MATHEMATICAL ENGINEERING
TECHNICAL REPORTS

An Automatic Fusion Mechanism for Variable-Length List Skeletons in SkeTo

Kento EMOTO and Kiminori MATSUZAKI

METR 2013-04 February 2013

DEPARTMENT OF MATHEMATICAL INFORMATICS
GRADUATE SCHOOL OF INFORMATION SCIENCE AND TECHNOLOGY
THE UNIVERSITY OF TOKYO
BUNKYO-KU, TOKYO 113-8656, JAPAN

WWW page: http://www.keisu.t.u-tokyo.ac.jp/research/techrep/index.html

An Automatic Fusion Mechanism for Variable-Length List Skeletons in SkeTo

Kento Emoto Kiminori Matsuzaki

Abstract

Skeletal parallel programming is a promising approach to easy parallel programming, in which user programmers easily build their parallel programs by simply combining some of a given set of ready-made parallel computation patterns called skeletons. In exchange for the easiness, this approach has an efficiency problem caused by its compositional style programming. Fusion transformation is a solution of this problem, which optimizes naively-composed skeleton programs by eliminating redundant intermediate data structures. Several parallel skeleton libraries have implement automatic fusion mechanisms. However, no automatic fusion mechanism has been proposed for so-called variable-length list skeletons (VLL skeletons for short), although VLL skeletons are useful for practical problems. The main difficulty is that the previous fusion mechanisms are not applicable to VLL skeletons, which cannot complete the fusion. In this paper, we propose a novel fusion mechanism for VLL skeletons, which achieves both easy programming interface and the complete fusion. The proposed mechanism has been implemented by using expression templates technique in our skeleton library SkeTo, and shown to be very effective by experiment results.

1 Introduction

As parallel computers are widely spread, parallel programming has become important and inevitable. However, parallel programming is much more difficult than sequential programing in general, because programmers have to consider extra things such as complicated scheduling of tasks, data distribution, communication and synchronization between processors, etc. This situation calls for easy parallel programming.

Skeletal parallel programming has been proposed and studied as a promising approach to easy parallel programming, in which user programmers build their parallel programs by combining some of a given set of ready-made parallel computation patterns called *skeletons* [8, 9, 11], such as `map` to apply a function to every element of a list, and `reduce` to take a summation of a list with a binary operator. For example, we can easily build a parallel program for computing the variance of list `x` with its average `ave` by using the skeletons with user-defined simple functions `plus`, `square` and `sub` as follows.

```
double var = reduce(plus, map(square, map(sub(ave), x)));
```

In spite of the easiness of programming, skeleton programs suffer from inefficiency caused by production of intermediate data structures between successive skeletons due to the compositional style of programming. For example, the skeleton program above has three local loops for two `map`s and the final `reduce`, and two intermediate lists are produced between successive loops, although we can compute the variance sequentially in a single loop.

Fusion transformation has been studied and used to optimize skeleton programs by removing redundant intermediate data structures, which dramatically improves the efficiency of naively-composed skeleton programs. Indeed, optimization mechanisms based on fusion transformation

have been implemented in several skeleton libraries and systems [7, 8, 9] including our library SkeTo[1], and the fusion transformation has been shown to be actually effective. For example, although the skeleton program above appears to have three loops, it is optimized into the following single loop followed by the final global communication.

```
double r = 0.0;
for(int i = 0; i < x.local_size(); i++)
  r = plus(r, square(sub(ave)(x.local_get(i))));
global_reduce(plus, r);
```

Variable-length list skeletons (VLL skeletons for short), such as concatmap to concatenate the results of applying a function to every element, and filter to discard elements not satisfying a given predicate, are useful in practice [13]. These skeletons generate lists of length different from that of the input. For example, we can easily build a parallel program for the n-queen problem [13] by using these two skeletons like below.

```
dist_list<board> x; x.push_back(emptyBoard);
for(int i = 0; i < n; i++)
  x = filter(invalidBoard, concatmap(putNewQueen, x));
long answer = x.length();
```

Starting from an empty board, the program repeatedly generates a list of new valid boards by putting one more queen in every board in the current list. In each iteration, it first generates all possible boards by using concatmap with `putNewQueen` that generates a list of new boards, each of which has a new queen in the top row of the given board. Then, it discards invalid boards by using filter with `invalidBoard` that returns `true` if the given board contains no collision of queens. Although this program is clear, it seems inefficient due to the intermediate list generated between concatmap and filter. We hope that an automatic fusion mechanism can improve the efficiency.

In spite of their usefulness, unfortunately no fusion mechanism has been proposed for these VLL skeletons. This is mainly because the previous fusion mechanism for the fixed-length list skeletons cannot be applied to VLL skeletons. Therefore, naively-composed programs with VLL skeletons suffer from inefficiency caused by redundant intermediate data structures.

In this paper, we propose design and implementation of a novel fusion mechanism for VLL skeletons, so that users can enjoy both VLL skeletons and automatic fusion transformations to get efficient parallel programs for various problems in an easy way. Our main technical contributions are as follows.

- We propose a novel design of *collector-based fusion mechanism* that brings both simple programming interface and complete fusion results, which cannot be achieved by the previous mechanisms.
- The new fusion mechanism is implemented by using expression templates [14], so that users need only a C++ compiler to enjoy our proposing fusion.
- Our proposing mechanism can be used together with the previous fusion mechanism in SkeTo. This means that the proposed mechanism strictly widens the application area of fusion optimization.

[1] http://sketo.ipl-lab.org/

The rest of this paper is organized as follows. Section 2 reviews the previous fusion mechanism for fixed-length list skeletons, and Section 3 reviews VLL skeletons, the target of our proposing fusion mechanism. Section 4 discusses several approaches to a fusion mechanism for VLL skeletons, and Section 5 shows and evaluates the implementation of the proposed mechanism. Finally, Section 6 discusses related work, and Section 7 concludes this paper.

