A Calculational Framework for Parallelization of Sequential Programs

Zhenjiang Hu [*] Masato Takeichi

Summary.

A great deal of effort has been made on a systematic way for parallelization of sequential programs, because parallel programs are known to be much more difficult to write that their sequential counterparts. What seems to be unsatisfactory with current approaches, however, is either too general where many heuristics are needed or too restrictive where application scope is rather limited. In this paper, we propose a *calculational* framework for deriving parallel divide-and-conquer programs from naive sequential programs in a more systematic way. Being more constructive, our method is not only helpful in design of efficient parallel programs in general but also promising in construction of parallelization system. Several interesting examples are used for illustration.

Keywords: Parallel Programming, Parallelization, Program Calculation, Transformational Programming, Bird Meertens Formalism.

1 Introduction

Parallel programs are known to be more difficult to write than their sequential counterparts [CDG96, CTT97]. As an example, consider the *sbp* problem of determining whether the brackets '(' and ')' in a given string are correctly matched, e.g.,

$$\begin{array}{ll} sbp\ ''(()sd(12))'' & = \mathit{True} \\ sbp\ ''(as))'' & = \mathit{False} \end{array}$$

This problem has a straightforward linear sequential algorithm, in which the string is examined from left to right. A counter is initialized to 0, and incremented or decreased as opening and closing brackets are encountered.

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

It, however, turns out to be difficult to write a parallel program like in [GR88, BSS91, Col95] whose algorithms involved are actually non-trivial. Therefore, a

[*] Correspondence Address: Zhenjiang Hu, Takeichi Lab., Dept. of Information Engineering, Univ. of Tokyo, Tokyo 113 Japan. Email: hu@ipl.t.u-tokyo.ac.jp

good way for parallelizing sequential programs is of great importance in helping us design efficient parallel programs solving problems. Furthermore, it is our hope that machines could perform parallelization automatically.

Recently, much attention has been drawn to looking into a systematic way for parallelization of sequential programs. Basically, there are two kinds of approaches. One aims at a general derivation of parallel programs from sequential ones, e.g. [CTT97]. It utilizes some artificial intelligent technicals, such as synthesis from examples, and makes painstaking effort to systemize derivation process, while imposing as less restrictions as possible on the forms of sequential programs. This approach has the advantages of generality, but it usually requires heuristics and human insights in the derivation process, which seems a bit difficult to be made automatic.

The other approach, which is very popular, is to use Bird-Meertens formalism [Bir87, MFP91, Fok92] to synthesize parallel functional programs by *program calculation*[1], e.g., [Ski90, GDH94, Gor96a, Gor96b]. Different from the first approach whose emphasis is on the derivation process, its emphasis is on the restriction of sequential programs being described in some specific recursive forms (like left reductions or right reductions). Imposing restrictions on the forms of the sequential programs makes derivation straightforward; the prepared simple transformation rules (laws) can be directly applied. This calculational approach has the advantage of simple derivation process suiting mechanical implementation, as demonstrated in other applications [OHIT97, HITT97]. But for lack of descriptive power of the specific forms, the application scope is rather limited.

This paper is intended as the first investigation of a calculational framework for parallelization with the aim of combining the advantages of the above two approaches. We take the advantage of the first approach for deriving elementary parallelization laws, and the second approach for constructing our parallelization algorithm. Our main contributions are as follows.

- We develop several elementary but general parallelization laws (Section 4). By elementary, we mean that they contribute to the core transformations in our parallelization algorithm; and by general, we mean that they are more powerful than the previous ones [Ski92, Gor96a, Gor96b] and can be applied to synthesize several interesting parallel programs (as demonstrated in Section 4). Moreover, these laws can be directly implemented in a way of simple symbolic manipulation.

- We propose a *systematic* and *constructive* parallelization algorithm (Section 5) for derivation of divide-and-conquer parallel programs from naive sequential ones. It can be applied to a wider class of sequential programs covering all primitive recursive functions with which almost all algorithms of interest can be described. The distinguished point of our algorithm is its constructive way of deriving associative/distributive operators from the data types, and its effective use of the fusion and tupling calculation in the parallelizing process.

[1] By program calculation, we usually mean that program transformation by symbolic manipulation based on a set of simple rules.

- Our parallelization algorithm is given in a calculational way like those in [TM95, OHIT97, HITT97]. Therefore, it preserves the advantages of transformation in calculational form; being correct, guaranteeing to terminate, and being natural to be generalized to programs over any data types other than lists we studied in this paper. It would be not only helpful in design of efficient parallel programs but also promising in construction of parallelization system.

The organization of this paper is as follows. In Section 2, we review the notational conventions and some basic concepts used in this paper. After showing the extension of homomorphisms to mutumorphisms in Section 3, we focus ourselves on deriving several basic parallelization laws with some interesting examples in Section 4. Finally, we propose our parallelization algorithm in Section 5. Related work and conclusions are given in Section 6 and 7.

2 Preliminary

In this section, we briefly review the notational conventions known as Bird-Meertens Formalisms [Bir87] and some basic concepts which will be used in the rest of this paper.

2.1 Functions

Functional 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 $(f\,a)\,b$. Functional application is regarded as more binding than any other operator, so $f\,a \oplus b$ means $(f\,a) \oplus b$, but not $f\,(a \oplus b)$. *Functional composition* is denoted by a centralized circle $\circ$. By definition,

$$(f \circ g)\,a = f\,(g\,a).$$

Functional 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

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

The *projection* function π_i will be used to select the ith component of tuples, e.g., $\pi_1\,(a, b) = a$. The $\vartriangle$ and $\times$ are two important binary operators on tuples, defined by

$$\begin{array}{ll} (f \vartriangle g)\,a & = (f\,a,\, g\,a) \\ (f \times g)\,(a, b) & = (f\,a,\, g\,b). \end{array}$$

The $\vartriangle$ can be naturally extended to functions with two arguments. So, we have $a\,(\oplus \vartriangle \otimes)\,b = (a \oplus b,\, a \otimes b)$.

2.2 Lists

The data type of lists dominates functional programming; much of the subject is taken up with notations, and the names and properties of useful functions for manipulating them. Lists are finite sequences of values of the same type. There are two basic views of lists.

- *Parallel View*: 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 $xs \mathbin{+\!\!+} ys$ for the concatenation of xs and ys. We usually call the lists in the parallel view *append lists*.
- *Sequential View*: a list is either empty $[\,]$, or constructed by an element a and a list x with data constructor $:$ producing $a : x$ or with $\mathbin{\hat{:}}$ producing $x \mathbin{\hat{:}} a$. Equationally, we have

  $$\begin{array}{lcl} a : x & = & [a] \mathbin{+\!\!+} x \\ x \mathbin{\hat{:}} a & = & x \mathbin{+\!\!+} [a]. \end{array}$$

  To tell difference, we usually call the lists by the the former construction *cons lists*, the lists by the later construction *snoc lists*.

Concatenation is associative, and $[\,]$ is its unit. For example, the term $[1] \mathbin{+\!\!+} [2] \mathbin{+\!\!+} [3]$ denotes a list with three elements, often abbreviated to $[1, 2, 3]$. We also write $a : xs$ for $[a] \mathbin{+\!\!+} xs$.

2.3 Recursions on Lists

Functions over lists are usually defined in a recursive way. This section introduces some well-known recursive patterns over append, cons and snoc lists.

