Implementing Fusion-Equipped Parallel Skeletons by Expression Templates

Kiminori Matsuzaki[1] and Kento Emoto[2]

[1] School of Information,
Kochi University of Technology, Japan
`matsuzaki.kiminori@kochi-tech.ac.jp`
[2] Graduate School of Information Science and Technology,
University of Tokyo, Japan
`emoto@ipl.t.u-tokyo.ac.jp`

Abstract. Developing efficient parallel programs is more difficult and complicated than developing sequential ones. Skeletal parallelism is a promising methodology for easy parallel programming in which users develop parallel programs by composing ready-made components called parallel skeletons. We developed a parallel skeleton library *SkeTo* that provides parallel skeletons implemented in C++ and MPI for distributed-memory environments. In the new version of the library, the implementation of the parallel skeletons for lists is improved so that the skeletons equip themselves with fusion optimization. The optimization mechanism is implemented based on the programming technique called expression templates. In this paper, we illustrate the improved design and implementation of parallel skeletons for lists in the SkeTo library.

Keywords: Skeletal parallelism, fusion transformation, list skeletons, expression templates, template meta-programming.

1 Introduction

Hardware environments for parallel computing are now widely available. The popularization and growth of multicore CPUs call for more parallelism to utilize the potential of the hardware. Developing parallel programs, however, is more difficult and complex than developing sequential ones due to, for example, data distribution, communication, and load balancing.

Skeletal Parallelism [1] is a promising methodology for this problem. In the skeletal parallelism, parallel programs are developed by composing ready-made components, called *parallel skeletons*, which are abstract computational patterns often used in parallel programs. Parallel skeletons conceal many details of parallelism in their implementation and, thus, allow for the development of parallel programs as if they were sequential programs. This paper considers parallel skeletons for data-parallel computations in which large amounts of data are processed in parallel.

Our group has intensively studied skeletal parallelism for data-parallel computations since the late 90's. We have developed several methods for deriving skeletal parallel programs and for optimizing skeletal programs using fusion transformation [2] based on the constructive algorithmic theory [3]. To make these results easily available, we have developed a parallel skeleton library named *SkeTo* [4]: the name is from the abbreviation of *Skeleton Library in Tokyo* and it also means helper or supporter in Japanese. Three important features of the SkeTo library are:

– The library is implemented in standard C++ and MPI (Message Passing Interface), and we can widely use the library on distributed-memory environments as well as shared-memory ones. Users who know C++ can use the library without learning another language or library for parallel programming.
– The library provides parallel skeletons for data-parallel computation. Supported data structures are lists (one-dimensional arrays), matrices (two-dimensional arrays), and trees. Parallel skeletons over these data structures have similar interfaces.
– The library provides a mechanism of optimizing skeletal programs based on fusion transformation [5].

The SkeTo library version 0.3beta was released in January 2007. After this release, some problems were found that needed be resolved and the new version 1.0 of the SkeTo library was developed. Two important improvements of the library are:

– With the old version, users had to select the proper skeleton for their specific situation (e.g., a skeleton to overwrite lists). In the new version, selections are automatically done by the library.
– In the old version, the fusion optimization was implemented in OpenC++ [6]– a meta-programming language for C++. OpenC++ is now obsolete. In the new version, the fusion optimization mechanism is implemented using standard C++ in conjunction with the meta-programming technique called *expression templates* [7]. In addition, more powerful fusion rules than those of the old version are implemented.

The SkeTo library is available for several environments. In terms of the OS, it is available for Linux, Mac OS X, and Windows with cygwin; in terms of the Compiler, it is available for GCC versions 3.4 and 4.3, and Intel Compilers 9.1 and 11.1; in terms of the MPI library, it is available for mpich and OpenMPI.

This paper discusses the design and the implementation of the parallel list skeletons in the SkeTo library. The focus is on the self-optimization mechanism implemented with expression templates. The rest of the paper is organized as follows. Section 2 presents the sequential and the parallel definitions of the list skeletons provided in the SkeTo library and discusses how to optimize skeletal programs using fusion transformation. The implementation of the parallel list skeletons is discussed in Section 3. Section 4 evaluates the performance of the SkeTo library using two examples. Related work is reviewed in Section 5 and concluding remarks are presented in Section 6.

$$
\begin{aligned}
&\mathsf{generate}(f, n) = [f(0), f(1), \ldots, f(n-1)] \\
&\mathsf{map}(f, [a_0, a_1, \ldots, a_{n-1}]) = [f(a_0), f(a_1), \ldots, f(a_{n-1})] \\
&\mathsf{zipw}(f, [a_0, a_1, \ldots, a_{n-1}], [b_0, b_1, \ldots, b_{n-1}]) = [f(a_0, b_0), f(a_1, b_1), \ldots, f(a_{n-1}, b_{n-1})] \\
&\mathsf{reduce}(\oplus, [a_0, a_1, \ldots, a_{n-1}]) = a_0 \oplus a_1 \oplus \cdots \oplus a_{n-1} \\
&\mathsf{scan}(\oplus, e, [a_0, a_1, \ldots, a_{n-1}], \mathit{ptr}) = [e, e \oplus a_0, \ldots, e \oplus a_0 \oplus a_1 \oplus \cdots \oplus a_{n-2}] \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{where } \mathit{ptr} \leftarrow e \oplus a_0 \oplus a_1 \oplus \cdots \oplus a_{n-2} \oplus a_{n-1} \\
&\mathsf{scanr}(\oplus, e, [a_0, a_1, \ldots, a_{n-1}], \mathit{ptr}) = [a_1 \oplus \cdots \oplus a_{n-2} \oplus a_{n-1} \oplus e, \ldots, a_{n-1} \oplus e, e] \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{where } \mathit{ptr} \leftarrow a_0 \oplus a_1 \oplus \cdots \oplus a_{n-2} \oplus a_{n-1} \oplus e \\
&\mathsf{shift}_{\gg}(e, [a_0, a_1, \ldots, a_{n-1}], \mathit{ptr}) = [e, a_0, \ldots, a_{n-2}] \quad \text{where } \mathit{ptr} \leftarrow a_{n-1} \\
&\mathsf{shift}_{\ll}(e, [a_0, a_1, \ldots, a_{n-1}], \mathit{ptr}) = [a_1, \ldots, a_{n-1}, e] \quad \text{where } \mathit{ptr} \leftarrow a_0
\end{aligned}
$$

Fig. 1. The sequential definition of list skeletons. Updates of values through pointers are denoted by $\mathit{ptr} \leftarrow a$ to make the definition consistent with the implementation in the SkeTo library.

