Domain-Specific Optimization Strategy for Skeleton Programs

Kento Emoto, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi

Graduate School of Information Science and Technology
University of Tokyo
{emoto,kmatsu}@ipl.t.u-tokyo.ac.jp
{hu,takeichi}@mist.i.u-tokyo.ac.jp

Abstract. Skeletal parallel programming enables us to develop parallel programs easily by composing ready-made components called *skeletons*. However, a simply-composed skeleton program often lacks efficiency due to overheads of intermediate data structures and communications. Many studies have focused on optimizations by fusing successive skeletons to eliminate the overheads. Existing fusion transformations, however, are too general to achieve adequate efficiency for some classes of problems. Thus, a specific fusion optimization is needed for a specific class. In this paper, we propose a strategy for domain-specific optimization of skeleton programs. In this strategy, one starts with a normal form that abstracts the programs of interest, then develops fusion rules that transform a skeleton program into the normal form, and finally makes efficient parallel implementation of the normal form. We illustrate the strategy with a case study: optimization of skeleton programs involving neighbor elements, which is often seen in scientific computations.

1 Introduction

Recently, the increasing popularity of parallel machines like PC clusters and multi-core CPUs attracts more and more users. However, development of efficient parallel programs is difficult due to synchronization, interprocessor communications, and data distribution that complicate the parallel programs. Many researchers have addressed themselves to developing methodology of parallel programming with ease. As one promising solution, skeletal parallel programming [1, 2] has been proposed.

In skeletal parallel programming users develop parallel programs by composing *skeletons*, which are abstracted basic patterns in parallel programs. Each skeleton is given as a higher order function that takes concrete computations as its parameters, and conceals low-level parallelism from users. Therefore, users can develop parallel programs with the skeletons in a similar way to developing sequential programs.

Efficiency is one of the most important topics in the research of skeletal parallel programming. Since skeleton programs are developed in a compositional style, they often have overheads of redundantly many loops and unnecessary intermediate data. To make skeleton programs efficient, not only each skeleton is implemented efficiently in parallel, but also optimizations over multiple skeletons are necessary.

There have been several studies on the optimizations over multiple skeletons based on *fusion transformations* [3–7], which were studied in depth in the field of functional programming [8, 9]. In particular, general fusion optimizations [3–6] have achieved good results both in theory and in practice. For example, Hu et al. [5] proposed a set of fusion rules based on a general form of skeletons named accumulate.

Although the general fusion optimizations so far are reasonably powerful, there is still large room for further optimizations. Due to the generality of their fusion transformations, some overheads in skeleton programs are left through the general fusion optimizations. In many cases such overheads can be removed if we provide a program-specific implementation. Thus, some specific optimizations are important for efficiency of skeleton programs.

In this paper, we study domain-specific optimizations to make skeleton programs more efficient. The target skeleton programs of these optimizations are domain-specific in the sense that the programs are built with a fixed set of skeletons and have some specific way of compositions of the skeletons. With the knowledge of the domain-specific properties, we expect to develop more efficient domain-specific programs.

The main contribution of the paper is a new strategy for domain-specific optimization of skeleton programs. The strategy proposed is as follows. First, we formalize a *normal form* that captures the domain-specific computations. Then, we develop *fusion rules* that transform a skeleton program into the normal form. Finally, we provide an efficient *parallel implementation* of the normal form.

We confirm the usability and effectiveness of the strategy with a case study of optimizing skeleton programs that involve neighbor elements, which is often seen in scientific computations. We formalize a normal form and fusion rules for the class of skeleton programs and developed a small system for fusing skeleton programs into the normal form implemented efficiently in parallel. The experiment results show effectiveness of the domain-specific optimization.

The rest of this paper is organized as follows. Section 2 explains our strategy for domain-specific optimization of skeleton programs. Section 3 gives a case study of optimization for skeleton programs involving neighbor elements. Section 4 discusses the applicability of our strategy and related work. Section 5 concludes this paper.

2 A General Strategy for Domain-Specific Optimization

In skeletal parallel programming, domain-specific programs are often developed with a fixed set of skeletons composed in a specific manner. Based on this observation, we propose the following strategy for developing domain-specific optimizations.

1. Design a normal form that abstracts target computations.
2. Develop fusion rules that transform a skeleton program into the normal form.
3. Implement the normal form efficiently in parallel.