2 Preliminaries

In this section, after introducing notation for formal discussion, we briefly review the previous fusion mechanism for fixed-length list skeletons.

The notation in this paper is reminiscent of Haskell [3]. Function application is denoted by a space and the argument may be written without brackets, so that $f\ a$ means $f(a)$ in ordinary notation. Functions are curried: they always take one argument and return a function or a value, and the function application associates to the left and binds more strongly than any other operator, so that $f\ a\ b$ means $(f\ a)\ b$ and $f\ a \otimes b$ means $(f\ a) \otimes b$. Function composition is denoted by $\circ$, and $(f \circ g)\ x = f\ (g\ x)$ according to its definition. Binary operators can be used as functions by sectioning as follows: $a \oplus b = (a\oplus)\ b = (\oplus b)\ a = (\oplus)\ a\ b$. A list is denoted by enclosing its elements by square brackets [and], e.g., $[a]$ represents a singleton list of element a, and $[a, b, c]$ a list of elements a, b and c. The list concatenation operator is denoted by $+\!\!+$, so that $[a, b] +\!\!+ [c, d] = [a, b, c, d]$. An empty list is denoted by $[\,]$. Function $[\cdot]$ creates a singleton list of the given element, so that $[\cdot]\ a = [a]$.

2.1 Fixed-length List Skeletons

We briefly review a small subset of our fixed-length list skeletons (FLL skeletons for short) [9]. Their intuitive definitions are as follows.

$$
\begin{array}{lcl}
\mathsf{map}\ f\ [a_1, \ldots, a_n] & = & [f\ a_1, \ldots, f\ a_n] \\
\mathsf{reduce}\ (\oplus)\ [a_1, \ldots, a_n] & = & a_1 \oplus \cdots \oplus a_n \\
\mathsf{zip}\ [a_1, \ldots, a_n]\ [b_1, \ldots, b_n] & = & [(a_1, b_1), \ldots, (a_n, b_n)] \\
\mathsf{shiftl}\ e\ [a_1, \ldots, a_n] & = & [a_2, \ldots, a_n, e] \\
\mathsf{shiftr}\ e\ [a_1, \ldots, a_n] & = & [e, a_1, \ldots, a_{n-1}]
\end{array}
$$

The skeleton map applies the given function f to every element a_i of the given list $[a_1, \ldots, a_n]$, to produce the new list $[f\ a_1, \ldots, f\ a_n]$. The skeleton reduce takes an associative binary operator $\oplus$ and an input list $[a_1, \ldots, a_n]$ to sum up its elements by using the operator. The skeleton zip builds a list of pairs of corresponding elements of given two lists, and skeletons shiftl and shiftr move elements to the left and right, respectively.

For example, if we want to take a summation of a given list after doubling its even numbers, we can easily make a parallel program for this by combining these skeletons:

$$
\begin{array}{l}
\mathit{evenDblSum} = \mathsf{reduce}\ (+) \circ \mathsf{map}\ \mathit{evenDbl} \\
\quad \textbf{where}\ \mathit{evenDbl}\ a = \textbf{if}\ \mathit{even}\ a\ \textbf{then}\ a + a\ \textbf{else}\ a
\end{array}
$$

Here, map is used to double even numbers by applying user function *evenDbl*, and reduce is used to take a summation of the results.

2.2 Fusion of FLL Skeletons

Our skeletons [9] have been designed based on a special recursive function called *homomorphism*, to have good optimizability by fusion. Given an associative binary operator $\oplus$ and a function

f, a homomorphism $(\!|\oplus, f|\!)$ is defined as follows.

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

Here, $\imath_\oplus$ is the identity element of $\oplus$, i.e., $a \oplus \imath_\oplus = \imath_\oplus \oplus a = a$ for any a. For example, the skeleton map is defined as $\mathsf{map}\ f = (\!|+\!\!+, [\cdot] \circ f|\!)$, and the skeleton reduce as $\mathsf{reduce}\ (\oplus) = (\!|\oplus, id|\!)$, in which id is the identity function, i.e., $id\ a = a$ for any a.

Homomorphisms have good fusability [2], and thus our skeletons have good fusability too. For example, we have the following fusion rules for the skeletons above.

$$\begin{array}{lcl} \mathsf{map}\ f \circ \mathsf{map}\ g & = & \mathsf{map}\ (f \circ g) \\ \mathsf{reduce}\ (\oplus) \circ \mathsf{map}\ f & = & (\!|\oplus, f|\!) \end{array}$$

In each rule, the left hand side has two skeletons and thus an intermediate list between them, while the right hand side has only one skeleton (homomorphism) and no intermediate list. Thus, the right hand side is expected to be faster than the left hand side. Indeed, this has been shown to be true by experiment results [9].

For example, from the example skeleton program $evenDblSum = \mathsf{reduce}\ (+) \circ \mathsf{map}\ evenDbl$, we can get a faster program $evenDblSum_{opt} = (\!|+, evenDbl|\!)$ by using the second fusion rule.

2.3 Implementation of FLL Skeleton Fusion via Expression Templates

The fusion of skeletons has been implemented in our skeleton library SkeTo [9] by using expression templates (ET for short) [14] with an index-based access method. We briefly review the mechanism.

Here is an example user code implementing the example program *evenDblSum*, which uses skeletons `map` and `reduce` with user-defined function object `evenDbl` and the STL plus operator.

```
int evenDblSum(dist_list<int> z) {
  return reduce(plus<int>(), map(evenDbl, z));
}
```

The user function `evenDbl` is defined as a function object like below, in which it extends the base class to tell its type to the library.

```
struct evenDbl_t : function_base<int(int)> {
  int operator()(int a) const { return even(a) ? a + a : a; }
} evenDbl;
```

In the ET-based library, production of the resulting list of a skeleton is postponed and the skeleton returns an *expression object* representing its computation, so that the computation can be fused into successive computations. For example, the function `map` is defined to build an object of `MapObj` that has two field `f` and `x` to represent the computation of $\mathsf{map}\ f\ x$, as shown in Figure 1. The object also has index-based access method `local_get` that returns the result of applying `f` to the ith element of `x`, which is the ith element of the resulting list computed by this expression. This method is used to generate elements *on demand*, which can avoid storing the intermediate results, to lead to the fusion.

