An Accumulative Parallel Skeleton for All

Zhenjiang HU[1,3], Hideya IWASAKI[2], Masato TAKEICHI[1]

[1] The University of Tokyo
{hu,takeichi}@ipl.t.u-tokyo.ac.jp
[2] The University of Electro-Communications
iwasaki@cs.uec.ac.jp
[3] PRESTO 21, Japan Science and Technology Corporation

Abstract. Parallel skeletons intend to encourage programmers to build a parallel program from ready-made components for which efficient implementations are known to exist, making the parallelization process simpler. However, it is neither easy to develop *efficient* parallel programs using skeletons nor to use skeletons to manipulate *irregular* data, and moreover there lacks a *systematic* way to optimize skeletal parallel programs. To remedy this situation, we propose a *novel* parallel skeleton, called accumulate, which not only efficiently describes data dependency in computation but also exhibits nice algebraic properties for manipulation. We show that this skeleton significantly eases skeletal parallel programming in practice, efficiently manipulating both regular and irregular data, and systematically optimizing skeletal parallel programs.

1 Introduction

With the increasing popularity of parallel programming environments such as PC cluster, more and more people, including those who have little knowledge of parallel architecture and parallel programming, are hoping to write parallel programs. This situation eagerly calls for models and methodologies which can assist programming parallel computers effectively and correctly.

The *data parallel* model [HS86] turns out to be one of the most successful ones for programming massively parallel computers. To support parallel programming, this model basically consists of two parts:

- *a parallel data structure* to model a uniform collection of data which can be organized in a way that each element can be manipulated in parallel; and
- *a fixed set of parallel skeletons* on the parallel data structure to abstract parallel structures of interest, which can be used as building blocks to write parallel programs. Typically, these skeletons include element-wise arithmetic and logic operations, reductions, prescans, and data broadcasting.

This model not only provides programmers an easily understandable view of a *single execution stream* of a parallel program, but also makes the parallelizing process easier because of explicit parallelism of the skeletons.

Despite these promising features, the application of current data parallel programming suffers from several problems which prevent it from being practically used. Firstly, because parallel programming relies on a set of parallel primitive skeletons for specifying parallelism, programmers often find it hard to choose proper ones or to integrate them well in order to develop *efficient* parallel programs. Secondly, the skeletal parallel programs are difficult to be optimized. The major difficulty lies in the construction of rules meeting the *skeleton-closed* requirement for transformation among skeletons [Bir87,SG99]. Thirdly, skeletons are assumed to manipulate regular data structure. Unfortunately, for *irregular* data like nested lists where the sizes of inner lists are remarkably different, the parallel semantics of skeletons would lead to load imbalance which may nullify the effect of parallelism in skeletons. For more detailed discussion of these problems, see Section 3.

To remedy this situation, we propose in this paper a *new* parallel skeleton, which can significantly eases skeletal parallel programming, efficiently manipulating both regular and irregular data, and systematically optimizing skeletal parallel programs. Our contributions, which make skeletal programming more practical, can be summarized as follows.

– We define a *novel* parallel skeleton (Section 4), called accumulate, which can not only efficiently describe data dependency in a computation through an *accumulating* parameter, but also exhibits nice algebraic properties for manipulation. It can be considered as a higher order list homomorphism, which abstracts a computation requiring more than one pass and provides a better recursive interface for parallel programming.
– We give a single but general rule (Theorem 4 in Section 5), based on which we construct a framework for systematically optimizing skeletal parallel programs. Inspired by the success of the shortcut deforestation [GLJ93] for optimizing sequential programs in compilers, we give a specific shortcut law for fusing compositional style of skeletal parallel programs, but paying much more attention to guaranteeing skeleton-closed parallelism. Our approach using a single rule is in sharp contrast to the existing ones [SG99,KC99] based on a huge set of transformation rules developed in a rather ad-hoc way.
– We propose a flattening rule (Theorem 5 in Section 6), enabling accumulate to deal with both regular and irregular nested data structures efficiently. Compared to the work by Blelloch [Ble89,Ble92] where the so-called *segmented scan* is proposed to deal with irregular data, our rule is more general and powerful, and can be used to systematically handle a wider class of skeletal parallel programs.

The organization of this paper is as follows. After briefly reviewing the notational conventions and some basic concepts in the BMF parallel model in Section 2, we illustrate with a concrete example the problems in skeletal parallel programming in Section 3. To resolve these problems, we begin by proposing a new general parallel skeleton called accumulate, and show how one can easily program with this new skeleton in Section 4. Then in Section 5, we develop a general rule for optimization of skeletal parallel programs. Finally, we give a powerful theorem showing that accumulate can be used to efficiently manipulate irregular data in Section 6. Related work and discussions are given in Section 7.

