SkeTo: Parallel Skeleton Library
Manual for Version 1.0

SkeTo Project
Web site: `http://www.ipl.t.u-tokyo.ac.jp/sketo/`
Contact email: `sketo@ipl.t.u-tokyo.ac.jp`

December 2, 2009

Abstract

SkeTo (Skeletons in Tokyo) is a constructive parallel skeleton library written in C++ with MPI intended for distributed environments such as PC clusters. SkeTo provides data parallel skeletons for lists, matrices, and trees. (The version 1.0 only include parallel skeletons for lists, but other parallel skeletons will be included in the later versions.) SkeTo enables users to write parallel programs as if they were sequential, since the distribution, gathering, and parallel computation of data are concealed within constructors of data types or definitions of parallel skeletons.

This document consists of two parts. The first part provides tutorials. After showing how to install the SkeTo library, we demonstrate programming with the SkeTo library through an example. The second part is the reference manual.

Contents

Chapter 1

Overview

SkeTo (Skeletons in Tokyo) is a constructive parallel skeleton library written in C++ with MPI intended for distributed environments such as PC clusters. SkeTo provides data parallel skeletons for lists (distributed one-dimensional arrays), matrices (distributed two-dimensional arrays), and trees (distributed binary trees)[1]. SkeTo enables users to write parallel programs as if they were sequential, since the distribution, gathering, and parallel computation of data are concealed within constructors of data types or definitions of parallel skeletons. SkeTo is named after the Japanese word *Suketto*, whose meaning is helper or supporter, in the hope that SkeTo library will help programmers easily develop efficient parallel programs.

The SkeTo library is the results of the research in the “SkeTo Project”, which is a research project working on skeletal parallelism (or algorithmic skeletons). The members of the SkeTo project are from The University of Tokyo, The University of Electro-Communications in Japan, National Institute of Informatics, and Kochi University of Technology. The SkeTo project has been partially supported by: HPC Systems Inc., PRESTO program by Japan Science and Technology Agency (JST), Grant-in-Aid for Scientific Research (B), No. 17300005, Japan Society for the Promotion of Science, and Grant-in-Aid for Scientific Research (C), No. 20500029, Japan Society for the Promotion of Science.

1.1 Quick Start

You can install the SkeTo library by the following four steps.

1. Install a C++ compiler (e.g., GCC) and an MPI library (e.g., mpich).

2. Download an archive of the source files (SkeTo-x.xx.tar.gz or SkeTo-x.xx.zip) from the website of the SkeTo project (http://www.ipl.t.u-tokyo.ac.jp/sketo/download.html) and extract the files.

3. Configure the package for your system by the following command.

   ```
   ./configure
   ```

   You can specify the place to which the SkeTo library is installed by `--prefix` option. You can also specify the C++ compiler and the MPI library you want to use. For details, please see the help by “`./configure --help`”.

[1]The version 1.0 only include parallel skeletons for lists, but other parallel skeletons will be included in the later versions.

4. Compile the package and install the files.

   ```
   make && make install
   ```

   The library file (`lib/libsketo.a`), header files (in directory `include/sketo`), and scripts (`bin/sketocxx` and `bin/sketorun`) will be installed by this command.

For more details, please see the INSTALL file included in the archive.

You can try the SkeTo library with several examples included in the directory `samples` of the package. For example, you can compile the program `variance.cpp`[2] by the following command (You may need to specify the full-path to the `sketocxx` script installed so far). The script `sketocxx` invokes C++/MPI compiler with the proper options for the SkeTo library.

```
sketocxx -O2 -o variance variance.cpp
```

Then you can execute the file by the `sketorun` script. For example, if you want to execute it with four processes, you type as follows. Note that some options may be different on your MPI library.

(For mpich user) `sketorun -np 4 variance 10 1`

(For mpich2 user: after executing `mpd`) `sketorun -n 4 variance 10 1`

The script `sketorun` starts the program with MPI library.