Definition 1 (List Homomorphism) A function h satisfying the following three equations is called a *list homomorphism*:

$$\begin{array}{lcl} h\,[\,] & = & \iota_{\oplus} \\ h\,[x] & = & f\,x \\ h\,(xs \mathbin{+\!\!+} ys) & = & h\,xs \oplus h\,ys \end{array}$$

where $\oplus$ is an *associative* binary operator with unit $\iota_{\oplus}$. We write $(\![f, \oplus]\!)$[2] for the unique function h. Usually, even a function h defined by the last two equations is considered to be a list homomorphism too. $\Box$

For example, we have $sum = (\![id, +]\!)$, which sums up all elements in a list. Note when it is clear from the context, we usually abbreviate "list homomorphisms" to "homomorphism."

Two important list homomorphisms are *map* and *reduction*. Map is the operator which applies a function to every item in a list. It is written as an infix $*$. Informally, we have

$$f * [x_1, x_2, \cdots, x_n] = [f\,x_1, f\,x_2, \cdots, f\,x_n].$$

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

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

List homomorphisms are good characterizations of parallel computational models [Ski92, GDH94, Col95]. Intuitively, the definition of list homomorphisms means

[2] Strictly speaking, we should write $(\![\iota_{\oplus}, f, \oplus]\!)$ to denote the unique function h. We can omit the $\iota_{\oplus}$ because it is the unit of $\oplus$.

that the value of h on the larger list depends in a particular way (using binary operation $\oplus$) on the values of h applied to the two pieces of the list. The computations of $h\,xs$ and $h\,ys$ are independent of each other and can thus be carried out in parallel. This simple equation can be viewed as expressing the well-known divide-and-conquer paradigm. A number of works on efficiently mapping list homomorphisms to particular parallel architectures can be found in [Ski92, GDH94, Gor96a]. It follows that if we can derive list homomorphisms from sequential programs we can have corresponding parallel programs.

Definition 2 (Left Reduction / Right Reduction) List function h is called a *left reduction* if there exists a binary operator $\oplus$ and a value e such that

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

Dually, h is called a *right reduction* if there exists a binary operator $\oplus$ and a value e such that

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

In contrast to the parallelism in homomorphisms, left and right reductions stipulate computation order, leading to sequential programs. Obviously, it is easy to specialize homomorphisms to left/right reductions. What is difficult but interesting is the reverse direction; going from sequential programs in left/right reductions to parallel programs in homomorphisms. This is sort of parallelization which we would like to do. Indeed it has attracted many researches [BSS91, GDH94, SB95, Gor95, Gor96a] because of the attractive theorem known as the third homomorphism theorem, as will be discussed later.

3 From Homomorphisms to Mutumorphisms

Surprisingly, there is a a fairly well known theorem called the *third homomorphism theorem* in program calculation community.

Theorem 1 (The Third Homomorphism Theorem [Bir87, Gib96]) Function h is a homomorphism if it is a left and a right reductions. □

It states that a function that can be computed by both a *left reduction* and a *right reduction* is necessarily a *list homomorphism* which can be computed according to any parenthesization of the list. It was conjectured by Richard Bird and was proved first by Lambert Meertens in 1989. Later, it was presented systematically by Gibbons [Gib96]. This theorem looks very attractive because it claims that if a problem can be defined by two specific sequential programs, then it can be defined by a list homomorphism which can be implemented in parallel as argued in Section 2.3.

However, there remain two major problems with this "attractive" theorem. First, there are a lot of useful and interesting list functions that are not list homomorphisms and thus have no corresponding $\oplus$. One example is the function *mss* known as *maximum segment sum problem* [Col95], which finds the maximum of the sums of contiguous segments within a list of integers. For example, $mss\ [3, -4, 2, -1, 6, -3] = 7$, where the result is contributed by the segment

$[2,-1,6]$. The *mss* is not a list homomorphism, since knowing *mss xs* and *mss ys* is not enough to allow computation of *mss* $(xs \mathbin{+\!\!+} ys)$. Second, as pointed by Gorlatch [Gor95, Gor96b], although the existence of an associative binary operator is guaranteed the theorem does not address the question of the existence — let alone the construction — of a direct and efficient way of calculating it.

To solve these problems, rather than using list homomorphisms, we choose *mutumorphisms* (mutual morphisms) [Fok89, Fok92] on append lists as our parallel computation model.

Definition 3 (List Mutumorphisms) The $h_1, \cdots, h_n$ are called *list mutumorphisms* if they are mutually defined in the following way:

$$\begin{array}{lcl} h_i\ [\,] & = & \iota_{\oplus_i} \\ h_i\ [x] & = & f_i\, x \\ h_i\ (xs \mathbin{+\!\!+} ys) & = & ((\Delta_1^n h_i)\ xs) \oplus_i ((\Delta_1^n h_i)\ ys) \end{array} \qquad \Box$$

List mutumorphisms (mutumorphisms for short in this paper) is a generalization of homomorphisms, which have enough descriptive power covering all primitive recursive functions. Moreover, it can be automatically turned into efficient list homomorphisms via tupling calculation [HIT96a, HITT97].

Theorem 2 (Tupling [HIT96a]) Let $h_1, \cdots, h_n$ be mutumorphisms as defined in Definition 3. Then,

$$\Delta_1^n h_i = (\![\Delta_1^n f_i,\ \Delta_1^n \oplus_i]\!)$$

and $(\iota_{\oplus_1}, \cdots, \iota_{\oplus_n})$ is the unit of $\Delta_1^n \oplus_i$. $\Box$

Therefore, like homomorphisms, mutumorphisms can be considered as a good characterization of computational model as well. In the rest of this paper, we shall focus on how to derive mutumorphisms from sequential programs in a very general form that is powerful to describe most algorithms of interest.

4 Parallelization Laws

Before giving our parallelization algorithm, we shall develop several elementary parallelization laws for parallelizing sequential programs, each of which is to capture one basic *syntactic structure* of expressions in the definition body. We develop these laws basically based on the parallel synthesis method (second order generalization + induction) [CTT97]. We omit the discussion of the detailed development process where some extension of the parallel synthesis method has been done [HT97]. This is beyond the scope of this paper.

4.1 Basic form

Sequential programs are usually specified in the following recursive way

$$f\ (x : xs) = g\ x\ xs \oplus f\ xs$$

reading that the result of f over a list $x : xs$ consists of two parts: $g\ x\ xs$, the result of another function being applied to the whole list, and $f\ xs$, the recursive partial result. These two parts are then grouped together by a binary operator $\oplus$.

Certainly, not all sequential programs have efficient parallel counterparts. If the $\oplus$ is associative, we then have the following divide-and-conquer parallel program for f.

Theorem 3 (Associativity) Given is a sequential program

$$\begin{aligned} f\ [\,] &= e \\ f\ (x : xs) &= g\ x\ xs \oplus f\ xs \end{aligned}$$

where $\oplus$ denotes an associative binary operator. Then, for any non-empty lists xs and ys, we have

$$\begin{aligned} f\ [x] &= g\ x\ [\,] \oplus e \\ f(xs \mathbin{+\!\!+} ys) &= G\ xs\ ys \oplus f\ ys \end{aligned}$$

where G is a function defined by

$$\begin{aligned} G\ [x]\ z &= g\ x\ z \\ G\ (xs \mathbin{+\!\!+} ys)\ z &= G\ xs\ (ys \mathbin{+\!\!+} z) \oplus G\ ys\ z \end{aligned}$$

Proof: We prove the new definition of f by induction on the length of the non-empty list xs.