2 BMF and Parallel Computation

We will address our method on the BMF data parallel programming model [Bir87,Ski90], though the method itself is not limited to the BMF model. We choose BMF because it can provide us a concise way to describe both programs and transformation of programs. Those who are familiar with the functional language Haskell [JH99] should have no problem in understanding the programs in this paper. From the notational viewpoint, the main difference is that we use more symbols or special parentheses to shorten the expressions so that manipulation of expressions can be performed in a more concise way.

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 $(f\,a) \oplus b$, 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 or binary functions by $a \oplus b = (a\,\oplus)\,b = (\oplus\,b)\,a = (\oplus)\ a\ b$.

Parallel Data Structure: Join Lists. *Join lists* (or *append lists*) are finite sequences of values of the same type. A list is either the empty, a singleton, or the concatenation of two other lists. We write $[\,]$ for the empty list, $[a]$ for the singleton list with element a (and $[\cdot]$ for the function taking a to $[a]$), and $x +\!\!+\, y$ for the concatenation (join) of two lists x and y. Concatenation is associative, and $[\,]$ is its unit. For example, $[1] +\!\!+ [2] +\!\!+ [3]$ denotes a list with three elements, often abbreviated to $[1, 2, 3]$. We also write $a : x$ for $[a] +\!\!+\, x$. If a list is constructed on by the constructor of $[\,]$ and $:$, we call it *cons list*.

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

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

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

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

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

Scan is the skeleton that accumulates all intermediate results for computation of reduce. Informally, for an associative binary operator $\oplus$ and an initial value e, we have

$$\oplus\!\!\not{\!\!/}\!\!\not{\!\!/}_e\ [x_1, x_2, \ldots, x_n] \;=\; [e, e \oplus x_1, e \oplus x_1 \oplus x_2, \ldots, e \oplus x_1 \oplus x_2 \oplus \cdots \oplus x_n].$$

Note that this definition is a little different from that in [Bir87]; the e there is assumed to be the unit of $\oplus$. In fact efficient implementation of the scan skeleton does not need this restriction.

Zip is the skeleton that merges two lists into a single one by paring the corresponding elements. The resulting list has the same length as that of shorter one. Informally, we have

$$[x_1, x_2, \ldots, x_n]\, \Upsilon\, [y_1, y_2, \ldots, y_n] = [(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)].$$

It has been shown that these four operators have nice massively parallel implementations on many architectures [Ski90,Ble89]. If k and $\oplus$ need $\mathrm{O}(1)$ parallel time, then $k*$ can be implemented using $\mathrm{O}(1)$ parallel time, and both $\oplus/$ and $\oplus\!\!\not{\!\!/}\!\!\not{\!\!/}_e$ can be implemented using $\mathrm{O}(\log N)$ parallel time, where N denotes the size of the list. For example, $\oplus$ can be computed in parallel on a tree-like structure with the combining operator $\oplus$ applied in the nodes, while $k*$ is computed in parallel with k applied to each of the leaves. The study on efficient parallel implementation of $\oplus\!\!\not{\!\!/}\!\!\not{\!\!/}_e$ can be found in [Ble89], though the implementation may be difficult to understand.

3 Limitations of the Existing Skeletal Parallel Programming

In this section, we are using a simple but practical example, the *lines-of-sight problem* (*los* for short), to explain in detail the limitations (problems) of the existing approach to parallel programming using skeletons, clarifying the motivation and the goal of this work.

Given a terrain map in the form of a grid of altitudes, an observation point, and a set of rays, the lines-of-sight problem is to find which points are visible along these rays originating at the observation point (as in Figure 1). A point on a ray is visible if and only if no other point between it and the observation point has a greater vertical angle. More precisely, $los\ :\ Point \to [[Point]] \to [[Bool]]$ accepts as input an observation point p_0 and a list of rays where each ray is a list of points, and returns a list of lists where corresponding element is a boolean value showing whether the point is visible or not.

This problem is of practical interest, and a simpler version (considering only a single line) was informally studied in [Ble89] where an efficient parallel program was given without explanation how it was obtained. Now the question is how to make use of the four BMF skeletons in Section 2 to develop an efficient parallel program for this problem.

Fig. 1. Altitude Map

3.1 Programming Efficient Skeletal Programs

Developing *efficient* programs with skeletons is hard because it requires both a proper choice of skeletons and an efficient combination of them. Using skeletons, one often tries to solve a problem by composition of several passes so that each pass can be described in terms of a parallel skeleton. Considering the subproblem $los1$ which just checks whether the points in a *single* ray ps are visible or not from the observation point p_0, one may solve the problem by the following three passes.

1. Compute the vertical angles for each point.
2. For each point, compute the maximum angle among all the points between this point and the observation point, which can again be solved by two passes:
   (a) gathering all angles of the points between this point and the observation point;
   (b) computing the maximum of the angles.
3. For each point, compare its angle with the maximum angle obtained in step 2.

