and reduce in parallel programming has greatly motivated us to study this simple diffusion in a more general and practical manner.

### 2.3 Limitation of the Simple Diffusion

The homomorphism lemma, a simple diffusion transformation, is nice, but a closer look at the lemma reveals a practical limitation in the case where the result of the application of $([k, \oplus])$ to a list returns a function instead of a basic value. Take a look at the following homomorphism $h$ for *psums* (computing the prefix sums of a list) derived in [HTC98].

$$\begin{array}{lcl} psums & : & [Int] \to [Int] \\ psums\ x & = & s\ \textbf{where}\ (s, g) = h\ x\ 0 \\ h\ [a]\ c & = & ([c + a], a) \\ h\ (x +\!\!+ y)\ c & = & \textbf{let}\ (s_x, g_x) = h\ x\ c \\ & & \quad\ (s_y, g_y) = h\ y\ (c + g_x) \\ & & \textbf{in}\ (s_x +\!\!+ s_y,\ g_x + g_{yx}) \end{array}$$

Function $h$ can be described in a more explicit homomorphic way by

$$\begin{array}{lcl} h & : & [Int] \to (Int \to [Int]) \\ h & = & ([k, \oplus]) \\ & & \textbf{where} \\ & & \quad k\ a = \lambda c.\,([c + a], a) \\ & & \quad h_x \oplus h_y = \lambda c.\,\textbf{let}\ (s_x, \underline{g_x}) = h_x\ c \\ & & \qquad\qquad\qquad\quad (s_y, g_y) = h_y\ \underline{(c + g_x)} \\ & & \qquad\qquad\quad \textbf{in}\ (s_x +\!\!+ s_y,\ g_x + g_y) \end{array}$$

The problem lies in the definition for $\oplus$ where $h_x$ and $h_y$ are functions. As indicated by the underlined parts above, the computation of $h_y$ cannot be performed until it receives $g_x$, one of the results from the computation of $h_x$. Thus, a naive implementation of such $\oplus$ may lead to a big function-closure, resulting in a sequential program. So we need to study carefully this dependency reflected in the use of accumulating parameter (like $c$ used in $h$) in the initial definition (for $h$).

Another practical concern [Gor96a, CTT97, HTC98] is that defining a function, say $h$, over join lists (as we do with list homomorphism) like

$$\begin{array}{lcl} h\ [a]\ c & = & \ldots \\ h\ (x +\!\!+ y)\ c & = & \ldots \end{array}$$

is much more difficult than doing over cons lists like

$$\begin{array}{lcl} h\ [\,]\ c & = & \ldots \\ h\ (a : x)\ c & = & \ldots \end{array}$$

Fortunately, with the technique of parallelization transformation [HTC98], we are able to allow functions to be defined in the latter easier form which will be handled later by our proposing diffusion.

## 3 Diffusion

Diffusion is a transformation turning a recursive definition into a composition of our higher order functions, namely *map*, *reduce* and *scan*. To be useful, our proposing diffusion transformation should satisfy the following two requirements.

- First, the diffusion transformation should be *powerful* enough to be applied to a wide class of recursive forms of interest.
- Second, the result parallel programs should be *efficient*, in the sense that if the original program uses $t$ sequential time, then the derived parallel one should take at most $O(\log N)$ times of the sequential time. Here and after, we often use $N$ to denote the size of the input list.

### 3.1 Diffusion Theorem

To meet our requirements, we should carefully choose proper recursive forms to diffuse We will be generous with paper space to show how we reach our target form in this section.

#### The simplest form: diffused to `reduce`

We start by considering the following most natural and simplest recursive form defined over the cons lists:

$$\begin{array}{lcl} h\ [\,] & = & e \\ h\ (a:x) & = & a\ \oplus\ h\ x. \end{array}$$

Here, by choosing suitable $e$ and $\oplus$, we are able to define many functions of our interest. This kind of definition should be familiar enough to the functional programming community, which is known to be *catamorphism* [MFP91] or *foldr* in Haskell. To diffuse this recursive form into our parallel primitives, we require $\oplus$ to be associative, and thus have

$$h\ x = (\oplus/x)\ \oplus\ e.$$

The correctness of this simple diffusion is obvious, and the efficiency requirement is satisfied: the sequential program uses $O(N \times t_\oplus)$ sequential time while the derived version use $O(\log N \times t_\oplus)$ parallel time, where $t_\oplus$ denotes the time to compute $\oplus$.

#### The form with computation on elements: diffused to `reduce` and `map`

We, however, should not be satisfied with the power of the simplest form. Even for the following simple function for squaring each element of a list

$$\begin{array}{lcl} sqrs\ [\,] & = & [\,] \\ sqrs\ (a:x) & = & sqr\ a\ :\ sqrs\ x \end{array}$$

we cannot diffuse it, because the $\oplus$ here is defined by

$$a \oplus b = sqr\ a : b$$

which is not associative since $a$ and $b$ have the different types. Nevertheless, with the normalization algorithm given in [HTC98], we can turn the second equation of $sqrs$ to

$$sqrs\ (a:x) = \underline{[sqr\ a]}\ +\!\!+\ \underline{sqrs\ x}.$$

Here $+\!\!+$ is an associative operator used to combine a *computation* on $a$ with the recursive part, as indicated by the underlined parts. In fact we can and should allow such computation on $a$. This leads to the following improved recursive form, permitting a computation on $a$ by any function $k$.

$$\begin{array}{lcl} h\ [\,] & = & e \\ h\ (a:x) & = & k\ a\ \oplus\ h\ x \end{array}$$

This recursion can be diffused into the following if $\oplus$ is associative.

$$h\ x = ((\oplus/) \circ k*)\ x\ \oplus\ e$$

If $e$ is the unit of $\oplus$, this can be reduced to

$$h = (\oplus/) \circ k*$$

which is the same as the homomorphism lemma except that $h$ is defined as a recursion on cons lists rather than that on join lists.