1.2 Tutorial: Computing Variance

Variance is the average of the square derivation. Assume that the input data are given as an array $[a_0, a_1, \ldots, a_{n-1}]$, then the mathematical definition of the variance is given as follows.

$$var = \frac{1}{n}\sum_{i=0}^{n-1}(a_i - ave)^2 \quad \text{where } ave = \frac{1}{n}\sum_{i=0}^{n-1} a_i$$

A simple translation of the above definition into C++ program yields the following sequential program.

```
int main(int, char**) {
  // ... initialization of the input array a[] ...

  double sum = 0;
  for (int i = 0; i < n; i++) {
    sum += a[i];
  }
  double ave = sum / n;
  double sqsum = 0;
  for (int i = 0; i < n; i++) {
    sqsum += (a[i] - ave) * (a[i] - ave);
  }
  double var = sqsum / n;
  std::cout << var << std::endl;
  return 0;
}
```

Now we develop a parallel program for computing variance based on the sequential program above.

[2]You can find this in the directory `samples/list`.

Outline of Program In this example, we use parallel list skeletons that manipulate distributed arrays in parallel. To use them, you first need to include `list_skeletons.h` file. A program that uses the SkeTo library usually starts from the `sketo::main` function. The `sketo::main` function takes two arguments of type `int` and `char**` as the usual `main` function does. Thus, the outline of the program with the SkeTo library becomes as follows.

```
#include <list_skeletons.h>

int sketo::main(int, char**) {
  // ... initialization of the input array a[] ...

  // ... data distribution ...
  // ... parallel computation ...
  // ... output result to console ...
  return 0;
}
```

Then, we fill the three missing parts of this program.

Data Distribution The SkeTo library provides classes for distributed data (Currently, the SkeTo version 1.00 only provides distributed list structure). A distributed array is given as an instance of the `dist_list` class. In this example, we use a constructor that takes a sequential array.

```
  sketo::dist_list<int> da(a, n);
```

The elements of `a` are distributed to processes in this constructor, and we need not be aware of the data distribution.

Parallel Computation We then manipulate distributed lists with parallel list skeletons. The definition of the parallel list skeletons is given in Section 2.2. In this example, we use the following two parallel list skeletons:

- map: a parallel skeleton that applies a given function to each element of the list, and
- reduce: a parallel skeleton that computes the summation with a given associative binary operator.

The functions or operators for parallel skeletons should be function objects (a function object is an instance of a class/structure that implements `operator()` method). You can also use function objects defined in the STL `<functional>` and in `<sketo/functions.h>`.

An example code is given as follows[3].

```
  double ave = sketo::list_skeletons::reduce(std::plus<double>(), da) / size;

  da = sketo::list_skeletons::map(std::bind2nd(std::plus<double>(), -ave), da);
  da = sketo::list_skeletons::map(sketo::functions::square<double>(), da);
  double var = sketo::list_skeletons::reduce(std::plus<double>(), da) / size;
```

[3]In the following program, `std::plus<double>()` is a function object that takes two doubles and returns the sum of them, `std::bind2nd(std::plus<double>(), -ave)` is a function object that is equivalent to the function defined as `double f(double x) { return x - ave; }`, `sketo::functions::square<double>()` is a function that returns the squared value of the input. The list of function objects defined in `<sketo/functions.h>` is given in Section 2.3

We can also simplify this code by using the default namespaces as follows (or with the aliases of namespaces).

```
using namespace sketo::list_skeletons;
using namespace sketo::functions;

double ave = reduce(std::plus<double>(), da) / size;

da = map(std::bind2nd(std::plus<double>(), -ave), da);
da = map(square<double>(), da);
double var = reduce(std::plus<double>(), as) / size;
```

Output Result to Console Since the SkeTo library is based on the MPI library, if we use `std::cout` the output is repeated by the number of processors. Therefore, we use `sketo::cout` instead of `std::cout` for the output to console as follows.

```
sketo::cout << "variance: " << var << std::endl;
```

Note that we use `std::endl` (not `sketo::endl`) when we output a newline.

Chapter 2

Reference Manual

2.1 Class dist_list: Distributed List Structure

The class dist_list provides containers for distributed lists. This class is a template class dist_list⟨A⟩ where A denotes the type of elements of a distributed list. This class provides only a small number of methods for its manipulation. Instead, users can manipulate distributed lists by using parallel list skeletons defined in the namespace sketo :: list_skeletons, which are listed in Section 2.2.

The following code shows some methods provided in the dist_list class.

```
int array[] = {10, 11, 12, 13, 14, 15, 16, 17};
sketo::dist_list<int> as(8, array);
sketo::cout << as.get(6) << std::endl;  // This outputs 16.

as.set(5, 20);
sketo::cout << as.get(5) << std::endl;  // This outputs 20.
```

In the background of the dist_list class, the elements are distributed to processes. As you see in the example above, when you pass a usual sequential array to the constructor it implicitly distributes the elements and you do not need to know how the data are distributed. You can use the gather method to restore a sequential array from a distributed list. The methods get and set implicitly do communications among processes.