Therefore one could come up with the following program:

$$
\begin{array}{lll}
los1\ p_0\ ps = \textbf{let} & & \\
\quad\quad as &= (angle\ p_0) * ps & \text{— Pass 1} \\
\quad\quad ass &= (+\!\!+)\,\#\!\!\#_{[]}\,([\cdot] * as) & \text{— Pass 2 (a)} \\
\quad\quad mas &= maximum * ass & \text{— Pass 2 (b)} \\
\quad\quad vs &= (\lambda\,(x, y) \rightarrow x > y) * (as\ \Upsilon\ mas) & \text{— Pass 3} \\
\quad \textbf{in} & & \\
\quad vs & &
\end{array}
$$

However, this multi-pass program has the problem of introducing many intermediate data structures (such as as, ass, and mas) passed between skeletons. This may result in a terribly inefficient programs (the above definition $los1$ is such an example) with high computation and communication cost especially in a distributed system [KC99]; these intermediate data structures have to be distributed to processors and each result must be gathered to the master processor.

As a matter of fact, even for a simpler subproblem like Pass 2 (a), solving it in terms of skeleton is actually not an easy task. One approach for coping with this problem is to derive a *homomorphism* [Col95,Gor96,HTC98], resulting in skeletal parallel programs in the form of $(\oplus\,/)\circ(f*)$. Though homomorphisms may deal well with programming using *map* and *reduce* skeletons, they cannot directly support programming with the skeleton of *scan* [Ble89] which uses an accumulating parameter.

3.2 Skeleton Composition

As argued, most inefficiency specific to skeletal parallel programs originates from many intermediate data structures passed from one skeleton to another. Therefore, it is essential to fuse compositions of skeletons to eliminate unnecessary intermediate data structures, to save both computation and communication cost.

The major difficulty for this fusion lies in the construction of rules meeting the *skeleton-closed* requirement that the result of fusion of skeletons should give a skeleton again [Bir87,SG99]. Recall the program for Pass 2 (a):

$$ass \;=\; +\!\!+ \,\#_{[]}([\cdot] * as).$$

It cannot be fused into a program using a single skeleton of *map*, *reduce*, or *scan*. One may hope to fuse it into a form something like $(\underline{\lambda\, e\, \lambda\, a\ \rightarrow\ e\ +\!\!+ [a]})\#_{[]}$, but this is incorrect because the underlined binary operator is not associative. The key problem for optimization turns out to be how to systematically fuse skeletal parallel programs.

3.3 Nested Parallelism

Consider the program of Pass 2 (b) in the definition of $los1$:

$$mas \;=\; maximum * ass.$$

Let $ass = [as_1, as_2, \ldots, as_n]$. This expression is quite inefficient when the lengths of as_1, as_2, ..., as_n are quite different. To see this more clearly, consider an extreme case of

$$maximum * [[1],[2],[1,2,3,4,\ldots,100]],$$

and assume that we have three processors. If we naively use one processor to compute *maximum* on each element list, computation time will be dominated by the processor which computes $maximum\ [1,2,\ldots,100]$, and the load imbalance cancels the effect of parallelism in the *map* skeleton.

Generally, the nested parallelism problem can be formalized as: "under what condition of f, the function $f*$ can be implemented efficiently no matter how different the sizes of the element lists are?" Blelloch [Ble89,Ble92] gave a *case study*, showing that if f is $\oplus\not\!\!\#_e$, then $f*$ is called *segmented scan* and can be implemented efficiently. But how to systematically cope with skeletal parallel programs remains unclear.

A concrete but more involved example is the lines-of-sight problem, which can be solved by

$$los\ p_0\ pss \ = \ (los1\ p_0) * pss$$

where pss may be an imbalanced (irregular) data, and $los1\ p_0$ is a rather complicated function.

4 An Accumulative Parallel Skeleton

From this section, we propose a new general parallel skeleton to resolve the problems raised in Section 3, showing how one can easily program with this new skeleton, how skeletal parallel programs can be systematically optimized, and how nested parallelism can be effectively dealt with.

4.1 The Skeleton accumulate

We believe that the skeleton itself should be able to describe data dependency naturally. Map and zip describe parallel computation without data dependency, reduce describes parallel computation with an *upward* (bottom-up) data dependency, and scan describes parallel computation with an *upward* and a simple *downward* (top-down) data accumulation. Compared with these skeletons, our new accumulative skeleton can describe both upward and downward data dependency in a more general way.

Definition 1 (accumulate)**.** Let g, p, q be functions, and $\oplus$ and $\otimes$ be associative operators. The skeleton accumulate is defined by

$$\begin{aligned} \mathsf{accumulate}\ [\,]\ e &= g\ e \\ \mathsf{accumulate}\ (a : x)\ e &= p(a, e)\ \oplus\ \mathsf{accumulate}\ x\ (e \otimes q\ a). \end{aligned}$$

We write $[\![g, (p, \oplus), (q, \otimes)]\!]$ for the function accumulate. □

It is worth noting that this skeleton, looking a bit complicated, is the simplest form combining two basic recursive forms of *foldl* and *foldr* [Gib96]; $(p, \oplus)$ corresponds to *foldr*, $(q, \otimes)$ corresponds to *foldl*, and g is to connect these two parts.