#### The form with accumulating parameters: diffused to `reduce`, `map` and `scan`

After discussing the two recursive forms that can be diffused, we are ready to solve the problem in Section 2.3, in which $h$ is a function whose application to a list still gives a function. To simplify our presentation, we shall focus ourselves on those the functions that have an accumulating parameter as in the following recursive form.

$$\begin{array}{lcl} h & : & [I] \to A \to O \\ h\ [\,]\ c & = & g_1\ c \\ h\ (a:x)\ c & = & k(a,c)\ \oplus\ h\ x\ (c \otimes g_2\ a) \end{array}$$

Note that $h$ has two parameters; the first is the inductive one and the second is the accumulating one. This recursive form extends the above second one with an accumulating parameter. To be specific, the computation on element $a$ can include the use of the accumulating parameter $c$, and the accumulating parameter in the recursive call can be updated with a combination of some computation on $a$ using a binary, associative operator $\otimes$.

We need to deal with such additional accumulating parameter $c$. Our idea is to remove it from the recursion by precomputation. We make the accumulating parameter look like a constant by precomputing all the $c$, say $cs$, that will be used during computation of $h$. The trick, and also an important point here, is our use of scan, an efficient parallel primitive, to perform such precomputation in a parallel way:

$$cs = (c\otimes) * (\otimes\!\!\#\ (g_2 * x)).$$

Now using the diffusion for the form that does not contain accumulating parameters while paying attention to the access to corresponding elements of $cs$, we can come up with our diffusion theorem.

**Theorem 1 (Diffusion)** Given a function $h$ defined in the above recursive form, if $\oplus$ and $\otimes$ are associative and have units, then $h$ can be diffused into the following.

$$\begin{array}{lcl} h\ x\ c & = & \textbf{let}\ cs' +\!\!+ [c'] = (c\otimes) * (\otimes\!\!\#(g_2 * x)) \\ & & \quad ac = zip\ x\ cs' \\ & & \textbf{in}\ (\oplus/\ (k * ac)) \oplus (g_1\ c') \end{array}$$

*Proof*: We can prove that the newly defined $h$ is equivalent to the original, by induction on the inductive parameter $x$, as shown by the following calculation.

- Base case $x = [\,]$.

$$\begin{array}{lcl} & & h\ [\,]\ c \\ & = & \quad \{\ \text{by the new definition}\ \} \\ & & \textbf{let}\ cs' +\!\!+ [c'] = (c\otimes) * (\otimes\!\!\#(g_2 * [\,])) \\ & & \quad ac = zip\ x\ cs' \\ & & \textbf{in}\ (\oplus/\ (k * ac)) \oplus (g_1\ c') \\ & = & \quad \{\ \text{map}\ \} \\ & & \textbf{let}\ cs' +\!\!+ [c'] = (c\otimes) * (\otimes\!\!\#[\,]) \\ & & \quad ac = zip\ x\ cs' \\ & & \textbf{in}\ (\oplus/\ (k * ac)) \oplus (g_1\ c') \end{array}$$

$$
\begin{array}{ll}
= & \quad \{ \text{ scan } \} \\
& \textbf{let } cs' +\!\!+ [c'] = (c\otimes) * [\iota_\otimes] \\
& \quad\; ac = zip\ x\ cs' \\
& \textbf{in } (\oplus/\,(k * ac)) \oplus (g_1\ c') \\
= & \quad \{ \text{ map, } c \otimes \iota_\otimes = c \ \} \\
& \textbf{let } cs' +\!\!+ [c'] = [c] \\
& \quad\; ac = zip\ x\ cs' \\
& \textbf{in } (\oplus/\,(k * ac)) \oplus (g_1\ c') \\
= & \quad \{ \text{ by pattern matching: } cs' = [\,],\ c = c' \ \} \\
& \textbf{let } e = g_1\ c \\
& \quad\; ac = zip\ x\ [\,] \\
& \textbf{in } (\oplus/\,(k * ac)) \oplus (g_1\ c') \\
= & \quad \{ \text{ zip, let } \} \\
& (\oplus/\,(k * [\,])) \oplus (g_1\ c) \\
= & \quad \{ \text{ map, } \oplus \ \} \\
& \iota_\oplus \oplus (g_1\ c) \\
= & \quad \{ \ \iota_\oplus \text{ is a unit of } \oplus \ \} \\
& g_1\ c
\end{array}
$$

- Inductive case $x = a : x'$.