A skeleton like reduce that does not produce a list receives an expression object built so far and carries out the *fusion* in its computation. For example, the skeleton reduce is implemented

```
template <typename F, typename X>
struct MapObj {
  const F& f; const X x;
  MapObj(const F& f, const X& x) : f(f), x(x) { }
  typename F::result_type local_get(int i) const { return f(x.local_get(i)); }
  int local_size() const { return x.local_size(); }
};

template <typename F, typename X>
MapObj<F, X> map(const F& f, const X& x) { return MapObj<F, X>(f, x); }
```

Figure 1: ET implementation of skeleton map, which returns an expression object `MapObj`.

```
template <typename OP, typename X>
typename OP::result_type reduce(const OP& op, const X& x) {
  typename OP::argument_type r = identity_element<OP>::val;
  for(int i = 0; i < x.local_size(); i++) r = op(r, x.local_get(i));
  global_reduce(op, r);
  return r;
}
```

Figure 2: ET implementation of skeleton reduce, which does fusion by using the index-based access method `local_get`.

by a single local loop followed by a global communication as shown in Figure 2. In the local loop, it calls the method `local_get` to get the ith element of the given expression `x`. For example, in the program `evenDblSum`, the function `reduce` receives an object `MapObj(evenDbl, z)` built by `map(evenDbl, z)`, and thus the whole code becomes the following code.

```
int r = 0;
for(int i = 0; i < z.local_size(); i++)
  r = plus<int>()(r, evenDbl(z.local_get(i)));
global_reduce(plus<int>(), r);
```

This code does not produce the intermediate list, i.e., the result of *map evenDbl z*. Indeed, it implements the fused computation $evenDblSum_{opt}$.

An important observation here is that the fused code uses the user-defined function object `evenDbl` *as is*. Actually, this point makes the fusion mechanism quite simple so that it can be implemented by the simple index-based access method. Unfortunately, this does not hold for variable-length list skeletons.

3 Variable-Length List Skeletons

In this section, we introduce variable-length list skeletons (VLL skeletons for short) with their examples and programming interface [13].

```
dist_list<board> x; x.push_back(emptyBoard);
for(int i = 0; i < n; i++)
  x = filter(invalidBoard, concatmap(putNewQueen, x));
long answer = x.length();
```

Figure 3: Examples use of VLL skeletons: n-queens problem.

```
dist_list<int> quicksort(const dist_list<int> &l)
{
  if(l.get_global_size() < 2) return l;
  int pv = list.get(0);
  return append(quicksort(filter(less_than(pv), l))
                append(filter(equal(pv), l),
                       quicksort(filter(greater_than(pv), l))));
}
```

Figure 4: Examples use of VLL skeletons: Quicksort.

3.1 Definition and Example Use of VLL Skeletons

Intuitive definitions of the VLL skeletons are given as follows.

$$\begin{array}{lcl} \mathsf{concatmap}\ f\ [a_1, \ldots, a_n] & = & f\ a_1 +\!\!+ \cdots +\!\!+ f\ a_n \\ \mathsf{filter}\ p\ [a_1, \ldots, a_n] & = & [a_{i_1}, \ldots, a_{i_k}] \\ \quad \textbf{where}\ (\forall i, i \notin \{i_1, \ldots, i_k\} \Leftrightarrow p\ a_i = \mathsf{false}) \wedge (\forall j, i_j < i_{j+1}) & & \\ \mathsf{append}\ x\ y & = & x +\!\!+ y \end{array}$$

The skeleton concatmap, taking a function f to produce a list from the given argument, applies f to every element of the given list and concatenates the resulting lists. The skeleton filter, taking a predicate (a function returning Boolean value) and a list, removes its elements not satisfying the predicate. The skeleton append simply concatenates the given two lists.

The formal definition of concatmap is given by the homomorphism: $\mathsf{concatmap}\ f = (\![+\!\!+, f]\!)$. Then, based on this definition, filter is defined as $\mathsf{filter}\ p = \mathsf{concatmap}\ (\lambda a.\textbf{if}\ p\ a\ \textbf{then}\ [a]\ \textbf{else}\ [\,])$.

VLL skeletons are useful in practice [13], widening the application area of skeletal parallel programming. For example, we can easily build a parallel program for the n-queen problem by using these two skeletons as shown in Figure 3. Here, given a board, function `putNewQueen` generates a list of new boards, each of which has a new queen in the top row, and function `invalidBoard` returns true if the given board contains no collision of queens. In general, by using these skeletons we can easily implement a parallel breadth-first search.

Another important application of VLL skeletons is an irregular divide-and-conquer algorithm, including the convex hull, the quicksort, etc. For example, the quicksort is implemented by using filter and append as shown in Figure 4.

Interested readers may find other examples in the previous work [13].

3.2 Programming Interface of Naively Implemented VLL Skeletons

We briefly review the programming interface of concatmap in the previous work [13], in which the VLL skeletons are implemented naively without any fusion.

The interface of the skeleton concatmap is the following template function.

```
struct evenDup_t {
  vector<int> operator()(int a) const {
    vector<int> v;
    if(even(a)) { v.push_back(a); v.push_back(a) } else { v.push_back(a); };
    return v;
  }
} evenDup;
```

Figure 5: Vector-based implementation of *evenDup* $a =$ **if** *even* a **then** $[a, a]$ **else** $[a]$.

```
template<typename F, typename T, typename S>
dist_list<T> concatmap(const F&f, const dist_list<S> &l);
```

Here, the function object `f` (of type `F`) is expected to return an instance of `vector<T>`, as function f in concatmap f returns a list.