In designing a normal form, we should have the following requirements in mind. A normal form is specified to describe any computation of target programs but should not be too general. A normal form should be specific to the target programs, and should enable us to develop efficient implementation for it. In addition, a normal form should be closed under the fusion rules to maintain the result of optimization in the form. Once we formalize a normal form with fusion rules and efficient implementation, we can perform the optimization easily: we first transform a skeleton program into the normal form with the fusion rules, and then we translate the program in the normal form to an efficient program.

We now demonstrate our strategy with a toy example. We consider programs described by compositions of the following two skeletons map and $\mathsf{shift}_{\gg}$.

$$\begin{aligned}\mathsf{map}(f,[a_1,\ldots,a_n]) &= [f(a_1),\ldots,f(a_n)]\\ \mathsf{shift}_{\gg}(e,[a_1,\ldots,a_n]) &= [e,a_1,\ldots,a_{n-1}]\end{aligned}$$

Here, a list is denoted by lining elements up between '[' and ']' separated by ','. Skeleton map applies given function f to each element of the input list. Skeleton $\mathsf{shift}_{\gg}$ shifts elements of the input list to the right by one, and inserts given value e as the leftmost element. The last element of the input is discarded. An instance of the target skeleton programs is shown below. Since one skeleton has one loop in its implementation, this program has four loops as well as two communication phases in two $\mathsf{shift}_{\gg}$s.

$$ys = \mathsf{shift}_{\gg}(e_0, \mathsf{map}(f, \mathsf{shift}_{\gg}(e_1, \mathsf{map}(g, xs)))) \ .$$

First, we design a normal form by abstracting computation of the target programs. Each resulting list consists of two parts in terms of its generation: some left elements are computed from constants introduced by skeleton $\mathsf{shift}_{\gg}$, and the other elements are computed by applying functions of map skeletons to the input list. Thus, we can define a normal form for the programs as a triple $\langle [c_1,\ldots,c_r],[f_1,\ldots,f_m],xs\rangle$: a list of constants, a list of functions, and an input list. For example, the above example program for ys can be described in the normal form as $\langle [e_0,e_1'],[g,f],xs\rangle$ where $e_1' = f(e_1)$.

Then, we define fusion rules to transform any instance of target programs into the normal form where each skeleton is fused with the normal form.

$$\begin{aligned}&xs \Rightarrow \langle [],[],xs\rangle\\ &\mathsf{map}(f,\langle [c_1,\ldots,c_r],[f_1,\ldots,f_m],xs\rangle) \Rightarrow \langle [f(c_1),\ldots,f(c_r)],[f,f_1,\ldots,f_m],xs\rangle\\ &\mathsf{shift}_{\gg}(e,\langle [c_1,\ldots,c_r],[f_1,\ldots,f_m],xs\rangle) \Rightarrow \langle [e,c_1,\ldots,c_r],[f_1,\ldots,f_m],xs\rangle\end{aligned}$$

The rule for map applies the given function f to each constant element, and inserts the function to the list of functions. The rule for $\mathsf{shift}_{\gg}$ inserts the given constant e to the list of constants. It is straightforward to check that the instance above can be transformed into the normal form as shown above. It is worth remarking that any instance of the target programs can be transformed into the normal form using these rules.

Finally, we develop an efficient parallel implementation of the normal form. The programs in the normal form above can be implemented with a single loop and a single communication. For instance, the example program in the normal form is implemented as follows. Here, the input list xs is divided into blocks of the length $bsize$ (>2) and each processor has one of these blocks. Note that indices of arrays start from one.

```
if(proc != last_proc)  send_to_next_proc(xs[bsize-1], xs[bsize]);
if(proc != first_proc) recv_from_prev_proc(v0, v1);
for(i = 3; i <= bsize; i++) { ys[i] = f(g(xs[i-2])); }
```

```
if(proc == first_proc) { ys[1] = e0; ys[2] = f(e1); }
else                   { ys[1] = f(g(v0)); ys[2] = f(g(v1)); }
```

In this implementation, skeletons of the original program have been fused into one loop with one communication. As illustrated so far, the example skeleton program has been optimized with the normal form and the fusion rules.