2 Parallel List Skeletons in the SkeTo Library

The parallel skeletons provided in the SkeTo library are computational patterns in the Bird-Meertens Formalism (BMF) [3] that was originally studied for sequential programming. This section defines the parallel list skeletons from two viewpoints: the sequential definition from the user's point of view and the parallel definition from the implementer's point of view. This sections also discuss how to apply fusion transformation to optimize programs with parallel list skeletons.

2.1 Sequential Definition of List Skeletons

Figure 1 shows some of the list skeletons available in the SkeTo library. Users develop their programs based on this sequential definition.

Skeletons $\mathsf{generate}$, map, and zipw are element-wise computational patterns. Skeleton $\mathsf{generate}(f, n)$ returns a list of length n whose elements are the results of the function f applied to the indices $[0, \ldots, n-1]$. Skeleton $\mathsf{map}(f, as)$ applies the function f to each element of the list as. Skeleton $\mathsf{zipw}(f, as, bs)$ applies the function f to each pair of corresponding elements of the lists as and bs.

Skeleton $\mathsf{reduce}(\oplus, as)$ computes the reduction of the list as with the associative binary operator $\oplus$. Skeleton $\mathsf{scan}(\oplus, e, as, \mathit{ptr})$ computes accumulation on the list as from the left to the right (also called prefix-sums) with the associative binary operator $\oplus$. The accumulation starts at e, and the fully accumulated result is returned through the pointer ptr. Skeleton $\mathsf{scanr}(\oplus, e, as, \mathit{ptr})$ accumulates from the right to the left.

Skeleton $\mathsf{shift}_{\gg}(e, as, \mathit{ptr})$ (`shiftr` in the program code) returns a list whose elements are shifted to the right, where the leftmost value is e and the original rightmost value is returned through ptr. Skeleton $\mathsf{shift}_{\ll}(e, as, \mathit{ptr})$ (`shiftl` in the program code) is a shift computation from the right to the left.

```
$\mathsf{generate}(f, n)$
    $= \mathbf{let}\ bs_i = \mathsf{generate}_{\mathrm{local}}(f, \lceil i * n/p \rceil, \lceil (i+1) * n/p \rceil - 1)$    for $i \in [0, p-1]$
      $\mathbf{in}\ [bs_0, \ldots, bs_{p-1}]$

$\mathsf{map}(f, [as_0, \ldots, as_{p-1}])$
    $= \mathbf{let}\ bs_i = \mathsf{map}_{\mathrm{local}}(f, as_i)$    for $i \in [0, p-1]$
      $\mathbf{in}\ [bs_0, \ldots, bs_{p-1}]$

$\mathsf{zipw}(f, [as_0, \ldots, as_{p-1}], [bs_0, \ldots, bs_{p-1}])$
    $= \mathbf{let}\ cs_i = \mathsf{zipw}_{\mathrm{local}}(f, as_i, bs_i)$    for $i \in [0, p-1]$
      $\mathbf{in}\ [cs_0, \ldots, cs_{p-1}]$

$\mathsf{reduce}(\oplus, [as_0, \ldots, as_{p-1}])$
    $= \mathbf{let}\ b_i = \mathsf{reduce}_{\mathrm{local}}(\oplus, as_i)$    for $i \in [0, p-1]$
      $\mathbf{in}\ \mathsf{reduce}_{\mathrm{global}}(\oplus, [b_0, \ldots, b_{p-1}])$

$\mathsf{scan}(\oplus, e, [as_0, \ldots, as_{p-1}], ptr)$
    $= \mathbf{let}\ bs_i = \mathsf{scan}_{\mathrm{local}}(\oplus, \iota_\oplus, as_i, c_i)$    for $i \in [0, p-1]$
            $[d_0, \ldots, d_{p-1}] = \mathsf{scan}_{\mathrm{global}}(\oplus, e, [c_0, \ldots, c_{p-1}], ptr)$
            $es_i = \mathsf{map}_{\mathrm{local}}((d_i \oplus), bs_i)$    for $i \in [0, p-1]$
      $\mathbf{in}\ [es_0, \ldots, es_{p-1}]$

$\mathsf{shift}_{\gg}(e, [as_0, \ldots, as_{p-1}], ptr)$
    $= \mathbf{let}\ b_i = last(as_i)$    for $i \in [0, p-1]$
            $[c_0, \ldots, c_{p-1}] = \mathsf{shift}_{\gg\mathrm{global}}(e, [b_0, \ldots, b_{p-1}], ptr)$
            $ds_i = \mathsf{shift}_{\gg\mathrm{local}}(c_i, as_i, NULL)$    for $i \in [0, p-1]$
      $\mathbf{in}\ [ds_0, \ldots, ds_{p-1}]$
```

Fig. 2. The parallel definition of list skeletons based on the sequential definition given in Figure 1. The subscript "local" indicates that a skeleton is used as a local computation and the subscript "global" indicates that a skeleton is used as a global computation. The definition of $\mathsf{generate}_{\mathrm{local}}$ is a bit different from that in Figure 1. It takes the first and the last indices of the list as input. Function *last* returns the last element of the given list and $\iota_\oplus$ is the unit of the binary operator $\oplus$.