For example, if we want to duplicate every even number in a given list `x`, we can use `concatmap` with user-defined function object `evenDup` (Figure 5) implementing a user function *evenDup* $a =$ **if** *even* a **then** $[a, a]$ **else** $[a]$ as follows.

```
x = concatmap(evenDup, x);
```

The function object `evenDup` implements the function *evenDup* straightforwardly in the functional style: It simply returns a vector of one or two elements. Since the functional style has been shown suitable for parallel programming [12, 8, 9, 11] and our skeletons are designed in the functional style, we can say that this simple programming interface is good.

4 Fusion Mechanism for Variable-Length List Skeletons

In this section, we discuss three approaches to a fusion mechanism of the VLL skeletons, to find the best one that achieves both good programability and good efficiency. Since filter is a special case of concatmap, and append is simply concatenates the two lists, we focus on a fusion mechanism for concatmap.

4.1 Target Fusion Transformation

First of all, we clarify our target fusion transformation of concatmap, by using the following example program *evenDupSum*.

$$\begin{aligned} \mathit{evenDupSum} \;=\; & \mathsf{reduce}\ (+) \circ \mathsf{concatmap}\ \mathit{evenDup} \\ & \textbf{where}\ \mathit{evenDup}\ a = \textbf{if}\ \mathit{even}\ a\ \textbf{then}\ [a, a]\ \textbf{else}\ [a] \end{aligned}$$

Given a list, *evenDupSum* first duplicates every even number in the list by using concatmap with *evenDup*, and then it takes a summation of the resulting list by using reduce. It is easily seen that *evenDupSum* is equivalent to *evenDblSum* in Section 2.1.

Since the program *evenDupSum* above generates an intermediate data structure (list) between reduce and concatmap, it seems inefficient, and we want to fuse these skeletons to get an efficient program. What should be the resulting program of fusion? Since *evenDupSum* is

equivalent to *evenDblSum*, we expect the result of fusion to be the following $evenDupSum_{opt}$, which is the same as $evenDblSum_{opt}$ that does not produce any intermediate list.

$$\begin{aligned} evenDupSum_{opt} &= (\![+, evenDup']\!) \\ &\quad \textbf{where } evenDup'\ a = \textbf{if } even\ a \textbf{ then } a + a \textbf{ else } a \end{aligned}$$

Actually, this can be obtained by using the fusion theorem of homomorphisms.

The goal of our fusion mechanism is to obtain the efficient $evenDupSum_{opt}$ from the naive *evenDupSum*, but there is a difficulty that did not appear in the previous fusion (Section 2.2).

The difficult point is that in the fused program $evenDupSum_{opt}$ the user function *evenDup* is *not used as is*, which means that a fusion mechanism needs to create the new function $evenDup'$ from the definition of *evenDup* and +. However, in many programming languages it is difficult to get the body of a user function and create a new function from it, so that a fusion mechanism has to use a user function as is. Therefore, we need a certain trick in defining a user function to implement a fusion mechanism for VLL skeletons. This situation is quite different from that of the FLL skeletons, in which the fusion mechanism can use a user function as is. This is one of the reasons why the previous fusion mechanism is not applicable to VLL skeletons.

In the following sections, we will discuss three approaches to a fusion mechanism of VLL skeletons, focusing on how users define their functions and what code a fusion mechanism can produce, e.g., from the following main user code for *evenDupSum*.

```
int evenDupSum(dist_list<int> z) {
  return reduce(plus<int>(), concatmap(evenDup, z));
}
```

4.2 Vector-based Approach

In this approach, a user function f used in concatmap f is implemented by a function object `f` that returns a concrete vector, which is a straightforward implementation of f that returns a list. For example, the user function *evenDup* is implemented as the function object `evenDup` shown in Figure 5, in which it returns a concrete vector of one or two elements.

This approach has an advantage of good programability: It provides a simple functional programming style for defining a user function. Actually, this style is the same as the previous mechanism, and quite natural in programming with our skeletons that have functional style definitions.

However, this approach has a big disadvantage: It suffers from *incomplete fusion*. Figure 6 shows the local loop of the fused program of *evenDupSum* in this approach. In the main loop, the user-defined `evenDup` creates a small vector `v` at every iteration, and the inner loop runs on the vector `v` to sum up its elements to the accumulator `r`. Since we cannot change the definition of `evenDup` at compile time, this production of small vectors is not avoidable. Therefore, the fusion is incomplete. Actually, the code implements not our goal $evenDupSum_{opt}$ but the following incompletely-fused program $evenDupSum''$.

$$\begin{aligned} evenDupSum'' &= (\![+, evenDup'']\!) \\ &\quad \textbf{where } evenDup''\ a = \mathsf{reduce}\ (+)\ (evenDup\ a) \end{aligned}$$

At a glance, this program looks successfully fused because the composition of reduce and concatmap has been replaced with the homomorphism $(\![+, evenDup'']\!)$. However, *the fusion is incomplete* in the sense that it creates intermediate data structures (lists) inside the new function $evenDup''$. This incompleteness raises a serious efficiency problem when a user function returns a big list.

```
for(int i = 0; i < z.local_size(); i++) {
  vector<int> v = evenDup(z.local_get(i));
  for(int j = 0; j < v.size(); j++) r = plus<int>()(r, v[j]);
}
```

Figure 6: The main loop of the fused program of *evenDupSum* in the vector-based approach.

```
for(int i = 0; i < z.local_size(); i++) {
  iterator<int> it = evenDup(z.local_get(i));
  while(it.has_next()) r = plus<int>()(r, it.next());
}
```

Figure 7: The main loop of the fused program of *evenDupSum* in the iterator-based approach.

The main problem of this approach is that the fused program produces many vectors—some of which are possibly big—inside the main loop, and we cannot avoid this as long as a user-defined function object returns a concrete vector. To avoid this incompleteness of the fusion, we need a user function not returning a concrete vector.