- *Base*: In case $xs = [x]$, we have

$$\begin{aligned} f\ (xs \mathbin{+\!\!+} ys) = &\quad \{\ \text{Assumption}\ \} \\ & f\ ([x] \mathbin{+\!\!+} ys) \\ = &\quad \{\ \text{trivial}\ \} \\ & f\ (x : ys) \\ = &\quad \{\ \text{By the given definition of } f\ \} \\ & g\ x\ ys \oplus f\ ys \\ = &\quad \{\ \text{Definition of } G\ \} \\ & G\ [x]\ ys \oplus f\ ys \\ = &\quad \{\ \text{Assumption}\ \} \\ & G\ xs\ ys \oplus f\ ys \end{aligned}$$

- *Induction*: In case $xs = x : xs'$, we have

$$\begin{aligned} f\ (xs \mathbin{+\!\!+} ys) = &\quad \{\ \text{Assumption}\ \} \\ & f\ ((x : xs') \mathbin{+\!\!+} ys) \\ = &\quad \{\ \text{trivial}\ \} \\ & f\ (x : (xs' \mathbin{+\!\!+} ys)) \\ = &\quad \{\ \text{Definition of } f\ \} \\ & g\ x\ (xs' \mathbin{+\!\!+} ys) \oplus f\ (xs' \mathbin{+\!\!+} ys) \\ = &\quad \{\ \text{Definition of } G\text{, and Inductive hypothes}\ \} \\ & G\ [x]\ (xs' \mathbin{+\!\!+} ys) \oplus (G\ xs'\ ys \oplus f\ ys) \\ = &\quad \{\ \text{Associativity of } \oplus\ \} \\ & (G\ [x]\ (xs' \mathbin{+\!\!+} ys) \oplus G\ xs'\ ys) \oplus f\ ys \\ = &\quad \{\ \text{Definition of } G\ \} \\ & G\ ([x] \mathbin{+\!\!+} xs')\ ys \oplus f\ ys \\ = &\quad \{\ \text{Since } xs = x : xs'\ \} \\ & G\ xs\ ys \oplus f\ ys \end{aligned}$$ □

This theorem shows a mechanical way to turn a sequential definition of f into mutumorphisms which can be automatically transformed into efficient[3] homomorphisms by application of the tupling theorem. Notice that the sequential programs

[3] By efficiency, we mean that redundant computations due to multiple data traversals of the input by several functions in the mutumorphisms are removed.

that can be dealt with here are much more general than left/right reductions in the sense that the xs is allowed to be used by g.

One problem with the theorem is the increasing size of the second parameter of G in G's definition, i.e., $ys \mathbin{+\!\!+} z$. This may introduce redundant computations (due to multiple data traversals of the input data by several functions) which cannot be eliminated by the tupling transformation. To remedy this situation, we make explicit the computations on xs in g in Theorem 3, as shown in the following corollary.

Corollary 4 Given is a sequential program

$$\begin{aligned} f\ [\,] &= e \\ f\ (x : xs) &= g\ x\ (g'\ xs) \oplus f\ xs \end{aligned}$$

where $\oplus$ denotes an associative binary operator and g' is a homomorphism $([f', \oplus'])$. Then, for any non-empty lists xs and ys, we have

$$\begin{aligned} f\ [x] &= g\ x\ (g'\ [\,]) \oplus e \\ f(xs \mathbin{+\!\!+} ys) &= G\ xs\ (g'\ ys) \oplus f\ ys \end{aligned}$$

where G is a function defined by

$$\begin{aligned} G\ [x]\ z &= g\ x\ z \\ G\ (xs \mathbin{+\!\!+} ys)\ z &= G\ xs\ (g'\ ys \oplus' z) \oplus G\ ys\ z \end{aligned}$$

□

Although we restrict g' to a homomorphism, it indeed covers more general mutumorphisms, because a mutumorphism can be turned into a composition of a projection function with a homomorphism and the projection function can be moved to g.

On the other hand, when there is no function in g that is applied to xs in Theorem 3, we can simply eliminate the second parameter of G, as shown in the following.

Corollary 5 Given is a sequential program

$$\begin{aligned} f\ [\,] &= e \\ f\ (x : xs) &= g\ x \oplus f\ xs \end{aligned}$$

where $\oplus$ denotes an associative binary operator. Then, for any non-empty lists xs and ys, we have

$$\begin{aligned} f\ [x] &= g\ x \oplus e \\ f(xs \mathbin{+\!\!+} ys) &= G\ xs \oplus f\ ys \end{aligned}$$

where G is a function defined by

$$\begin{aligned} G\ [x] &= g\ x \\ G\ (xs \mathbin{+\!\!+} ys) &= G\ xs \oplus G\ ys \end{aligned}$$

□

To give an example of the use of the above theorem and corollaries, consider a simple simulation program (with a single server and queue) to compute the departure and arrival times for a sequences of events $[(s_n, a_n), \cdots, (s_1, a_1)]$, where

$a_1, \cdots, a_n$ are the inter-arrival time gaps between n events, and $s_1, \cdots, s_n$ are the corresponding service time.

$$\begin{array}{lcl}
\mathit{depart}\ [\,] & = & 0 \\
\mathit{depart}\ ((s,a) : xs) & = & (s + a + \mathit{arrive}\ xs) \uparrow \mathit{depart}\ xs \\
\mathit{arrive}\ [\,] & = & 0 \\
\mathit{arrive}\ ((s,a) : xs) & = & a + \mathit{arrive}\ xs
\end{array}$$

Here $\uparrow$ is an associative operator which accepts two values and returns the bigger. Applying Corollary 5 to the sequential program *arrive* gives

$$\begin{array}{lcl}
\mathit{arrive}\ [(s,a)] & = & a \\
\mathit{arrive}\ (xs \mathbin{+\!\!+} ys) & = & G_a\ xs + \mathit{arrive}\ ys \\
G_a\ [(s,a)] & = & a \\
G_a\ (xs \mathbin{+\!\!+} ys) & = & G_a\ xs \mathbin{+\!\!+} G_a\ ys
\end{array}$$

Notice $g\ (s,a) = a$ and $\oplus = +$. Although we could see that G_a is equal to *arrive*, our calculation approach should avoid comparing two functions which is impossible in general. Instead, we apply the tupling transformation to turn *arrive* to the composition of a projection and a homomorphism.

$$\begin{array}{lcl}
\mathit{arrive} & = & \pi_1 . g' \\
g'\ [(s,a)] & = & (a,a) \\
g'\ (xs \mathbin{+\!\!+} ys) & = & g'\ xs \oplus' g'\ ys \\
 & & \quad \text{where } (x_1,y_1) \oplus' (x_2,y_2) = (y_1 + x_2, y_1 + y_2).
\end{array}$$

Here, $g'\ [\,] = (0, -)$ where $-$ stands for a "don't-care" value which is not necessary during computation. So much for *arrive*. Now we turn to *depart* by using Corollary 4. In this case, we have $g\ (s,a)\ z = s + a + \pi_1\ z$ and g' as defined above. Therefore, we get

$$\begin{array}{lcl}
\mathit{depart}\ [(s,a)] & = & s + a + 0 \\
\mathit{depart}\ (xs \mathbin{+\!\!+} ys) & = & G_d\ xs\ (g'\ ys) \uparrow \mathit{depart}\ ys \\
G_d\ [(s,a)]\ z & = & s + a + \pi_1\ z \\
G_d\ (xs \mathbin{+\!\!+} ys)\ z & = & G_d\ xs\ (g'\ xs \oplus' z) \uparrow G_d\ ys\ z
\end{array}$$