$$
\begin{array}{ll}
& h\ (a : x')\ c \\
= & \quad \{ \text{ by the new definition } \} \\
& \textbf{let } cs' +\!\!+ [c'] = (c\otimes) * (\otimes\!/\!/(g_2 * (a : x'))) \\
& \quad\; ac = zip\ x\ cs' \\
& \textbf{in } (\oplus/\,(k * ac)) \oplus (g_1\ c') \\
= & \quad \{ \text{ map } \} \\
& \textbf{let } cs' +\!\!+ [c'] = (c\otimes) * (\otimes\!/\!/(g_2\ a : g_2 * x')) \\
& \quad\; ac = zip\ x\ cs' \\
& \textbf{in } (\oplus/\,(k * ac)) \oplus (g_1\ c') \\
= & \quad \{ \text{ definition of scan [Bir87] } \} \\
& \textbf{let } cs' +\!\!+ [c'] = (c\otimes) * (\iota_\otimes : (g_2\ a \otimes) * \\
& \qquad\qquad\qquad \otimes\!/\!/(g_2 * x')) \\
& \quad\; ac = zip\ x\ cs' \\
& \textbf{in } (\oplus/\,(k * ac)) \oplus (g_1\ c') \\
= & \quad \{ \text{ map, associativity of } \otimes \ \} \\
& \textbf{let } cs' +\!\!+ [c'] = c : ((c \otimes g_2\ a) \otimes) * (\otimes\!/\!/(g_2 * x')) \\
& \quad\; ac = zip\ x\ cs' \\
& \textbf{in } (\oplus/\,(k * ac)) \oplus (g_1\ c') \\
= & \quad \{ \ ((c \otimes g_2\ a) \otimes) * (\otimes\!/\!/(g_2 * x')) \text{ is not empty } \} \\
& \textbf{let } cs'' +\!\!+ [c''] = ((c \otimes g_2\ a) \otimes) * (\otimes\!/\!/(g_2 * x')) \\
& \quad\; ac = zip\ (a : x')\ (c : cs'') \\
& \textbf{in } (\oplus/\,(k * ac)) \oplus (g_1\ c'') \\
= & \quad \{ \ \text{zip } \} \\
& \textbf{let } cs'' +\!\!+ [c''] = ((c \otimes g_2\ a) \otimes) * (\otimes\!/\!/(g_2 * x')) \\
& \quad\; ac'' = zip\ x'\ cs'' \\
& \quad\; ac = (a, c) : ac'' \\
& \textbf{in } (\oplus/\,(k * ac)) \oplus (g_1\ c'') \\
= & \quad \{ \ \text{map, } \oplus \ \} \\
& \textbf{let } cs'' +\!\!+ [c''] = ((c \otimes g_2\ a) \otimes) * (\otimes\!/\!/(g_2 * x')) \\
& \quad\; ac'' = zip\ x'\ cs'' \\
& \textbf{in } k(a, c) \oplus (\oplus/\,(k * ac'')) \oplus (g_1\ c'') \\
= & \quad \{ \ \text{let, associativity of } \oplus \ \} \\
& k(a, c) \oplus \\
& \textbf{let } cs'' +\!\!+ [c''] = ((c \otimes g_2\ a) \otimes) * (\otimes\!/\!/(g_2 * x')) \\
& \quad\; ac'' = zip\ x'\ cs'' \\
& \textbf{in } (\oplus/\,(k * ac'')) \oplus (g_1\ c'') \\
= & \quad \{ \ \text{inductive hypothesis } \} \\
& k(a, c) \oplus h\ x'\ (c \oplus g_2\ a)
\end{array}
$$

□

Note that we use the matching notation of $cs' +\!\!+ [c']$ to extract the leading part and the last element from a list. It is not difficult to check that this diffusion indeed meets our requirements as given at the beginning of this section.

To see a simple use of this theorem, consider the following function *sbp* to solve a simplified bracket matching problem: determining whether a single type (not many types) of brackets '(' and ')' are matched in a given string. It uses a counter (starting with zero) to increase upon meeting '(' and to decrease upon meeting ')':

$$
\begin{array}{lcl}
sbp\ [\,]\ c & = & c == 0 \\
sbp\ (a : x)\ c & = & \textbf{if } a == \text{'('} \textbf{ then } sbp\ x\ (c + 1) \\
& & \textbf{else if } a == \text{')'} \textbf{ then} \\
& & \quad c > 0\ \wedge\ sbp\ x\ (c - 1) \\
& & \textbf{else } sbp\ x\ c.
\end{array}
$$

Merging all recursive calls by the normalization algorithm in [CDG96] will give

$$
\begin{array}{lcl}
sbp\ [\,]\ c & = & g_1\ c \\
sbp\ (a : x)\ c & = & k(a, c)\ \wedge\ sbp\ x\ (c + g_2\ a)
\end{array}
$$

where

$$
\begin{array}{lcl}
g_1\ c & = & c == 0 \\
k(a, c) & = & \textbf{if } a == \text{'('} \textbf{ then } True \\
& & \textbf{else if } a == \text{')'} \textbf{ then } c > 0 \textbf{ else } True \\
g_2\ a & = & \textbf{if } a == \text{'('} \textbf{ then } 1 \\
& & \textbf{else if } a == \text{')'} \textbf{ then } (-1) \textbf{ else } 0
\end{array}
$$

which is in the right form that the diffusion theorem can be applied to get an efficient parallel program for *sbp*. It is worth noting that this problem was considered as a kind of hard parallelization problem in [Col95]. By using the diffusion theorem, its efficient parallel program can be obtained by a straightforward program calculation.

### 3.2 Diffusion Algorithm

Although we have argued that our recursive form is powerful and general, user's programs may not be exactly in our form. Therefore, we turn to find a way to transform more general programs into the form that our diffusion theorem can be applied. The diffusion algorithm is for this purpose.

We shall illustrate our algorithm by the derivation of an explicit parallel algorithm for bracket matching in terms of our parallel primitives from the naive program given in the introduction.

#### Step 1: Linearizing Recursive Calls

It is required by the diffusion theorem that the occurrences of the recursive call should appear once. If there are many occurrences, we need to merge them into a single one. Recall the definition of *bm* in the introduction. In the branch of $(a : x)$, there are three occurrences of the recursive call to *bm* in the right hand side. We can merge them based on the normalization algorithm [CDG96].

$$
\begin{array}{lcl}
bm\ [\,]\ s & = & isEmpty\ s \\
bm\ (a : x)\ s & = & g_1\ (a, s)\ \wedge\ bm\ x\ (g_2\ a\ s)
\end{array}
$$

where

$$
\begin{array}{lcl}
g_1\ (a, s) & = & \textbf{if } isOpen\ a \textbf{ then } True \\
& & \textbf{else if } isClose\ a \textbf{ then} \\
& & \quad noEmpty\ s\ \wedge \\
& & \quad match\ a\ (top\ s) \\
& & \textbf{else } True \\
g_2\ a\ s & = & \textbf{if } isOpen\ a \textbf{ then } push\ a\ s \\
& & \textbf{else if } isClose\ a \textbf{ then } pop\ s \textbf{ else } s.
\end{array}
$$

### Step 2: Identifying Associative Operators

Central to our diffusion theorem is the use of associativity of the binary operators $\oplus$ and $\otimes$. Clearly, $\oplus$ should be an associative operator over the resulting domain of function $h$, while $\otimes$ is an associative operator over the resulting domain of the accumulating parameter (e.g., $s$ for $bm$).