3 A Case Study: Optimizing Skeleton Programs Involving Neighbor Elements

We demonstrate our strategy by a case study of optimizing skeleton programs that involve neighbor elements, which is often seen in scientific computations. Due to space limitation, we omit the details of formal definitions. The details can be found in the technical report [10].

3.1 Target Skeleton Programs

Our target skeleton programs involve neighbor elements using combination of $\mathsf{shift}_{\ll}$, $\mathsf{shift}_{\gg}$, zip and map.

$$\begin{aligned} \mathit{Program} ::= {} & \mathsf{map}(f, \mathit{Program}) && \mid \mathsf{zip}(\mathit{Program}, \mathit{Program}) \\ \mid {} & \mathsf{shift}_{\ll}(e, \mathit{Program}) && \mid \mathsf{shift}_{\gg}(e, \mathit{Program}) \\ \mid {} & x \end{aligned}$$

Here, f means a function, e means an element, and x means an input list. We introduce two skeletons zip and $\mathsf{shift}_{\ll}$ as well as previously defined skeletons map and $\mathsf{shift}_{\gg}$. Skeleton zip makes a list of pairs of corresponding elements in the given two lists of the same length. Skeleton $\mathsf{shift}_{\ll}$ shifts elements to the left and inserts the given value as the rightmost element. The first element of the input is discarded.

$$\begin{aligned} \mathsf{zip}([a_1, \ldots, a_n], [b_1, \ldots, b_n]) &= [(a_1, b_1), \ldots, (a_n, b_n)] \\ \mathsf{shift}_{\ll}(e, [a_1, \ldots, a_n]) &= [a_2, \ldots, a_n, e] \end{aligned}$$

Note that any instance of $\mathit{Program}$ takes a list and returns a list of the same length, and that each skeleton has parallel implementation [11].

As our running example, consider a simple program for the following recurrence equation obtained by rearranging a difference equation for physical simulation. Here, u_i^n denotes a value of a field u at time n and at location i, and we consider simple boundary conditions: $u_0^n = b_L$ and $u_{N+1}^n = b_R$ for a fixed N.

$$u_i^{n+1} = c_{-1} u_{i-1}^n + c_0 u_i^n + c_1 u_{i+1}^n$$

A skeleton program next that computes the values at the next time form the current values of u in parallel is given as follows.

$$\begin{aligned} \mathit{next}(u) = \textbf{let } & v'_{-1} = \mathsf{map}(c_{-1}\times, \mathsf{shift}_{\gg}(b_L, u)) \\ & v'_0 = \mathsf{map}(c_0\times, u) \\ & v'_1 = \mathsf{map}(c_1\times, \mathsf{shift}_{\ll}(b_R, u)) \\ \textbf{in } & \mathsf{map}(\mathit{add}, \mathsf{zip}\ (v'_{-1}, \mathsf{map}(\mathit{add}, \mathsf{zip}(v'_0, v'_1)))) \end{aligned}$$

Here, we use intermediate variables v'_{-1}, v'_0, and v'_1 for readability, although the definition of target programs $Program$ does not have variables. The correspondence of the program $next$ and the above recurrence equation is as follows. First, to generate a list corresponding to the first term $c_{-1}u^n_{i-1}$, we apply $\mathsf{shift}_{\gg}$ to shift the elements, and apply map to multiply the coefficient c_{-1}. Similarly, lists corresponding to the second and the third terms are generated by using $\mathsf{shift}_{\ll}$ and map. Then, zipping these three lists by zip and adding elements by $\mathsf{map}\ add$, we obtain the final result. Since each of the three lists are shifted by $\mathsf{shift}_{\gg}$ and $\mathsf{shift}_{\ll}$, the ith element of the resulting list is $c_{-1}u_{i-1} + c_0u_i + c_1u_{i+1}$.

In the rest of this paper, we explain our idea by using this example $next$.

3.2 Normal Form

The first step of our strategy is to design a normal form that can describe any computation of target programs. In this section, we give a normal form for $Program$ (see Section 3.1) that involves neighbor elements by using map, zip, $\mathsf{shift}_{\gg}$ and $\mathsf{shift}_{\ll}$.

Generally, any computation of a target skeleton program is denoted by a triple $[\![\, ls, ce, rs \,]\!]$. Here, ls is a list of computational trees for the left edge of the resulting list, ce is a common computational tree for the center part, and rs is a list of computational trees for the right edge. Thus, we use this triple as our normal form, and denote this triple by using special brackets for readability.