One important thing to be noted is that the distributed lists by the dist_list class have semantics different from usual vector or list in the STL. The semantics of dist_list is similar to that of an array in Java. An instance of the dist_list class has a reference to a concrete distributed list. Allocation or deallocation of memory are performed automatically based on the reference counting technique. Note that when they are copied only their references are copied, and if you want one distributed list that is allocated independently from another you should use the clone method explicitly. For example, look at the following code; the distributed lists as and bs point the same concrete distributed list while the distributed list cs points a different one.

```
sketo::dist_list<int> as(1);
as.set(0, 10);              // Generate a distributed list of one element.

sketo::dist_list<int> bs = as;
sketo::dist_list<int> cs = as.clone();

as.set(0, 20);

sketo::cout << as.get(0) << std::endl; // This outputs 20.
sketo::cout << bs.get(0) << std::endl; // This outputs 20.
sketo::cout << cs.get(0) << std::endl; // This outputs 10.
```

2.1.1 Constructors

Type Signature:

```
template ⟨typename A⟩
dist_list⟨A⟩ :: dist_list();

template ⟨typename A⟩
dist_list⟨A⟩ :: dist_list(int  size);

template ⟨typename A⟩
dist_list⟨A⟩ :: dist_list(int       size,
                          const A*  array);
```

The first constructor is the default constructor. This constructor does nothing and no distributed list is allocated here.

The second constructor takes an integer *size* and allocates a distributed list of the size. The values of the elements will not be initialized by this constructor.

The third constructor takes an integer *size* and an array *array*. This constructor allocates a distributed list of the size and the elements are initialized with those of *array*.

In addition to these constructors, we provide copy constructors. The copy constructor has semantics different from that of STL containers and much similar to that of arrays in Java as we stated above.

2.1.2 get_global_size

Type Signature:

```
template ⟨typename A⟩
int dist_list⟨A⟩ :: get_global_size() const;
```

This method returns the (whole) length of the distributed list.

2.1.3 get

Type Signature:

```
template ⟨typename A⟩
A dist_list⟨A⟩ :: get(int  index) const;
```

This method returns the value of the element at the index *index* of the distributed list.

2.1.4 set

Type Signature:

```
template ⟨typename A⟩
void dist_list⟨A⟩ :: set(int  index,
                         A    value);
```

This method sets the value *value* at the index *index* of the distributed list.

2.1.5 clone

Type Signature:

```
template ⟨typename A⟩
dist_list⟨A⟩ dist_list⟨A⟩ :: clone() const;
```

This method generates another copy of the distributed list.

As we stated above, the semantics of the dist_list is similar to that of the array in Java. We need to use this method if we want an independent distributed list from another.

2.1.6 gather

Type Signature:

```
template ⟨typename A⟩
void dist_list⟨A⟩ :: gather(A* buffer) const;
```

This method copies the value of the elements of the distributed list to the array *buffer*. User should allocate enough space (i.e., the size given by get_global_size) for the array *buffer*.

> **Detail:** For reasons of performance, a user can pass the value *NULL* for *buffer* on some processes. When the value of *buffer* is *NULL* on a process, then the values of the distributed list will not be copied to the process. This feature is effective when the gathered values are used only on some processes, for example when we output the distributed list from the root process.

2.2 Namespace list_skeletons: Skeletons for Distributed Lists

The namespace list_skeletons provides parallel list skeletons that manipulate distributed lists in parallel. The list skeletons provided are categorized as follows.

- Data generation: generate
- Apply-to-all (map) and its variants: map, map_with_index, zip, zipwith
- Reduction: reduce
- Prefix-sums (scan) and its variants: scan, postscan, gsacnl
- Suffix-sums (scanr) and its variants: scanr, postscanr, gsacnr
- Shift operations: shiftl, shiftr

In this section, we use two notations for showing informal definition of the parallel list skeletons: one is mathematical definition and the other is C-like program. The type signature is provided for explanation and is different from the actual definition in the library. This is due to the optimization mechanism with C++-template programming technique. We use the notation like Fobj$\langle C(A,B)\rangle$ for the type of function objects (Fobj$\langle C(A,B)\rangle$ denotes the type of a function object that takes two arguments of types A and B and returns a value of type A).