For $bm$, it is easy to find that $\oplus$ is $\wedge$, but not so easy to find what corresponds to $\otimes$ which is supposed to combine two stacks, satisfying

$$g_2\ a\ s = s \otimes g_2'\ a.$$

Consider the following stack we would like to use in $bm$:

$$Stack\ \alpha\ =\ Empty \mid Push\ \alpha\ Stack \mid Pop\ Stack$$

From this definition, we are able to systematically derive the following associative operator $\otimes$ for combining two stacks as shown in [SF93, HT99].

$$\begin{array}{lcl} s \otimes Empty & = & s \\ s \otimes (Push\ a\ s') & = & Push\ a\ (s \otimes s') \\ s \otimes (Pop\ s') & = & Pop\ (s \otimes s') \end{array}$$

Using this $\otimes$, we thus have

$$\begin{array}{lcl} g_2\ a\ s & = & s \otimes\ g_2'\ a \\ g_2'\ a & = & \textbf{if}\ isOpen\ a\ \textbf{then}\ Push\ a\ Empty \\ & & \textbf{else if}\ isClose\ a\ \textbf{then} \\ & & \quad Pop\ Empty\ \textbf{else}\ Empty. \end{array}$$

### Step 3: Applying the Diffusion Theorem

After merging recursive call occurrences and identifying associative operators, we are ready to apply the diffusion theorem.

For $bm$, it follows from the diffusion theorem that

$$\begin{array}{lcl} bm\ x\ c & = & \textbf{let}\ cs' +\!\!+ [c'] = (c\otimes) * (\otimes /\!\!/ (g_2 * x)) \\ & & \quad ac = zip\ x\ cs' \\ & & \textbf{in}\ (\wedge/\ (g_1 * ac)) \wedge (isEmpty\ c') \end{array}$$

### Step 4: Optimizing Operators

So far we have derived a parallel program that is described in terms of our parallel primitives. According to our cost model for parallel primitives, we should continue to find efficient implementation for the operations like $g_1$, $g_2$, $k$, $\oplus$ and $\otimes$ that are used in each parallel primitive to obtain a more efficient parallel program.

For $bm$, we need to show that $\otimes$ as well as the stack operations can be implemented in $O(1)$ parallel time if we want an $O(\log N)$ parallel program for bracket matching. Notice that with the property of $Pop\ (Push\ a\ s) = s$, our stack should, as discussed in [HT99], keep the form of

$$Push\ a_1\ (\ldots(Push\ a_n\ (\underbrace{Pop\ (\ldots(Pop}_{m}\ Empty)))))),$$

that can be naturally represented by

$$([a_1, \cdots, a_n], n, m)$$

Here, $m$ denotes the number of $Pop$ occurrences, and the second component $n$ is added to incrementally compute the length of the first component. With this new representation, we can refine all operations on stack to those using $O(1)$ parallel time as follows.

$$\begin{array}{lcl} Empty & = & ([\,], 0, 0) \\ isEmpty\ ([\,], 0, 0) & = & True \\ isEmpty\ _ & = & False \\ Push\ c\ (cs, n, m) & = & ([c] +\!\!+ cs, n+1, m) \\ Pop\ (c : cs, n+1, m) & = & (cs, n, m) \\ Pop\ ([\,], 0, m) & = & ([\,], 0, m+1) \end{array}$$

And

$$\begin{array}{l} (cs_1, n_1, m_1) \otimes (cs_2, n_2, m_2) \\ \quad = \textbf{if}\ m_1 \geq n_2\ \textbf{then}\ (cs_1, n_1, m_1 - n_2 + m_2) \\ \qquad \textbf{else}\ (cs_1 +\!\!+ drop\ m_1\ cs_2, n_1 + n_2 - m_1, m_2) \end{array}$$

Function $drop\ n\ x$ is to drop the first $n$ elements from list $x$. According to the fact that $+\!\!+$ and $drop$ can be implemented using constant parallel time (e.g., under the PRAM parallel model [Ski94]), our final operators of $\otimes$, $g_1$, and $g_2$ can be implemented using $O(1)$ parallel time, and we thus got an $O(\log N)$ parallel program for bracket matching.

It has been shown that the bracket matching problem can be solved in $O(\log N)$ parallel time [GR88] where $N$ denotes the length of the input string, but the algorithm involved is rather complicated and its correctness is difficult to prove. To resolve this problem, Cole [Col95] proposed an *informal* development of an *suboptimal* $O(\log^2 n)$ parallel algorithm. In contrast, we propose a formal development of a novel parallel one to solve this problem. In [HT99], we proposed an homomorphic algorithm for the same problem but left as an open work for the derivation of an explicit Nesl parallel program in terms of parallel primitives.

## 4 Polytypic Diffusion

In this section, we highlight how to generalize the idea of diffusion of recursive definitions on lists to those on other data structures like trees. Rather than giving a formal study of this generalization, we shall concentrate ourselves on trees, and explain our idea in a concrete manner, It should not be difficult at all to generalize from trees to other data structures.

### 4.1 Tree Parallel Primitives

We consider binary trees defined by

$$Tree\ \alpha = Leaf\ \alpha \mid Node\ \alpha\ (Tree\ \alpha)\ (Tree\ \alpha).$$

Based on the constructive algorithmics [MFP91, Fok92a], we can define a set of tree parallel primitives by a natural generalization of those primitives on lists.