As a quick example of this skeleton, $los1$ in Section 3 can be defined as

$$\begin{aligned} los1\ p_0 = [\![& \lambda m \to [\,], \\ & (\lambda(p, maxAngle) \to \mathbf{if}\ a > maxAngle\ \mathbf{then}\ [\mathsf{T}]\ \mathbf{else}\ [\mathsf{F}], +\!\!\!+\,), \\ & (id, max)]\!]. \end{aligned}$$

More examples can be found in Section 4.4.

4.2 Parallelizability of accumulate

To see that accumulate is indeed a parallel skeleton, we will show that it can be implemented efficiently in parallel. As a matter of fact, the recursive definition for accumulate belongs to the class of parallelizable recursions defined in [HTC98]. The following theorem gives the resulting parallel version for accumulate.

Theorem 1 (Parallelization). The function accumulate defined in Definition 1 can be parallelized to the following divide-and-conquer program.

$$
\begin{array}{lll}
\mathsf{accumulate}\ []\ e & = & g\ e \\
\mathsf{accumulate}\ x\ e & = & \mathit{fst}\ (\mathsf{accumulate}'\ x\ e) \\
\mathsf{accumulate}'\ [a]\ e & = & (p\ (a,e) \oplus g\ (e \otimes q\ a), p\ (a,e), q\ a) \\
\mathsf{accumulate}'\ (x +\!\!\!+\, y)\ e & = & \mathbf{let}\ (r_1, s_1, t_1) = \mathsf{accumulate}'\ x\ e \\
 & & \qquad (r_2, s_2, t_2) = \mathsf{accumulate}'\ y\ (e \otimes t_1) \\
 & & \mathbf{in}\ (s_1 \oplus r_2,\ s_1 \oplus s_2,\ t_1 \otimes t_2)
\end{array}
$$

Proof. Apply the parallelization theorem in [HTC98] followed by the tupling calculation [HITT97]. □

It is worth noting that $\mathsf{accumulate}'$ can be implemented in parallel on a multiple processor system supporting bidirectional tree-like communication, using the time of $O(\log N)$ where N denotes the length of the input list, provided that $\oplus$, $\otimes$, p, q and g can be computed in a constant time. Two passes are employed; an upward pass in the computation can be used to compute the third component of $\mathsf{accumulate}'\ x\ e$ before a downward pass is used to compute the first two values of the triple.

4.3 An Abstraction of Multi-Pass Computation

Let us see why it is necessary to have this special skeleton accumulate rather than implementing it using existing skeletons, although it can be described in terms of the existing skeletons according to the diffusion theorem [HTI99].

Theorem 2 (Diffusion). The skeleton accumulate can be diffused into the following composition of skeletal functions.

$$
\begin{array}{rl}
\mathsf{accumulate}\ x\ e = & \mathbf{let}\ y +\!\!\!+\, [e'] = \otimes\#_e(q * x)) \\
 & \qquad z = x\, \Upsilon\, y \\
 & \mathbf{in}\ (\oplus/\ (p * z)) \oplus (g\ e')
\end{array}
$$

□

The resulting skeletal program after diffusion cannot be efficiently implemented just according to the parallel semantics of each skeleton. To see this, consider the simplest composition of two skeleton of $\oplus/(p * z)$ when computed on a distributed parallel machine. It would be performed by the following two passes.

1. Compute $p * z$ by distributing data z among processors, performing $p*$ in a parallel way, and collecting data from processors to form data w.

2. Compute $\oplus/w$ by distributing data w among processors, performing $\oplus/$ in a parallel way, and collecting data from processors to form the result.

The underlined two parts are obviously not necessary, but this multiple-pass computation would not be avoidable unless $(\oplus/) \circ (p*)$ can be fused into a single existing skeleton[1]. To resolve this problem, one must introduce a new skeleton to capture (abstract) this kind of multiple-pass program. Our skeleton accumulate is exactly designed for this abstraction, while it can be efficiently implemented without actual intermediate data distribution and collection.

4.4 Parallel Programming with accumulate

The skeleton accumulate uses an accumulating parameter which can be used to describe complicated dependency of computation in a natural way. In contrast, the existing skeletal parallel programming requires that all dependency must be explicitly specified by using intermediate data.

We have shown in Section 4.1 that computation function $los1$ can be easily described by accumulate. In fact, the new skeleton is so powerful and general that it can be used to describe the skeletons without sacrificing the performance in order, as summarized in the following theorem.