4.3 Iterator-based Approach

In this approach, to avoid the production of intermediate data structures (vectors) in the fused program, a user implements a function object to return an iterator (an object that yields elements one by one) instead of a concrete vector. Use of iterators to avoid intermediate data structures is natural in practical C++ programming.

The advantage of this approach is that we can achieve the complete fusion, avoiding the problem of the vector-based approach. Figure 7 shows the code of the fused program in this approach, in which `evenDup` returns an iterator `it` instead of a concrete vector. The inner loop sums up elements yielded by the method `next` of `it` while `it` has elements to be yielded. There is no production of any intermediate data structure in the main loop, and this fused code can successfully implement our goal program $evenDupSum_{opt}$.

Although this approach can achieve the complete fusion, unfortunately it has two disadvantages: difficulty of user programming and a chance of incomplete fusion.

The main disadvantage is the difficulty of user programming: Defining a user function to return an iterator is much more difficult than returning a concrete vector. Figure 8 shows an implementation of the user function *evenDup*, in which the function object `evenDup` returns an iterator. Clearly, the code is much more complicated than the code (Figure 5) in the vector-based approach: A new structure `evenDup_iterator` is needed to implement the iterator, and it needs some computation to count the number of elements yielded so far, which are not required in the vector-approach. Even though the function *evenDup* is quite simple, we cannot understand what the function object `evenDup` computes at a glance.

The other disadvantage is that this approach has a chance of incomplete fusion, mainly due to the difficulty of user programming. Usually, implementing a new iterator is complicated and difficult for user programmers, and they may take a simpler way to avoid such difficulty: creating a vector and returning its iterator. Figure 9 shows a simple but problematic implementation of *evenDup* in this style[2]. This code is simple, and easy to write and understand. However, the

[2]The code is simplified to make the problem clear: Please ignore problems related to temporary objects.

```
struct evenDup_t {
  struct evenDup_iterator {
    const int a; int cnt;
    evenDup_iterator(int a) : a(a), cnt(even(a) ? 2 : 1) { }
    bool has_next() { return cnt > 0; }
    int next() { cnt--; return a; }
  };
  evenDup_iterator operator()(int a) const {
    return evenDup_iterator(a);
  }
} evenDup;
```

Figure 8: Iterator-based implementation of user function *evenDup*.

```
struct evenDup_t {
  iterator<int> operator()(int a) const {
    vector<int> v;
    if(even(a)) { v.push_back(a); v.push_back(a) } else { v.push_back(a); };
    return v.begin();
  }
} evenDup;
```

Figure 9: Simple but problematic implementation of user function *evenDup* in iterator-based approach.

fused program using this function object produces many vectors implicitly inside the calls of `evenDup`, which implements the incompletely-fused program *evenDupSum″*.

The main problem of this approach is the difficulty of the user programming, and this difficulty is caused by adopting the functional style such that a function object returns something. To avoid this difficulty and achieve both good programability and good efficiency, we need to change our thinking from the functional style to a slightly imperative style.

4.4 Collector-based Approach

In this approach, adopting a slightly imperative style, a user function is implemented to *receive* a *collector* (an object that receives elements one by one), which is a dual of the iterator-based approach. This approach can achieve both good programability and good efficiency, as shown below.

Figure 10 shows an implementation of the user function *evenDup* in this approach. The function object `evenDup` receives a collector `c`, and puts elements into the collector by calling `c.push_back(a)`. The code looks almost the same as the code in the vector-based approach (Figure 5). The only difference is the place where the elements are emitted into: The former puts elements into the given collector, while the latter puts elements into its created vector. Therefore, the programability of this approach is as good as the vector-approach, and much better than the iterator-approach. It is worth noting that this style of defining a user function is also adopted in Hadoop [1], a practical implementation of the MapReduce model [6].

Figure 11 shows the fused program of *evenDupSum* in this approach. The fused program

```
struct evenDup_t {
  void operator()(int a, Collector &c) const {
    if(even(a)) { c.push_back(a); c.push_back(a) } else { c.push_back(a); };
  }
} evenDup;
```

Figure 10: Collector-based implementation of user function *evenDup*.

```
  struct ReduceCollector {
    int &r;
    ReduceCollector(int &r) : r(r) { }
    void push_back(int a) { r = plus<int>()(r, a); }
  };
  int r = 0;
  ReduceCollector c(r);
  for(int i = 0; i < z.local_size(); i++) evenDup(z.local_get(i), c);
```

Figure 11: The main loop of the fused program of *evenDupSum* in the collector-based approach.

uses a collector defined as a new structure `ReduceCollector`, of which method `push_back` adds the given element `a` into its accumulator variable `r`. In each iteration of the main loop, the user function `evenDup` receives the collector `c` as well as the ith element `z.local_get(i)` of the input list `z`, and puts one or two copies of the element into the collector. Since the collector immediately adds the given element into the accumulator, there is no production of intermediate vectors in this code. Therefore, this code successfully implements our goal program $evenDupSum_{opt}$.

This approach can fuse multiple concatmaps. To explain this, we use the following program with two concatmaps, which computes a doubled summation of even numbers only.

$$\begin{aligned} evenDupNoOddSum \;\; = \;\; & \mathsf{reduce}\ (+) \circ \mathsf{concatmap}\ noOdd \circ \mathsf{concatmap}\ evenDup \\ & \textbf{where}\ noOdd\ a = \textbf{if}\ even\ a\ \textbf{then}\ [a]\ \textbf{else}\ [\,] \end{aligned}$$

Figure 12 shows a collector-based implementation of the user function *noOdd*. Our desired fused program is basically the following program.

$$\begin{aligned} evenDupNoOddSum_{opt} = {} & (\![+, evenDupNoOdd]\!) \\ & \textbf{where}\ evenDupNoOdd\ a = \textbf{if}\ even\ a\ \textbf{then}\ a + a\ \textbf{else}\ 0 \end{aligned}$$

What we need to do to fuse multiple concatmaps is just to build new collectors from user functions. We use a new structure `Collector` shown in Figure 13, which has two fields to hold a user function `f` and another collector `c`. The method `push_back` of `Collector` simply supplies the given element `a` and the collector `c` to the user function `f`.