2.2 Parallel Definition of List Skeletons

The SkeTo library is a parallel skeleton library for distributed-memory environments. We adopt the SPMD (Single Program/Multiple Data) computation model in which each process has its own data. In this model, we implement a list as a nested list whose elements are local lists allocated by processes. More concretely, in an environment with p processes we represent a list of n elements $a_0, a_1, \ldots, a_{n-1}$ as follows.

$$[[a_0, \ldots, a_{\lceil n/p \rceil - 1}], \ldots, [a_{\lceil (p-1)*n/p \rceil}, \ldots, a_{n-1}]]$$

The parallel implementation of the list skeletons consists of local computation parts in which each process computes independently with its local lists and of global computation parts for which inter-process communication occurs. Figure 2 shows the definition of the list skeletons for parallel implementation. We omit the definition of scanr (and $\mathsf{shift}_{\ll}$), since it is similar to that of scan (and $\mathsf{shift}_{\gg}$).

Since skeletons generate, map, and zipw are element-wise computational patterns, they can be easily implemented with local computations. Skeleton reduce first performs local reduction on each local list, and then reduces the local results with a global computation. Skeleton scan is implemented in three steps: (1) we compute scan for each local list, (2) compute scan globally on the results of the local scans, and (3) update the results of local scan with local map for each local list. Skeleton $\mathsf{shift}_{\gg}$ is implemented by global $\mathsf{shift}_{\gg}$ applied to the last elements of local lists followed by local $\mathsf{shift}_{\gg}$ applied to each local list.

Note that there exists another three-step implementation of scan that consists of local reduce, global scan, and local scan. This implementation is not used, because applying the fusion transformation to the local scan is, in general, complicated.

2.3 Target of Fusion Optimization

In the skeletal parallelism, users develop parallel programs by composing several skeletons. One potential drawback of such a methodology is the overhead caused by many calls of skeletons with intermediate data passed between skeletons. The fusion transformation is an important optimization technique that removes such overhead, and there have been several studies on this topic [8, 9, 2, 5]. We will review these studies in Section 5.

In the new version of the SkeTo library, we implemented the fusion transformation focusing on realistic and important parts of skeletal programs. The idea is to fuse the local computation parts only, instead of applying the fusion transformation over whole skeletons. As we defined in Figure 2 the skeletons are implemented with local computations and global computations, and we apply the fusion transformation to the consecutive local computations between global computations. Figure 3 shows an example of the targets of the fusion transformation. It is worth noting that almost all the skeletal programs to which the fusion mechanism of the old version of the SkeTo library can be applied to can be optimized. Moreover, programs can be optimized using the scan and shift skeletons.

Now the targets of the fusion transformation are formalized. First, in the implementation of the $\mathsf{shift}_{\gg}$ and $\mathsf{shift}_{\ll}$ skeletons the global shift computation is moved before the local shift computation. Based on this fact and on the definition of skeletons in Figure 2, observe that the local computations between global computations have a specific form: almost element-wise computations (map, zipw, $\mathsf{shift}_{\gg}$, and $\mathsf{shift}_{\ll}$) occur in some order first and then scanning on local lists (reduce or scan) may follow. Therefore, there is a fusion transformation implementation for this specific form. Implementation details are given in Section 3.

The fusion transformation considered here is known as the loop-fusion optimization. It is worth noting that loop-fusion often makes a program faster when the loop computations are rather small. Sometimes, however, loop fusion makes the program slower due to the increased number of registers needed.

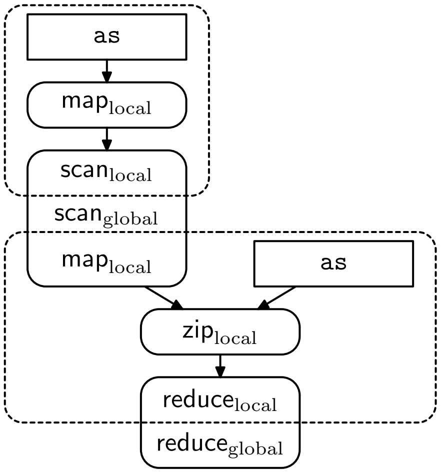

Fig. 3. Two targets of the fusion transformation (denoted by dashed lines). After the fusion transformation, the number of local computations (loops) decreases from 5 to 2. You may think that this example is artificial, but this combination of skeletons comes from parallelization of a very common form of recursive functions on lists [10].

3 Implementation of Parallel List Skeletons in the SkeTo Library

Before discussing the implementation details, example programs to compute the variance of n values $[a_0, \ldots, a_{n-1}]$ (where $a_i = i^5 \bmod 100$) using the following definition below are displayed.

$$\begin{aligned} ave &= \textstyle\sum_{i=0}^{n-1} a_i/n \\ var &= \textstyle\sum_{i=0}^{n-1} (a_i - ave)^2/n \end{aligned}$$

Figure 4 shows a program with simple for-loops, Figure 5 shows a program with the SkeTo library, and Figure 6 shows a program with the STL library.

3.1 Interface

Distributed List Structure. In the SkeTo library, distributed lists are provided as instances of the template class `dist_list`. Data distribution is concealed in the constructors of the `dist_list` class and users do not need to know how the elements of a list are distributed to the processes.