#### Map

*Map* is to apply two functions on a tree; one to all leaves and the other to all internal nodes.

$$\begin{array}{lcl} map\ f_1\ f_2\ (Leaf\ a) & = & Leaf\ (f_1\ a) \\ map\ f_1\ f_2\ (Node\ a\ l\ r) & = & Node\ (f_2\ a) \\ & & \quad (map\ f_1\ f_2\ l) \\ & & \quad (map\ f_1\ f_2\ r) \end{array}$$

The parallelism in map should be obvious. For example, using enough processors we can easily implement it in $O(max(T(f_1),T(f_2)))$ parallel time, where $T(f_1)$ and $T(f_2)$ denote the time for computing $f_1$ and $f_2$ respectively.

**Scan**

Formal study of binary tree scans (downwards and upwards accumulations) can be found in [Gib92, BdM96]. We have two kinds of scan: scanning a tree upwards or downwards. They will be called *upward scan*, denoted by $scan_u$, and *downward scan*, denoted by $scan_d$, respectively.

Upward scan computes sum of all elements with a binary operator $\oplus$, while keeping all running sums during upwards computation. Like list scans requiring a binary operator that are associative, our tree scans relies on a binary operator that are both associative and commutative, which is sufficient (not necessary) to guarantee their efficient parallel implementation.

Given an associative and commutative operator $\oplus : A \to A \to A$, $scan_u$ is defined by

$$\begin{array}{lcl} scan_u\ (\oplus)\ (Leaf\ a) & = & Leaf\ a \\ scan_u\ (\oplus)\ (Node\ a\ l\ r) & = & \textbf{let}\ l' = scan_u\ (\oplus)\ l \\ & & \quad\ r' = scan_u\ (\oplus)\ r \\ & & \textbf{in}\ Node \\ & & \quad (a \oplus root\ l' \oplus root\ r') \\ & & \quad l'\ r' \end{array}$$

where

$$\begin{array}{lcl} root\ (Leaf\ a) & = & a \\ root\ (Node\ a\ l\ r) & = & a. \end{array}$$

Downward scan $scan_d$ is to propagate information from the root to the leaves with some computation at the internal nodes by an associative $\oplus$.

$$\begin{array}{l} scan_d\ (\oplus)\ g_1\ g_2\ (Leaf\ a)\ c \\ \quad = Leaf\ c \\ scan_d\ (\oplus)\ g_1\ g_2\ (Node\ a\ l\ r)\ c \\ \quad = Node\ c\ (scan_d\ (\oplus)\ g_1\ g_2\ l\ (c \oplus g_1\ a))) \\ \qquad\qquad (scan_d\ (\oplus)\ g_1\ g_2\ r\ (c \oplus g_2\ a)) \end{array}$$

Efficient implementation of the scans is not so obvious. Fortunately, so many studies have been devoted to show that the tree contraction technique [LF80, TV84, MR85, GMT87, ADKP87, Ble89] can be applied to implement our scans efficiently, and some more concrete studies can be found [GCS94, Gib96, Ski96]. We do not recapitulate them, rather we summarize the result. For the upward scan, the parallel time is $O(T(\oplus) \times \log N)$ with $N/\log N$ processors, where $N$ denotes the number of tree nodes (no matter how unbalanced the tree is). For the downward scan, the parallel time is $O(T(\oplus) \times \log N + max(T(g_1), T(g_2)))$.

**Reduce**

Generalizing reduce from that on lists is straightforward. Given an associative and commutative operator $\oplus : A \to A \to A$, *reduce* is defined by

$$\begin{array}{l} reduce\ (\oplus)\ (Leaf\ a) = a \\ reduce\ (\oplus)\ (Node\ a\ l\ r) \\ \quad = a\ \oplus\ reduce\,(\oplus)\,l\ \oplus\ reduce\,(\oplus)\,r \end{array}$$

The reduce can be implemented in parallel by using the tree contraction technique similarly to the upward scan.

**Zip**

Zip merges two trees of the same form into one by pairwisely gluing elements.

$$\begin{array}{l} zip\ (Leaf\ a_1)\ (Leaf\ a_2) = Leaf\ (a_1, a_2) \\ zip\ (Node\ a_1\ l_1\ r_1)\ (Node\ a_2\ l_2\ r_2) \\ \quad = Node\ (a_1, a_2)\ (zip\ l_1\ l_2)\ (zip\ r_1\ r_2) \end{array}$$

This definition can be extended from two data to any number of data. The parallelism in zip is also obvious.

### 4.2 Tree Diffusion Theorem

We now generalize the diffusion theorem from list functions to tree functions.

**Theorem 2 (Tree Diffusion)** Let $h : Tree\ \alpha \to A \to O$ be defined in the following recursive way:

$$\begin{array}{lcl} h\ (Leaf\ a)\ c & = & k_1(a, c) \\ h\ (Node\ a\ l\ r)\ c & = & k_2(a, c)\ \oplus \\ & & \quad h\ l\ (c \otimes g_1\ a)\ \oplus \\ & & \quad h\ r\ (c \otimes g_2\ a) \end{array}$$

where $\oplus : O \to O \to O$ is an associative and commutative operator, $\otimes : A \to A \to A$ is an associative operator, and $k_1$, $k_2$, $g_1$ and $g_2$ are given functions. Then, $h$ can be equivalently defined by

$$\begin{array}{lcl} h\ x\ c & = & \textbf{let}\ cs = scan_d\ (\otimes)\ g_1\ g_2\ x\ c \\ & & \quad\ ac = zip\ x\ cs \\ & & \textbf{in}\ reduce\ (\oplus)\ (map\ k_1\ k_2\ ac). \end{array}$$

*Proof Sketch*: This can be proved by induction on the structure of $x$, quite similar to what we did for the the proof of Theorem 1. □

It is worth noting that the tree diffusion theorem is quite similar to the list diffusion theorem but in a more compact form due to our generalized definition of map and scan.