2.2.1 generate

Type Signature:

$$\begin{array}{lll} \texttt{template}\ \langle\texttt{typename}\ A\rangle & & \\ \mathsf{dist\_list}\langle A\rangle\ \ \mathsf{generate}(\mathit{int} & & n, \\ \qquad\qquad\qquad\ \texttt{Fobj}\langle A\ (\mathit{int})\rangle & & f); \end{array}$$

Mathematical Definition:

$$\begin{array}{l} [a_0, a_1, \ldots, a_{n-1}] = \mathsf{generate}(n, f) \\ \qquad \Longrightarrow a_i = f(i) \end{array}$$

C-like Definition:

```
for (int i = 0; i < n; i++) { as[i] = f(i); }
return as;
```

This function returns a distributed list of n elements generated by a generator function f.

An important use of this function is to generate a list of increasing integers, $[0, 1, \ldots, n-1]$. In the following code, `sketo::functions::identity<int>()` is a function object for the identity function on integers.

```
dist_list<int> as
  = sketo::list_skeletons::generate(n, sketo::functions::identity<int>());
```

2.2.2 map

Type Signature:

$$\begin{array}{lll} \texttt{template}\ \langle\texttt{typename}\ A, \texttt{typename}\ B\rangle & & \\ \mathsf{dist\_list}\langle B\rangle\ \ \mathsf{map}(\texttt{Fobj}\langle B\ (A)\rangle & & f, \\ \qquad\qquad\qquad\ \mathsf{dist\_list}\langle A\rangle & & as); \end{array}$$

Mathematical Definition:

$$\begin{array}{l} [b_0, b_1, \ldots, b_{n-1}] = \mathsf{map}(f, [a_0, a_1, \ldots, a_{n-1}]) \\ \qquad \Longrightarrow b_i = f(a_i) \end{array}$$

C-like Definition:

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

This function applies the argument function f to every element of the distributed list as.

2.2.3 map_with_index

Type Signature:

$$\begin{array}{lll} \texttt{template}\ \langle\texttt{typename}\ A, \texttt{typename}\ B\rangle & & \\ \mathsf{dist\_list}\langle B\rangle\ \ \mathsf{map\_with\_index}(\texttt{Fobj}\langle B\ (\mathit{int}, A)\rangle & & f, \\ \qquad\qquad\qquad\qquad\qquad\ \mathsf{dist\_list}\langle A\rangle & & as); \end{array}$$

Mathematical Definition:

$$\begin{array}{l} [b_0, b_1, \ldots, b_{n-1}] = \mathsf{map\_with\_index}(f, [a_0, a_1, \ldots, a_{n-1}]) \\ \qquad \Longrightarrow b_i = f(i, a_i) \end{array}$$

C-like Definition:

```
for (int i = 0; i < n; i++) { bs[i] = f(i, as[i]); }
return bs;
```

This function applies the argument function f to every element of the distributed list as. The difference from the map skeleton is that the argument function takes an 0-based index in addition to the value of the element.

2.2.4 zip

Type Signature:

$$\begin{aligned}&\texttt{template}\ \langle \texttt{typename}\ A, \texttt{typename}\ B\rangle\\ &\mathsf{dist\_list}\langle std :: pair\langle A, B\rangle\rangle\ \ \mathsf{zip}(\mathsf{dist\_list}\langle A\rangle\ \ as,\\ &\qquad\qquad\qquad\qquad\qquad\quad \mathsf{dist\_list}\langle B\rangle\ \ bs);\end{aligned}$$

Mathematical Definition:

$$\begin{aligned}&[c_0, c_1, \ldots, c_{n-1}] = \mathsf{zip}([a_0, a_1, \ldots, a_{n-1}], [b_0, b_1, \ldots, b_{n-1}])\\ &\qquad \Longrightarrow c_i = (a_i, b_i)\end{aligned}$$

C-like Definition:

```
for (int i = 0; i < n; i++) { cs[i] = mkpair(as[i], bs[i]); }
return cs;
```

Condition: The length of two argument lists must be the same.

This function takes two distributed lists of the same length and makes a pair of the values for each index. This skeleton is a special case of the following zipwith skeleton.