One difference of the new implementation from the previous one is that the real buffer of a distributed list is managed with its reference count in another class `dist_list_buffer` to which the `dist_list` class has a pointer. With this change, we can implement automatic allocation/release of memory and automatic dispatching to specialized skeletons that overwrite the results on the inputs. We illustrate the difference with an example. In the previous version, users had to call `delete` explicitly to release the memory used by the distributed lists as follows:

```
#include <iostream>
using namespace std;
const int n = 10000000;

int main(int, char**) {
  int *as = new int[n];
  double ave = 0;
  for (int i = 0; i < n; ++i) {
    as[i] = i*i*i*i*i % 100;
    ave += as[i];
  } ave /= n;

  double var = 0;
  for (int i = 0; i < n; ++i) {
    var += (as[i]-ave) * (as[i]-ave);
  } var /= n;

  cout << var << endl;
  delete [] as;
}
```

Fig. 4. A program using for-loops

```
#include <iostream>
#include <sketo/sketo.h>
#include <sketo/list_skeletons.h>
const int n = 10000000;

using namespace std;
using namespace sketo;
using namespace sketo::list_skeletons;

struct gen
  : public functions::base<int (int)> {
  int operator()(int i) const {
    return i*i*i*i*i % 100;
  }
};

int sketo::main(int, char**) {
  dist_list<int> as;
  as = generate(n, gen());
  double ave
    = reduce(plus<double>(), as) / n;

  double var
    = reduce(plus<double>(),
        map(functions::square<double>(),
          map(bind2nd(minus<double>(), ave),
            as))) / n;

  sketo::cout << var << endl;
}
```

Fig. 5. A program using the SkeTo library

```
#include <iostream>
#include <vector>
#include <functional>
#include <algorithm>
#include <numeric>
using namespace std;

const int n = 10000000;

struct gen {
  mutable int index;
  gen() : index(0) {};
  double operator()() const {
    const int i = index++;
    return i*i*i*i*i % 100;
  }
};

struct minus_ave_sqr {
  double ave;
  minus_ave_sqr(double ave) : ave(ave) { }
  double operator()(double x) const {
    return (x - ave) * (x - ave);
  }
};

int main() {
  vector<double> as(n);

  generate(as.begin(), as.end(), gen());
  double ave
    = accumulate(as.begin(), as.end(),
                 0.0, plus<double>()) / n;

  transform(as.begin(), as.end(),
            as.begin(),
    minus_ave_sqr(ave));
  double var
    = accumulate(as.begin(), as.end(),
                 0.0, plus<double>()) / n;

  cout << var << endl;
}
```

Fig. 6. A program using STL

```
dist_list<int> *as = new dist_list<int>(array, size);
dist_list<int> *bs = list_skeletons::map(f, as);
  ...
delete bs;
delete as;
```

In the new version of the SkeTo library, the `dist_list` class is responsible for memory management and, thus, programmers can simplify as follows:

```
dist_list<int> as(array, size);
dist_list<int> bs = list_skeletons::map(f, as);
  ...
```

Parallel Skeletons. The parallel list skeletons that manipulate distributed lists are defined in the name space `list_skeletons`. The interfaces of the parallel list skeletons are essentially the same as before.

In the SkeTo library, the argument functions for parallel skeletons are function objects (objects that implement an `operator()` method). With function objects instead of function pointers, compilers can find the concrete definition of the functions and optimize the function calls by inline expansion. The inline expansion works quite well when programs are the composition of several small components. Function objects passed to parallel skeletons are instances of classes that inherit one of the template classes `sketo::functions::base` for the declaration of the types of the arguments and the return value. For example, a function object that takes a value of type `A` and returns a value of type `B` should inherit the template class `sketo::functions::base<B (A)>`. These base classes are implemented in a similar way to the `boost::function`. The reason for reimplementation is that `boost::function` cannot be inline-expanded due to its implementation. For the same reason, anonymous functions, `boost::lambda`, have problems of efficiency. It is worth noting that the function objects provided by `<functional>` in STL are available in SkeTo.

The most important change to the interface of parallel list skeletons is that we only provide a single function for each skeleton. In the previous implementation, we provided two or more functions for a skeleton. For example, for the `map` skeleton there were three functions: normal `map` function with two arguments, `map` function with three arguments, and specialized implementation `map_ow` for overwriting. In the new version, we unify those implementations into a single interface. In fact, based on the reference count in `dist_list_buffer` and the expression template technique, the library dispatches skeleton calls to specific implementations. The details of the implementation with expression templates are shown in the next subsection.

To illustrate the differences consider the following example. In the previous version, to overwrite the results of `map` onto its inputs a specialized version of the `map` skeleton, namely `map_ow`, had to be used as follows:

```
list_skeletons::map_ow(f, as);
list_skeletons::map_ow(g, as);
v = list_skeletons::reduce(plus, 0, as);.
```

With the new version the code is written as follows:

```
as = list_skeletons::map(f, as);
as = list_skeletons::map(g, as);
v = list_skeletons::reduce(plus, as);.
```

The library automatically selects the specialized code. Furthermore, since the result of the `map` is `dist_list`, we can also write it in the following nested way:

```
v = list_skeletons::reduce(plus,
      list_skeletons::map(g,
        list_skeletons::map(f, as)));.
```

3.2 Optimization Mechanism by Expression Templates

The new version of the SkeTo library uses expression templates [7] to implement fusion transformations and uses overwriting for memory reuse. This section introduces the expression template technique and illustrates how the optimization mechanisms are implemented.

Expression Templates. This subsection introduces the meta-programming technique called expression templates [7]. This technique has been used to implement efficient libraries for linear-algebraic computations [11] and for domain-specific regular expressions.

When an expression in C++ is evaluated, the sub-expressions are evaluated one by one. Consider evaluating the following code where the variables `A`, `B`, `C` and `D` are vectors:

```
D = A + B - C;
```

Usually, the sub-expression `A + B` is evaluated first which generates, `E`, an intermediate vector. Then the subtraction, `E - C`, is computed which also generates, `F`, another intermediate vector. Finally, the vector `F` is assigned to `D`.