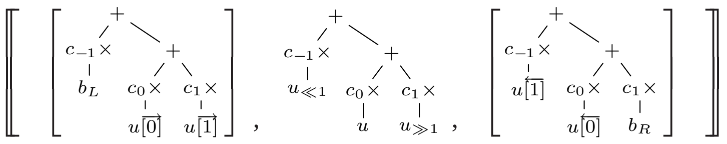
For the example $next$, the leftmost element l_1 and the rightmost element r_1 of the resulting list are calculated by the following expressions. Here, for a fixed index i, $u\overrightarrow{[i]}$ and $u\overleftarrow{[i]}$ denote the ith elements of u from the left and the right respectively.

$$\begin{aligned} l_1 &= add((c_{-1} \times b_L, add((c_0 \times u\overrightarrow{[0]}, c_1 \times u\overrightarrow{[1]})))) \\ r_1 &= add((c_{-1} \times u\overleftarrow{[1]}, add((c_0 \times u\overleftarrow{[0]}, c_1 \times b_R)))) \end{aligned}$$

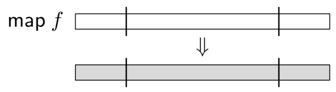
Each computation involves variables ($u\overrightarrow{[0]}$ and $u\overrightarrow{[1]}$, or $u\overleftarrow{[1]}$ and $u\overleftarrow{[0]}$) and a constant (b_L or b_R) introduced by $\mathsf{shift}_{\gg}$ or $\mathsf{shift}_{\ll}$. On the other hand, each element in the center part is calculated by evaluating the following expression ce against each index i. Here, we omit the index i since only the difference from the index is important. Thus, u denotes the ith element, $u_{\ll 1}$ denotes the element on the left of the ith element, and $u_{\gg 1}$ denotes the element on the right.

$$ce = add((c_{-1} \times u_{\ll 1}, add((c_0 \times u, c_1 \times u_{\gg 1}))))$$

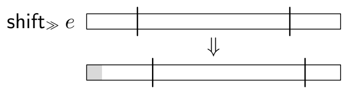
Summarizing these observations, we can denote the whole computation of $next$ by a triple $[\![\, [l_1], ce, [r_1] \,]\!]$. Thus, we use this triple as the normal form. These computational trees have the following structures.



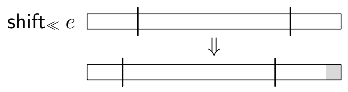
(1)



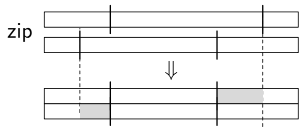
(a) Fusion of map composes the given function to each tree of the normal form. The structure of the normal form is not changed.



(b) Fusion of shift$_{\gg}$ discards the rightmost computational tree and introduces a new constant tree to the leftmost. The amount of shifting in the tree for center part is updated by one.



(c) Fusion of shift$_{\ll}$ discards the leftmost computational tree and introduces a new constant tree to the rightmost. The amount of shifting in the tree for center part is updated by -1.



(d) Fusion of zip instantiates the shaded parts so that the sizes of lists of trees for edges become uniform, and zips the corresponding trees.

Fig. 1. An image of fusion rules. Rectangles show the resulting lists. The three parts separated by vertical lines correspond to the triple of a normal form. Changed parts are shaded.

3.3 Fusion Rules for Transformation to a Normal Form

The second step of our strategy is to define fusion rules to transform a skeleton program to a normal form. These rules should be able to transform any of the target skeleton programs to a normal form. In this section, we give fusion rules to transform a skeleton program $Program$ (see Section 3.1) into the normal form.

The transformation is done one by one using fusion rules. Figure 1 shows an image of the fusion rules (formal definitions can be found in the technical report [10]). Since our target program involves four kinds of skeletons, there are four fusion rules. As an explanation of these rules, we transform the example $next$ into the normal form.