2.2.5 zipwith

Type Signature:

$$\begin{aligned}&\texttt{template}\ \langle \texttt{typename}\ A, \texttt{typename}\ B, \texttt{typename}\ C\rangle\\ &\mathsf{dist\_list}\langle C\rangle\ \ \mathsf{zipwith}(\texttt{Fobj}\langle C\ (A, B)\rangle\ \ f,\\ &\qquad\qquad\qquad\qquad \mathsf{dist\_list}\langle A\rangle\qquad as,\\ &\qquad\qquad\qquad\qquad \mathsf{dist\_list}\langle B\rangle\qquad bs);\end{aligned}$$

Mathematical Definition:

$$\begin{aligned}&[c_0, c_1, \ldots, c_{n-1}] = \mathsf{zipwith}(f, [a_0, a_1, \ldots, a_{n-1}], [b_0, b_1, \ldots, b_{n-1}])\\ &\qquad \Longrightarrow c_i = f(a_i, b_i)\end{aligned}$$

C-like Definition:

```
for (int i = 0; i < n; i++) { cs[i] = f(as[i], bs[i]); }
return cs;
```

Condition: The length of two argument lists must be the same.

This function takes two distributed lists of the same length, and applies the argument function f to each pair of the values of the same index.

2.2.6 reduce

Type Signature:

```
template ⟨typename A⟩
A  reduce(Fobj⟨A (A, A)⟩  ⊕,
          dist_list⟨A⟩       as);
```

Mathematical Definition:

$b = \mathsf{reduce}(\oplus, [a_0, a_1, \ldots, a_{n-1}])$
$\quad \Longrightarrow b = a_0 \oplus a_1 \oplus \cdots \oplus a_{n-1}$

C-like Definition:

```
b = as[0];
for (int i = 1; i < n; i++) { b = oplus(b, as[i]); }
return b;
```

Condition: The operator $\oplus$ should be associative: equation $a \oplus (b \oplus c) = (a \oplus b) \oplus c$ should hold for any values a, b, and c.

This function collapses the distributed lists *as* into a single value with the associative binary operator $\oplus$.

2.2.7 scan

Type Signature:

```
template ⟨typename A⟩
dist_list⟨A⟩  scan(Fobj⟨A (A, A)⟩  ⊕,
                   A                 e,
                   dist_list⟨A⟩      as,
                   A*                last = NULL);
```

Mathematical Definition:

$[b_0, b_1, \ldots, b_{n-1}] = \mathsf{scan}(\oplus, e, [a_0, a_1, \ldots, a_{n-1}], last)$
$\quad \Longrightarrow b_i = e \oplus a_0 \oplus a_1 \oplus \cdots \oplus a_{i-1} \qquad last = e \oplus a_0 \oplus a_1 \oplus \cdots \oplus a_{n-1}$

C-like Definition:

```
for (int i = 0; i < n; i++) { bs[i] = e; e = oplus(e, as[i]); }
if (last) *last = e;
return bs;
```

Condition: The operator $\oplus$ should be associative: equation $a \oplus (b \oplus c) = (a \oplus b) \oplus c$ should hold for any values a, b, and c. In the implementation, we use the unit of the associative operator $\oplus$, which is defined by `identity_element` function.

This function computes accumulation of the values of *as* from left to right starting at e. The leftmost value of the distributed list becomes the argument e. The rightmost value of the original distributed list will not be used for computation of the returned lists. Instead, the totally accumulated value will be given through the pointer *last*, if *last* is not `NULL` (parameter *last* can be omitted).

2.2.8 scanr

Type Signature:

```
template ⟨typename A⟩
dist_list⟨A⟩  scanr(Fobj⟨A (A, A)⟩  ⊕,
                    A                e,
                    dist_list⟨A⟩     as,
                    A*               first = NULL);
```

Mathematical Definition:

$$[b_0, b_1, \dots, b_{n-1}] = \mathsf{scanr}(\oplus, e, [a_0, a_1, \dots, a_{n-1}], \mathit{first})$$
$$\Longrightarrow b_i = a_{i+1} \oplus a_{i+2} \oplus \cdots \oplus a_{n-1} \oplus e$$
$$\mathit{first} = a_0 \oplus a_1 \oplus \cdots \oplus a_{n-1} \oplus e$$

C-like Definition:

```
for (int i = n-1; i >= 0; i--) { bs[i] = e; e = oplus(as[i], e); }
if (first) *first = e;
return bs;
```

Condition: The operator $\oplus$ should be associative: equation $a \oplus (b \oplus c) = (a \oplus b) \oplus c$ should hold for any values a, b, and c. In the implementation, we use the (right) unit of the associative operator $\oplus$, which is defined by `identity_element` function.