The use of the expression template technique generates certain structures representing the computation (often tree structures like abstract syntax trees) and delay the computation until the results are required. By delaying the computation, efficient code can be generated for the whole expression. In the example above, an instance of template type `plus<vec,vec>` is generated for the sub-expression `A + B`, then the right-hand side of the expression is given as an instance of template type `minus<plus<vec,vec>,vec>`, and finally the member function

```
vec::operator=(minus<plus<vec,vec>,vec>)
```

is called. Proper code is generated for these member functions with the help of the template structures. In the example above, a fast implementation corresponding to the following loop is generated:

```
for (int i = 0; i < n; i++) { D[i] = A[i] + B[i] - C[i]; }.
```

Implementation of Fusion Transformation. In Section 2.3, we stated that the target of the fusion transformation is a set of consecutive local computations between global computations. Due to the lack of type inference in C++, in the new version of the SkeTo, we apply the fusion transformation to local computations that are written as a single expression. For example, in the following code from Figure 5

```
double var
  = reduce(plus<double>(),
      map(functions::square<double>(),
        map(bind2nd(minus<double>(), ave), as))) / n;
```

the two `maps` and the `reduce` are the target of the fusion transformation.

In the implementation, template types for local map, local zipw, local $\mathsf{shift}_{\gg}$, and so on are defined. For example, the template object for local map, `ls_mapobj`, is defined as follows:

```
template <typename F, typename AS>
struct ls_mapobj {
  F f;
  const AS as;
  ls_mapobj(const F &f, const AS &as) : f(f), as(as) { }

  typedef typename F::result_type element_type;
  element_type local_get(int i) const { return f(as.local_get(i)); }

  ...
};.
```

This template object stores the function object and the argument list, and the computation of the map skeleton is executed in the member function `local_get`. The map skeleton just generates this template object as follows:

```
template <typename F, typename AS>
_impl::ls_mapobj<F, AS> map(F f, const AS& as) {
  return _impl::ls_mapobj<F, AS>(f, as);
}.
```

The computation of a parallel skeleton is delayed until either reduce, scan, scanr or an assignment to a list occurs. For example, in the implementation of reduce, the computation of skeletons is triggered through the member function `local_get` as shown in the following code:

```
A result = as.local_get(0);
{
  const int n = as.get_local_size();
  for (int i = 1; i < n; ++i) {
    result = oplus(result, as.local_get(i));
  }
}.
```

To illustrate how the fusion transformation with expression templates is done consider, once again, the sample code above. The skeleton `reduce` takes a value of type

```
ls_mapobj<G,ls_mapobj<F,dist_list<double> > >
```

where `F` represents the type of `bind2nd(minus<double>(),ave)`, and `G` represents the type of `functions::square<double>()`. Then, in the computation of `reduce`, the local computation represented by two `ls_mapobj`s are fused to the local reduction because the `local_get` functions of the two `ls_mapobj` are called in a nested way. After fusion optimization and inline expansion, the main loop of the generated code becomes the same as the following simple loop:

```
double result = (as[0] - ave) * (as[0] - ave);
for (int i = 1; i < n; ++i) {
  result = result + (as[i] - ave) * (as[i] - ave);
}.
```

The performance effects of the fusion transformation are discussed in Section 4.

The current implementation of the fusion transformation for programs including $\mathsf{shift}_{\gg}$ or $\mathsf{shift}_{\ll}$ has room for improvement. For example, for the following code

```
bs = map(f, shiftr(e, as));
```

the current implementation generates code corresponding to the following loop.

```
for (int i = 0; i < n; ++i) {
  bs[i] = f( (i==0) ? e : as[i-1] );
}
```

However, the following loop is faster in many cases.

```
bs[0] = f(e);
for (int i = 1; i < n; ++i) {
  bs[i] = f( as[i-1] );
}
```

This improvement of the fusion transformation is a part of our future work.

Implementation of Specialized Skeletons. Overwriting the results of parallel skeletons onto their inputs is an important optimization in terms of memory consumption and the cost of memory allocation/release. The implementation of this optimization is also attained by expression templates.

A single line of the skeletal programs usually has the following form.

`as =` *skeleton_calls*;

With the expression templates, the right-hand side *skeleton_calls* forms a tree structure representing the skeleton calls. The following template member function was added to `dist_list`

```
template <typename BS> void operator=(const BS &bs);
```

and the dispatch mechanism was implemented in the member function.

The results of skeletons are overwritten when all the following conditions hold (where `as` denotes the distributed list on the left-hand side):

1. The buffer is already allocated for `as`.
2. The length of `as` is the same as that of resulting list of the right-hand side.
3. The reference count in `as` is greater by one than the number of occurrences of `as` on the right-hand side. Note that the reference count of `as` increases if it appears on the right-hand side, and this condition means that `as` does not share the array with other variables.
4. The tree structure has no $\texttt{shift}_{\gg}/\texttt{shift}_{\ll}$ applied to `as` except for the root.

Note that in condition 4 we permit the inclusion of `scan` and `scanr` because they allocate another distributed list.

Revealing Errors in Programs Developed with Expression Templates. Expression templates are an important programming technique for implementing efficient libraries. However, a problem occurs when we use expression templates for implementing a skeleton library: unreadable error messages appear when we fail to compile template programs. It is worth noting that the following discussion is relevant to the use of GCC. The Intel Compiler checks errors before expanding expression templates and, thus, the following "tricks" are unnecessary.

When using expression templates for linear-algebraic computations, the primary operators used in user programs are `+` and `*` and, as such, programs have fewer errors. However, in skeletal parallel programming, users can specify any function for parallel skeletons and thus user programs tend to have errors. For example, in the code displayed in Figure 5, a programmer may mistakenly pass a unary function, like `square<double>()`, as the first argument to `bind2nd`. This single mistake causes 20 lines of error messages. A sample error message line is a line:.

```
sketo/list_skeletons_with_fusion.h: In member function 'typename F::
result_type sketo::_impl::ls_mapobj<F, AS>::local_get(int) const [
with F = sketo::functions::square<double>, AS = sketo::_impl::
ls_mapobj<std::binder2nd<sketo::functions::square<double> >, sketo::
dist_list<double> >]': .
```

Note that this error is detected inside the library code even though the bug is in the user code. Users not familiar with the implementation details of the SkeTo library cannot determine the reason for this annoying error message.

