Parallelization in Calculational Forms

Zhenjiang Hu
Department of Information Engineering
University of Tokyo
(hu@ipl.t.u-tokyo.ac.jp)

Masato Takeichi
Department of Information Engineering
University of Tokyo
(takeichi@u-tokyo.ac.jp)

Wei-Ngan Chin
Department of Information Systems & Computer Science
National University of Singapore
(chinwn@iscs.nus.edu.sg)

Abstract

The problems involved in developing efficient parallel programs have proved harder than those in developing efficient sequential ones, both for programmers and for compilers. Although program calculation has been found to be a promising way to solve these problems in the sequential world, we believe that it needs much more effort to study its effective use in the parallel world. In this paper, we propose a *calculational framework* for the derivation of efficient parallel programs with two main innovations:

- We propose a novel inductive synthesis lemma based on which an elementary but powerful parallelization theorem is developed.
- We make the first attempt to construct a calculational algorithm for parallelization, deriving associative operators from data type definition and making full use of existing fusion and tupling calculations.

Being more constructive, our method is not only helpful in the design of efficient parallel programs in general but also promising in the construction of parallelizing compiler. Several interesting examples are used for illustration.

1 Introduction

Consider a language recognition problem for determining whether the brackets '(' and ')' in a given string are correctly matched. 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 increased or decreased as opening and closing brackets are encountered.

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

To appear in *25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Diego, California, USA. January 1998. ACM Press.

It is, however, quite difficult to write a parallel program like those in [BSS91, Col95] whose algorithms are actually non-trivial.

Our work on parallelization in calculational forms is motivated by the successful application of *program calculation* to the optimization of sequential programs. Program calculation is a kind of transformational programming approach [Dar81] to program construction, in which a clear and understandable but may be terribly inefficient program is successively transformed into more and more efficient versions by means of *equational reasoning* in Bird-Meertens Formalisms (BMF) [Bir87, Bir89, Mal89, Bac89, Fok92]. BMF, also known as *constructive algorithmics*, was first proposed as the theory of lists [Bir87], and was then extended to be a general theory of datatypes. It has proved to be very useful not only in deriving various kinds of efficient sequential programs [Gib92, dM92, Jeu93], but also in constructing optimization passes of compilers [GLJ93, SF93, TM95, HIT96, OHIT97, HITT97]. Its success owes much to its concise description of transformation algorithms and its strong theoretical foundation based on category theory.

We do believe that it is both worthwhile and challenging to apply the calculational approach in a *practical* way to develop efficient *parallel* programs as well as to construct *parallelizing* compilers. Different from the previous studies, this work attains several new characteristics.

- *Making the BMF parallel model more practical.*

  Many studies have been devoted to showing that BMF is a good parallel computation model and a suitable parallel programming language [Ski90, Ski94b]. To enable extraction of parallelism, programs are expected to be written in terms of a small fix set of specific higher-order functions, such as *map* and *reduction*. These higher-order functions enjoy useful manipulation properties for program transformation [Bir87] and are suitable for parallel implementation [Ski92, Ski94a]. However, it is not practical to force programmers to write programs this way. In contrast, our parallelization will target general recursive programs.

- *Enriching calculational laws and theorems for parallelization.*

  BMF provides a general theory for program calculation, which should be specialized with respect to different application fields, e.g., dynamic programming

[dM92], circuit design [JS90], and optimization of functional programs [TM95, HIT96, HITT97]. In each specialization, new laws and theorems need to be developed in order to handle specific problems. However, in the field of *parallelization* (i.e., development of efficient parallel program) [GDH96, Gor96a, Gor96b], there is a lack of powerful parallelization laws and theorems, which greatly limits its scope. Our calculational framework should remedy this situation.

In this paper, we shall report our first attempt to construct a *calculational framework* specifically for parallelization. Our main contributions are as follows.

- We propose a novel inductive synthesis lemma in which two well-known synthesis techniques, namely *generalization* and *induction*, are elegantly embedded. Based on it, we develop an elementary, but general calculational theorem for parallelization (Section 4). By elementary, we mean that it contributes to the core transformations in our parallelization algorithm; and by general, we mean that it is more powerful than all the previous laws and theorems [Ski92, GDH96, Gor96a, Gor96b] and thus can be applied to synthesize many interesting parallel programs (as demonstrated in Section 4). Moreover, this theorem can be directly implemented by way of simple symbolic manipulation.
- We propose a *systematic* and *constructive* parallelization algorithm (Section 5) for the derivation of parallel programs. It can be applied to a wide class of general programs covering all primitive recursive functions with which almost all algorithms of interest can be described. Two distinguishing points of our algorithm are its constructive way of deriving associative/distributive operators from algebraic datatypes, and the effective use of the fusion and tupling calculation in the parallelizing process.
- Our parallelization algorithm is given in a calculational way like those in [OHIT97, HITT97]. Therefore, it preserves the advantages of transformation in calculational form; being correct and guaranteed to terminate. In addition, it can be naturally generalized to programs over other linear algebraic datatypes rather than only lists as used in this paper. It is not only helpful in the design of efficient parallel programs but also promising in the construction of parallelization systems.

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 making clear the parallelization problem in Section 3, we propose our new synthesis lemma from which several basic parallelization laws and the parallelization theorem are derived in Section 4. We propose our parallelization algorithm in Section 5, and highlight some future work in Section 6. Related work and conclusion are given in Section 7.

2 BMF and Parallel Computation

In this section, we briefly review the notational conventions and some basic concepts in BMF [Bir87], and point out some related results which will be used in the rest of this paper. In order to simplify our presentation, we will not formulate our calculational idea in terms of the general theory of *constructive algorithmics* as we did in [HIT96, HITT97] (see Section 6 for some related discussion.) Rather, we illustrate our idea using the theory of lists.

2.1 Functions

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

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

The *projection* function π_i selects the ith component of tuples, e.g., $\pi_1\,(a, b) = a$. Also, $\vartriangle$ is a binary operator on tuples, defined by

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

Lastly, $f_1 \vartriangle \cdots \vartriangle f_n$ is abbreviated to $\Delta_1^n f_i$.

2.2 Lists

The list datatype dominates our daily programming. Lists are finite sequences of values of the same type. There are two basic views of lists.

- *Parallel view of lists.* A list is either empty, a singleton, or a concatenation of two lists. We write $[\,]$ for the empty list, $[a]$ for the singleton list with element a, and $x +\!\!+ y$ for the concatenation of x and y. Lists in the parallel view are also called *append* (or *join*) lists.
- *Sequential view of lists.* A list is either empty $[\,]$, or is constructed by an element a and a list x by the constructor $:$ which is denoted by $a : x$. Lists in this sequential view are also called *cons* lists.

Concatenation is associative, and $[\,]$ is its unit. For example, the term $[1] +\!\!+ [2] +\!\!+ [3]$ denotes a list with three elements, often abbreviated to $[1, 2, 3]$. In addition, we write $a : x$ for $[a] +\!\!+ x$, and vice versa.

2.3 Homomorphisms, Mutumorphism, and Parallel Computational Model

List homomorphisms (or *homomorphisms* for short) [Bir87] are those functions on finite lists that *promote* through list concatenation, as precisely defined below.

Definition 1 (List Homomorphism) 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 $(\![id, +]\!)$. Two important homomorphisms are *map* and *reduction*. Map is the operator which applies a function to every element in a list. It is written as an infix $*$. Informally, we have

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

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

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

It has been argued that $*$ and / have simple massively parallel implementations on many architectures [Ski90]. 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 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 written as the composition of a reduction and a map. One can also observe that homomorphisms express the well-known divide-and-conquer parallel paradigm. More detailed studies can be found in [Ski92, GDH96, Col95] on showing why homomorphisms are a good characterization of parallel computational models and can be effectively implemented on modern parallel architectures.