Figure 14 shows the main loop of the fused program of *evenDupNoOddSum*, which simply supplies elements to the new collector built from the user functions. Figure 15 shows the computation flow of the new collector, in which `zi` corresponds to `z.local_get(i)` in the main loop. When `zi` is an even number (the solid line), by definition, `c2.push_back(zi)` calls `evenDup(zi, c1)` once, and the call of `evenDup` makes two calls of `c1.push_back(zi)`. Each call of `c1.push_back(zi)` invokes `noOdd(zi, c)` once, and this `noOdd` makes one call of `c.push_back(zi)`. Therefore, when `zi` is even, `zi` is added to the accumulator `r` twice. On the

```
struct noOdd_t {
  void operator()(int a, Collector &c) const {
    if(even(a)) { c.push_back(a) };
  }
} noOdd;
```

Figure 12: Collector-based implementation of user function *noOdd*.

```
template<typename NextCollector, typename F>
struct Collector {
  NextCollector c; const F f;
  Collector(const F&f, NextCollector &c) : f(f), c(c) {}
  void push_back(const int& a) { f(a, c); }
};
```

Figure 13: Structure of combined collectors for fusing multiple concatmaps.

other hand, when `zi` is an odd number (the dashed line), the call of `evenDup` makes one call of `c1.push_back(zi)`, and it invokes `noOdd(zi, c)` once. Since `noOdd(zi, c)` does nothing when `zi` is odd, the accumulator `r` is kept unchanged in this case. Clearly, the main loop implements our desired fused program.

Now, we have got a good design of a fusion mechanism to achieve both easy programming interface and the complete fusion. Its concrete implementation will be explained in Section 5.1.

5 Implementation and Evaluation

We have implemented the fusion mechanism for VLL skeletons in our library SkeTo [9] by using expression templates technique [14]. We briefly explain it and report some experiment results to show the impact of the fusion mechanism.

5.1 Implementation of the Fusion Mechanism for VLL Skeletons

Figure 16 shows the implementation of the fusion mechanism of the collector-based approach. In the explanation below, we use as an example the following code implementing *evenDupNoOddSum*.

```
int sum = reduce(plus<int>(), concatmap(noOdd, concatmap(evenDup, z)));
```

The skeleton function `concatmap` returns an expression object of `CMapObj`, to postpone its computation to have a chance of fusion. The object has two fields: a user function `f` and an expression object `x` that represents its target list. It also has several methods and type declaration, which will be explained later. For example, `concatmap`s in the example program create an object `CMapObj(noOdd, CMapObj(evenDup, z))`.

The skeleton function `reduce` receives a `CMapObj` object that represents its target list, as well as an associative binary operator `op`. Before executing the main loop, it asks the object to find the initial list in the chain of `concatmap`s and build a combined collector from the initial collector `ic` of the generalized `ReduceCollector` that accumulates given elements to the accumulator `res` by `op`. For example, the initial list of the example above is `z`, and the combined collector is

```
int r = 0; ReduceCollector c(r);
Collector<ReduceCollector, noOdd_t> c1(noOdd, c);
Collector<Collector<ReduceCollector, noOdd_t>, evenDup_t> c2(evenDup, c1);
for(int i = 0; i < z.local_size(); i++) c2.push_back(z.local_get(i));
```

Figure 14: The main loop of the fused program of *evenDupNoOddSum*.

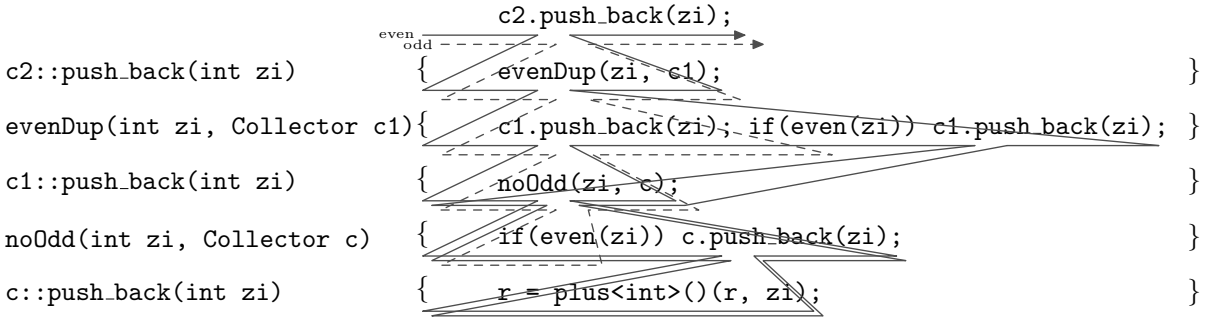

Figure 15: The call chain of collectors inside the fused program of *evenDupNoOddSum*.

(equivalent to) the one explained in the last of Section 4.4. The extraction of the initial list and construction of the combined collector can be implemented by simple recursive methods `getCollector` and `getInitialList` on expression objects. Then, the main loop supplies each element of the initial list to the combined collector.

The above mechanism implements our desired fusion for concatmaps.

It is easily seen that we can use the previous fusion mechanism for FLL skeletons in the main loop of the fused program, because it uses the index-based access method `local_get`. This means that we can fuse FLL skeletons followed by a chain of VLL skeletons into one loop.

Finally, it should be noted that a resulting list of a chain of concatmap can be computed efficiently with fusion in a similar way. We can simply use a `vector` as an initial collector instead of `ReduceCollector`.

5.2 Experiment Results

To evaluate the implemented fusion mechanism, we measured execution time of skeleton programs for three problems *evenDupSum*, *n*-queens and *sumOfPeaks* with and without the fusion. Table 1 shows measured execution time on a cluster consisting of 32 nodes, each of which has Intel(R) Xeon(R) CPU E5645 and 12GB memory, and is connected to Gigabit Ethernet. The programs were compiled with GCC 4.6.3. An empty cell means that the program was not run due to the shortage of the memory. The size means the number of elements of the input list, or the size n of boards for n-queens problem. We used one core per a node. Basically, skeleton programs show good scalability regardless of the fusion.