Theorem 3 (Skeletons in accumulate).

$$\begin{aligned} f * x &= [\![\lambda\_ \rightarrow [\,], (\lambda(a,\_) \rightarrow [f\ a], +\!\!+), (\_,\_)]\!]\ x\ \_ \\ \oplus/\ x &= [\![\lambda\_ \rightarrow \iota_{\oplus}, (\lambda(a,\_) \rightarrow a, \oplus), (\_,\_)]\!]\ x\ \_ \\ \oplus\#_e &= [\![[\cdot], (\lambda(a,e) \rightarrow [e], +\!\!+), (id, \oplus)]\!] \end{aligned} \qquad \square$$

Note that in the above theorem, $\_$ is used to represent a "don't care" value with consistent type.

Powerful and general, accumulate can be used to solve many problems in a rather straightforward way, that is *not* more difficult than solving the problems in sequential setting [HIT01]. Consider a simple example computing a polynomial value, a case study in [SG99] and an exercise in [Ble90]:

$$poly\ [a_1, a_2, \ldots, a_N]\ x \ = \ a_1 \times x + a_1 \times x^2 + \cdots + a_N \times x^N.$$

It can be easily defined by the following recursive definition with an accumulating parameter storing x^i.

$$\begin{aligned} &poly\ as\ x = poly'\ as\ 1 \\ &\qquad \textbf{where}\ poly'\ [\,]\ e = 0 \\ &\qquad\qquad poly'\ (a : as)\ e = a \times e + poly'\ as\ (e \times x) \end{aligned}$$

That is,

$$poly\ as\ x = [\![\lambda e \rightarrow 0, (\lambda(a,e) \rightarrow a \times e, +), (const\ x, \times)]\!]\ as\ 1.$$

It follows from the Parallelization Theorem that we have obtained an $O(\log N)$ parallel time program for evaluating a polynomial.

[1] Note that we cannot fuse it into $\otimes/$ where $a \otimes r = p\ a \oplus r$, because $\otimes$ may not be associative.

5 Optimizing Skeletal Parallel Programs

To fuse several skeletons into one for eliminating unnecessary intermediate data structures passed between skeletons, one would try to develop rules for performing algebraic transformations on skeletal parallel program like [SG99]. For instance, here is a possible algebraic transformation which eliminates an intermediate list:

$$f * (g * x) = (f \circ g) * x.$$

Unfortunately, one would need a huge set of rules to account for all possible combinations of skeletal functions. In this paper, we borrow the idea of shortcut deforestation [GLJ93] for optimization of sequential programs, and reduce the entire set to a *single* rule, by standardizing the way in which *join* lists are consumed by accumulate and standardizing the way in which they are produced.

5.1 The Fusion Rule

First, we explain the shortcut deforestation theorem, known as foldr-build rule [GLJ93].

Lemma 1 (foldr-build Rule). If we have $gen : \forall\beta.\,(A \to \beta \to \beta) \to \beta \to \beta$ for some fixed A, then

$$\mathsf{foldr}\ (\oplus)\ e\ (\mathsf{build}\ gen) = gen\ (\oplus)\ e,$$

where foldr and build are defined by

$$\begin{array}{ll} \mathsf{foldr}\ (\oplus)\ e\ [] & = e \\ \mathsf{foldr}\ (\oplus)\ e\ (a : x) & = a \oplus \mathsf{foldr}\ (\oplus)\ e\ x \\ \mathsf{build}\ gen & = gen\ (:)\ []. \end{array}$$

□

Noticing that accumulate can be described in terms of foldr as

$$\mathsf{accumulate}\ =\ \mathsf{foldr}\ (\lambda a \lambda r\ \to (\lambda e \to p(a,e) \oplus r(e \otimes q\ a)))\ g,$$

we obtain a rule for fusion of accumulate from Lemma 1:

$$[\![g, (p, \oplus), (q, \otimes)]\!]\ (\mathsf{build}\ gen) = gen\ (\lambda a \lambda r\ \to (\lambda e \to p(a,e) \oplus r(e \otimes q\ a)))\ g.$$

However, this rule has a practical problem for being used to fuse skeletal parallel programs. The reason is that skeletal functions produce *join* lists rather than cons lists, due to the requirement of associativity in their definitions. For example, for the definition of $f*$:

$$f * x = [\![\lambda\_ \to [], (\lambda(a,\_) \to [f\ a], +\!\!+), (\_,\_)]\!]\ x\ \_$$

it would be more natural to consider $f*$ as a production of a join list using the constructors of $[]$, $[\cdot]$, and $+\!\!+$. To resolve this problem, we standardize the production of join lists by defining a new function buildJ as

$$\mathsf{buildJ}\ gen = gen\ (+\!\!+)\ [\cdot]\ [\,],$$

and accordingly standardize the list consumption by transforming $[\![g, (p, \oplus), (q, \otimes)]\!]$ based on the parallelization theorem to

