

SkeTo: Parallel Skeleton Library

Manual for Version 1.10

SkeTo Project

Web site: <http://www.ipl.t.u-tokyo.ac.jp/sketo/>

Contact email: sketo@ipl.t.u-tokyo.ac.jp

February 22, 2013

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 1.10 includes parallel skeletons for lists and matrices. 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 three parts. The first part provides quick-start tutorials. After showing how to install the SkeTo library, we demonstrate programming with the SkeTo library through several examples. The second part shows some advanced topics on use of the SkeTo library. The third part is the reference manual.

Contents

1	Quick Start	3
1.1	Installing the SkeTo Library	3
1.2	Tutorial: Computing Variance	4
1.3	Tutorial: Computing π by Monte-Carlo Method	6
1.4	Tutorial: Computing an Addition of Multi-Precision Numbers	9
2	Advanced Use of the SkeTo Library	13
2.1	Automatic Fusion Optimization	13
2.2	Function Objects	14
2.3	User-defined Data Structures	15
2.3.1	Simple Data Structures	15
2.3.2	Data structures on Heap Area	15
2.3.3	More Complex Data structures	16
3	Experimental Features in SkeTo	18
3.1	Use of Lambda Expressions	18
3.2	Fusion of Separate Expressions	19
3.3	Use of Any Function Objects	19
4	Reference Manual	20
4.1	Class <code>dist_list</code> : Distributed List Structure	20
4.1.1	Constructors	21
4.1.2	<code>clone</code>	21
4.1.3	<code>get_global_size</code>	21
4.1.4	<code>get</code>	21
4.1.5	<code>set</code>	22
4.1.6	<code>gather</code>	22
4.2	Namespace <code>list_skeletons</code> : Skeletons for Distributed Lists	22
4.2.1	<code>generate</code>	23
4.2.2	<code>map</code>	23
4.2.3	<code>map_with_index</code>	23
4.2.4	<code>zip</code>	24
4.2.5	<code>zipwith</code>	24
4.2.6	<code>reduce</code>	25
4.2.7	<code>scan</code>	25
4.2.8	<code>scanr</code>	26
4.2.9	<code>postscan</code>	26
4.2.10	<code>postscanr</code>	27
4.2.11	<code>gscanl</code>	27

4.2.12	<code>gscanr</code>	28
4.2.13	<code>shiffl</code>	29
4.2.14	<code>shiftr</code>	29
4.3	Class <code>dist_matrix</code> : Distributed Matrix Structure	30
4.3.1	Matrix Indices and Sizes	31
4.3.2	Constructors	31
4.3.3	<code>clone</code>	31
4.3.4	<code>get_global_size</code>	32
4.3.5	<code>get</code>	32
4.3.6	<code>set</code>	32
4.3.7	<code>gather</code>	32
4.4	Namespace <code>matrix_skeletons</code> : Skeletons for Distributed Matrices	33
4.4.1	<code>generate</code>	33
4.4.2	<code>map</code>	34
4.4.3	<code>map_with_index</code>	34
4.4.4	<code>zip</code>	35
4.4.5	<code>zipwith</code>	35
4.4.6	<code>reduce</code>	36
4.4.7	<code>scan</code>	36
4.4.8	<code>scanr</code>	37
4.4.9	<code>shift</code>	38
4.5	Namespace <code>functions</code> : Function Objects	38
4.5.1	Base classes	38
4.5.2	<code>identity</code> , <code>caster</code>	39
4.5.3	<code>fst</code> , <code>snd</code> , <code>mkpair</code> , <code>left</code> , <code>right</code>	39
4.5.4	<code>square</code> , <code>max</code> , <code>min</code>	40

Chapter 1

Quick Start

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 Installing the SkeTo Library

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.ip1.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`.

¹The 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 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 $[a_0, a_1, \dots, a_{n-1}]$, 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**) {
    int n;
    // ... 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;
}
```

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

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

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**) {
    int n;
    // ... 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. 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);
```

Since the elements of `a` are distributed among processes by this constructor, 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 4.2. 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

```
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;
```

In the above 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 4.5.

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>(), da) / size;
```

Output Result to Console

Since the SkeTo library is based on the MPI library, if we use `std::cout` the output will be repeated by the number of processors. Therefore, we use `sketo::cout` instead of `std::cout` to obtain a single output to console as follows.

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

We can use manipulators such as `std::endl` defined in STL here.

1.3 Tutorial: Computing π by Monte-Carlo Method

Let's consider computing the π ($= 3.1415926535\dots$) by the Monte-Carlo method. The basic idea is to estimate the area A_q of a quarter circle whose radius is 1, and obtain the π as $\pi = 4 * A_q$. The following method gives us a way to estimate the area A_q (and thus the π).

Given a sequence $[(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})]$ of random points whose components (i.e., x_i and y_i) independently follow the uniform distribution $U(0, 1)$, an estimation $\bar{\pi}$ of the π is given by counting the number of those points whose distance from $(0, 0)$ is less than or equal to 1.

$$\bar{\pi} = 4 * \left(\sum_{i=0}^{n-1} (\text{if } x_i^2 + y_i^2 \leq 1 \text{ then } 1 \text{ else } 0) \right) / n$$

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