This is the parallel version we'd like to get in this paper, although it is currently inefficient as there are multiple recursive calls in the RHS which operate on the same input. But this can be automatically improved by tupling calculation as intensively studied in [HIT96a, HIT96c, HITT97]. For instance, we can tuple *depart*, G_d, and g', (i.e., $\mathit{tup}\ xs\ c = (\mathit{depart}\ xs, G_d\ xs\ c, g'\ xs)$), and automatically get the following final efficient parallel program for *depart*.

$$\begin{array}{lcl}
\mathit{depart}\ xs & = & x,\ \text{where } (x, y, (z, w)) = \mathit{tup}\ xs\ (0, -) \\
\mathit{tup}\ [(s,a)]\ (z,w) & = & (s + a, s + a + \pi_1\ z, (a,a)) \\
\mathit{tup}\ (xs \mathbin{+\!\!+} ys)\ (z,w) & = & \text{let } (x1, y1, (z1, w1)) = \mathit{tup}\ xs \\
 & & \qquad\ (x2, y2, (z2, w2)) = \mathit{tup}\ ys \\
 & & \text{in } (y1\ \underline{z2} \uparrow x2, y1\ \underline{(w1 + z, w1 + w)}, (w1 + z2, w1 + w2)))
\end{array}$$

It is worth noting that *tup* can be parallelly implemented with multiple processor system supporting bidirectional tree-like communication, using the time of $O(\log n)$ where n denotes the length of the input list based on the algorithm in [Ble89, Gib92]. Two passes are employed; an upward pass in the computation can be used to compute the third component of $tup\ xs\ c$ (in order to get the values of the underlined parts) before a downward pass is used to compute the first two values of the tuple.

This example has also been studied in [GLM90] and [CTT97]. Differently, our derivation turns out to be a mechanical symbolic manipulation.

4.2 Accumulation

Another important syntactic structure in a recursive definition is *accumulating parameters* which are helpful to store information for later computation. One example is the *sbp* problem in the introduction, where a counter is used for accumulation. The following theorem is a natural extension of Theorem 3 in order to deal with accumulating parameters.

Theorem 6 (Accumulation) Given is a sequential program

$$\begin{array}{ll} f\ []\ c & = g_1\ c \\ f\ (x : xs)\ c & = g_2\ x\ xs\ c \oplus f\ xs\ (g_3\ x \otimes c) \end{array}$$

where $\oplus$ and $\otimes$ are two associative binary operators. Then, for any non-empty lists xs and ys, we have

$$\begin{array}{ll} f\ [x]\ c & = g_2\ x\ []\ c \oplus g_1\ (g_3\ x \otimes c) \\ f(xs +\!\!+ ys)\ c & = G_2\ xs\ ys\ c \oplus f\ ys\ (G_3\ xs \otimes c) \end{array}$$

where G_2 and G_3 are functions defined by

$$\begin{array}{ll} G_2\ [x]\ z\ c & = g_2\ x\ z\ c \\ G_2\ (xs +\!\!+ ys)\ z\ c & = G_2\ xs\ (ys +\!\!+ z)\ c \oplus G_2\ ys\ z\ (G_3\ xs \otimes c) \\ G_3\ [x] & = g_3\ x \\ G_3\ (xs +\!\!+ ys) & = G_3\ ys \otimes G_3\ xs \end{array}$$

□

Again, Theorem 6 can be improved in a similar way to what we did for Theorem 3, which will not be repeated here. It should be noted that we place the restriction that the value is accumulated by an associative operator $\otimes$. This makes room for parallelizing accumulation computation.

For a use of the theorem, recall the *sbp* problem given in the introduction. By the technique for manipulating conditional structure in [CDG96], we can turn the definition of sbp' into the following.

$$\begin{array}{ll} sbp'\ (x : xs)\ c = & (\text{if } x =' (' \text{ then } True \text{ else } (\text{if } x =')' \text{ then } c > 0 \text{ else } True)) \wedge \\ & sbp'\ xs\ (\text{if } x =' (' \text{ then } c + 1 \text{ else } (\text{if } x =')' \text{ then } c - 1 \text{ else } c)) \end{array}$$

Now matching it with the sequential program in Theorem 6 yields

$$\begin{array}{ll} g_1\ c & = c = 0 \\ g_2\ x\ xs\ c & = \text{if } x =' (' \text{ then } True \text{ else } (\text{if } x =')' \text{ then } c > 0 \text{ else } True) \\ g_3\ x & = \text{if } x =' (' \text{ then } 1 \text{ else } (\text{if } x =')' \text{ then } (-1) \text{ else } 0) \\ \oplus & = \wedge \\ \otimes & = + \end{array}$$

It follows directly from Theorem 6[4] that

$$sbp'\ (xs +\!\!+ ys)\ c = G_2\ xs\ c\ \wedge\ sbp'\ ys\ (G_3\ xs + c)$$

where

$$\begin{array}{ll}
G_2\ [x]\ c & = \text{if } x =' (' \text{ then } \mathit{True} \text{ else (if } x =')' \text{ then } c > 0 \text{ else } \mathit{True})\\
G_2\ (xs +\!\!+ ys)\ c & = G_2\ xs\ c\ \wedge\ G_2\ ys\ (G_3\ xs + c)\\
G_3\ [x] & = \text{if } x =' (' \text{ then } 1 \text{ else (if } x =')' \text{ then } (-1) \text{ else } 0)\\
G_3\ (xs +\!\!+ ys) & = G_3\ ys + G_3\ xs
\end{array}$$

This result can be easily improved by tupling sbp', G_2 and G_3 using the algorithm in [HITT97]:

$$\begin{array}{ll}
sbp'\ xs\ c & = s,\ \text{where } (s, g_2, g_3) = s23\ xs\ c\\
s23\ [x]\ c & = \text{if } x =' (' \text{ then } (c+1 = 0, \mathit{True}, c+1) \text{ else}\\
 & \quad \text{(if } x =')' \text{ then } (c-1 = 0, c > 0, c-1) \text{ else } (c = 0, \mathit{True}, c))\\
s23\ (xs +\!\!+ ys)\ c & = \text{let } (sx, g_2x, g_3x) = s23\ xs;\ (sy, g_2y, g_3y) = s23\ ys\\
 & \quad \text{in } (g_2x\ c\ \wedge\ sy\ (g_3x + c),\ g_2x\ c\ \wedge\ g_2y\ (g_3x + c),\ g_3x + g_3y)
\end{array}$$

Similar to the discussion for the final parallel program of *depart*, $s23$ can be parallelly implemented using the time of $O(\log n)$ where n denotes the length of the input list based on the algorithm in [Ble89, Gib92].

This example is taken from [Col95] where only an informal and intuitive derivation was given. Although our derived program is a bit different, it is as efficient as that in [Col95].

4.3 Conditional Structure

Conditional structure is important in a recursive definition. Related work can be found in [FG94, CDG96], where transformation on conditional expressions is proposed. Take a look at the following sequential program solving the *least sorted prefix* (*lsp* for short) problem [Gib96].

$$\begin{array}{ll}
lsp\ [x] & = [x]\\
lsp\ (x : xs) & = \text{if } x < hd\ xs \text{ then } [x] +\!\!+ lsp\ xs \text{ else } [x]
\end{array}$$

Our parallelization law with regard to the conditional structure is as follows.