The tree diffusion theorem can be degenerated to the following corollary where $h$ does not use any accumulating parameter.

**Corollary 3** Let $h : Tree\ \alpha \to O$ be defined in the following recursive way:

$$\begin{array}{lcl} h\ (Leaf\ a) & = & k_1\ a \\ h\ (Node\ a\ l\ r) & = & k_2\ a\ \oplus\ h\,l\ \oplus\ h\,r \end{array}$$

where $\oplus : O \to O \to O$ is an associative and commutative operator, $k_1$ and $k_2$ are given functions. Then,

$$h\ x\ =\ reduce\ (\oplus)\ (map\ k_1\ k_2\ x)$$ □

This corollary is similar to the homomorphism lemma in Section 2.2. Another corollary, focusing on treating manipulation of the accumulating parameter, is obtained by eliminating the last reduce step in the new definition of $h$ in the tree diffusion theorem.

**Corollary 4** Let $h : Tree\ \alpha \to A \to Tree\ \beta$ be defined in the following recursive way:

$$\begin{array}{lcl} h\ (Leaf\ a)\ c & = & Leaf\ (k_1(a, c)) \\ h\ (Node\ a\ l\ r)\ c & = & Node\ (k_2(a, c)) \\ & & \quad (h\ l\ (c \otimes g_1\ a)) \\ & & \quad (h\ r\ (c \otimes g_2\ a)) \end{array}$$

where $\otimes : A \to A \to A$ is associative, and $k_1$, $k_2$, $g_1$ and $g_2$ are given functions. Then, $h$ can be equivalently defined by

$$\begin{array}{lcl} h\ x\ c & = & \textbf{let}\ cs = scan_d\ (\otimes)\ g_1\ g_2\ x\ c \\ & & \textbf{in}\ map\ k_1\ k_2\ (zip\ x\ cs). \end{array}$$

□

To see how tree diffusion theorem works, consider the following naive solution to the problem to number each node and leaf of a tree in an infix traversing order:

$$\begin{array}{lcl} nt\ (Leaf\ a)\ c & = & Leaf\ c \\ nt\ (Node\ a\ l\ r)\ c & = & Node\ (c + size\ l) \\ & & \quad (nt\ l\ c) \\ & & \quad (nt\ r\ (c + size\ l + 1)). \end{array}$$

Here $size$, computing the number of nodes of a tree, is defined by

$$\begin{array}{lcl} size\ (Leaf\ a) & = & 1 \\ size\ (Node\ a\ l\ r) & = & 1 + size\ l + size\ r \end{array}$$

or simply by

$$size\ t = reduce\ (+)\ (map\ (\lambda x.1)\ (\lambda x.1)\ t).$$

We number a tree by using a counter $c$ (starting from an initial value). It is actually not easy to derive an $O(\log N)$ ($N$ denotes the number of tree nodes) parallel program, because of two seemingly sequential factors in the above naive specification, the counter $c$ and a probably very unbalanced tree, which may sequentialize the visit of each node.

We cannot directly apply the tree diffusion theorem to parallelize $nt$, because $nt$ uses an auxiliary function $size$. Fortunely, this can be easily handled by sort of memoisation; for the given tree $t$, we derive the following $scansize$ from $size$, which can memoise the size of the tree rooted at each tree node

$$scansize\ t = scan_u\ (+)\ (map\ (\lambda x.1)\ (\lambda x.1)\ t).$$

Suppose that we have a parallel operation $getchildren$ that can replace each node in parallel with a pair of the root values of its two children, then we can associate the children's sizes to each node of the original tree by

$$tup\ t = zip\ (getchildren\ (scansize\ t))\ t.$$

We then define

$$nt\ t = nt'\ (tup\ t)$$

and a simple calculation can yield the following definition for $nt'$.

$$\begin{array}{lcl} nt'\ (Leaf\ ((s_l, s_r), a)\ c & = & Leaf\ c \\ nt'\ (Node\ ((s_l, s_r), a)\ l\ r)\ c & = & Node\ (c + s_l) \\ & & \quad (nt\ l\ c) \\ & & \quad (nt\ r\ (c + (s_l + 1))). \end{array}$$

Now we can apply Corollary 4 to obtain the following explicit parallel code for $nt'$.

$$\begin{array}{lcl} nt'\ x\ c & = & \textbf{let}\ cs = scan_d\ (\otimes)\ g_1\ g_2\ x\ c \\ & & \textbf{in}\ map\ k_1\ k_2\ (zip\ x\ cs) \end{array}$$

where

$$\begin{array}{lcl} k_1\ (((s_l.s_r), a), c) & = & c \\ k_2\ (((s_l.s_r), a), c) & = & c + s_l \\ g_1\ ((s_l.s_r), a) & = & 0 \\ g_2\ ((s_l.s_r), a) & = & s_l + 1. \end{array}$$

## 5 Related Work and Discussions

Besides the related work as in the introduction, our work is closely related to three kinds of active research work, namely parallel programming in BMF, parallel programming with scans, and polytypic programming.

Parallel Programming in BMF has been attracting many researchers. The initial BMF [Bir87] was designed as a calculus for deriving (sequential) efficient programs on lists. Skillicorn [Ski90] showed that BMF could also provide an architecture-independent parallel model for parallel programming because a small fixed set of higher order functions in BMF such as map, reduce can be mapped efficiently to a wide range of parallel architectures. Along with the extension of BMF from the theory of lists to the uniform theory of most data types, Skillicorn [Ski93b, Ski94, Ski96] coincided these data types as *categorical data types*, and established an architecture-independent cost model for generic catamorphisms. This influence our definitions of parallel primitives over data structures like trees.