The base case is the transformation of the argument list u. A list u needs only the common computational tree u that is just the element of u. So,

$$u \Rightarrow [\![\, [\,], u, [\,] \,]\!].$$

Next, we fuse shift$_{\gg}$ to transform shift$_{\gg}(b_L, u)$. Since shift$_{\gg}$ introduces the constant b_L to the leftmost element, a new computational tree of the constant b_L is introduced to the new normal form. Also, the amount of shifting in the common tree is updated by one.

$$\mathsf{shift}_{\gg}(b_L, [\![\, [\,], u, [\,] \,]\!]) \Rightarrow [\![\, [b_L], u_{\gg 1}, [\,] \,]\!]$$

Then, we fuse map to the above result to get the following normal form.

$$\mathsf{map}\ (c_{-1}\times, [\![\, [b_L], u_{\gg 1}, [\,] \,]\!]) \Rightarrow [\![\, [c_{-1}\times b_L], c_{-1}\times u_{\gg 1}, [\,] \,]\!]$$

The constant b_L is replaced by $c_{-1} \times b_L$, and the function $c_{-1}\times$ is composed to the root of the common tree. Similarly, applications of shift$_{\ll}$ and map to the input list u result in these normal forms:

$$\begin{aligned}&\mathsf{map}\ (c_0\times, u) \Rightarrow [\![\, [\,], c_0 \times u, [\,]\,]\!]\ ,\\ &\mathsf{map}\ (c_1\times, \mathsf{shift}_{\ll}(b_R, u)) \Rightarrow [\![\, [\,], c_1 \times u_{\ll 1}, [c_1 \times b_R]\,]\!]\ .\end{aligned}$$

In the last transformation, a new tree is introduced by $\mathsf{shift}_{\ll}$ to the rightmost, and the amount of shifting in the common tree is updated by one to the left.

Next, we perform fusion of zip to transform the following program part.

$$\mathsf{zip}(v'_0, v'_1) = \mathsf{zip}([\![\, [\,], c_0 \times u, [\,]\,]\!], [\![\, [\,], c_1 \times u_{\ll 1}, [c_1 \times b_R]\,]\!])$$

Since the lengths of edge lists of two normal forms to be zipped are not the same (i.e. $[\,]$ and $[c_1 \times b_R]$ for the right edges), we have to make the lengths uniform by instantiating the common trees for center parts. The instantiation means to fix the indices in the common trees for elements on the edges. The instantiation and the zip of trees result in the following normal form. Here, the instantiation of the common tree of the first normal from $c_0 \times u$ is $c_0 \times u\overleftarrow{[0]}$, and it is zipped with the rightmost tree of the second normal form to make the new rightmost tree.

$$[\![\, [\,], (c_0 \times u, c_1 \times u_{\ll 1}), [(c_0 \times u\overleftarrow{[0]}, c_1 \times b_R)]\,]\!]$$

Similarly, we obtain the following normal form for $\mathsf{zip}\ (v'_{-1}, \mathsf{map}(add, \mathsf{zip}(v'_0, v'_1)))$:

$$\begin{aligned}&[\![[(c_{-1} \times b_L, add((c_0 \times u\overrightarrow{[0]}, c_1 \times u\overrightarrow{[1]})))]\ ,\\ &\qquad (c_{-1} \times u_{\ll 1}, add((c_0 \times u, c_1 \times u_{\gg 1})))\ , [(c_{-1} \times u\overleftarrow{[1]}, add((c_0 \times u\overleftarrow{[0]}, c_1 \times b_R)))]\,]\!]\end{aligned}$$

Continuing these fusions, we finally obtain the normal form of the example $next$, which is shown in Eq. (1) of Section 3.2.

These four fusion rules and the base case rule can transform any skeleton program defined by $Program$ into the normal form. We conclude this fact as a theorem.

Theorem 1. *Any skeleton program defined by $Program$ can be transformed into the normal form by using the four fusion rules and the base case rule.*

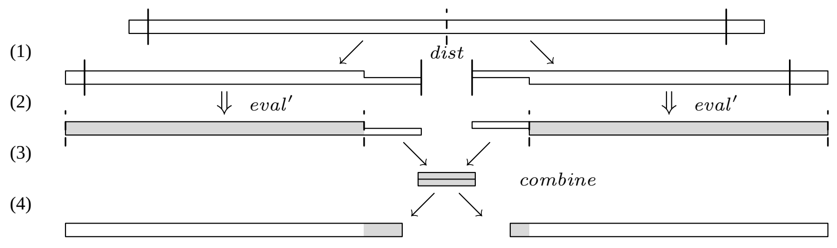
Proof. This is proven by induction on the structure of $Program$. The base case is shown by the transformation of an input list. Induction cases are shown by the four fusion rules. A formal proof is given in the technical report [10]. □

3.4 Parallel Implementation of Normal Form

The third step of our strategy is to design parallel implementation of the normal form. In this section, we explain it briefly. The details are shown in the technical report [10].