Theorem 7 (Condition) Given is a sequential program

$$\begin{array}{ll}
f\ [\,] & = e\\
f\ (x : xs) & = \text{if } g_1\ x\ xs \text{ then } g_2\ x\ xs \oplus f\ xs \text{ else } g_3\ x\ xs
\end{array}$$

where $\oplus$ denotes an associative binary operator. Then, for any non-empty lists xs and ys, we have

$$\begin{array}{ll}
f\ [x] & = \text{if } g_1\ x\ [\,] \text{ then } g_2\ x\ [\,] \oplus e \text{ else } g_3\ x\ [\,]\\
f(xs +\!\!+ ys)\ c & = \text{if } G_1\ xs\ ys \text{ then } G_2\ xs\ ys \oplus f\ ys \text{ else } G_3\ xs\ ys
\end{array}$$

[4] Note that the second parameter of G_2 is a dead one (i.e., not necessary) and has been removed. It is similar to the case in Corollary 5.

where G_1, G_2 and G_3 are functions defined by

$$\begin{array}{lcl} G_1\ [x]\ z & = & g_1\ x\ z \\ G_1\ (xs +\!\!+ ys)\ z & = & G_1\ xs\ (ys +\!\!+ z) \wedge G_1\ ys\ z \\ G_2\ [x]\ z & = & g_2\ x\ z \\ G_2\ (xs +\!\!+ ys)\ z & = & G_2\ xs\ (ys +\!\!+ z) \oplus G_2\ ys\ z \\ G_3\ [x]\ z & = & g_3\ x\ z \\ G_3\ (xs +\!\!+ ys)\ z & = & \text{if}\ G_1\ xs\ (ys +\!\!+ z)\ \text{then}\ G_2\ xs\ (ys +\!\!+ z) \oplus G_3\ ys\ z \\ & & \quad \text{else}\ G_3\ xs\ (ys +\!\!+ z) \end{array} \qquad \Box$$

We will not address how to improve this theorem by extracting recursive functions on xs from g_1, g_2 and g_3, and by deleting xs in case xs is not used by g_1, g_2 and g_3, just like Corollary 4 and 5. Returning to the *lsp* sequential program, we can get the following parallel program according to this theorem.

$$lsp\ (xs +\!\!+ ys) = \text{if}\ G_1\ xs\ (hd\ ys)\ \text{then}\ G_2\ xs \oplus f\ ys\ \text{else}\ G_3\ xs$$

where G_1, G_2 and G_3 are functions defined by

$$\begin{array}{lcl} G_1\ [x]\ z & = & x < z \\ G_1\ (xs +\!\!+ ys)\ z & = & G_1\ xs\ (hd\ ys) \wedge G_1\ ys\ z \\ G_2\ [x] & = & [x] \\ G_2\ (xs +\!\!+ ys) & = & G_2\ xs\ +\!\!+ G_2\ ys \\ G_3\ [x] & = & [x] \\ G_3\ (xs +\!\!+ ys) & = & \text{if}\ G_1\ xs\ ys\ \text{then}\ G_2\ xs \oplus G_3\ ys\ \text{else}\ G_3\ xs \end{array}$$

In fact, $G_1\ xs\ z$ defines a predicate which is *True* when xs is in increasing order and the last element of xs is less than z, G_2 is an identity function, and G_3 is the same as *lsp*. We ask the readers to apply tupling transformation to the above program so as to get a final efficient parallel program.

4.4 Multiple Recursive Calls

So far we have considered linear recursions, i.e, recursions with a single recursive call in the definition body. In this section, we shall provide our parallelization law for nonlinear recursions. For instance, the following *lfib* is a tricky nonlinear recursion on lists, which computes the fibonacci number of the length of a given list, mimicking the fibonacci function on natural numbers.

$$\begin{array}{lcl} lfib\ [\,] & = & 1 \\ lfib\ (x : xs) & = & lfib\ xs + lfib'\ xs \\ lfib'\ [\,] & = & 0 \\ lfib'\ (x : xs) & = & lfib\ xs \end{array}$$

To handle nonlinear recursive sequential programs, we need to make use of distributive property in order to parallelize them.

Theorem 8 (Distributivity) Assume that f_1 and f_2 are mutually recursive functions defined by

$$\begin{array}{lcl} f_1\ [\,] & = & e_1 \\ f_1\ (x : xs) & = & g_1\ x\ xs \oplus (g_{11} \otimes f_1\ xs) \oplus (g_{12} \otimes f_2\ xs) \\ f_2\ [\,] & = & e_2 \\ f_2\ (x : xs) & = & g_2\ x\ xs \oplus (g_{21} \otimes f_1\ xs) \oplus (g_{22} \otimes f_2\ xs) \end{array}$$

where $\oplus$ is associative and commutative, and $\otimes$ is associative and distributive w.r.t $\oplus$, i.e., for any x, y and z,

$$x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z).$$

Then, for any non-empty lists xs and ys, we have

$$\begin{array}{lll}
f_1\ [x] & = & g_1\ x\ [\,] \oplus (g_{11} \otimes e_1) \oplus (g_{12} \otimes e_2) \\
f_1\ (xs \mathbin{+\!\!+} ys) & = & G_1\ xs\ ys \oplus (G_{11}\ xs \otimes f_1\ ys) \oplus (G_{12}\ xs \otimes f_2\ ys) \\
f_2\ [x] & = & g_2\ x\ [\,] \oplus (g_{21} \otimes e_1) \oplus (g_{22} \otimes e_2) \\
f_2\ (xs \mathbin{+\!\!+} ys) & = & G_2\ xs\ ys \oplus (G_{21}\ xs \otimes f_1\ ys) \oplus (G_{22}\ xs \otimes f_2\ ys)
\end{array}$$

where

$$\begin{array}{lll}
G_1\ [x]\ z & = & g_1\ x\ z \\
G_1\ (xs \mathbin{+\!\!+} ys) & = & G_1\ xs\ (ys \oplus z) \oplus (G_{11}\ xs \otimes G_1\ ys\ z) \oplus (G_{12}\ xs \otimes G_2\ ys\ z) \\
G_2\ [x]\ z & = & g_2\ x\ z \\
G_2\ (xs \mathbin{+\!\!+} ys)\ z & = & G_2\ xs\ (ys \oplus z) \oplus (G_{21}\ xs \otimes G_1\ ys\ z) \oplus (G_{22}\ xs \otimes G_2\ ys\ z) \\
G_{11}\ [x] & = & g_{11} \\
G_{11}\ (xs \mathbin{+\!\!+} ys) & = & (G_{11}\ xs \otimes G_{11}\ ys) \oplus (G_{12}\ xs \otimes G_{21}\ ys) \\
G_{12}\ [x] & = & g_{12} \\
G_{12}\ (xs \mathbin{+\!\!+} ys) & = & (G_{11}\ xs \otimes G_{22}\ ys) \oplus (G_{12}\ xs) \otimes G_{22}\ ys) \\
G_{21}\ [x] & = & g_{21} \\
G_{21}\ (xs \mathbin{+\!\!+} ys) & = & (G_{21}\ xs \otimes G_{11}\ ys) \oplus (G_{22}\ xs\ \otimes G_{21}\ ys) \\
G_{22}\ [x] & = & g_{22} \\
G_{22}\ (xs \mathbin{+\!\!+} ys)\ z & = & (G_{21}\ xs \otimes G_{12}\ ys) \oplus (G_{22}\ xs \otimes G_{12}\ ys)
\end{array}$$

□

We have two remarks. First, Theorem 8 can be easily generalized from two functions that are mutually defined to n functions. Second, like Theorem 3 Theorem 8 only gives a parallelization rule for recursions with multiple calls in a very basic form. Other syntactic structures, like accumulating parameters and conditional structure, can be dealt in a similar way as we did before, which will not be made detailed in this paper. The interested readers are referred to [HT97].