$$
\begin{aligned}
[\![g, (p, \oplus), (q, \otimes)]\!]\ x &= \mathit{fst} \circ \mathsf{accumulate}' \\
\mathsf{accumulate}'\ [\,] &= \lambda e \to (g\ e, \_, \_) \\
\mathsf{accumulate}'\ [a] &= \lambda e \to (p\ (a, e) \oplus g\ (e \otimes q\ a), p\ (a, e), q\ a) \\
\mathsf{accumulate}'\ (x +\!\!+\, y) &= \mathsf{accumulate}'\ x \odot_{\oplus,\otimes} \mathsf{accumulate}'\ y
\end{aligned}
$$

where $\odot_{\oplus,\otimes}$ is defined by

$$
\begin{aligned}
(u \odot_{\oplus,\otimes} v)\ e = \mathbf{let}\ & (r_1, s_1, t_1) = u\ e \\
& (r_2, s_2, t_2) = v\ (e \otimes t_1) \\
\mathbf{in}\ & (s_1 \oplus r_2,\ s_1 \oplus s_2,\ t_1 \otimes t_2).
\end{aligned}
$$

Therefore, we obtain the following general and practical fusion theorem for accumulate.

Theorem 4 (Fusion (Join Lists)). If for some fixed A we have $gen : \forall \beta. (\beta \to \beta \to \beta) \to (A \to \beta) \to \beta \to \beta$ then

$$
\begin{aligned}
& [\![g, (p, \oplus), (q, \otimes)]\!]\ (\mathsf{buildJ}\ gen)\ e \\
& \quad = \mathit{fst}\ (gen\ (\odot_{\oplus,\otimes})\ (\lambda a \to (\lambda e \to (p\ (a, e) \oplus g\ (e \otimes q\ a), p\ (a, e), q\ a))) \\
& \qquad\qquad (\lambda e \to (g\ e, \_, \_))\ e) \qquad \square
\end{aligned}
$$

For the skeletons in the form of $[\![\lambda\_ \to e, (\lambda(a, \_) \to p'\ a, \oplus), (\_, \_)]\!]$, like *map* and *reduce*, which do not need a accumulating parameter, we can specialize Theorem 4 to the following corollary for fusion with these skeletons.

Corollary 1. If for some fixed A we have $gen : \forall \beta. (A \to \beta \to \beta) \to (A \to \beta) \to \beta \to \beta$ and then

$$
\begin{aligned}
& [\![\lambda\_ \to d, (\lambda(a, \_) \to p' a, \oplus), (\_, \_)]\!]\ (\mathsf{buildJ}\ gen)\ \_ \\
& \quad = gen\ (\oplus)\ (\lambda a \to p'\ a \oplus d)\ d \qquad \square
\end{aligned}
$$

5.2 Warm Fusion

To apply Theorem 4 for fusion, we must standardize those skeletal parallel programs to be fused in terms of accumulate for consuming join lists and of buildJ for producing join lists. We may deal with this by the following two methods:

- *Standardizing library functions by hand.* We can standardize frequently used functions by hand. For example, the followings define *map* and *scan* in the required form.

  $$
  \begin{aligned}
  f * x &= \mathsf{buildJ}\ (\lambda c \lambda s \lambda n \to [\![\lambda\_ \to n, (\lambda(a, \_) \to s\ (f\ a), c), (\_, \_)]\!]\ x\ \_) \\
  \oplus\!\!\#_e x &= \mathsf{buildJ}\ (\lambda c \lambda s \lambda n \to [\![s, (\lambda(a, e) \to s\ e, c), (id, \oplus)]\!]\ x\ e)
  \end{aligned}
  $$

  Following this idea, one may redefine their library functions such as *maximum* in this form, as done in sequential programming like [GLJ93] which rewrites most prelude functions of Haskell in the form of foldr-build form. This approach is rather practical [GLJ93], but needs preprocessing.

– *Standardizing* accumulate *automatically.* For a user-defined function in terms of accumulate, we may build a type inference system to automatically abstract the data constructors of join lists appearing in the program to derive its buildJ form. Many studies have been devoted to this approach on the context of sequential programming [LS95,Chi99], which may be adapted to our use.