This function computes accumulation of the values of *as* from right to left starting at *e*. The rightmost value of the distributed list becomes the argument *e*. The leftmost value of the original distributed list will not be used for computation of the returned lists. Instead, the totally accumulated value will be given through the pointer *first*, if *first* is not `NULL` (parameter *first* can be omitted).

When you use an associative but noncommutative operator for $\oplus$, pay attention to the order of the arguments of the operator.

2.2.9 postscan

Type Signature:

```
template ⟨typename A⟩
dist_list⟨A⟩  postscan(Fobj⟨A (A, A)⟩  ⊕,
                       dist_list⟨A⟩     as);
```

Mathematical Definition:

$$[b_0, b_1, \dots, b_{n-1}] = \mathsf{postscan}(\oplus, [a_0, a_1, \dots, a_{n-1}])$$
$$\Longrightarrow b_i = a_0 \oplus a_1 \oplus \cdots \oplus a_i$$

C-like Definition:

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

Condition: The operator $\oplus$ should be associative: equation $a \oplus (b \oplus c) = (a \oplus b) \oplus c$ should hold for any values a, b, and c.

This function computes accumulation of the values of *as* from left to right inside *as*. The leftmost value of the returned distributed list is equal to the leftmost value of the original list, and the rightmost value of the returned distributed list becomes the result of reduction, reduce$(\oplus, as)$.

2.2.10 postscanr

Type Signature:

```
template ⟨typename A⟩
dist_list⟨A⟩ postscanr(Fobj⟨A (A, A)⟩ ⊕,
                       dist_list⟨A⟩     as);
```

Mathematical Definition:

$$[b_0, b_1, \ldots, b_{n-1}] = \text{scanr}(\oplus, [a_0, a_1, \ldots, a_{n-1}])$$
$$\Longrightarrow b_i = a_i \oplus a_{i+1} \oplus \cdots \oplus a_{n-1}$$

C-like Definition:

```
bs[n-1] = as[n-1];
for (int i = n-2; i >= 0; i--) { bs[i] = oplus(as[i], bs[i+1]); }
return bs;
```

Condition: The operator $\oplus$ should be associative: equation $a \oplus (b \oplus c) = (a \oplus b) \oplus c$ should hold for any values a, b, and c.

This function computes accumulation of the values of as from right to left inside as. The rightmost value of the returned distributed list is equal to the rightmost value of the original list, and the leftmost value of the returned distributed list becomes the result of reduction, reduce$(\oplus, as)$.

When you use an associative but noncommutative operator for $\oplus$, pay attention to the order of the arguments of the operator.

2.2.11 gscanl

Type Signature:

```
template ⟨typename A, typename B, typename C⟩
dist_list⟨B⟩ gscanl(Fobj⟨B (B, A)⟩ f,
                    B             c,
                    dist_list⟨A⟩  as,
                    Fobj⟨C (A)⟩   g,
                    Fobj⟨C (C, C)⟩ ⊕,
                    Fobj⟨B (B, C)⟩ ⊖,
                    B*            last = NULL);
```

Mathematical Definition:

$$[b_0, b_1, \ldots, b_{n-1}] = \text{gscanl}(f, c, [a_0, a_1, \ldots, a_{n-1}], g, \oplus, \ominus, \mathit{last})$$
$$\Longrightarrow b_i = f(\cdots f(f(c, a_0), a_1), \ldots, a_{i-1})$$
$$\mathit{last} = f(\cdots f(f(c, a_0), a_1), \ldots, a_{n-1})$$

C-like Definition:

```
for (int i = 0; i < n; i++) { bs[i] = c; c = f(c, a[i]); }
if (last) *last = c;
return bs;
```

Condition: On the functions f and g and the operators $\oplus$ and $\ominus$, the following three equations should hold for any values a, b, c_1, c_2 and c_3 (the alphabets are related to the type of values).

$$f(b,a) = b \ominus g(a)$$
$$(b \ominus c_1) \ominus c_2 = b \ominus (c_1 \oplus c_2)$$
$$c_1 \oplus (c_2 \oplus c_3) = (c_1 \oplus c_2) \oplus c_3$$

As you see above, the operator $\oplus$ should be associative, and the operators $\oplus$ and $\ominus$ have the same relation as $+$ and $-$ $((b - c_1) - c_2 = b - (c_1 + c_2))$. In the implementation, we use the unit of the associative operator $\oplus$, which is defined by the `identity_element` function.