It follows that if we can derive list homomorphisms, then we can get corresponding parallel programs. Unfortunately, there remains a major problem; a lot of interesting list functions are not homomorphisms themselves because there exists no appropriate operator $\oplus$ [Col95]. To solve this problem, Cole proposed the idea of a *near* homomorphism, the composition of a projection function and a homomorphism. Following Cole's idea and using the result in [HIT97, HITT97], we shall choose list *mutumorphisms* [Fok92] as our parallel computation model.

Definition 2 (List Mutumorphisms) The functions h_1, ..., h_n are called *list mutumorphisms* (or *mutumorphisms* for short) if they are mutually defined in the following way:

$$\begin{array}{lcl} h_j\ [a] & = & k_j\ a \\ h_j\ (x +\!\!+\ y) & = & ((\Delta_1^n h_i)\ x) \oplus_j ((\Delta_1^n h_i)\ y). \end{array}$$

Particularly, a single function, say h_i, is said to be a list mutumorphism, if there exist a set of functions h_1, ..., h_{i-1}, h_{i+1}, ..., h_n which together with h_i satisfy the above equational form. □

Compared to homomorphisms, mutumorphisms provide a better characterization of a parallel computational model.

- Mutumorphisms win over homomorphisms because of their more powerful descriptive power. They are considered as the most general recursive functions defined in an inductive manner [Fok92], being capable of describing most interesting functions.
- Mutumorphisms enjoy many useful manipulation properties for program transformation. Particularly, they can be automatically turned into efficient homomorphisms via the following tupling calculation (see Section 4.2.2 for an example). Therefore, they possess a similar parallel property as homomorphisms.

Theorem 1 (Tupling [HIT97]) Let $h_1, \ldots, h_n$ be mutumorphisms as defined in Definition 2. Then,

$$\Delta_1^n h_i = (\![\Delta_1^n k_i,\ \Delta_1^n \oplus_i]\!).$$ □

It then follows from the theorem that any mutumorphism h can be transformed to be a composition of a projection function and a homomorphism, i.e.,

$$h_j = \pi_j \circ (\![\Delta_1^n k_i,\ \Delta_1^n \oplus_i]\!).$$

3 Parallelization Problem

Before addressing our calculational framework for parallelization, we should be more specific about the parallelization problem we would like to resolve in this paper.

Recall that in the process of program calculation to obtain efficient sequential programs, we start with a naive and concise program without concern for its efficiency and then transform it into more and more efficient program by equational reasoning. Similarly, in the process of program calculation to parallelize programs, we should start with a naive and concise program without concern for its parallelism and then transform it into a parallel version.

Specifically, we would like to start with a general recursive program (see the definition of sbp' for an example) usually defined in the following form.

$$f\ (a : x)\ =\ \cdots f\ x\ \cdots\ q\ x\ \cdots\ a\ \cdots$$

The function f is inductively defined on lists. The definition body is an expression probably containing occurrences of recursive call $f\ x$, variable a, and even application of some other functions to x. By looking at general functional programs in Haskell [HPJWe92], we could find that most of the recursive functions over lists are defined in (or can be easily turned to) this form. Here we need not consider parallelism at all. In order to simplify our presentation, we shall mostly consider single recursive definitions rather than mutual ones (except for Section 4.2.4), and we assume that recursive functions induct over a single parameter rather than multiple ones. To be precise, we give the following definition of our specification program (initial programs to be parallelized).

Definition 3 (Specification) The programs to be parallelized are those that can be turned into the form

$$f\ (a : x)\ =\ E[\langle t_i \rangle_{i=1}^m, \langle q_j\ x \rangle_{j=1}^n, \langle f\ x \rangle_1^k]$$

where

- $E[\,]$ denotes an *expression context* with three groups of holes $\langle\,\rangle$. It contains no occurrence of a, x and f.
- $\langle t_i \rangle_{i=1}^m$ denotes a group of m holes being filled with m terms $t_1, \ldots, t_m$ respectively. It is allowed to contain occurrences of a, but not those of x.

- $\langle q_j\ x\rangle_{j=1}^{n}$ denotes a group of n holes being filled with n function applications, $q_1\ x, \ldots, q_n\ x$, where q_j's are mutumorphisms (parallelized functions).
- $\langle f\ x\rangle_1^k$ denotes a group of k holes each being filled with the same term $f\ x$. □

Clearly, this sort of specification is quite general and can describe most interesting algorithms. In fact, it can define all primitive recursive functions [Mee92]. The essential restriction we imposed on the specification is that the parameter to each recursive call to f in the RHS should be x. In other words we allow $f\ x$, but we do not allow any computation on the argument of f, e.g., $f\ (2:x)$. Several remarks should be made here.

- Our specification programs need not be given in terms of a context; rather they are general programs like *sbp* given in the introduction. We only require the existence of an expression context, which usually can be obtained in a simple way. The following is an example of sbp' in terms of a context:

$$\begin{array}{rcl} sbp'\ (a:x) & = & E[\langle a == \text{'('},\ a == \text{')'}\rangle, \\ & & \langle\rangle, \\ & & \langle sbp'\ x,\ sbp'\ x,\ sbp'\ x\rangle] \end{array}$$

  where the context E is defined by

$$\begin{array}{l} E[\langle t_1, t_2\rangle, \langle\rangle, \langle f_1, f_2, f_3\rangle] \\ = \ \lambda c.\, \text{if } t_1 \text{ then } f_1\ (c+1) \\ \qquad \text{else (if } t_2 \text{ then } c > 0\ \wedge\ f_2\ (c-1) \text{ else } f_3\ c). \end{array}$$

- For a given program, there may exist many different potential contexts. As will become clear later, it is of great importance in our parallelization framework to derive a proper one from the program. This may involve some normalizing transformation on the program prior to a context extraction.
- We have omitted the base equation for the definition of f in Definition 3:

$$f\ [\,] = e$$

  where e is an expression without occurrence of f. For brevity, we shall even omit it in the rest of the paper.

Our parallelization problem turns out to be equivalent to calculate a new parallel version for f in Definition 3. According to the discussion in Section 2, we know that mutumorphisms can be considered as a good characterization of parallel computations. We thus want this parallel version of f to be in a mutumorphic form, i.e.,

$$f\ (x +\!\!+ y) \ = \ \cdots\ f\ x\ \cdots\ f\ y\ \cdots.$$

4 Calculational Laws and Theorem for Parallelization

In this section, we first propose a new synthesis lemma in which two well-known synthesis techniques, namely *generalization* and *induction*, are well embedded. Then, based on the synthesis lemma we develop several parallelizing laws and conclude with our parallelization theorem. All of them are the basis of our parallelization algorithm as discussed in Section 5.

4.1 Synthesis Lemma

Parallelizing f in this paper means to derive a mutumorphism for f. To this end, we shall propose our *synthesis lemma*, which neatly combines two known synthesis techniques, namely generalization and induction, which have been proven to be very useful in [CTT97].

Lemma 2 (Synthesis) Given is a specification program

$$f\ (a:x) \ = \ E[\langle t_i\rangle_{i=1}^{m}, \langle q_j\ x\rangle_{j=1}^{n}, \langle f\ x\rangle_1^k] \qquad (1)$$

whose context $E[\,]$ satisfies the *fusible property* with respect to the q_j's, f and x, if there exist terms $t'_1, \ldots, t'_m$ such that for any A_i's, B_i's and y we have

$$\begin{array}{l} E[\langle A_i\rangle_{i=1}^{m}, \langle q_j\,(y +\!\!+ x)\rangle_{j=1}^{n}, \\ \quad \langle E[\langle B_i\rangle_{i=1}^{m}, \langle q_j\ x\rangle_{j=1}^{n}, \langle f\ x\rangle_1^k]\rangle_1^k] \\ = E[\langle t'_i\rangle_{i=1}^{m}, \langle q_j\ x\rangle_{j=1}^{n}, \langle f\ x\rangle_1^k]. \end{array}$$

Then we can obtain the following parallel version. For any nonempty x',

$$f\ (x' +\!\!+ x) = E[\langle G_i\ x'\rangle_{i=1}^{m}, \langle q_j\ x\rangle\rangle_{j=1}^{n}, \langle f\ x\rangle_1^k] \qquad (2)$$

where the new functions $G_1, \ldots, G_m$ are defined by[1]

$$\begin{array}{rcl} G_i\ [a] & = & t_i \\ G_i\ (x'_1 +\!\!+ x'_2) & = & t'_i[(A_i \mapsto G_i\ x'_1)_{i=1}^{m}, \\ & & \quad (B_i \mapsto G_i\ x'_2)_{i=1}^{m}, \\ & & \quad y \mapsto x'_2] \end{array}$$

Proof. We prove (2) by induction on the nonempty list x'.