To resolve this problem, we provide another implementation of the SkeTo library that does not optimize skeletal programs by expression templates. Users can easily switch the implementations: defining a macro `__SKETO_NO_FUSION__` at the preprocessing stage is enough and no change to the program code is needed. The bug can easily be found with this alternative library implementation. For the above example, the error messages are reduced to 13 lines and the bug can directly be identified in:

```
variance.cpp:26: error: no matching function for call to 'map(std::
binder2nd<sketo::functions::square<double> >, sketo::dist_list<
double>&)'
```

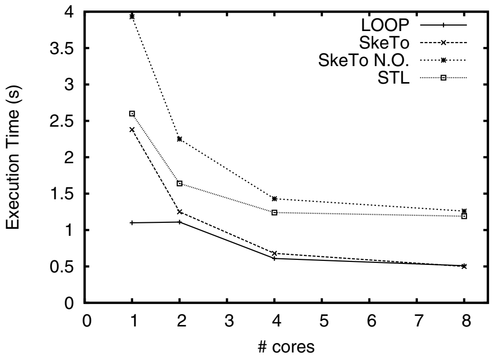

Fig. 7. The execution time for computing variance

Table 1. The execution time for computing variance (in seconds)

#cores	1	2	4	8
LOOP	1.10	1.11	0.61	0.51
SkeTo	2.38	1.25	0.68	0.50
SkeTo N.O.	3.93	2.25	1.43	1.26
STL	2.60	1.64	1.24	1.19

4 Experiments

To evaluate the performance of the SkeTo library, experiments with computing variance (Figures 4, 5, and 6), the bracket matching problem [10], and the N-queen problem were conducted.

Variance computation was used to evaluate the sequential performance, the speed-up, and the overhead of the parallel list skeletons. In the experiments, a list of length 200,000,000 was used. The experiments were carried out on a desktop PC with two Intel Xeon E5430 (2.66GHz, quad-cores) CPUs and 8 GByte memory. The compiler and MPI library were GCC 4.4.0 and mpich 1.2.7p1. Figure 7 and Table 1 show the results of experiments. LOOP indicates the program with simple loops in Figure 4 parallelized with OpenMP, SkeTo indicates the program with the SkeTo library in Figure 5, SkeTo N.O. indicates the same program as SkeTo but no optimization is applied, and STL indicates the program with STL in Figure 6 parallelized with GCC libstdc++ parallel mode [12]. With the fusion optimization using expression templates, the program utilizing SkeTo is optimized so that it achieves almost the same performance as the simple for-loops with OpenMP. It is worth noting that the SkeTo library and the program in Figure 5 are also available on distributed-memory environments. The program without fusion optimization and the program with STL are slower due to the overhead caused by multiple list traversals. Note that the relatively small speedups in this example are due to memory bandwidth saturation.

The bracket matching problem [10] was used to investigate the effects of the fusion optimization using expression templates. The outline of the skeletal program for this problem is the same as that in Figure 3, but the concrete function objects for skeletons are a bit more complicated. The complete definition can be found in [10]. The main part of the program is the first `map` and `scan`: the function g_2' for `map` is

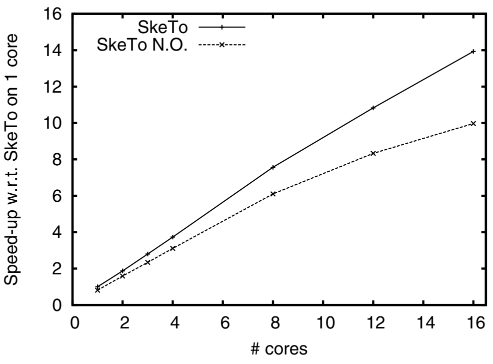

Fig. 8. The speedup for the bracket matching problem

Table 2. The execution time for the bracket matching problem (in seconds)

# core	1	4	8	16
SkeTo	12.26	3.29	1.62	0.88
SkeTo N.O.	15.21	3.94	2.01	1.23

$$\begin{aligned} g'_2(a) = &\ \mathbf{if}\ \mathit{isOpen}(a)\ \mathbf{then}\ ([a],1,0) \\ &\ \mathbf{elseif}\ \mathit{isClose}(a)\ \mathbf{then}\ ([\,],0,1) \\ &\ \mathbf{else}\ ([\,],0,0) \end{aligned}$$

and the operator $\otimes$ for scan is

$$\begin{aligned} (cs_1,n_1,m_1)\otimes(cs_2,n_2,m_2) = &\ \mathbf{if}\ m_1 \geq n_2\ \mathbf{then}\ (cs_1,n_1,m_1-n_2+m_2) \\ &\ \mathbf{else}\ (cs_1 \mathbin{+\!\!+} \mathit{drop}(m_1,cs_2), n_1+n_2-m_1, m_2) \end{aligned}$$

where the operator $+\!\!+$ concatenates two lists and the function $\mathit{drop}(m_1, cs_2)$ drops the first m_1 elements from the list cs_2.

In the experiment, the string length is 100,000,000, the different types of brackets is 4, and the maximum nesting of brackets is 10. The hardware environment is a cluster of four PCs with an Intel Core2Quad 2.4GHz CPU and 4 GByte memory connected with Gigabit Ethernet. The compiler and library used are Intel C++ Compiler 9.0 and MPICH 1.2.7p1. The optimized version and the non-optimized version of the skeletal program were executed varying the number of cores from 1 to 16. Table 2 shows the results of the experiments and Figure 8 plots the speed-up with respect to the execution of the optimized version on one core. The optimized version is 20% faster than the non-optimized version independent of the number of cores.