Let's use this theorem to parallelize *lfib* function. To use the theorem, we should notice that $e_1 = 1$, $e_2 = 0$, $g_1\ x\ xs = g_2\ x\ xs = 0$, $g_{11} = g_{12} = g_{21} = 1$, $g_{22} = 0$, $\oplus = +$, and $\otimes = \times$. And we can get the following parallel program after noticing that $G_1\ xs = G_2\ xs = 0$.

$$\begin{array}{lll}
\mathit{lfib}\ [x] & = & 1 \\
\mathit{lfib}\ (xs \mathbin{+\!\!+} ys) & = & (G_{11}\ xs \times \mathit{lfib}\ ys) + (G_{12}\ xs \times \mathit{lfib}'\ ys) \\
\mathit{lfib}'\ [x] & = & 1 \\
\mathit{lfib}'\ (xs \mathbin{+\!\!+} ys) & = & (G_{21}\ xs \times \mathit{lfib}\ ys) + (G_{22}\ xs \times \mathit{lfib}'\ ys)
\end{array}$$

where

$$\begin{array}{lll}
G_{11}\ [x] & = & 1 \\
G_{11}\ (xs \mathbin{+\!\!+} ys) & = & (G_{11}\ xs \times G_{11}\ ys) + (G_{12}\ xs \times G_{21}\ ys) \\
G_{12}\ [x] & = & 1 \\
G_{12}\ (xs \mathbin{+\!\!+} ys) & = & (G_{11}\ xs \times G_{22}\ ys) + (G_{12}\ xs \times G_{22}\ ys) \\
G_{21}\ [x] & = & 1 \\
G_{21}\ (xs \mathbin{+\!\!+} ys) & = & (G_{21}\ xs \times G_{11}\ ys) + (G_{22}\ xs \times G_{21}\ ys) \\
G_{22}\ [x] & = & 0 \\
G_{22}\ (xs \mathbin{+\!\!+} ys)\ z & = & (G_{21}\ xs \times G_{12}\ ys) + (G_{22}\ xs \times G_{12}\ ys)
\end{array}$$

which is an efficient $O(\log n)$ parallel program. It would be interesting to see that we have actually derived an $O(\log n)$ sequential algorithm for computing the standard *fib* function. This could be seen if we replace all xs and ys in the program by the length of xs and ys respectively.

5 Parallelization Algorithm

Several important parallelization laws have been given in the previous section. In this section, we are going to propose our parallelization algorithm based on these laws. Basically, we have to make clear the following issues.

- How to recognize associative and distributive operators in a program?
- How to apply these parallelization laws in a systematic way?
- How to turn a sequential program into the specific form that our laws can be applied?

5.1 Recognizing Associative and Distributive Operators

Central to our parallelization laws is the use of associativity of a binary operator $\oplus$ as well as distributivity of $\otimes$. As the first step, we must be able to recognize them in a program. There are several ways. We may restrict our application scope so that all associative and distributive operators can be made explicit, e.g. in [FG94, CTT97]. Or, we may adopt some artificial methods like anti-unification [Hei94] to synthesize them. However, these approaches are not so satisfactory to be used practically in a parallelization system. In this paper, rather than recognizing all associative and distributive operators, we are interested in the associative and distributive operators that are derivable from the resulting data type of the given sequential program.

Associative Operators from Data Types

The use of the associative binary operator $\oplus$ in our parallelization laws indicates that it should have the type

$$R \to R \to R$$

where R is the type of the given function that are to be parallelized. Such binary functions are no1t uncommon. In fact every type R which has a zero constructor C_Z (a constructor with no arguments like $[\,]$ for lists) has a function that is associative, and that has the zero C_Z for both a left and right identity. Such function $\odot$ is called *zero replacement function* in [SF94]:

$$x \odot y$$

which replaces all C_Z in x with y. Rather than being involved in complicated discussions, let's look at several examples. For the type of cons lists, we have a $\odot$ defined by

$$\begin{array}{ll} [\,] \odot y & = y \\ (x : xs) \odot y & = x : (xs \odot y) \end{array}$$

which is the list append operator $++$; for the type of natural numbers, we have a $\odot$ defined by

$$\begin{array}{lcl} 0 \odot y & = & y \\ (Succ\ n) \odot y & = & Succ\ (n \odot y) \end{array}$$

which the integer addition $+$; and for the type of booleans, we have two such operators $\odot_1$ and $\odot_2$ corresponding to choosing *True* and *Flase* as a zero respectively, which are defined by

$$\begin{array}{lcl} True \odot_1 y & = & y \\ Flase \odot_1 y & = & Flase \end{array}$$

and

$$\begin{array}{lcl} True \odot_2 y & = & True \\ Flase \odot_2 y & = & y \end{array}$$

It is not difficult to see that they are exactly the boolean $\wedge$ and $\vee$.

Distributive Operators

Now that we have derived associative operators from data types. Associating with some associative operator $\oplus$, we may derive a most natural distributive operator $\otimes$. We avoid formal addressing here. For example, for the type of natural numbers, associating with $+$ we have a distributive operator $\otimes$ defined by:

$$\begin{array}{lcl} (x\otimes)\ 0 & = & 0 \\ (x\otimes)\ (Succ\ n) & = & x + x \otimes n \end{array}$$

Clearly, $\otimes$ is our familiar $\times$. This natural distributive operator is useful when we deal with nonlinear recursions like the *lfib* function.

The ideas of derivation of associative and distributive operator from data types are not new [SF93, SF94]. However, previous studies were essentially for the purpose of automatic construction of monadic operators from type definitions. We brought them here for our parallelization purpose.

5.2 Main Algorithm

In order to simplify our presentation and to make it clear the point of our parallelization algorithm, we shall consider input programs to be *single* (not mutual) recursions. So an input to our algorithm is the following sequential definition

$$\begin{array}{lcl} f & : & [A] \to C \to R \\ f\ [\,]\ c & = & g_1\ c \\ f\ (x : xs)\ c & = & body \end{array}$$

where *body* is an expression. The accumulating parameter is probably unnecessary which can then be eliminated. We shall use *scan* (or called *prefix sums*) [Ble89, FG94, Gor95] as our running example.

$$\begin{array}{lcl} scan\ [\,] & = & [\,] \\ scan\ (x : xs) & = & x : (x+) * scan\ xs \end{array}$$

The parallelization algorithm consists of five steps, as summarized below.