- *Base Case* $[a]$. This is established by the following calculation.

$$\begin{array}{rl} & f\ (x' +\!\!+ x) \\ = & \quad \{\ \text{Assumption: } x' = [a]\ \} \\ & f\ ([a] +\!\!+ x) \\ = & \quad \{\ \text{Equation (1)}\ \} \\ & E[\langle t_i\rangle_{i=1}^{m}, \langle q_j\ x\rangle_{j=1}^{n}, \langle f\ x\rangle_1^k] \\ = & \quad \{\ \text{Definition of } G_i\ \} \\ & E[\langle G_i\,[a]\rangle_{i=1}^{m}, \langle q_j\ x\rangle_{j=1}^{n}, \langle f\ x\rangle_1^k] \\ = & \quad \{\ \text{Since } x' = [a]\ \} \\ & E[\langle G_i\ x'\rangle_{i=1}^{m}, \langle q_j\ x\rangle_{j=1}^{n}, \langle f\ x\rangle_1^k] \end{array}$$

[1] We use $t[x \mapsto y]$ to denote a term obtained from t with all occurrences of x being replaced by y.

- *Inductive Case* $x'_1 +\!\!+ x'_2$. This is established by the following calculation.

$$\begin{array}{ll} & f\ (x' +\!\!+ x) \\ = & \quad \{ \text{ Assumption: } x' = x'_1 +\!\!+ x'_2 \ \} \\ & f\ ((x'_1 +\!\!+ x'_2) +\!\!+ x) \\ = & \quad \{ \text{ Associativity of } +\!\!+ \ \} \\ & f\ (x'_1 +\!\!+ (x'_2 +\!\!+ x)) \\ = & \quad \{ \text{ Inductive hypothesis w.r.t. } x'_1 \ \} \\ & E[\langle G_i\ (x'_1)\rangle_{i=1}^m, \langle q_j\ (x'_2 +\!\!+ x)\rangle_{j=1}^n, \langle f\ (x'_2 +\!\!+ x)\rangle_1^k] \\ = & \quad \{ \text{ Inductive hypothesis w.r.t. } x'_2 \ \} \\ & E[\langle G_i\ x'_1\rangle_{i=1}^m, \langle q_j\ (x'_2 +\!\!+ x)\rangle_{j=1}^n, \\ & \quad \langle E[\langle G_i\ x'_2\rangle_{i=1}^m, \langle q_j\ x\rangle_{j=1}^n, \langle f\ x\rangle_1^k]\rangle_1^k] \\ = & \quad \{ \text{ Fusible: } E[\,] \text{ with } A_i = G_i\ x'_1,\ B_i = G_i\ x'_2 \ \} \\ & E[\langle t'_i[(A_i \mapsto G_i\ x'_1)_{i=1}^m, \\ & \qquad (B_i \mapsto G_i\ x'_2)_{i=1}^m, y \mapsto x'_2]\rangle_{i=1}^m, \\ & \qquad \langle q_j\ x\rangle_{j=1}^n, \langle f\ x\rangle_1^k]] \\ = & \quad \{ \text{ Definition of } G_i \ \} \\ & E[\langle G_i\ (x'_1 +\!\!+ x'_2)\rangle_{i=1}^m, \langle q_j\ x\rangle_{j=1}^n, \langle f\ x\rangle_1^k] \\ = & \quad \{ \text{ Since } x' = x'_1 +\!\!+ x'_2 \ \} \\ & E[\langle G_i\ x'\rangle_{i=1}^m, \langle q_j\ x\rangle_{j=1}^n, \langle f\ x\rangle_1^k] \end{array}$$ □

The synthesis lemma tells us that a recursive definition whose expression context satisfies the fusible property can be parallelized to a mutumorphic form of Equation (2)[2]. The fusible property is employed to guarantee that unfolding the functions q_j and f in a fusible context should only change the holes' contents while preserving the context structure. This lemma is motivated by the parallelizing procedure described in [CTT97], but it has two distinguishing features. First, it does not rely on comparison of functions which is unavoidable in [CTT97]. We achieve this by induction directly on append lists ($[a]$ and $x'_1 +\!\!+ x'_2$) rather than on cons lists ($[\,]$ and $a : x$). Second, it is formal and precise, which is in sharp contrast to the previous informal study. This makes it possible to construct our calculational framework for parallelization.

To see a simple use of this lemma, consider the function *length*, computing the length of a list,

$$length\ (a : x) = 1 + length\ x.$$

It can be expressed using a context as

$$\begin{array}{lcl} length\ (a : x) & = & E[\langle 1\rangle, \langle\rangle, \langle length\ x\rangle] \\ E[\langle t_1\rangle, \langle\rangle), \langle f_1\rangle] & = & t_1 + f_1. \end{array}$$

As $E[\langle A\rangle, \langle\rangle, \langle E[\langle B\rangle, \langle\rangle, \langle f\ x\rangle]\rangle] = E[\langle A + B\rangle, \langle\rangle, \langle f\ x\rangle]$ (i.e., $E[\,]$ meets the fusible condition), it follows from the synthesis lemma, after expansion of the context, that we obtain the following mutumorphic definitions.

$$\begin{array}{lcl} length\ (x' +\!\!+ x) & = & G_1\ x' + length\ x \\ G_1\ [a] & = & 1 \\ G_1\ (x'_1 +\!\!+ x'_2) & = & G_1\ x'_1 + G_1\ x'_2 \end{array}$$

It should be noted that we can go further to make this result more efficient (although this improvement is beyond parallelization itself) if we adopt syntactical comparison of function definitions to check if the new functions are equivalent to known functions being parallelized and thus reduce the number of newly introduced functions. A simple check (which compares both the base and induction equations) confirms that $G_1 = length$, showing that G_1 can be replaced by *length* and hence can be removed safely. An alternative way is to put off this syntactic comparison until after tupling transformation, as we will see for *scan'* in Section 5.

[2] This can be seen from the definition of mutumorphism where we may choose f, q_1, ..., q_n, G_1, ..., G_m to be the h_i's.

4.2 Deriving Laws and Theorem for Parallelization

Most previous studies on parallelization by program transformation [GDH96, Bra94] are essentially based on the following calculational law[3]:

$$\frac{f\ (a : x) = f\ [a] \oplus f\ x,\ \oplus \text{ is associative}}{f\ (x' +\!\!+ x) = f\ x' \oplus f\ x}.$$

However, in the present form, this law is restricted in scope and has rather limited use in practice. We shall extend it in several ways making use of our synthesis lemma. In the following, after proposing several typical extensions of the Bird's law, we conclude with a general theorem.

4.2.1 First Extension

Notice that in the RHS of the given definition of f the x is not allowed to be accessed by other functions except f. So our first extension is to remove this restriction.