The measured execution time of fused `evenDupSum` compared with that of non-fused version shows the basic impact of the proposed fusion mechanism: The fusion improves the efficiency dramatically, achieving 30× speedup. The fused program achieves the absolute speed slightly faster than the fused version of `evenDblSum`, which is fused by the previous fusion mechanism, and `evenDupSumHand` that is the following hand-written single sequential loop.

```
for(i=0; i < n; i++) r += (x[i]&1) ? x[i] : x[i] + x[i];
```

```
template <typename F, typename X>
struct CMapObj {
  const F f; const X x;
  CMapObj(const F &f, const X &x) : f(f), x(x) { }
  typedef typename X::InitialType InitialType;
  const InitialType &getInitialList() const { return x.getInitialList(); }
  template <typename NextCollector>
  /* omit the type */
  getCollector(NextCollector &c) const {
    return x.getCollector(Collector<NextCollector>(f, c));
  }
};

template <typename F, typename X>
CMapObj<F, X> concatmap(const F& f, const X& x){ return CMapObj<F, X>(f, x); }

template <typename OP, typename A>
struct ReduceCollector {
  const OP& op; A &r;
  ReduceCollector(const OP&op, A &r) : op(op), r(r) { }
  void push_back(const A& a) { r = op(r, a); }
};

template <typename OP, typename F, typename X>
typename OP::result_type
reduce(const OP &op, const CMapObj<F, X> &cmobj) {
  const typename X::InitialType &l = cmobj.getInitialList();
  typename OP::result_type res = get_identity<OP>();
  ReduceCollector<OP, typename OP::result_type> ic(op, res);
  /* omit the type */ c = cmobj.getCollector(ic);
  for(int i = 0; i < l.local_size(); i++) c.push_back(l.local_get(i));
  global_reduce(op, res);
  return res;
}
```

Figure 16: Expression templates implementation of the collector-based fusion mechanism

```
typedef pair<uint, pair<uint, uint> > triple;

struct peak : public functions::base <bool (triple)> {
  bool operator()(const triple &x) const {
    return (x.first < x.second.first) && (x.second.first > x.second.second);
  }
};

struct peak_m : public functions::base <triple (triple)> {
  triple operator()(const triple &x) const {
    return (x.first < x.second.first) && (x.second.first > x.second.second) ?
     x : triple_zero;
  }
};

uint sumOfPeaks = reduce(plus<uint>(), map(fst, map(snd,
                    filter(peak, zip(shiftr(0U, x),zip(x, shiftl(0U, x)))))));
uint sumOfPeaks_m = reduce(plus<uint>(), map(fst, map(snd,
                     map(peak_m, zip(shiftr(0U, x),zip(x, shiftl(0U, x)))))));
```

Figure 17: Concrete programs of *sumOfPeaks* and $sumOfPeaks_m$.

The difference between the compiled code of fused `evenDupSum` and that of `evenDupSumHand` is that the former uses a conditional branch instruction while the latter uses a conditional move instruction. This compiler's different choice of instructions made the difference of execution time. Anyway, the results show that the proposed fusion mechanism produces truly efficient code comparable with hand-written code.

Comparison of the measured times of `nqueen` with and without fusion shows the impact of the fusion on practical programs: It achieves more than 2× speedup for the practical program.

Finally, we compare fused programs of the following equivalent programs, each of which computes a summation of elements bigger than their immediate neighbors. Here, *sumOfPeaks* uses both FLL and VLL skeletons, while $sumOfPeaks_m$ uses only FLL skeleton.

$$
\begin{array}{lcl}
sumOfPeaks\ x & = & \mathsf{reduce}\ (+)\ (\mathsf{map}\ fst\ (\mathsf{map}\ snd \\
 & & \quad (\mathsf{filter}\ peak\ (\mathsf{zip}\ (\mathsf{shiftr}\ 0\ x)\ (\mathsf{zip}\ x\ (\mathsf{shiftl}\ 0\ x))))))) \\
 & & \mathbf{where}\ peak\ (p,(c,s)) = p < c \wedge c > s \\
sumOfPeaks_m\ x & = & \mathsf{reduce}\ (+)\ (\mathsf{map}\ fst\ (\mathsf{map}\ snd \\
 & & \quad (\mathsf{map}\ peak'\ (\mathsf{zip}\ (\mathsf{shiftr}\ 0\ x)\ (\mathsf{zip}\ x\ (\mathsf{shiftl}\ 0\ x))))))) \\
 & & \mathbf{where}\ peak'\ (p,(c,s)) = \mathbf{if}\ p < c \wedge c > s\ \mathbf{then}\ (p,(c,s)) \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ (0,(0,0))
\end{array}
$$

Figure 17 shows concrete implementation of these programs, in which trivial definitions of user functions are omitted. The implementation of skeleton function `filter` is given in Figure 18, which implements its formal definition in Section 3. The proposed fusion mechanism combined with the previous one successfully optimizes the mixed skeleton program *sumOfPeaks*, to achieve the same performance as the fused code of $sumOfPeaks_m$ produced by the previous fusion. This shows that the proposed fusion mechanism works well together with the previous one.

```
template <typename P>
struct FilterFunction {
  const P p; FilterFunction(const P&p) : p(p) { }
  template <typename Collector>
  void operator()(const typename P::argument_type &a, Collector c) {
    if(p(a)) { c.push_back(a); }
  }
};

template <typename P, typename X>
CMapObj<FilterFunction<P>, X> filter(const P&p, const X&x)
{
  return concatmap(FilterFunction<P>(p), x);
}
```