Step 1: Making Associative Operator Explicit

First of all, we need to make the associative operator $\oplus$ be explicit in our program. Such $\oplus$ is not an arbitrary associative operator; rather it is the zero replacement operator derivable from the resulting type R which has a zero constructor C_Z. To this end, we represent (recursive) data constructors, used for producing the result, in terms of $\oplus$. For instances, when R is the cons list type (whose zero constructor is $[\,]$ and whose associate operator is $+\!\!+$), we have the rule

$$x : e \Rightarrow [x] +\!\!+ e$$

where we extract the list e from the constructor expression $x : e$. When R is the type of natural numbers, we have

$$Succ\ e \Rightarrow (Succ\ 0) + e.$$

Returning to our running example, we should get

$$\begin{array}{lcl} scan\ [\,] & = & [\,] \\ scan\ (x : xs) & = & [x] +\!\!+ (x+) * scan\ xs \end{array}$$

Step 2: Normalizing *body*

In order to apply our parallelization laws, we shall turn the definition body into our required forms. Based on the associative property of $\oplus$ and the following rule concerning *if* expressions:

$$\begin{array}{l} \text{if } p \text{ then } e_1 \oplus e_2 \oplus e_2 \text{ else } e_1' \oplus e_2' \oplus e_3' \\ \quad \Rightarrow (\text{if } p \text{ then } e_1 \text{ else } e_1') \oplus (\text{if } p \text{ then } e_2 \text{ else } e_2') \oplus (\text{if } p \text{ then } e_3 \text{ else } e_3') \end{array}$$

we can transform *body* into the following *normal form.*

$$\mathrm{e_1 \oplus e_2 \oplus \cdots \oplus e_n}$$

where e_i is

(i) an expression without recursive calls (to f), or

(ii) a recursive call (to f), or

(iii) a function application, say $g\ e$, where e is an expression of (ii) and g is another function, or

(iv) an *if* expression, say if $e1'$ then $e2'$ else $e3'$, where $e2'$ or $e3'$ are expressions of form (ii) or (iii).

Looking at the *scan* example, we simply normalize the body to

$$\underline{[x]} +\!\!+ \underline{(x+) * scan\ xs}$$

in which the first underlined expression is of from (i) and the second can be considered as a function application $g\ (scan\ xs)$ where $g\ r = (x+) * r$.

Step 3: Removing Recursive Calls by Fusion

Recall that the parallelization laws require that recursive calls be exposed to associative operators in the *body* rather than being wrapped in a function application, and that the predicate part in a *if* expression does not contain any recursive call. However, as seen in Step 2, the normalized body may contain some expressions violating this requirement. Fortunately, we may apply fusion calculation [TM95, HIT96b, OHIT97] to remove the recursive calls and turn transform the unexpected expressions into expected ones.

As an example, consider our running example of *scan* where the recursive call *scan xs* does not directly expose to the associative operator $+\!\!+$; being included in the expression $(x+) * scan\ xs$. Let $scan'\ xs\ x = (x+) * scan\ xs$. We apply the fusion calculation to $scan'$ and obtain the following result.

$$\begin{array}{lll} scan\ (x:xs) & = & [x] +\!\!+ scan'\ xs\ x \\ scan'\ [\,]\ y & = & [\,] \\ scan'\ (x:xs)\ y & = & [x+y] +\!\!+ scan'\ xs\ (x+y) \end{array}$$

The new $scan'$ can be parallelized by Theorem 6, leading to a parallel *scan*.

One question remained is whether this fusion succeeds and if it succeeds whether the fused program are suited for parallelization. Our current parallelization algorithm will give up if the fusion calculation fails. If it succeeds, our parallelization algorithm will parallelize the fused program as well.

Step 4: Applying Parallelization Laws

Now we are ready to use the parallelization laws to derive a parallel f. There are three cases according to the structure of the normalized *body*.

- First, the transformed *body* has no recursive call to f. In this case, we step to parallelize other functions in the *body*. For instance, for the following new *scan* definition
$$scan\ (x:xs) = [x] +\!\!+ scan'\ xs\ x$$
we should turn to parallelize $scan'$.
- Second, the transformed *body* has a single recursive call to f (a direct recursive call or a recursive call inside a if structure), denoted by E[f] here and after. We have three subcases.
  - $body = e \oplus E[f]$. We apply Theorem **??** or 7 for parallelization, while trivially introducing a function from the expression e by defining $g\ x\ xs = e$. For the example of *scan'* whose body is
$$[x+y] +\!\!+ scan'\ xs\ (x+y)$$
we apply Theorem 6 to it while noticing $g_2\ x\ xs\ c = [x+c]$, $g_3\ x = x$, $\oplus = +\!\!+$, and $\otimes = +$, and we get the following parallel version for $scan'$ after the elimination of the second parameter of G_2.
$$\begin{array}{lll} scan'\ (xs +\!\!+ ys)\ c & = & G_2\ xs\ c +\!\!+ scan'\ ys\ (G_3\ xs + c) \\ G_2\ [x]\ c & = & [x+c] \\ G_2\ (xs +\!\!+ ys)\ c & = & G_2\ xs\ c +\!\!+ G_2\ ys\ (G_3\ xs) \\ G_3\ [x] & = & g_3\ x \\ G_3\ (xs +\!\!+ ys) & = & G_3\ ys + G_3\ xs \end{array}$$
  - $body = E[f] \oplus e$. Defining a new associative operator $\hat{\oplus}$ by $x \hat{\oplus} y = y \oplus x$, we turn the body into the first subcase, i.e., $body = e \hat{\oplus} E[f]$.

  - *body* $= e_1 \oplus E[f] \oplus e_2$. Here, we need to check if $\oplus$ is commutative. If so, we exchange the positions of e_2 and e_3, transforming it to the first subcase. Otherwise, we give up parallelizing.

- Third and last, the transformed *body* has over one recursive calls, say two for simplicity. In this case, we require that $\oplus$ should be commutative and should have a corresponding distributive operator $\otimes$ with the identity unit say $\iota_\otimes$. If this requirement is satisfied, we can transform *body* to the form

$$e \oplus E_1[f] \oplus E_2[f]$$

Then we can introduce a new function f' (in fact, f' is the same as f) and have

$$\begin{aligned} f\ (x : xs)\ c\ &= e \oplus E_1[f] \oplus E_2[f'] \\ f'\ (x : xs)\ c &= e \oplus E_1[f] \oplus E_2[f'] \end{aligned}$$

Now we are able to apply, e.g., Theorem 8 (see more discussion in Section 4.4), for parallelizing mutually defined functions f and f'.

Step 5: Optimizing by Tupling Calculation

As demonstrate in the examples of the *depart* and the *sbp'*, we need to perform tupling calculation based on Theorem 2 in order to obtain final efficient parallel programs. More detailed studies on tupling calculation can be found in [HIT96a, HITT97].

It is worth noting that our parallelization algorithm is correct and guarantees to terminate. Although it gives up in case the conditions in the algorithm cannot meet, our parallelization algorithm can be applied to a wider class of recursive functions including many interesting programs, such as *scans*, *lsp*, and *depart*, which are considered to be difficult by some of the previous approaches.

6 Related Work

It has been attracting much attention to make use of list homomorphisms in parallel programming [Ski92, Col95, Gor95, Gor96a, GDH94, HIT96a, HIT96c], because they ideally suit the divide-and-conquer parallel paradigm. In fact, list homomorphisms are good characterizations of parallel computational models, and there are a number of researches [Ski92, GDH94, Gor96a] on efficiently mapping list homomorphisms to particular parallel architectures. Our work has been much influenced by these work. We are particularly interested in how to derive list homomorphisms.

One popular way, known as calculational way [Ski90, Ski92], for derivation of homomorphism is the use of program calculational laws in Bird Meertens Formalism [Bir87]. It forces the initial programs to be described in terms of a small set of specialized homomorphisms such as *map* and *reduction*, from which a more complicated homomorphism are derived based on calculational laws such as *promotion rules*. As illustrated in the paper, homomorphisms are rather limiting, excluding many interesting programs. To remedy this situation, Cole [Col95] proposed the idea of *near homomorphism* (or called *almost homomorphism*), a composition of projection function with a homomorphsm, and gave a quite informal way showing how to write a new homomorphism to solve a problem. This idea was then formalized by [HIT96a, HIT96c] where any natural programs defined over append lists can be structured to be a composition of mutumorphisms and then be turned into near homomorphisms. Basically, all the above approaches require programs to be initially defined over append lists, the parallel view of lists.