Lemma 3 (Primitive Form) Given is a program

$$f\ (a : x) \quad = \quad g\ a\ (q\ x) \oplus f\ x$$

where $\oplus$ denotes an associative binary operator and q is a homomorphism $([k_q, \oplus_q])$. Then, for any non-empty lists x' and x, we have

$$f(x' +\!\!+ x) \quad = \quad G\ x'\ (q\ x) \oplus f\ x$$

where G is a function defined by

$$\begin{array}{lcl} G\ [a] & = & g\ a \\ G\ (x'_1 +\!\!+ x'_2) & = & \lambda z.\ (G\ x'_1\ (q\ x'_2 \oplus_q z) \oplus G\ x'_2\ z) \end{array}$$

Proof Sketch. Notice that

$$f\ (a : x) = E[\langle g\ a\rangle, \langle q\ x\rangle, \langle f\ x\rangle]$$

where

$$E[\langle t_1\rangle, \langle q_1\rangle, \langle f_1\rangle] = t_1\ q_1 \oplus f_1.$$

$E[\,]$ is fusible w.r.t. q, f and x, because

$$\begin{array}{l} E[\langle A\rangle, \langle q\ (y +\!\!+ x)\rangle, \langle E[\langle B\rangle, \langle q\ x\rangle, \langle f\ x\rangle]\rangle] \\ \quad = E[\langle \lambda z.\ (A\ (q\ y \oplus_q z) \oplus B\ z)\rangle, \langle q\ x\rangle, \langle f\ x\rangle]. \end{array}$$

Therefore, this lemma follows from the synthesis lemma. □

4.2.2 Second Extension

Our second extension is to allow the specification program f to use an *accumulating parameter*.

[3] Note we do not think that the third homomorphism theorem [Gib96] is a calculational law, because it tells the existence of a parallel program but does not address how to calculate them. More discussion can be found in Section 7.

Lemma 4 (Accumulation) Given is a program

$$f\ (a:x)\ c \quad = \quad g_1\ a\ (q\ x)\ c\ \oplus\ f\ x\ (g_2\ a \otimes c)$$

where $\oplus$ and $\otimes$ are two associative binary operators, and $q = ([k_q, \oplus_q])$. Then, for any non-empty lists x' and x, we have

$$f\ (x' +\!\!+ x)\ c \quad = \quad G_1\ x'\ (q\ x)\ c \oplus f\ x\ (G_2\ x' \otimes c)$$

where G_1 and G_2 are functions defined by

$$\begin{array}{lcl} G_1\ [a]\ z\ c & = & g_1\ a\ z\ c \\ G_1\ (x_1' +\!\!+ x_2')\ z\ c & = & G_1\ x_1'\ (q\ x_2' \oplus_q z)\ c \oplus \\ & & \quad G_1\ x_2'\ z\ (G_2\ x_1' \otimes c) \\ G_2\ [a] & = & g_2\ a \\ G_2\ (x_1' +\!\!+ x_2') & = & G_2\ x_2' \otimes G_2\ x_1' \end{array}$$

Proof Sketch. First we move the accumulating parameter from LHS to RHS by means of a lambda abstraction, i.e.,

$$f\ (a:x) = \lambda c.\,(g_1\ a\ (q\ x)\ c \oplus f\ x\ (g_2\ a \otimes c))$$

Then we define a fusible expression context by

$$E[\langle t_i \rangle_{i=1}^2, \langle q_1 \rangle, \langle f_1 \rangle] = \lambda c.\,(t_1\ q_1\ c \oplus f_1\ (t_2 \otimes c))$$

such that

$$f\ (a:x) = E[\langle g_i\ a \rangle_{i=1}^2, \langle q\ x \rangle, \langle f\ x \rangle].$$

Now, it is not difficult to verify that $E[\,]$ is fusible and thus the synthesis lemma can be applied. □

Before going on with other extensions, we pause with a more concrete use of this lemma. Consider the *sbp* problem given in the introduction. Normalizing the if-structures in the definition [CDG96] gives (omitting initial equation)

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

In order to use the lemma, we introduce two functions g_1 and g_2 to abstract two subexpressions.

$$\begin{array}{lcl} sbp'\ (a:x)\ c & = & g_1\ a\ c\ \wedge\ sbp'\ x\ (g_2\ a + c) \\ g_1\ a\ c & = & \text{if } a == \text{'('} \text{ then } True \\ & & \text{else if } a == \text{')'} \text{ then } c > 0 \text{ else } True \\ g_2\ a & = & \text{if } a == \text{'('} \text{ then } 1 \\ & & \text{else if } a = \text{')'} \text{ then } -1 \text{ else } 0 \end{array}$$

It follows from Lemma 4[4] that

$$sbp'\ (x' +\!\!+ x)\ c = G_1\ x'\ c\ \wedge\ sbp'\ x\ (G_2\ x' + c)$$

where

$$\begin{array}{lcl} G_1\ [a]\ c & = & \text{if } a == \text{'('} \text{ then } True \\ & & \text{else if } a == \text{')'} \text{ then } c > 0 \\ & & \text{else } True \\ G_1\ (x_1' +\!\!+ x_2')\ c & = & G_1\ x_1'\ c\ \wedge\ G_1\ x_2'\ (G_2\ x_1' + c) \\ G_2\ [a] & = & \text{if } a == \text{'('} \text{ then } 1 \\ & & \text{else if } a == \text{')'} \text{ then } (-1) \text{ else } 0 \\ G_2\ (x_1' +\!\!+ x_2') & = & G_2\ x_2' + G_2\ x_1' \end{array}$$

[4]Note that the auxiliary $q\ x$ call can be made optional in both Lemma 3 and Lemma 4 when x does not appear outside of the recursive f call. This occurs for the sbp' definition.

This is the parallel version we aim to get in this paper, although it is currently inefficient because of multiple traversals of the same input list by several functions. But this can be automatically improved by the tupling calculation as intensively studied in [Chi93, HITT97]. For instance, we can obtain the following program by tupling sbp', G_1 and G_2.

$$\begin{array}{lcl} sbp'\ x\ c & = & s \text{ where } (s, g_1, g_2) = tup\ x\ c \\ tup\ [a]\ c & = & \text{if } a == \text{'('} \text{ then} \\ & & \quad (c + 1 == 0, True, 1) \\ & & \text{else if } a == \text{')'} \text{ then} \\ & & \quad (c - 1 == 0, c > 0, -1) \\ & & \text{else } (c == 0, True, 0) \\ tup\ (x +\!\!+ y)\ c & = & \text{let } (s_x, g_{1x}, g_{2x}) = tup\ x\ c \\ & & \quad (s_y, g_{1y}, g_{2y}) = tup\ y\ (g_{2x} + c) \\ & & \text{in } (g_{1x}\ \wedge\ s_y,\ g_{1x}\ \wedge\ g_{1y},\ g_{2x} + g_{2y}) \end{array}$$

It seems not so apparent that the above gives an efficient parallel program. Particularly, the second recursive call $tup\ y\ (g_{2x} + c)$ relies on g_{2x}, an output from the first recursive call $tup\ x\ c$. Nevertheless, this version of *tup* can be effectively implemented in parallel on a multiple processor system supporting bidirectional tree-like communication with $O(\log n)$ complexity where n denotes the length of the input list, by using a algorithm similar to that in [Ble89]. Two passes are employed; an upward pass in the computation is used to compute the third component of $tup\ x\ c$ before a downward pass is used to compute the first two values of the tuple.

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].

4.2.3 Third Extension

The importance of *conditional* structure in a definition has been highlighted in [FG94, CDG96]. Our third extension is to allow explicit conditional structure.