Consider the function *alleven* defined by $alleven\ x = \wedge / (even * x)$, which judges whether all the elements of a list are even. We can perform fusion systematically (automatically) as follows. Note that this program cannot be fused in the existing framework.

$$
\begin{array}{ll}
 & \wedge / (even * x) \\
= & \quad \{ \text{ def. of } reduce \text{ and } map \ \} \\
 & [\![\lambda\_ \to \mathsf{T}, (\lambda(a,\_) \to a, \wedge), (\_,\_)]\!] \\
 & \quad (\mathsf{buildJ}\ (\lambda c \lambda s \lambda n \to [\![\lambda\_ \to n, (\lambda(a,\_) \to s\ (even\ a), c), (\_,\_)]\!]\ x\ \_))\ \_ \\
= & \quad \{ \text{ Corollary 1 } \} \\
 & [\![\lambda\_ \to \mathsf{T}, (\lambda(a,\_) \to even\ a \wedge \mathsf{T}, \wedge), (\_,\_)]\!]\ x\ \_ \\
= & \quad \{ \text{ simplification } \} \\
 & [\![\lambda\_ \to \mathsf{T}, (\lambda(a,\_) \to even\ a, \wedge), (\_,\_)]\!]\ x\ \_
\end{array}
$$

Theorem 4 can be applied to a wider class of skeletal parallel programs, including the useful program patterns such as (1) $(f_1*) \circ (f_2*) \circ \cdots \circ (f_n*)$, (2) $(\oplus /) \circ (f*)$, (3) $(\oplus /) \circ (\otimes \#_e)$, (4) $(f*) \circ (\oplus \#_e) \circ (g*)$, (5) $(\oplus_1 \#_{e_1}) \circ (\oplus_2 \#_{e_2})$. Recall the lines-of-sight problem in Section 3 where we obtain the following compositional program for Pass 2 (b) after expansion of ass and as:

$$mas = (maximum/) * ((+\!\!+) \#_{[]} ([\cdot] * (angle * ps))).$$

We can fuse it into a single one [HIT01].

6 Dealing with Nested Skeletons

Our new skeleton accumulate can deal with nested data structures very well, especially *irregular* ones whose elements may have quite different sizes. To be concrete, given a list of lists, we are considering efficient implementation of a computation which maps some function f in terms of accumulate to every sublist.

In order to process a given nested (maybe irregular) list[2] efficiently, we first use $flatten : [[a]] \to [(Bool,\ a)]$ to transform the nested list into a flat list of pairs [Ble92]. Each element in this flat list is a pair of *flag*, a boolean value, and an element of inner list of the original nested list. If the element is the first of an inner list, *flag* is T, otherwise *flag* is F. For example, a nested list

$$[[x_1, x_2, x_3], [x_4, x_5], [x_6], [x_7, x_8]]$$

[2] For simplicity, we assume that each element list is nonempty. Actually, it not difficult to introduce an additional tag to deal with empty list.

is flattened into the list

$$[(\mathsf{T}, x_1), (\mathsf{F}, x_2), (\mathsf{F}, x_3), (\mathsf{T}, x_4), (\mathsf{F}, x_5), (\mathsf{T}, x_6), (\mathsf{T}, x_7), (\mathsf{F}, x_8)].$$

Using the flattened representation, each processor can be assigned almost the same number of data elements, and therefore, reasonable load balancing between processors can be achieved. For the above example, if there are four processors, they are assigned to $[(\mathsf{T}, x_1), (\mathsf{F}, x_2)]$, $[(\mathsf{F}, x_3), (\mathsf{T}, x_4)]$, $[(\mathsf{F}, x_5), (\mathsf{T}, x_6)]$, and $[(\mathsf{T}, x_7), (\mathsf{F}, x_8)]$, respectively. Note that elements of the same inner list may be divided and assigned to more than one processor.

Our theorem concerning nested lists states that mapping the function accumulate to every sublist can be turned into a form applying another accumulate to the flattened representation of the given nested list.

Theorem 5 (Flattening). If xs is a nested list which is not empty and does not include empty list, then

$$\begin{array}{l}
(\lambda\, x \;\to\; [\![g, (p, \oplus), (q, \otimes)]\!]\; x\; e_0) \;*\; xs \\
\quad = \; v \text{ where } (_, v, _) \;=\; [\![g', (p', \oplus'), (q', \otimes')]\!]\; (\mathit{flatten}\; xs)\; (\mathsf{T}, e_0) \\
\quad \textbf{where} \\
\qquad \begin{array}{lcl}
g'\; (z, a) & = & (\mathsf{T},\; [\,],\; a) \\
p'\; ((\mathsf{T}, x),\; (z, a)) & = & (\mathsf{T},\; [p\; (x, e_0)],\; a) \\
p'\; ((\mathsf{F}, x),\; (z, a)) & = & (\mathsf{F},\; [p\; (x, a)],\; a) \\
q'\; (\mathsf{T}, x) & = & (\mathsf{T},\; e_0 \otimes q\; x) \\
q'\; (\mathsf{F}, x) & = & (\mathsf{F},\; q\; x) \\
(z,\; vs \;+\!\!+\; [v],\; a) \;\oplus'\; (\mathsf{T},\; ws,\; b) & = & (z,\; vs +\!\!+ [v \oplus g\; b] +\!\!+ ws,\; a) \\
(z,\; vs +\!\!+ [v],\; a) \oplus' (\mathsf{F},\; [w] +\!\!+ ws,\; b) & = & (z,\; vs +\!\!+ [v \;\oplus\; w] +\!\!+ ws,\; a) \\
(z, a) \;\otimes'\; (\mathsf{T}, b) & = & (\mathsf{T}, b) \\
(z, a) \;\otimes'\; (\mathsf{F}, b) & = & (z,\; a \;\otimes\; b)
\end{array}
\end{array}$$

□

Due to space limination, we cannot give a proof of the theorem. The key point of this theorem is that the transformed program is just a simple application of accumulate to the flattened list. It follows from the implementation of accumulate that we obtain an efficient implementation for the map of accumulate.