The sequential definition of this function is to accumulate the values of the distributed list *as* with the argument function f from left to right starting at the value e. The totally accumulated value will be given through the pointer *last*, if *last* is not `NULL` (parameter *last* can be omitted).

The name of this skeleton is from *general scan(l)*. This skeleton is said general in the sense that the function f used in the accumulation may be nonassociative but the parallel computation is performed through an associative operator $\oplus$ related to the function f. The three equations in the condition above form a sufficient condition for enabling parallel computation.

2.2.12 gscanr

Type Signature:

```
template ⟨typename A, typename B, typename C⟩
dist_list⟨A⟩ gscanr(Fobj⟨B (B, A)⟩  f,
                    B               c,
                    dist_list⟨A⟩    as,
                    Fobj⟨C (A)⟩     g,
                    Fobj⟨C (C, C)⟩  ⊕,
                    Fobj⟨B (B, C)⟩  ⊖,
                    B*              first = NULL);
```

Mathematical Definition:

$$\begin{aligned} &[b_0, b_1, \ldots, b_{n-1}] = \mathsf{gscanr}(f, c, [a_0, a_1, \ldots, a_{n-1}], g, \oplus, \ominus, \mathit{first}) \\ &\quad \Longrightarrow b_i = f(\cdots f(f(c, a_{n-1}), a_{n-2}), \ldots, a_{i+1}) \\ &\qquad \mathit{first} = f(\cdots f(f(c, a_{n-1}), a_{n-2}), \ldots, a_0) \end{aligned}$$

C-like Definition:

```
for (int i = n-1; i >= 0; i--) { bs[i] = c; c = f(c, a[i]); }
if (first) *first = c;
return bs;
```

Condition: On the functions f and g and the operators $\oplus$ and $\ominus$, the following three equations should hold for any values a, b, c_1, c_2 and c_3 (the alphabets are related to the type of values).

$$f(b,a) = b \ominus g(a)$$
$$(b \ominus c_1) \ominus c_2 = b \ominus (c_1 \oplus c_2)$$
$$c_1 \oplus (c_2 \oplus c_3) = (c_1 \oplus c_2) \oplus c_3$$

As you see above, the operator $\oplus$ should be associative, and the operators $\oplus$ and $\ominus$ have the same relation as $+$ and $-$ $((b - c_1) - c_2 = b - (c_1 + c_2))$. In the implementation, we use the unit of the associative operator $\oplus$, which is defined by the `identity_element` function.

The sequential definition of this function is to accumulate the values of the distributed list *as* with the argument function f from right to left starting at the value e. The totally accumulated value will be given through the pointer *first*, if *first* is not `NULL` (parameter *first* can be omitted).

The name of this skeleton is from *general scanr*. This skeleton is said general in the sense that the function f used in the accumulation may be nonassociative but the parallel computation is performed through an associative operator $\oplus$ related to the function f. The three equations in the condition above form a sufficient condition for enabling parallel computation.

When you use a function f that is noncommutative, pay attention to the order of the argument of the function. The order is different from that of the scanr and postscanr skeletons.

2.2.13 shiftl

Type Signature:

```
template ⟨typename A⟩
dist_list⟨A⟩ shiftl(A             fromR,
                    dist_list⟨A⟩  as,
                    A∗            toL = NULL);
```

Mathematical Definition:

$$\begin{aligned}&[b_0, b_1, \ldots, b_{n-1}] = \mathsf{shiftl}(\mathit{fromR}, [a_0, a_1, \ldots, a_{n-1}], \mathit{toL})\\ &\quad\Longrightarrow b_i = a_{i+1}\\ &\qquad\quad b_{n-1} = \mathit{fromR}\\ &\qquad\quad \mathit{toL} = a_0\end{aligned}$$

C-like Definition:

```
for (int i = 0; i < n-1; i++) { bs[i] = as[i+1]; }
bs[n-1] = fromR;
if (toL) *toL = a[0];
return bs;
```

This function shifts the elements of the distributed list *as* from right to left and the name of this skeleton is from *shift to left*. The rightmost element of the returned list becomes *fromR*. The leftmost value of the original distributed list well be given through the pointer *toL*, if *toL* is not `NULL` (parameter *toL* can be omitted.)