Lemma 5 (Conditional) Given is a program

$$\begin{array}{lcl} f\ (a:x) & = & \text{if } g_1\ a\ (q_1\ x) \text{ then } g_2\ a\ (q_2\ x) \oplus f\ x \\ & & \text{else } g_3\ a\ (q_3\ x) \end{array}$$

where $\oplus$ denotes an associative binary operator, and $q_i = ([k_{q_i}, \oplus_{q_i}])$ for $i = 1, 2, 3$. Then, for any non-empty lists x' and x, we have

$$\begin{array}{lcl} f\ (x' +\!\!+ x) & = & \text{if } G_1\ x'\ (q_1\ x) \text{ then } G_2\ x'\ (q_2\ x) \oplus f\ x \\ & & \text{else } G_3\ x'\ (q_3\ x) \end{array}$$

where G_1, G_2 and G_3 are defined by

$$\begin{array}{lcl} G_1\ [a]\ z & = & g_1\ a\ z \\ G_1\ (x_1' +\!\!+ x_2')\ z & = & G_1\ x_1'\ (q_1\ x_2' \oplus_{q_1} z)\ \wedge\ G_1\ x_2'\ z \\ G_2\ [a]\ z & = & g_2\ a\ z \\ G_2\ (x_1' +\!\!+ x_2')\ z & = & G_2\ x_1'\ (q_2\ x_2' \oplus_{q_2} z)\ \oplus\ G_2\ x_2'\ z \\ G_3\ [a]\ z & = & g_3\ a\ z \\ G_3\ (x_1' +\!\!+ x_2')\ z & = & \text{if } G_1\ x_1'\ (q_1\ x_2' \oplus_{q_1} z) \text{ then } G_3\ x_2'\ z \\ & & \text{else } G_3\ x_1'\ (q_3\ x_2' \oplus_{q_3} z) \end{array}$$

Proof Sketch. We can define f by

$$f\ (a:x) = E[\langle g_i\ a \rangle_{i=1}^3, \langle q_i\ x \rangle_{i=1}^3, \langle f\ x \rangle]$$

where

$$E[\langle t_i\rangle_{i=1}^3, \langle q_i\rangle_{i=1}^3, \langle f_1\rangle] = \text{if } t_1\ q_1 \text{ then } t_2\ q_2 \oplus f_1 \text{ else } t_3\ q_3.$$

The context $E[\,]$ is fusible since

$$\begin{array}{l} E[\langle A_i\rangle_{i=1}^3, \langle q_j(y +\!\!+ x)\rangle_{j=1}^3, \langle E[\langle B_i\rangle_{i=1}^3, \langle q_j\ x\rangle_{j=1}^3, \langle f\ x\rangle]\rangle] \\ \quad = E[\langle t'_i\rangle_{i=1}^3, \langle q_j\ x\rangle_{j=1}^3, \langle f\ x\rangle] \end{array}$$

where

$$\begin{array}{lcl} t'_1 & = & \lambda z.\,(A_1\,(q_1\ y \oplus_{q_1} z) \wedge B_1\ z) \\ t'_2 & = & \lambda z.\,(A_2\,(q_2\ y \oplus_{q_2} z) \oplus B_2\ z) \\ t'_3 & = & \lambda z.\,(\text{if } A_1\,(q_1\ y \oplus_{q_1} z) \text{ then } B_3\ z \\ & & \qquad \text{else } A_3\,(q_3\ y \oplus_{q_3} z)) \end{array}$$

□

4.2.4 Fourth Extension

So far we have considered linear recursions, i.e, recursions with a single recursive call in the definition body. In this section, we 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.

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

To handle nonlinear recursions properly, we make use of distributive and commutative properties.

Lemma 6 (Multiple Recursive Calls) Assume that f_1 and f_2 are mutually recursive functions defined by

$$\begin{array}{lcl} f_1\ (a:x) & = & g_1\ a\ (q_1\ x) \oplus (p_{11} \otimes f_1\ x) \oplus (p_{12} \otimes f_2\ x) \\ f_2\ (a:x) & = & g_2\ a\ (q_2\ x) \oplus (p_{21} \otimes f_1\ x) \oplus (p_{22} \otimes f_2\ x) \end{array}$$

where $q_i = (\!|k_{q_i}, \oplus_{q_i}|\!)$ for $i = 1, 2$, $\oplus$ is associative and commutative, and $\otimes$ is an associative operator which is distributive over $\oplus$, i.e., for any x, y and z,

$$\begin{array}{lcl} x \otimes (y \oplus z) & = & (x \otimes y) \oplus (x \otimes z) \\ (y \oplus z) \otimes x & = & (y \otimes x) \oplus (z \otimes x). \end{array}$$

Then, for any non-empty lists x' and x, we have

$$\begin{array}{lcl} f_1\ (x' +\!\!+ x) & = & G_1\ x'\,(q_1\ x) \oplus (G_{11}\ x' \otimes f_1\ x) \\ & & \qquad \oplus (G_{12}\ x' \otimes f_2\ x) \\ f_2\ (x' +\!\!+ x) & = & G_2\ x'\,(q_2\ x) \oplus (G_{21}\ x' \otimes f_1\ x) \\ & & \qquad \oplus (G_{22}\ x' \otimes f_2\ x) \end{array}$$

where

$$\begin{array}{lcl} G_1\ [a]\ z & = & g_1\ a\ z \\ G_1\ (x'_1 +\!\!+ x'_2)\ z & = & G_1\ x'_1\,(q_1\ x'_2 \oplus_{q_1} z) \oplus \\ & & (G_{11}\ x'_1 \otimes G_1\ x'_2\ z) \oplus \\ & & (G_{12}\ x'_1 \otimes G_2\ x'_2\ z) \\ G_2\ [a]\ z & = & g_2\ a\ z \\ G_2\ (x'_1 +\!\!+ x'_2)\ z & = & G_2\ x'_1\,(q_2\ x'_2 \oplus_{q_2} z) \oplus \\ & & (G_{21}\ x'_1 \otimes G_1\ x'_2\ z) \oplus \\ & & (G_{22}\ x'_1 \otimes G_2\ x'_2\ z) \\ G_{11}\ [a] & = & p_{11} \\ G_{11}\ (x'_1 +\!\!+ x'_2) & = & (G_{11}\ x'_1 \otimes G_{11}\ x'_2) \oplus \\ & & (G_{12}\ x'_1 \otimes G_{21}\ x'_2) \\ G_{12}\ [a] & = & p_{12} \\ G_{12}\ (x'_1 +\!\!+ x'_2) & = & (G_{11}\ x'_1 \otimes G_{22}\ x'_2) \oplus \\ & & (G_{12}\ x'_1) \otimes G_{22}\ x'_2) \\ G_{21}\ [a] & = & p_{21} \\ G_{21}\ (x'_1 +\!\!+ x'_2) & = & (G_{21}\ x'_1 \otimes G_{11}\ x'_2) \oplus \\ & & (G_{22}\ x'_1 \otimes G_{21}\ x'_2) \\ G_{22}\ [a] & = & p_{22} \\ G_{22}\ (x'_1 +\!\!+ x'_2)\ z & = & (G_{21}\ x'_1 \otimes G_{12}\ x'_2) \oplus \\ & & (G_{22}\ x'_1 \otimes G_{12}\ x'_2) \end{array}$$

□

The proof of the lemma is omitted, which is not difficult using induction on the input of f_1 and f_2. In fact, the calculational law in this lemma is synthesized using a natural extension of the synthesis lemma, from single contexts to mutual ones. This lemma can be easily generalized from two mutually recursive functions to n functions. On the other hand, in case that f_1 and f_2 are the same, this lemma is specialized to deal with a single function whose definition body contains multiple occurrences of the recursive call.

Let us use this theorem to parallelize the *lfib* function. Noticing that $\oplus = +$ and $\otimes = \times$, we get the following parallel program where $G_1\ x\ z = G_2\ x\ z = 0$.

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

where

$$\begin{array}{lcl} G_{11}\ [a] & = & 1 \\ G_{11}\ (x'_1 +\!\!+ x'_2) & = & (G_{11}\ x'_1 \times G_{11}\ x'_2) + \\ & & (G_{12}\ x'_1 \times G_{21}\ x'_2) \\ G_{12}\ [a] & = & 1 \\ G_{12}\ (x'_1 +\!\!+ x'_2) & = & (G_{11}\ x'_1 \times G_{22}\ x'_2) + \\ & & (G_{12}\ x'_1 \times G_{22}\ x'_2) \\ G_{21}\ [a] & = & 1 \\ G_{21}\ (x'_1 +\!\!+ x'_2) & = & (G_{21}\ x'_1 \times G_{11}\ x'_2) + \\ & & (G_{22}\ x'_1 \times G_{21}\ x'_2) \\ G_{22}\ [a] & = & 0 \\ G_{22}\ (x'_1 +\!\!+ x'_2)\ z & = & (G_{21}\ x'_1 \times G_{12}\ x'_2) + \\ & & (G_{22}\ x'_1 \times G_{12}\ x'_2) \end{array}$$

This result can be mechanically transformed into an efficient $O(\log n)$ parallel program by the tupling calculation [HITT97]. As an interesting side result, we have actually derived an $O(\log n)$ sequential algorithm for computing the standard *fib* function. This can be seen by first replacing all x, x', x'_1, and x'_2 in the program by their lengths respectively, and then applying the tupling calculation (followed by even/odd case analysis).