What is more challenging is to derive homomorphisms from sequential programs defined over cons or snoc lists, the sequential view of lists. To this end, some skeletons of sequential programs are defined whose list homomorphisms can be easily derived, e.g., in [GDH94, Gor96a]. However, the prepared skeletons are slightly less general and depend heavily on associativity of the operators in them where how to find or determine an associative operator was not clear. Compared to them, our approach does not restricted to any skeletons, giving a general parallelization algorithm. Furthermore, our approach gives a way to recognize the associative operator from the resulting data type.

Another idea in the calculational approach to derivation of list homomorphims from sequential programs is the use the third homomorphism theorem [Gib96]. Barnard et al [BSS91] tried it for the language recognition problem. As pointed by [Gor95], although the existence of an associative binary operator is guaranteed, the theorem does not address any efficient way of calculating it. Gorlatch [Gor95, Gor96a] proposed an idea of synthesizing list homomorphisms by generalizing both leftward and rightward reduction functions. Since his idea was studied in an informal way, and the generalization algorithm was not given, it is not so clear how to do it in general.

Our work was greatly inspired by the parallel synthesis algorithm in [CDG96, CTT97]. After determining a desired pre-parallel form for the initial recursive equation based on the idea of *synthesis from examples*, sort of artificial intelligence method, it uses the second order generalization to obtain a template equation and uses an inductive derivation to derive unknown functions in the template. We brought the transformations of the second order generalization and the inductive derivation here for building our basic parallelization laws. What is different is that rather than determining a desired pre-parallel form from examples which requires heuristics, we propose a constructive way to do so as seen in our parallelization algorithm.

In traditional imperative languages there are also many ongoing efforts at developing sophisticated techniques for parallelizing iterative loop [FG94]. This method is based on a parallel reduction of function composition which are associative. It define a certain *template* form which can be efficiently parallelized. However, it needs a bit human insight to derive such template form from programs.

This work can be considered as a complementary of our previous work [HIT96a, HIT96c]. Previous work starts from the specification of a form which can be turned into mutumorphisms, while this work shows how to derive mutumorphisms from sequential specifications.

7 Conclusions

In this paper, we propose a calculational framework for parallelizing any naive sequential programs. Particularly, we give a constructive parallelization algorithm, by developing a set of elementary but powerful parallelization laws and deriving associative and distributive operators from the resulting data type. We illustrate with several interesting problems that our parallelization algorithm can be applied to a wide class of programs.

As to the future work, the current parallelization algorithm can be improved in two respects. One is to reduce the number of new functions introduced. The other is to enhence the power of the fusion calculation [TM95, OHIT97], enlarging the application scope of our parallelization algorithm.

Acknowledgements

We gratefully acknowledge inspiring discussions with N.W. Chin, from which some of the ideas in this paper came out. His clarification of his parallel synthesis algorithm helped us a lot in developing our parallelization laws. Thanks are also to the members of Tokyo CACA seminar (`http://www.ipl.t.u-tokyo.ac.jp/~caca/`) for many helpful suggestions for improvement of this paper.

References

[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.

[BSS91] D. Barnard, J. Schmeiser, and D. Skillicorn. Deriving associative operators for language recognition. In *Bulletin of* EATCS (43), pages 131–139, 1991.

[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.

[Col95] M. Cole. Parallel programming with list homomorphisms. *Parallel Processing Letter*, 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*, Hawaii, July 1997. ACM Press. to appear.

[FG94] A. Fischer and A. Ghuloum. Parallelizing complex scans and reductions. In *ACM PLDI*, pages 135–146, Orlando, Florida, 1994. ACM Press.

[Fok89] M. Fokkinga. Tupling and mutumorphisms. *Squiggolist*, 1(4), 1989.

[Fok92] M. Fokkinga. A gentle introduction to category theory — the calculational approach —. Technical Report Lecture Notes, Dept. INF, University of Twente, The Netherlands, September 1992.

[GDH94] Z.N. Grant-Duff and P.G. Harrison. Skeletons, list homomorphisms and parallel program transformation. Technical report, Department of Computing, Imperial College, 1994.

[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. The third homomorphism theorem. *Journal of Functional Programming*, 1996. to appear.

[GLM90] A. Greenberg, B. Lubachevsky, and I. Mitrani. Unboundedly parallel simulation vis recurrence relations. In *ACM SIG-METRICS*, pages 1–12, September 1990.

[Gor95] S. Gorlatch. Constructing list homomorphisms. Technical Report MIP-9512, Fakultät für Mathematik und Informatik, Universität Passau, August 1995.

[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. *Microprocessing and Microprogramming*, 41:571–578, 1996. (Also appears in PLILP'96).

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

[Hei94] B. Heinz. Lemma discovery by anti-unification of regular sorts. Technical report no. 94-21, FM Informatik, Technische Universitat Berlin, May 1994.

[HIT96a] Z. Hu, H. Iwasaki, and M. Takeichi. Construction of list homomorphisms by tupling and fusion. In *21st International Symposium on Mathematical Foundation of Computer Science, LNCS 1113*, pages 407–418, Cracow, September 1996. Springer-Verlag.

[HIT96b] Z. Hu, H. Iwasaki, and M. Takeichi. Deriving structural hylomorphisms from recursive definitions. In *ACM SIGPLAN International Conference on Functional Programming*, pages 73–82, Philadelphia, PA, May 1996. ACM Press.

[HIT96c] Z. Hu, H. Iwasaki, and M. Takeichi. Formal derivation of parallel program for 2-dimensional maximum segment sum problem. In *Annual European Conference on Parallel Processing, LNCS 1123*, pages 553–562, LIP, ENS Lyon, France, August 1996. Springer-Verlag.

[HITT97] Z. Hu, H. Iwasaki, M. Takeichi, and A. Takano. Tupling calculation eliminates multiple data traversals. In *ACM SIGPLAN International Conference on Functional Programming*, Amsterdam, The Netherlands, June 1997. ACM Press. to appear.

[HT97] Z. Hu and M. Takeichi. Synthsizing calculational laws for parallelization. under preparation, May 1997.

[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.

[OHIT97] Y. Onoue, Z. Hu, H. Iwasaki, and M. Takeichi. A calculational fusion system HYLO. In *IFIP TC 2 Working Conference on Algorithmic Languages and Calculi*, Le Bischenberg, France, February 1997. Chapman&Hall.

[SB95] J.P. Schmeiser and D.T. Barnard. Polylogorithmic parallel parsing of $p(k)$ languages. Technical report 95-384, Department of Computing and Information Science, Queen's University, Kingston, Canada, June 1995.

[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.

[SF94] T. Sheard and L. Fegaras. Optimizing algebraic programs. Technical Report Technical Report 94-004, Dept. of Computer Science and Engineering, Oregon Graduate Institution of Science and Technology, 1994.

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

[Ski92] D. B. Skillicorn. The Bird-Meertens Formalism as a Parallel Model. In *NATO ARW "Software for Parallel Computation"*, June 92.

[TM95] A. Takano and E. Meijer. Shortcut deforestation in calculational form. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 306–313, La Jolla, California, June 1995.