2.2.14 shiftr

Type Signature:

```
template ⟨typename A⟩
dist_list⟨A⟩ shiftr(A             fromL,
                    dist_list⟨A⟩  as,
                    A∗            toR = NULL);
```

Mathematical Definition:

$$\begin{aligned}&[b_0, b_1, \ldots, b_{n-1}] = \mathsf{shiftr}(\mathit{fromL}, [a_0, a_1, \ldots, a_{n-1}], \mathit{toR})\\ &\quad\Longrightarrow b_i = a_{i-1}\\ &\qquad\quad b_0 = \mathit{fromL}\\ &\qquad\quad \mathit{toR} = a_{n-1}\end{aligned}$$

C-like Definition:

```
for (int i = 1; i < n; i++) { bs[i] = as[i-1]; }
bs[0] = fromL;
if (toR) *toR = a[n-1];
return bs;
```

This function shifts the elements of the distributed list *as* from left to right and the name of this skeleton is from *shift to right*. The leftmost element of the returned list becomes *fromL*. The rightmost value of the original distributed list well be given through the pointer *toR*, if *toR* is not `NULL` (parameter *toR* can be omitted.)

2.3 Namespace functions: Function Objects

In the SkeTo library, concrete computations of parallel skeletons is specified by function objects. We can use function objects defined in the STL `<functional>` and `<sketo/functions.h>`, as well as user-defined function objects. When we use a user-defined function object, we should define it to inherit one of the base classes listed in Section 2.3.1 to inform the compiler about the type of the function object.

2.3.1 Base classes

Type Signature:

```
template ⟨typename R⟩
class base⟨R (void)⟩;

template ⟨typename R, typename A1⟩
class base⟨R (A1)⟩;

template ⟨typename R, typename A1, typename A2⟩
class base⟨R (A1, A2)⟩;

template ⟨typename R, typename A1, typename A2, typename A3⟩
class base⟨R (A1, A2, A3)⟩;
```

When we use a user-defined function object, the function object should inherit one of these classes. These classes are respectively for function objects with zero, one, two, and three arguments. For example, a function object that takes two integers and returns the difference of them can be defined as follows.

```
class diff_int : public sketo::functions::base<int (int, int)> {
public:
  int operator()(int x, int y) {
    return (x > y) ? x - y : y - x;
  }
};
```

2.3.2 identity, caster

Type Signature:

```
template ⟨typename A⟩
class identity {
  A operator()(A x)
};
```

This function object takes a value x and simply returns the value[1].

Type Signature:

```
template ⟨typename A, typename B⟩
class caster {
  B operator()(A x)
};
```

This function object takes a value x of type A, and returns the value by casting to type B.

2.3.3 fst, snd, mkpair, left, right

Type Signature:

```
template ⟨typename A, typename B⟩
class fst {
  A operator()(std :: pair⟨A, B⟩ x)
};
```

This function object takes a pair (x_1, x_2) and returns the first element of it x_1.

Type Signature:

```
template ⟨typename A, typename B⟩
class snd {
  B operator()(std :: pair⟨A, B⟩ x)
};
```

This function object takes a pair (x_1, x_2) and returns the second element of it x_2.

Type Signature:

```
template ⟨typename A, typename B⟩
class mkpair {
  std :: pair⟨A, B⟩ operator()(A x, B y)
};
```

This function object takes two values x and y and returns a pair of them (x, y).

Type Signature:

```
template ⟨typename A, typename B⟩
class left {
  A operator()(A x, B y)
};
```

[1]In the following of this section, we may omit "`public`" in the definition of a class.

This function object takes two values x and y and simply returns the first argument x.

Type Signature:

```
template ⟨typename A, typename B⟩
class right {
  B operator()(A x, B y)
};
```

This function object takes two values x and y and simply returns the second argument y.

2.3.4 square, max, min

Type Signature:

```
template ⟨typename A⟩
class square {
  A operator()(A x)
};
```

This function object takes a value x and returns the squared value of x, i.e. x^2.

Type Signature:

```
template ⟨typename A⟩
class max {
  A operator()(A x, A y)
};
```

This function object takes two values and returns the larger of them. This operator is associative and the unit is provided for $A = int, double$.

Type Signature:

```
template ⟨typename A⟩
class min {
  A operator()(A x, A y)
};
```

This function object takes two values and returns the smaller of them. This operator is associative and the unit is provided for $A = int, double$.