4.2.5 Main Theorem

We have demonstrated the key extensions together with the corresponding parallelization laws. And, we have simplified our presentation in several ways. First, we discussed each extension almost independently; for example, the accumulation parameter is not considered in Lemma 5 and 6. Second, in all the lemmas, the $f\ x$ are placed on the right side of $\oplus$ in the definition body. In fact, it can be put on the left side just by using a new associative operator $x \oplus' y = y \oplus x$, and it can even be put in the middle of two $\oplus$, as in $g_1\ a\ (q_1\ x) \oplus f\ x \oplus g_2\ a\ (q_2\ x)$. Taking all the above into consideration, we obtain a class of new recursive definitions that can be parallelized.

Theorem 7 (Parallelization) Let g_i be any function, q_i be a mutumorphism (say $([k_{q_i}, \oplus_{q_i}])$, $\oplus$ be an associative operators. Then, we consider the following definitions[5] of a function f

$$f\ (a : x)\ \{c\}\ =\ e_1 \oplus e_2 \oplus \cdots \oplus e_n$$

where every e_i is

- *a non-recursive expression*, i.e., $g_i\ a\ (q_i\ x)\ \{c\}$; or
- *a recursive call*: $f\ x\ \{g\ a \odot c\}$ where g is a function and $\odot$ is associative; or
- *a conditional expression wrapping recursive calls*: if $g_{i_1}\ a\ (q_{i_1}\ x)\ \{c\}$ then $e'_1 \oplus e'_2 \oplus \cdots \oplus e'_n$ else $g_{i_2}\ a\ (q_{i_2}\ x)\ \{c\}$, where at least one e'_i is a recursive call and the others are non-recursive expressions.

Functions of this form can be parallelized by calculation, provided (1) all occurrences of recursive calls to f are the same[6], and (2) if there are two or more occurrences of f, then $\oplus$ should be commutative and there should exist an associative operator $\otimes$ (with the unit $\iota_\otimes$) which is distributive over $\oplus$. □

This theorem summarizes a sufficient condition for a recursive definition to be parallelized. We will omit the proof (which is based on the lemmas in this section), but shall show how it will be used practically in the next section.

5 Parallelization Algorithm

Having given our parallelization theorem, we shall propose our parallelization algorithm, making it clear how to recognize associative and distributive operators in a program and how to transform a general program so that the parallelization theorem can be applied.

5.1 Recognizing Associative and Distributive Operators

Central to our parallelization laws and theorem is the use of associativity of a binary operator $\oplus$ and distributive operator $\otimes$. Therefore we must be able to recognize them in a program. There are several ways: limiting application scope by requiring all associative and distributive operators to be made explicit, e.g. in [FG94, CTT97], or adopting AI techniques like anti-unification [Hei94] to synthesize them. All of them need human insights.

Fortunately, we are able to constructively derive such associative operators from the resulting *type* of the function to be parallelized! It is known [SF93] that every type R that has a zero constructor C_Z (a constructor with no arguments like [] for lists) has a function $\oplus$, which is associative and has the zero C_Z for both a left and right identity. Such a function $\oplus$ is called *zero replacement function*, since $x \oplus y$ means to replace all C_Z in x with y. Here are two examples. For the type of cons lists, we have such a $\oplus$ defined by

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

which is the list concatenation operator $+\!\!+$; and for the type of natural numbers, it is defined by

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

which is the integer addition $+$.

Associated with $\oplus$, we can could derive a most natural distributive $\otimes$. 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 very important as seen in Lemma 6.

In summary, given a function f whose result has the type of R, our associative operator is nothing but the zero replacement function derived from R. At first sight, this seems to be rather restrictive, because we simply take one fixed operator for every data type (e.g., $+\!\!+$ for lists and $+$ for natural numbers). In fact, among the associative operators with the type of $R \to R \to R$, the zero replacement function is the most primitive one. This is because no matter how the associative operators are defined, they basically have to produce their results (of type R) using the data constructors of R for gluing whereas these data constructors could be expressed in terms of the zero constructor and the zero replacement function (see Step 1 in Section 5.2 for some examples.)

5.2 Main Algorithm

Suppose that we are given a function defined by

$$\begin{array}{l} f\ :\ [A] \to \{C\} \to R \\ f\ (a : x)\ \{c\}\ =\ body \end{array}$$

where *body* is any expression. The accumulating parameter may be unnecessary (and thus denoted as $\{c\}$) which can then be ignored. Certainly, not all functions have efficient parallel versions and so our algorithm will give up parallelization in case our conditions are not satisfied. Nevertheless, our algorithm can automatically parallelize a wider class of functions covering many interesting ones (e.g., *sbp*, *fib* and *scan*) than existing calculational methods.

[5] We use $\{c\}$ in the definition of f to denote that c is an option which may not appear.

[6] In the case of mutual-recursive functions, as seen in Lemma 6, we allow calls to f to be different but logically treat them simply as f calls.

We shall adopt *scan* (also known as *prefix sums*) [Ble89, FG94, Gor96b] as our running example, while specializing the general operator to be $+$ for readability. This example is interesting not only because of its importance in parallel computation [Ble89] but also because of the non-triviality of a correct and efficient parallel implementation, as argued in [O'D94].

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

Step 1: Making the associative operator explicit

First of all, we make the associative operator $\oplus$ explicit in our program to use our parallelization theorem. Note that this $\oplus$ is not an arbitrary associative operator; rather it is a zero replacement operator derivable from the resulting type R. Theoretically [SF93], for any linear datatype[7] R with a zero data constructor C_Z, a constructor expression, say $C_i\ e_1\ e_r\ e_2$ where e_r corresponds to the recursive component, can be transformed into a new expression combined with e_r by $\oplus$, i.e., $(C_i\ e_1\ C_Z\ e_2) \oplus e_r$. We apply this transformation to all those constructor expressions whose recursive component contains occurrences of a recursive call to f. For instance, when R is the list type (whose zero constructor is $[\,]$ and whose associative operator is $+\!\!+$), we apply the following rule to the body:

$$\frac{f\ x \text{ appears in } e_r}{e : e_r\ \Rightarrow\ [e] +\!\!+ e_r}.$$

Similarly, when R is the type of natural numbers, we have the rule:

$$\frac{f\ x \text{ appears in } e_r}{Succ\ e_r \Rightarrow (Succ\ 0) + e_r}$$

Returning to our running example, we should get

$$scan\ (a:x) \quad = \quad [a] +\!\!+ (a+) * scan\ x.$$

Step 2: Normalizing *body* by abstraction and fusion calculation

After $\oplus$ is made explicit and the normalization algorithm [CDG96] for conditional structure has been applied, the *body* should be the form $e_1 \oplus e_2 \oplus \cdots \oplus e_n$ or be a single conditional expression of the form if e_p then $e_1 \oplus e_2 \oplus \cdots \oplus e_n$ else $e'_1 \oplus e'_2 \oplus \cdots \oplus e'_n$. These e_i's and e'_i's can be classified into three groups.

- $\mathcal{A}$-group: the expression contains no recursive call to f.
- $\mathcal{B}$-group: the expression is just a recursive call to f.
- $\mathcal{C}$-group: the expression is an expression containing a recursive call to f as its component[8].

For an expression e in the $\mathcal{A}$-group, we turn it into the form $g_e\ a\ (q_e\ x)\ \{c\}$ where g_e is a function and q_e is a homomorphism. This can be done as follows. Assuming that $q_1, \ldots, q_n$ are all functions being applied to x in e, we first apply our parallelization algorithm to obtain their corresponding mutumorphisms, which then can be tupled [HITT97] to a homomorphism, say q_e. So we have $q_i\ x = \pi_i\ (q_e\ x)$. Now g_e with three arguments can be defined by $g_e = \lambda a\ q_x\ \{c\}.\ e[(q_i\ x \mapsto \pi_i\ q_x)_{i=1}^n]$.