Figure 18: Implementation of filter by concatmap

Table 1: Measured execution time (seconds) of skeleton programs

			#processes					
program	fusion	size	1	2	4	8	16	32
evenDupSum	w/o	400M	14.47	4.60	2.30	1.17	0.61	0.35
	w/	400M	0.50	0.27	0.15	0.10	0.07	0.09
	w/	2G			0.67	0.35	0.20	0.15
evenDblSum	w/	400M	0.59	0.32	0.18	0.12	0.10	0.09
	w/	2G			0.80	0.40	0.28	0.16
evenDblSumHand	w/	2G	0.59	—	—	—	—	—
nqueen	w/o	14				44.34	22.05	12.22
	w/	14	123.22	61.85	36.51	18.77	9.63	5.55
sumOfPeaks	w/	400M	2.73	1.36	0.70	0.37	0.20	0.12
	w/	2G			3.44	1.72	0.98	0.46
sumOfPeaks_m	w/	400M	2.73	1.38	0.70	0.37	0.20	0.12
	w/	2G			3.44	1.74	1.09	0.46

6 Related Work

Skeletal parallel programming was first proposed by Cole [5] and a number of systems (libraries) have been proposed so far. Among them, OSL [8] and SaC [7] as well as our library SkeTo [9] are ones equipped with fusion mechanisms to optimize skeleton programs. OSL is a skeleton library based on the BSP model implemented by using MPI and C++, and its fusion mechanism is implemented by using expression templates technique [14]. The set of its fusion rules is almost the same as our previous fusion mechanism [9]. SaC is an array programming language mainly suited for application areas such as numerically intensive applications and signal processing. It has the with-loop fusion mechanism that combines high-level program specifications with runtime efficiency similar to that of hand-optimized low-level specifications. Unfortunately, none of them provides VLL skeletons with a fusion mechanism. Data Parallel Haskell [4] provides parallel treatment of nested lists, and we can enjoy various parallel operations including our VLL skeletons thanks to the nature of Haskell [3], a powerful functional programming language. However, it only targets shared-memory multiprocessor environments.

7 Conclusion

We proposed a novel fusion mechanism for variable-length list skeletons (VLL skeletons for short), adopting the collector-based design for defining user functions. The proposed mechanism achieves both good programability and good performance. In addition, it can be used together with the previous fusion mechanism, so that a wide variety of skeleton programs can enjoy our fusion optimizations. The new fusion mechanism has been implemented by using expression templates technique in our skeleton library SkeTo, and its impact on efficiency has been shown by experiment results.

A VLL skeleton may cause an ill-balance of distributed data, and in such a case we need rebalancing of data before executing the following skeletons, to achieve the best performance. However, the fusion may remove the chance of rebalancing, though it can remove the redundant intermediate data structures. Therefore, we need to control this trade-off for the best parallel performance. Currently, a user can control the range of fusion by hand, and he can find the best setting by trial and error. Automatic control of fusion in such a case is one direction of our future work. In addition, it will be interesting to study shape-changing skeletons on other data structures, such as trees and matrices, by extending the results on VLL skeletons. In a practical direction, it will be an important task to reimplement the fusion mechanisms by using a sophisticated expression templates library like Boost.Proto [10], which will improve maintainability of libraries and raise a chance of stronger transformations.

Acknowledgements

This work was partially supported by Japan Society for the Promotion of Science, Grant-in-Aid for Young Scientists (B) 24700025. The authors would like to thank Liu Yu and Shigeyuki Sato for their fruitful discussions with the authors in the early stage of this work.

References

[1] Apache Software Foundation: Hadoop. `http://hadoop.apache.org`

[2] Bird, R.: An introduction to the theory of lists. In: Proceedings of NATO Advanced Study Institute on Logic of Programming and Calculi of Discrete Design, pp. 5–42 (1987)

[3] Bird, R.: Introduction to Functional Programming using Haskell. Prentice-Hall (1998)

[4] Chakravarty, M.M.T., Leshchinskiy, R., Jones, S.P., Keller, G., Marlow, S.: Data parallel haskell: a status report. In: DAMP '07: Proceedings of the 2007 workshop on Declarative aspects of multicore programming, pp. 10–18. ACM, New York, NY, USA (2007)

[5] Cole, M.: Algorithmic Skeletons : A Structured Approach to the Management of Parallel Computation. Research Monographs in Parallel and Distributed Computing (1989)

[6] Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. Communications of the ACM **51**(1), 107–113 (2008)

[7] Grelck, C., Scholz, S.B.: Merging compositions of array skeletons in SaC. Parallel Computing **32**(7-8), 507–522 (2006)

[8] Javed, N., Loulergue, F.: OSL: Optimized bulk synchronous parallel skeletons on distributed arrays. In: Advanced Parallel Processing Technologies, *Lecture Notes in Computer Science*, vol. 5737, pp. 436–451. Springer Berlin Heidelberg (2009)

[9] Matsuzaki, K., Emoto, K.: Implementing fusion-equipped parallel skeletons by expression templates. In: Implementation and Application of Functional Languages, *Lecture Notes in Computer Science*, vol. 6041, pp. 72–89. Springer Berlin Heidelberg (2011)

[10] Niebler, E.: Boost.proto. `http://www.boost.org/doc/libs/1_53_0/libs/proto/`

[11] Rabhi, F.A., Gorlatch, S. (eds.): Patterns and Skeletons for Parallel and Distributed Computing. Springer-Verlag (2002)

[12] Skillicorn, D.B.: The Bird-Meertens formalism as a parallel model. In: Software for Parallel Computation, volume 106 of NATO ASI Series F, pp. 120–133. Springer-Verlag (1993)

[13] Tanno, H., Iwasaki, H.: Parallel skeletons for variable-length lists in sketo skeleton library. In: Euro-Par 2009 Parallel Processing, *Lecture Notes in Computer Science*, vol. 5704, pp. 666–677. Springer (2009)

[14] Veldhuizen, T.: Expression templates. C++ Report **7**, 26–31 (1995)