7 Related Work and Discussions

Our design of accumulate for parallel programming is related to the Third Homomorphism Theorem [Gib96], which says that if a problem can be solved in terms of both *foldl* (top down) and *foldr* (bottom up), then it can be solved in terms of a list homomorphism which can be implemented in parallel in a divide-and-conquer way. However, it remains open how to construct such list homomorphism from two solutions in terms of *foldl* and *foldr*. Rather than finding a way for this construction, we provide accumulate for parallel programming, and it can be regarded as an *integration* of both *foldl* and *foldr*.

Optimizing skeletal parallel programs is a challenge, and there have been several studies. A set of optimization rules, together with performance estimation, have been proposed in [SG99], which are used to guide fusion of several skeletons into one. This, however, would need a huge set of rules to account for all possible combinations of skeletal functions. In contrast, we reduce this set to a *single* rule (Fusion Theorem), by standardizing both the way in which *join* lists are consumed by accumulate and the way in which they are produced. This idea is related to the shortcut deforestation [GLJ93,LS95,Chi99] which has proved to be practically useful for optimization of sequential programs. Another approach is to refine the library functions to reveal their internal structure for optimization in a compiler [KC99]. We deal with this problem in programming instead of compiler reconstruction.

As for nested parallelism or the form $f*$, our work is related to that by Blelloch [Ble89,Ble92] who showed that if f is $\oplus\#_e$, than we know that $(\oplus \#_e)*$ can be implemented efficiently. Here we treat more complicated f including *scan* as its special case, showing that $(\lambda x.[\![g,(p,\oplus),(q,\otimes)]\!]\ x\ e_0)*$ can be efficiently implemented.

This work is a continuation of our effort to apply the so-called program calculation technique to the development of efficient parallel programs [HTC98]. As a matter of fact, our new skeleton accumulate comes out of the recursive pattern which is parallelizable in [HTC98,HTI99]. Based on these results, this paper made a significant progress towards practical use of skeletons for parallel programming, showing how to program with accumulate, how to systematically optimize skeletal programs, and how to deal with irregular data.

Although we have implemented a small prototype system basically for testing the idea in this paper, we believe that it should be more important to see how efficient the idea in a real parallelizing compiler. In addition, we have not taken account of those skeletons for data communication.

Acknowledgments

We thank the anonymous referees for their suggestions and comments.

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.

[Ble90] G. E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, Carnegie-Mellon Univ., 1990.

[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.

[Chi99] O. Chitil. Type inference builds short cut to deforestation. In *Proceedings of 1999 ACM SIGPLAN International Conference on Functional Programming*, pages 249–260. ACM Press, 1999.

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

[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.

[Gor96] 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.

[GWL99] Sergei Gorlatch, Christoph Wedler, and Christian Lengauer. Optimization rules for programming with collective operations. In Mikhail Atallah, editor, *IPPS/SPDP'99. 13th Int. Parallel Processing Symp. & 10th Symp. on Parallel and Distributed Processing*, pages 492–499, 1999.

[HIT01] Z. Hu, H. Iwasaki, and M. Takeichi. An accumulative parallel skeleton for all. Technical Report METR 01–05, University of Tokyo, September 2001. Available from `http://www.ipl.t.u-tokyo.ac.jp/ ~hu/pub/metr01-05.ps.gz`.

[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.

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

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

[HTI99] Z. Hu, M. Takeichi, and H. Iwasaki. Diffusion: Calculating efficient parallel programs. In *1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 85–94, San Antonio, Texas, January 1999. BRICS Notes Series NS-99-1.

[JH99] S. Peyton Jones and J. Hughes, editors. *Haskell 98: A Non-strict, Purely Functional Language*. Available online: `http://www.haskell.org`, February 1999.

[KC99] G. Keller and M. M. T. Chakravarty. On the distributed implementation of aggregate data structures by program transformation. In J. Rolim et al., editor, *4th International Workshop on High-Level Parallel Programming Models and Supportive Environments* (LNCS 1586), pages 108–122, Berlin, Germany, 1999. Springer-Verlag.

[LS95] J. Launchbury and T. Sheard. Warm fusion: Deriving build-catas from recursive definitions. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 314–323, La Jolla, California, June 1995.

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