For all expressions in the $\mathcal{B}$-group, we check if they are in the same form as $f\ x\ \{(g\ a \odot c)\}$ w.r.t. the same function g and the associative operator $\odot$ derived from c's datatype. Note that if f has no accumulating parameter, we just check if they are the same as $f\ x$. If it succeeds, we continue; otherwise, we give up parallelization.

For an expression e in the $\mathcal{C}$-group, we introduce a new function $f_e\ x\ a\ c = e$ and try to derive a recursive definition for f_e by automatic fusion calculation [SF93, TM95, HIT96, OHIT97]. If this succeeds, we proceed to parallelize f_e by our parallelization algorithm. It is fusion calculation that helps us move expressions from the $\mathcal{C}$-group to the $\mathcal{A}$-group.

After doing so, we have got a program in the form that can be parallelized according to the parallelization theorem (Note that we may need to normalize conditional expressions to our required form by the algorithm in [CDG96]; this step is omitted here).

Returning to our running example, recall that we have reached the point where

$$body = \underline{[a]} +\!\!+ \underline{(a+) * scan\ x}.$$

The first underlined expression is in the $\mathcal{A}$-group which can be turned into $g_1\ a$ where $g_1\ a = [a]$, whereas the second is in the $\mathcal{C}$-group which needs fusion calculation. It is actually a fusion of two functions $(a+)*$ and *scan*. Let $scan'\ x\ a = (a+) * scan\ x$. Fusion calculation will give

$$\begin{array}{lcl} scan'\ [a]\ c & = & [a + c] \\ scan'\ (a:x)\ c & = & [a + c] +\!\!+ scan'\ x\ (a \underline{+} c). \end{array}$$

Notice the fact that the above underlined $+$ is an associative operator derived from the type of the accumulating result c, which actually plays an important role both in the previous fusion calculation and in the later parallelization of $scan'$. Now $scan\ (a:x) = [a] +\!\!+ scan'\ x\ a$, which indicates that *scan* is no longer a recursion and the task of parallelization has moved to parallelizing $scan'$. Repeating the above step to the body of $scan'$ would give

$$\begin{array}{lcl} scan'\ (a:x)\ c & = & g_1\ a\ c +\!\!+ scan'\ x\ (g_2\ a + c) \\ & & \text{where } g_1\ a\ c = [a + c],\ g_2\ a = a \end{array}$$

Step 3: Applying parallelization laws and optimizing by tupling calculation

Now we are ready to apply the parallelization theorem to derive a parallel version of f. There are three cases.

- If the transformed *body* has no recursive call to f, we need do nothing for f as seen for *scan*, but turn to parallelize functions used in the body.
- If the transformed *body* has a single occurrence of recursive call, we can apply the parallelization theorem directly. For instance, we can obtain the following parallel version for $scan'$:

$$\begin{array}{lcl} scan'\ (x +\!\!+ y)\ c & = & G_1\ x\ c +\!\!+ scan'\ y\ (G_2\ x + c) \\ G_1\ [a]\ c & = & [a + c] \\ G_1\ (x +\!\!+ y)\ c & = & G_1\ x\ c +\!\!+ G_1\ y\ (G_2\ x + c) \\ G_2\ [a] & = & a \\ G_2\ (x +\!\!+ y) & = & G_2\ y + G_2\ x \end{array}$$

[7] A datatype is linear if each branch has a single recursive component. So lists are linear but trees are not.

[8] It may be an conditional expression whose predicate part contains a recursive call to f.

- Otherwise, the transformed *body* has more than one occurrences of the recursive call. In this case, we require that $\oplus$ should be commutative and should have a corresponding distributive operator $\otimes$ with the unit say $\iota_\otimes$. Then, we can apply the parallelization theorem.

Application of the parallelization theorem gives a parallel program in an mutumorphic form as seen for $scan'$. To obtain the final version, we need further optimization using the automatic tupling calculation as demonstrated in Section 4.2.2. Following the same thought, we can calculate the following final version for $scan'$ (by tupling $scan'$, G_1 and G_2), which is a $O(\log n)$ parallel algorithm, as efficient as that in [Ble89].

$$\begin{array}{lcl} scan'\ x\ c & = & s \text{ where } (s, g_1, g_2) = tup\ x\ c \\ tup\ [a]\ c & = & ([a+c], [a+c], a) \\ tup\ (x +\!\!+ y)\ c & = & \text{let } (s_x, g_{1x}, g_{2x}) = tup\ x\ c \\ & & \quad (s_y, g_{1y}, g_{2y}) = tup\ y\ (g_{2x} + c) \\ & & \text{in } (g_{1x} +\!\!+ s_y,\ g_{1x} +\!\!+ g_{1y},\ g_{2y} + g_{2x}) \end{array}$$

Interestingly, this program can be improved again by the fact that the first and the second components produced by *tup* are always the same. Therefore, we can remove the second component and obtain the following program.

$$\begin{array}{lcl} scan'\ x\ c & = & s \text{ where } (s, g_2) = tup\ x\ c \\ tup\ [a]\ c & = & ([a+c], a) \\ tup\ (x +\!\!+ y)\ c & = & \text{let } (s_x, g_{2x}) = tup\ x\ c \\ & & \quad (s_y, g_{2y}) = tup\ y\ (g_{2x} + c) \\ & & \text{in } (s_x +\!\!+ s_y,\ g_{2y} + g_{2x}) \end{array}$$

6 Future Extensions

In this section, we highlight some future work. First of all, although the parallelization framework in this paper is defined for functions over lists, it can be *naturally* extended to other linear data types based on the theory of Constructive Algorithmics [Mal90, Fok92]. It will be interesting to formalize the parallelization framework in a *polytypic* way [JJ96], making this extension be more precise. Specifically, a straightforward extension is to generalize the present results to *linear* data types which permit multiple base data constructors and multiple recursive data constructors, with the latter having only one recursive argument each. Such a data type can be shown to have an associative decomposition operator, and a corresponding synthesis lemma. A simple example of this is natural numbers with + as its associative decomposition operator. The corresponding synthesis lemma for functions on natural numbers is given below.

Lemma 8 (Synthesis: Natural Numbers) Given is a program

$$f\ (Succ\ x) = E[\langle t_i \rangle_{i=1}^m, \langle g_j\ x \rangle_{j=1}^n, \langle f\ x \rangle_1^k]$$

whose context $E[\,]$ satisfies the *fusible property* with respects to g_j's, f and x, i.e., there exist terms $t'_1, \ldots, t'_m$ such that for any A_i's, B_i's, y and z we have

$$\begin{array}{l} E[\langle A_i \rangle_{i=1}^m, \langle g_j(y+x) \rangle_{j=1}^n, \langle E[\langle B_i \rangle_{i=1}^m, \langle g_j\ x \rangle_{j=1}^n, \langle f\ x \rangle_1^k] \rangle_1^k] \\ \quad = E[\langle t'_i \rangle_{i=1}^m, \langle g_j\ x \rangle_{j=1}^n, \langle f\ x \rangle_1^k] \end{array}$$

Then we can obtain the following parallel version: for any nonempty x',

$$f\ (x' + x) = E[\langle G_i\ x' \rangle_{i=1}^m, \langle g_j\ x) \rangle_{j=1}^n, \langle f\ x \rangle_1^k]$$

where the new functions $G_1, \ldots, G_m$ are defined by

$$\begin{array}{lcl} G_i\ 0 & = & t_i \\ G_i\ (x'_1 + x'_2) & = & t'_i[(A_i \mapsto G_i\ x'_1)_{i=1}^m, \\ & & \quad (B_i \mapsto G_i\ x'_2)_{i=1}^m, \\ & & \quad y \mapsto x'_2]. \end{array}$$

□

Another direction for enhancement is to generalize the present result to *nested* linear data types. As amply demonstrated by the NESL work [Ble92], *nested sequences* are particularly important for expressing irregular problems, such as sparse matrixes. Effective parallelization of such problems depend on the ability to parallelize flattened version of nested sequences. It may be interesting to see how our parallelization theorem can be extended to handle nested linear data types. This enhancement could be used to provide a more friendly front-end to NESL, by allowing conventional sequential programs to be used without being restricted to the available parallel primitives.