Finally, experiments to evaluate the scalability of the SkeTo library using the 18-Queens problem are presented. The hand-written code using MPI and C developed by Kise et al. [13] (qn24b) and a program with list skeletons using the SkeTo library (SkeTo) are used as benchmarks. The environment is a cluster of PCs with dual Xeon 2.4GHz CPUs and 2GByte memory connected with Gigabit Ethernet, GCC 4.1.2, and mpich 1.2.7p1. Figure 9 and Table 3 show the empirical results.

From these results, we can see that both programs achieve good speedups. In this example, nonoptimized version runs as fast as the optimized version, since

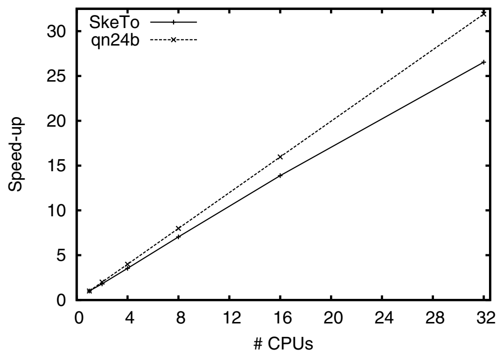

Fig. 9. The speed-up for 18-Queens problem

Table 3. The execution time for 18-Queens problems (in seconds)

#CPU	1	4	16	32
SkeTo	554	157	39.9	20.9
qn24b	596	149	37.3	18.7

almost all the execution time is spent in a single map skeleton. The program with the SkeTo library shows a bit worse scalability. This is due to the static scheduling policy of the SkeTo library: the program with the SkeTo library runs a bit faster on 1 CPU, but the loads may be ill-balanced on many CPUs. In the future version of the SkeTo library we would like to integrate dynamic load-balancing.

5 Related Work

Parallel Skeleton Libraries

Several parallel skeleton libraries have been implemented. Since the idea of skeletal parallel programming is closely related to functional programming, there are several implementations based on functional languages such as Haskell [14], Template Haskell [15], and SML [16].

There are also several implementations or widely used imperative languages like C, C++, and Java developed for efficiency reasons. For example, Muskel [17] and eSkel [18] provide parallel skeletons mainly for task-parallel computations; Muesli [19] provides a two-tier model of task- and data-parallel skeletons; Intel TBB (Thread Building Blocks) [20] is now being widely used for multicore parallel programming. Among these, Muesli is the skeleton library most related to the SkeTo library. It provides data-parallel skeletons for lists and matrices implemented in C++ and MPI (the new version of Muesli also uses OpenMP) and some task-parallel skeletons. Compared with Muesli, the SkeTo library offers the advantages of matrix skeletons that are defined based on the theory of constructive algorithmics [21], tree skeletons [22], and the optimization mechanism based on the fusion transformation.

Instead of developing a new library, providing a parallel implementation to an existing standard library is another approach to skeletal parallelism. For example, DatTel [23] and MCSTL [12] are parallel implementations for STL (the standard template library) in C++. In particular, the latter is now integrated into GCC with the name "libstdc++ parallel mode," and is used in the experiments in Section 4.

Optimization of Skeletal Programs by Fusion Transformation

The fusion transformation is an important optimization technique that removes overhead caused by too many skeleton calls with intermediate data between them. There have been several studies on this topic [2,5,8,9,24,25].

In the framework proposed by Aldinucci et al. [8], many transformation rules are used to optimize skeletal programs. They considered not only simple rules like map-map fusion, but also complex rules like fusing scan and reduce under certain conditions on operators. The number of optimization rules, however, easily becomes too large and it is unrealistic to implement all of them.

In the previous version of the SkeTo library [5], we implemented a fusion optimization mechanism based on normal forms that characterize data generation/consumption. The fusion rules on those normal forms were proposed in [2]. Though this method is simple and rather easy to implement, the fusion transformation often fails for `scan` and `shift` skeletons. Note that the fusion optimization by the expression templates covers almost all the cases that the fusion mechanism in the old version can be applied to.

Single assignment C (SAC) [25] is a programming language with high level array operations. The SAC compiler has a powerful fusion optimization mechanism called with-loop-folding [24], which combines consecutive array operations into a singe one. The basic idea for the fusion transformation is almost the same: to fuse almost element-wise computations. Since the optimization was implemented in the SAC compiler, it supports more powerful optimizations such as high-dimensional arrays, more complicated data movement, and changing the size of arrays.

Expression Templates

In the new implementation of the SkeTo library, we implemented the fusion transformation by the expression template technique. Expression templates are often used in efficient implementations for linear algebraic computation and in domain-specific computations such as those for regular expressions. For the linear algebraic computation, Blitz++ [11] and the uBLAS library in the Boost library[1] are two well-known implementations. As a research-level implementation, NT2 [26] implemented several nontrivial optimizations for parallel linear-algebraic computation with expression templates.

6 Conclusion

This paper discusses the new design and implementation of the parallel list skeletons of the SkeTo library. Based on the parallel definition of the list skeletons, we

[1] `http://www.crystalclearsoftware.com/cgi-bin/boost_wiki/wiki.pl?Effective_UBLAS`

formalized the target of the fusion transformation as consecutive local computations between global computations. The optimization mechanism in the new version of the SkeTo library was implemented using expression templates. The presented experiments confirm the good performance of the SkeTo library and, in particular, the good performance of the fusion optimization implemented in the new library.

As we stated in Section 3.2, the results of the fusion transformation for the computations with shift skeletons are not the best ones. Emoto et al. [9] proposed an optimization method for those computations with shift skeletons. Implementing this optimization for the SkeTo library is a part of our future work.