Despite the architecture-independent cost model for the extended BMF, we are lacking of powerful parallelization theorem and laws for calculating efficient parallel programs, which more or less prevents it from being widely used. To remedy this situation, Quite a lot of recent studies have been devoted to the development of powerful parallelization methods with BMF [Ski93a, Col95, Gor96b, Gor96a, GDH96, HIT97, HTC98]. As explained in Section 2, the main idea is based on derivation of list homomorphism from a naive specification. This is based on the fact that a list homomorphism can be efficiently implemented by a composition of two parallel primitives, namely reduce and map. Our uniform recursions for structuring the parallel primitives as in the diffusion theorem is more general and easier to be used in programming than list homomorphisms. And our diffusion theorem can be considered as an extension of the homomorphism lemma. Our explicit use of accumulating parameters in recursive definitions and our use of scan for memoization in a parallel way are quite new.

Parallel programming with scans (either on lists or trees) is not new. For example, scan on lists is argued to be an important parallel skeleton [Ble89], and is used as one of the two important parallel constructs in Nesl [Ble92]. However, if we look at those programs in Nesl, they only contain some very simple use of scan (with simple operations like +). It lacks systematic way to develop parallel programs with scans. It might be difficult, even for an Nesl expert, to write an efficient program to solve our running example of bracket matching, because scans with complicated operations needs to be carefully designed.

Formal study of binary tree scans (downwards and upwards accumulations) can be found in [Gib92, BdM96], but to ensure the existence of efficient parallel implementation the complicated "cooperation condition" must be checked. In contrast, we give a more natural definition using an explicit accumulating parameter, and simplify the condition to guarantee the existence of efficient parallel implementation.

Polytypic programming [JJ96, JJ97] are widely used in the Squigol community [Mal89, Fok92b, MFP91], but its importance in parallel programming has not been well recognized. Starting with [BdM96], more and more algorithmic problems have been considered in a polytypic setting [dM95, Jeu95, Mee96, JJ96]. In this paper, we made an at-

tempt to apply polytypic idea to the development of parallel algorithms.

This work is a continuation of our effort to apply the so-called program calculation technique [THT98] to the development of efficient parallel programs [HIT97, HTC98]. As a matter of fact, our diffusion theorem is much related to our previous parallelization theorem [HTC98]; the parallelization theorem only derives a homomorphic program whereas diffusion theorem gives an explicit parallel program using parallel primitives. Nevertheless, the algorithm to turn a general program to the form that the parallelization theorem can be applied [HTC98] has been borrowed here as shown in Section 3.2.

We are working on a precise definition of the class of recursive definitions that can be parallelized into parallel primitives, and on formalization of the diffusion algorithm in a more mechanical way. Although we have not verify our idea in a practical system yet, we believe that it is promising to be used practically. We are going to embed our idea in, for example, the Nesl-like system in the future.

Another interesting future work is to deal with diffusion of polymorphic recursive definitions into a set of communication primitives, which should be very important to manipulate data communications and distributions.

## Acknowledgement

This paper owes much to the thoughtful and helpful discussions with Wei Ngan Chin during his visit of Tokyo University on efficient implementation of list homomorphisms using scan. Thanks are also to Manuel M. T. Chakravarty who suggested us to try deriving parallel primitives from sequential programs rather than from list homomorphisms, and to Akihiko Takano and other Tokyo CACA members for their useful comments.

## References

[ADKP87] K. Abrahamson, N. Dadoun, D.G. Kirkpatrick, and T. Przytycka. A simple parallel tree contraction algorithm. In *Proceedings of the Twenty-Fifth Allerton Conference on Communication, Control and Computing*, pages 624–633, September 1987.

[BdM96] R.S. Bird and O. de Moor. *Algebras of Programming.* Prentice Hall, 1996.

[Bir87] R. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 5–42. Springer-Verlag, 1987.

[Ble89] Guy E. Blelloch. Scans as primitive operations. *IEEE Trans. on Computers*, 38(11):1526–1538, November 1989.

[Ble92] G.E. Blelloch. NESL: a nested data parallel language. Technical Report CMU-CS-92-103, School of Computer Science, Carnegie-Mellon University, January 1992.

[BW88] R. Bird and P. Waddler. *Introduction to Functional Programming.* Prentice Hall, 1988.

[CDG96] W. Chin, J. Darlington, and Y. Guo. Parallelizing conditional recurrences. In *Annual European Conference on Parallel Processing, LNCS 1123*, pages 579–586, LIP, ENS Lyon, France, August 1996. Springer-Verlag.

[Chi92] W. Chin. Safe fusion of functional expressions. In *Proc. Conference on Lisp and Functional Programming*, pages 11–20, San Francisco, California, June 1992.

[Col95] M. Cole. Parallel programming with list homomorphisms. *Parallel Processing Letters*, 5(2), 1995.

[CTT97] W. Chin, S. Tan, and Y. Teo. Deriving efficient parallel programs for complex recurrences. In *ACM SIGSAM/SIGNUM International Conference on Parallel Symbolic Computation*, pages 101–110, Hawaii, July 1997. ACM Press.

[dM95] O. de Moor. A generic program for sequential decision processes. In M. Hermenegildo and D. S. Swierstra, editors, *Programming Languages: Implementations, Logics, and Programs*, volume 982 of *Lecture Notes in Computer Science*, pages 1–23. Springer-Verlag, 1995.

[Fok92a] M. Fokkinga. A gentle introduction to category theory — the calculational approach —. Technical Report Lecture Notes, Dept. INF, University of Twente, The Netherlands, September 1992.

[Fok92b] M. Fokkinga. *Law and Order in Algorithmics.* Ph.D thesis, Dept. INF, University of Twente, The Netherlands, 1992.

[GCS94] J. Gibbons, W. Cai, and D. Skillicorn. Efficient parallel algorithms for tree accumulations. *Science of Computer Programming*, (23):1–18, August 1994.