Based on parallel implementation of existing skeletons [11], we design parallel implementation of the normal form with four steps: (1) distribution of input lists, (2) the first local computation, (3) global communication, (4) the second local computation. Figure 2 shows an image of computation of the normal form using two processors.

We will briefly explain the parallel implementation by the example $next$. First, we distribute the input list u by dividing it into two parts: $u = u_1 +\!\!+ u_2$. Each processor has



(1) Input lists in a normal form are divided into two parts to be distributed to processors.
(2) Each processor calculates elements that can be calculated by local data.
(3) To calculate the elements on the edges, communication is performed to complete the necessary data.
(4) Perform calculation of the elements on the edges. Combine the edge elements to the other elements.

Fig. 2. An image of parallel implementation of the normal form (two processors).

a part of the divided list. Second, in the first local computation, processors calculate partial results in parallel. This is the main part of the whole computation. The computation is performed in a single loop in which the common computational tree of the center part is evaluated against each index. Therefore, creating no redundant intermediate data, the main part of the computation is efficient. Since elements on the edges of the distributed results need elements of both u_1 and u_2, these elements are calculated after global communication. Third, in the global communication, neighboring processors communicate incomplete trees to each other to complete trees for their edges. Of course, the first local computation can hide the time of this communication phase. Fourth, in the second local step, each processor calculates the elements on the edges with those completed trees to complete the resulting distributed list. After these four steps, these completed results are gathered to the root processor, or become a new input to another normal form. Distribution of input will be skipped in the latter case.

3.5 Experimental Result

We implemented a small domain-specific optimizer for the case study. The system reads a skeleton program written with our parallel skeleton library SkeTo [11], and generates an optimized C++ code. For the example $next$, we measured running times of a skeleton program written with SkeTo and an optimized program. We used a PC cluster where each node connected with Gigabit Ethernet has a CPU of Intel® Xeon®2.80GHz and 2GB memory, with Linux 2.4.21, GCC 4.1.1, and mpich 1.2.7.

Table 1 shows measured running times and speedups. Running time is of applying $next$ 100 times to an input list of 10,000,000 elements. A speedup is a ratio of running time of a sequential program to running time of a parallel program.

The optimized program achieves ten times faster running time than the original skeleton program, and the same running time as a sequential program on one processor. This improvement was gained by elimination of redundant intermediate data and by covering communication time by the computation of center parts in a loop. Also, the optimized program achieves good speedups against the number of processors. These results show effectiveness of the proposed optimization.

Table 1. Running times and speedups of parallel programs against the number of processors.

#processors		1	2	4	8	16	24	32	48	64
next	time (s)	210.25	100.84	48.12	24.41	13.31	8.86	6.52	4.70	3.50
	speedup	0.094	0.20	0.41	0.81	1.49	2.24	3.04	4.23	5.67
next_opt	time (s)	19.86	9.64	4.93	2.44	1.26	0.87	0.70	0.54	0.47
	speedup	1.00	2.06	4.03	8.14	15.79	22.76	28.26	36.73	42.22

4 Discussion and Related Work

One of the simplest fusion optimizations so far uses a general form called cataJ [6]. This cataJ has a function applied to each element of the input list, and an associative binary operator used to perform reduction on elements. Thus, cataJ can describe any computation written as composition of any number of map and at most one reduce (a skeleton to perform reduction) at the last. In this sense, cataJ is a normal form of skeleton programs of such compositions.

Hu et al. [5] proposed a general fusion optimization using a general form called accumulate and a set of fusion rules. The accumulate can describe skeleton scan, which calculates an accumulation of the input list with an associative binary operator, as well as map and reduce. So, it can be a normal form of skeleton programs described with compositions of these skeletons. Although accumulate can describe also $\mathsf{shift}_{\gg}$ and $\mathsf{shift}_{\ll}$, it causes some overheads due to lack of consideration of elements on edges. Main overheads are as follows: (1) extensions of elements for uniform manipulation by the associative binary operator, and (2) logarithmic steps of interprocessor communications for general implementation of accumulation. Thus, we need to consider a specific fusion optimization, i.e. a normal form, fusion rules and efficient implementation.