References

1. Cole, M.: Algorithmic Skeletons: Structural Management of Parallel Computation. Research Monographs in Parallel and Distributed Computing. MIT Press, Cambridge (1989)
2. Hu, Z., Iwasaki, H., Takeichi, M.: An accumulative parallel skeleton for all. In: Le Métayer, D. (ed.) ESOP 2002. LNCS, vol. 2305, pp. 83–97. Springer, Heidelberg (2002)
3. Bird, R.S.: An introduction to the theory of lists. In: Logic of Programming and Calculi of Discrete Design. NATO ASI Series F, vol. 36, pp. 5–42. Springer, Heidelberg (1987)
4. Matsuzaki, K., Iwasaki, H., Emoto, K., Hu, Z.: A library of constructive skeletons for sequential style of parallel programming. In: InfoScale 2006: Proceedings of the 1st international conference on Scalable information systems. ACM International Conference Proceeding Series, vol. 152. ACM Press, New York (2006)
5. 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. LNCS, vol. 3149, pp. 644–653. Springer, Heidelberg (2004)
6. Chiba, S.: A metaobject protocol for C++. In: Proceedings of OOPSLA 1995, Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications. SIGPLAN Notices, vol. 30, pp. 285–299. ACM Press, New York (1995)
7. Veldhuizen, T.L.: Expression templates. C++ Report 7(5), 26–31 (1995); Reprinted in Lippman, S. (ed.): C++ Gems
8. Aldinucci, M., Gorlatch, S., Lengauer, C., Pelagatti, S.: Towards parallel programming by transformation: the FAN skeleton framework. Parallel Algorithms and Applications 16(2-3), 87–121 (2001)
9. Emoto, K., Matsuzaki, K., Hu, Z., Takeichi, M.: Domain-specific optimization strategy for skeleton programs. In: Kermarrec, A.-M., Bougé, L., Priol, T. (eds.) Euro-Par 2007. LNCS, vol. 4641, pp. 705–714. Springer, Heidelberg (2007)
10. Hu, Z., Takeichi, M., Iwasaki, H.: Diffusion: Calculating efficient parallel programs. In: Proceedings of the 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (1999)
11. Veldhuizen, T.L.: Arrays in Blitz++. In: Caromel, D., Oldehoeft, R.R., Tholburn, M. (eds.) ISCOPE 1998. LNCS, vol. 1505, pp. 223–230. Springer, Heidelberg (1998)
12. Singler, J., Sanders, P., Putze, F.: The multi-core standard template library. In: Kermarrec, A.-M., Bougé, L., Priol, T. (eds.) Euro-Par 2007. LNCS, vol. 4641, pp. 682–694. Springer, Heidelberg (2007)

13. Kise, K., Katagiri, T., Honda, H., Yuba, T.: Solving the 24-queens problem using MPI on a PC cluster. Technical Report UEC-IS-2004-6, Graduate School of Information Systems, The University of Electro-Communications (2004)
14. Klusik, U., Loogen, R., Priebe, S., Rubio, F.: Implementation skeletons in Eden: Low-effort parallel programming. In: Mohnen, M., Koopman, P. (eds.) IFL 2000. LNCS, vol. 2011, pp. 71–88. Springer, Heidelberg (2001)
15. Hammond, K., Berthold, J., Loogen, R.: Automatic skeletons in Template Haskell. Parallel Processing Letters 13(3), 413–424 (2003)
16. Scaife, N., Horiguchi, S., Michaelson, G., Bristow, P.: A parallel SML compiler based on algorithmic skeletons. Journal of Functional Programming 15(4), 615–650 (2005)
17. Aldinucci, M., Danelutto, M., Dazzi, P.: Muskel: an expandable skeleton environment. Scalable Computing: Practice and Experience 8(4), 325–341 (2007)
18. Benoit, A., Cole, M., Gilmore, S., Hillston, J.: Flexible skeletal programming with eSkel. In: Cunha, J.C., Medeiros, P.D. (eds.) Euro-Par 2005. LNCS, vol. 3648, pp. 761–770. Springer, Heidelberg (2005)
19. Kuchen, H.: A skeleton library. In: Monien, B., Feldmann, R.L. (eds.) Euro-Par 2002. LNCS, vol. 2400, pp. 620–629. Springer, Heidelberg (2002)
20. Reinders, J.: Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism. O'Reilly Media, Inc., Sebastopol (2007)
21. Emoto, K., Hu, Z., Kakehi, K., Takeichi, M.: A compositional framework for developing parallel programs on two-dimensional arrays. International Journal of Parallel Programming 35(6), 615–658 (2007)
22. Matsuzaki, K.: Efficient implementation of tree accumulations on distributed-memory parallel computers. In: Shi, Y., van Albada, G.D., Dongarra, J., Sloot, P.M.A. (eds.) ICCS 2007, Part II. LNCS, vol. 4488, pp. 609–616. Springer, Heidelberg (2007)
23. Bischof, H., Gorlatch, S., Leshchinskiy, R.: Generic parallel programming using C++ templates and skeletons. In: Lengauer, C., Batory, D., Consel, C., Odersky, M. (eds.) Domain-Specific Program Generation. LNCS, vol. 3016, pp. 107–126. Springer, Heidelberg (2004)
24. Scholz, S.-B.: With-loop-folding in SAC — condensing consecutive array operations. In: Clack, C., Hammond, K., Davie, T. (eds.) IFL 1997. LNCS, vol. 1467, p. 72. Springer, Heidelberg (1998)
25. Scholz, S.B.: Single assignment C — efficient support for high-level array operations in a functional setting. Journal of Functional Programming 13(6) (2003)
26. Falcou, J., Sérot, J., Pech, L., Lapresté, J.T.: Meta-programming applied to automatic SMP parallelization of linear algebra code. In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008. LNCS, vol. 5168, pp. 729–738. Springer, Heidelberg (2008)