[GDH96] Z.N. Grant-Duff and P. Harrison. Parallelism via homomorphism. *Parallel Processing Letters*, 6(2):279–295, 1996.

[Gib92] J. Gibbons. Upwards and downwards accumulations on trees. In *Mathematics of Program Construction* (LNCS 669), pages 122–138. Springer-Verlag, 1992.

[Gib96] J. Gibbons. Computing downwards accumulations on trees quickly. *Theoretical Computer Science*, 169(1):67–80, 1996.

[GMT87] H. Gazit, G.L. Miller, and S.-H. Teng. Optimal tree contraction in the EREW model. In S.K. Tewksbury, B.W. Dickinson, and S.C. Schwartz, editors, *Concurrent Computations: Algorithms, Architecture and Technology*, pages 139–156. Plenum Press, 1987.

[Gor96a] S. Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In *Annual European Conference on Parallel Processing, LNCS 1124*, pages 401–408, LIP, ENS Lyon, France, August 1996. Springer-Verlag.

[Gor96b] S. Gorlatch. Systematic extraction and implementation of divide-and-conquer parallelism. In *Proc. Conference on Programming Languages: Implementation, Logics and Programs, LNCS 1140*, pages 274–288. Springer-Verlag, 1996.

[GR88] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms.* Cambridge University Press, 1988.

[HIT97] Z. Hu, H. Iwasaki, and M. Takeichi. Formal derivation of efficient parallel programs by construction of list homomorphisms. *ACM Transactions on Programming Languages and Systems*, 19(3):444–461, 1997.

[HL93] P. Hammarlund and B. Lisper. Data parallel programming, a survey and a proposal for a new model. Technical Report 93/8-SE, Department of Teleinformatics, Royal Institute of Technology, September 1993.

[HS86] W.D. Hills and Jr. G. L. Steele. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[HT99] Z. Hu and M. Takeichi. Calculating an optimal homomorphic algorithm for bracket matching. *Parallel Processing Letters*, 9(1), 1999.

[HTC98] Z. Hu, M. Takeichi, and W.N. Chin. Parallelization in calculational forms. In *25th ACM Symposium on Principles of Programming Languages*, pages 316–328, San Diego, California, USA, January 1998.

[Jeu95] J. Jeuring. Polytypic pattern matching. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 238–248, La Jolla, California, June 1995.

[JJ96] J. Jeuring and P. Jansson. Polytypic programming. In *2nd International Summer School on Advanced Functional Programming Techniques, LNCS*. Springer Verlag, July 1996.

[JJ97] P. Jansson and J. Jeuring. Polyp - a polytypic programming language extension. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Language*, pages 470–482. ACM Press, January 1997.

[Kar87] A. Karp. Programming for parallelism. *IEEE Computer*, pages 43–57, May 1987.

[KC98] G. Keller and M. T. Chakravarty. Flatten trees. In *EuroPar'98, LNCS*. Springer-Verlag, September 1998.

[LF80] R.E. Ladner and M.J. Fisher. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, 1980.

[Mal89] G. Malcolm. Homomorphisms and promotability. In J.L.A. van de Snepscheut, editor, *Mathematics of Program Construction*, pages 335–347. Springer-Verlag, 1989.

[Mee96] L. Meertens. Calculate polytypically. In *Proc. Conference on PLILP, LNCS 1140*, pages 1–16. Springer Verlag, 1996.

[MFP91] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proc. Conference on Functional Programming Languages and Computer Architecture* (LNCS 523), pages 124–144, Cambridge, Massachuetts, August 1991.

[MR85] G.L. Miller and J. Reif. Parallel tree contraction and its application. In *26th IEEE Symposium on Foundations of Computer Science*, pages 478–489, 1985.

[NO94] S. Nishimura and A. Ohori. A calculus for exploiting data parallelism on recursively defined data (preliminary report). In *International Workshop on Theory and Practice on Parallel Programming*. LNCS 907, 1994.

[Pra92] T.W. Pratt. Kernel-control parallel versus data parallel: A technical comparison. In *Proceeding of a Workshop on Languages, Compilers and Run-Time Enviroments for Distributed Memory Multiprocessors, appeared as SIGPLAN Notices, Vol 28, No. 1, January 1993*, pages 5–8, September 1992.

[SF93] T. Sheard and L. Fegaras. A fold for all seasons. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 233–242, Copenhagen, June 1993.

[Ski90] D.B. Skillicorn. Architecture-independent parallel computation. *IEEE Computer*, 23(12):38–51, December 1990.

[Ski93a] D.B. Skillicorn. The Bird-Meertens Formalism as a parallel model. In J.S. Kowalik and L. Grandinetti, editors, *Software for Parallel Computation*, volume 106 of *NATO ASI Series F*, pages 120–133. Springer-Verlag, 1993.

[Ski93b] D.B. Skillicorn. Categorical data types. In *Second Workshop on Abstract Models for Parallel Computation*, Oxford University Press, 1993.

[Ski94] David B. Skillicorn. *Foundations of Parallel Programming*. Cambridge University Press, 1994.

[Ski96] D.B. Skillicorn. Parallel implementation of tree skeletons. *Journal of Parallel and Distributied Computing*, 39(0160):115–125, 1996.

[THT98] A. Takano, Z. Hu, and M. Takeichi. Program transformation in calculational form. *ACM Computing Surveys*, 1998. to appear.

[TV84] R.E. Tarjan and U. Vishkin. Finding biconnected components and computing tree functions in logarithmic parallel time. In *25th IEEE Symposium on Foundations of Computer Science*, pages 12–22, 1984.

[Wad88] P. Wadler. Deforestation: Transforming programs to eliminate trees. In *Proc. ESOP (LNCS 300)*, pages 344–358, 1988.