The normal form of the case study extends these fusions optimizations with consideration of elements on edges introduced by $\mathsf{shift}_{\gg}$ and $\mathsf{shift}_{\ll}$. The normal form separates computation of edges from that of center part, so that it does not introduce the overheads accumulate causes. The normal form, instead, cannot deal with scan.

As extension of the normal form, we can add scan to the domain of target programs. The extended normal form has an associative binary operator as well as the triple of the normal form of the case study. Using this normal form, we can successfully optimize, for example, a skeleton program for solving tridiagonal matrix equations. The details are shown in [10].

5 Conclusion

In this paper, we proposed a general strategy for domain-specific optimization of skeleton programs, and showed a case study for programs involving neighbor elements. Our strategy consists of the following three: (1) a normal form that abstracts computation of target skeleton programs, (2) a set of fusion rules to transform a skeleton program into the normal form, and (3) efficient parallel implementation of the normal form. The optimization is performed by transforming skeleton programs into normal forms with efficient implementation. A small system has been implemented and experiment results show effectiveness of proposed optimization. It is our future work to develop support tools for easy development of various domain-specific optimizations.

6 Acknowledgment

We are grateful to the referees for their detailed and helpful comments. This work was partially supported by Japan Society for the Promotion of Science, Grant-in-Aid for Scientific Research (B) 17300005 , and the Ministry of Education, Culture, Sports, Science and Technology, Grant-in-Aid for Young Scientists (B) 18700021.

References

1. Cole, M.: Algorithmic Skeletons: Structural Management of Parallel Computation. Research Monographs in Parallel and Distributed Computing. MIT Press (1989)
2. Rabhi, F.A., Gorlatch, S., eds.: Patterns and Skeletons for Parallel and Distributed Computing. Springer (2002)
3. Gorlatch, S., Wedler, C., Lengauer, C.: Optimization rules for programming with collective operations. In 13th International Parallel Processing Symposium / 10th Symposium on Parallel and Distributed Processing (IPPS / SPDP '99), 12–16 April 1999, San Juan, Puerto Rico, Proceedings. IEEE Computer Society (1999)
4. Wedler, C., Lengauer, C.: On linear list recursion in parallel. Acta Informatica **35**(10). Springer (1998)
5. Hu, Z., Iwasaki, H., Takeichi, M.: An accumulative parallel skeleton for all. In Métayer, D.L., ed.: Programming Languages and Systems, 11th European Symposium on Programming, ESOP 2002, held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8–12, 2002, Proceedings. Volume 2305 of Lecture Notes in Computer Science. Springer (2002)
6. Matsuzaki, K., Kakehi, K., Iwasaki, H., Hu, Z., Akashi, Y.: A fusion-embedded skeleton library. In Danelutto, M., Vanneschi, M., Laforenza, D., eds.: Euro-Par 2004 Parallel Processing, 10th International Euro-Par Conference, Pisa, Italy, August 31–September 3, 2004, Proceedings. Volume 3149 of Lecture Notes in Computer Science. Springer (2004)
7. Grelck, C., Scholz, S.B.: Merging compositions of array skeletons in SaC. Parallel Computing **32**(7–8). Elsevier (2006)
8. Wadler, P.: Deforestation: Transforming programs to eliminate trees. In Ganzinger, H., ed.: ESOP '88, 2nd European Symposium on Programming, Nancy, France, March 21–24, 1988, Proceedings. Volume 300 of Lecture Notes in Computer Science. Springer (1988)
9. Gill, A.J., Launchbury, J., Jones, S.L.P.: A short cut to deforestation. In FPCA '93 Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark, 9–11 June 1993. ACM Press (1993)
10. Emoto, K., Matsuzaki, K., Hu, Z., Takeichi, M.: Domain-specific optimization for skeleton programs involving neighbor elements. Technical Report METR2007-05, Department of Mathematical Informatics, University of Tokyo. (2007)
11. Matsuzaki, K., Iwasaki, H., Emoto, K., Hu, Z.: A library of constructive skeletons for sequential style of parallel programming. In InfoScale '06: Proceedings of the 1st international conference on Scalable information systems. Volume 152 of ACM International Conference Proceeding Series. ACM Press (2006)