7 Related Work and Conclusion

It is known to be very hard to give a *general* study of parallelization [Ski94a] because it requires a framework well integrating three general things: a general parallel programming language, a general parallelization algorithm, and a general parallel model. In this paper, we show that BMF can provide us with such a framework. Particularly, we propose a general parallelization algorithm in BMF, which has not received its worthy study. Being more constructive, our parallelization algorithm is not only helpful in design of efficient parallel programs in general but also promising in the construction of parallelizing compilers.

Besides the related work given in the introduction, our work is closely related to the studies of parallel programming (particularly with homomorphisms) in BMF [Ski92, Col95, Gor96b, Gor96a, GDH96, HIT97], and to the previous works on proving that homomorphisms are a good characterizations of parallel computational models [Ski92, GDH96, Gor96a]. They provide the basis of our work. Homomorphisms provide us with an excellent common interface for programmers, lower-level parallel implementation, and higher-level parallelizing transformation. They can be considered as a parallel skeleton, but are different from those in [Col89, DFH+93] which are mainly designed to capture various kinds of control structures of programs. In contrast, homomorphisms are derivable from the data structures that programs are defined on, thus their control structures are much simpler and can be mapped efficiently to wide range of parallel architectures.

Much work has been devoted to deriving efficient homomorphisms in BMF. We classify them into three approaches. The first approach [Ski92] requires initial programs to be written in terms of a small set of specific homomorphisms such as *map* and *reduction*, from which more complicated homomorphisms can be derived, based on calculational laws such as *promotion rules*. However, it is impractical to force programmers to write programs in this way.

The second approach is to derive homomorphism from recursive definitions. The main idea is to identify those recursive patterns whose list homomorphisms can be easily derived, e.g., in [GDH96, Gor96a]. However, the given patterns are less general and the application scope is rather

limited. Moreover, although it makes heavy use of associativity of operators, how to recognize and derive them in a program was not described at all. In contrast, our approach is not restricted to a specific recursive pattern, and we give a way to recognize associative operators from the resulting type information.

The third approach is based on the third homomorphism theorem [Gib96]. The third homomorphism theorem says that an algorithm h which can be formally described by two specific sequential algorithms (*leftward* and *rightward reduction* algorithms) is a list homomorphism. Barnard et al [BSS91] once tried it on the language recognition problem. Although the existence of an associative binary operator is guaranteed, the theorem does not address any efficient way of calculating it. Gorlatch [Gor96a, Gor96b] proposed an idea of synthesizing list homomorphisms by generalizing both leftward and rightward reduction functions, which was then well formalized by using the term rewriting technique [GG97]. Nevertheless, as shown in this paper, it is possible to parallelize systematically using only a leftward sequential program.

Our synthesis lemma was greatly inspired by the parallel synthesis algorithm in [CDG96, CTT97]. We extended and formalize the idea of second order generalization and inductive derivation in that algorithm.

In traditional imperative languages there are also many ongoing efforts at developing sophisticated techniques for parallelizing iterative loop [FG94]. Different from the usual way [Len96] based on the analysis of dependence graph, it is based on a parallel reduction of function composition which are associative, relying on the existence of a *template* form which can be efficiently maintained. Our idea of fusible contexts is related but both more formal and more general than that of a closed template form. We attempt to deal with more general recursive equations, and do not require heuristic simplification techniques.

Our work is much influenced by the ideas of derivation of associative and distributive operator from data types in [SF93]. 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.

This work is also related to our previous work [HIT97]. Previous work starts from the specification of compositions of mutumorphisms, while this work shows how to derive mutumorphisms from general sequential programs.

Acknowledgement

This paper owes much to the thoughtful and helpful discussions with Manuel Chakravarty, Fer-Jan de Vries, Masami Hagiya, Hideya Iwasaki, Gabriele Keller, Mizuhito Ogawa, Akihiko Takano, and other members of the Tokyo CACA seminars. Particularly, Manuel Chakravarty carefully read a draft of this paper and gave us many valuable comments. Thanks are also to POPL referees who provided detailed and helpful comments.

References

[Bac89] R. Backhouse. An exploration of the Bird-Meertens formalism. In *STOP Summer School on Constructive Algorithmics, Ameland*, September 1989.

[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.

[Bir89] R. Bird. Constructive functional programming. In *STOP Summer School on Constructive Algorithmics, Abeland*, 9 1989.

[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.

[Bra94] T. A. Bratvold. Parallelising a functional program using a list-homomorphism skeleton. In Hoon Hong, editor, *PASCO'94: First International Symposium on Parallel Symbolic Computation*, pages 44–53. World Scientific, September 1994.

[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.

[Chi93] W. Chin. Towards an automated tupling strategy. In *Proc. Conference on Partial Evaluation and Program Manipulation*, pages 119–132, Copenhagen, June 1993. ACM Press.

[Col89] M. Cole. *Algorithmic skeletons : a structured approach to the management of parallel computation.* Research Monographs in Parallel and Distributed Computing, Pitman, London, 1989.

[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.

[Dar81] J. Darlington. An experimental program transformation system. *Artificial Intelligence*, 16:1–46, 1981.

[DFH+93] J. Darlington, A.J. Field, P.G. Harrison, P.H.J. Kelly, D.W.N. Sharp, Q. Wu, and R.L. While. Parallel programming using skeleton functions. In *Parallel Architectures & Languages Europe.* Springer-Verlag, June 93.

[dM92] O. de Moor. *Categories, relations and dynamic programming.* Ph.D thesis, Programming research group, Oxford Univ., 1992. Technical Monograph PRG-98.

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

[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.

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

[GG97] A. Geser and S. Gorlatch. Parallelizing functional programs by generalization. In *Algebraic and Logic Programming. ALP'97*, Lecture Notes in Computer Science. Springer-Verlag, 1997.

[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*, 6(4):657–665, 1996.

[GLJ93] A. Gill, J. Launchbury, and S. Peyton Jones. A short cut to deforestation. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 223–232, Copenhagen, June 1993.

[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.

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

[HIT96] 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.

[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.

[HITT97] Z. Hu, H. Iwasaki, M. Takeichi, and A. Takano. Tupling calculation eliminates multiple data traversals. In *ACM SIGPLAN International Conference on Functional Programming*, pages 164–175, Amsterdam, The Netherlands, June 1997. ACM Press.

[HPJWe92] Paul Hudak, Simon L. Peyton Jones, and Philip Wadler (editors). Report on the programming language Haskell, a non-strict purely functional language (version 1.2). *SIGPLAN Notices*, Mar, 1992.

[Jeu93] J. Jeuring. *Theories for Algorithm Calculation.* Ph.D thesis, Faculty of Science, Utrecht University, 1993.

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

[JS90] Geraint Jones and Mary Sheeran. Circuit design in Ruby. In Staunstrup, editor, *Formal Methods for VLSI Design*, Amsterdam, 1990. Elsevier Science Publications.

[Len96] C. Lengauer. Automatic parallelization and high performance compiler. In *Annual European Conference on Parallel Processing, LNCS 1123*, pages 377–378, LIP, ENS Lyon, France, August 1996. Springer-Verlag.

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

[Mal90] G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, (14):255–279, August 1990.

[Mee92] L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.

[O'D94] J. O'Donnell. A correctness proof of parallel scan. *Parallel Processing Letters*, 4(3):329–338, 1994.

[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.

[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.

[Ski94a] David B. Skillicorn. *Foundations of Parallel Programming.* Cambridge University Press, 1994.

[Ski94b] D.B. Skillicorn. The categorical data type approach to general-purpose parallel computation. In B. Pehrson and I. Simon, editors, *Workshop on General-Purpose Parallel Computing, 13th IFIP World Congress*, volume IFIP Transactions A-51, pages 565–570. North-Holland, September 1994.

[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.