```
int main(int, char**) {
    int n;
```

```

// ... initialization of the random generator

int count = 0;
for (int i = 0; i < n; i++) {
    double x = rand();
    double y = rand();
    count += x * x + y * y <= 1.0 ? 1 : 0;
}
double pi = 4.0 * count / n;
std::cout << pi << std::endl;
return 0;
}

```

Now we develop a parallel program using our parallel skeletons. The archive of the SkeTo library includes the complete code of this example (`samples/list/mcpi.cpp`).

Outline of Program

In this example, we use parallel list skeletons that manipulate distributed arrays in parallel. Similar to the first tutorial that computes the variance, you first need to include `list_skeletons.h` file. The outline of the program with the SkeTo library becomes as follows. In this example, we use default namespaces.

```

#include <list_skeletons.h>
using namespace sketo;
using namespace sketo::list_skeletons;

typedef std::pair<double,double> point;

// ... user-defined function objects

int sketo::main(int, char**) {
    // ... initialization of the random generator ...

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

```

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

Data Distribution

A distributed array is given as an instance of the `dist_list` class. There are two ways to create a distributed array: distributing a sequential array, and generating elements by a generator function. The former has been already given in the variance example.

In this example, we use the latter way to generate a distributed sequence of random points. The `generate` skeleton creates a distributed array according to a given generator function to produce elements from its indices. Here, we use `point` (another name of `std::pair<double,double>`) as the type of points, and `randpair` is the generator function (a user-defined function object shown below).


```
dist_list< point > ps = generate(n, randpoint);;
```

The declaration of the user-defined function object `randpoint` is as follows. It should be defined before the function `sketo::main`.

```
struct randpoint_t : public sketo::functions::base<point (int)> {
    point operator()(int /* x */) const {
        return point((std::rand()*1.0 / RAND_MAX), (std::rand()*1.0 / RAND_MAX));
    }
} randpoint;
```

The superclass `sketo::functions::base<point (int)>` indicates the type of the function object; it takes an `int` (i.e., the index) and returns a `point`.

Parallel Computation

We then carry out computation on distributed lists with parallel list skeletons. In this example, we use the `map` and `reduce` skeletons.

An example code is given as follows.

```
dist_list<double> ds = map(distance, ps);
dist_list<double> inouts = map(inout, ds);
double pi = reduce(plus, inouts) / size * 4;
```

Here, we first use the `map` skeleton twice to compute the distance from the origin (0,0) for each point, and to determine the distances are less or equal to 1 or not, and then we use the `reduce` skeleton to count the number of points contributing to the π . Declaration of the user-defined function objects is shown below.

```
struct distance_t : public sketo::functions::base<double (point)> {
    double operator()(point x) const {
        return x.first * x.first + x.second * x.second;
    }
} distance;

struct inout_t : public sketo::functions::base<double (double)> {
    double operator()(double x) const {
        return x <= 1.0 ? 1.0 : 0.0;
    }
} inout;

struct plus_t : public sketo::functions::base<double (double, double)> {
    double operator()(double x, double y) const {
        return x + y;
    }
} plus;
```

Note that we can use `std::plus<double>()` instead of our-defined `plus`. [iwasaki: ♠pp.7-8 の sequential code からみると](#), `return x.first * x.first + x.second * x.second <= 1.0 ? 1 : 0;` とする `f` 関数オブジェクトを定義して `map` 1 回にすればよい(しかも結果は `dist_list<int>` で) [ように思えるが](#). ♠

Output Result to Console

We use `sketo::cout` instead of `std::cout` for the output to console as follows.

```
sketo::cout << "pi = : " << pi << std::endl;
```

1.4 Tutorial: Computing an Addition of Multi-Precision Numbers

Let's consider computing an addition of two multi-precision numbers represented as arrays of integers. Here, we assume that an n -digit base- b multi-precision number $X = \sum_{i=0}^{n-1} x_i b^{n-1-i}$ is represented by an integer array $x = [x_0, x_1, \dots, x_{n-1}]$ (where $0 \leq x_i < b$). Given two integer arrays $x = [x_0, x_1, \dots, x_{n-1}]$ and $y = [y_0, y_1, \dots, y_{n-1}]$ that represent numbers X and Y , the objective is to compute an array $z = [z_0, z_1, \dots, z_{n-1}]$ and the carry c that represent $Z = X + Y$ (i.e., Z is represented by $[c, z_0, z_1, \dots, z_{n-1}]$):

$$\sum_{i=0}^{n-1} z_i b^{n-1-i} + c b^n = \sum_{i=0}^{n-1} x_i b^{n-1-i} + \sum_{i=0}^{n-1} y_i b^{n-1-i}$$

where $0 \leq z_i < b, 0 \leq c < 2$

A simple sequential program to obtain z and c is given as follows.

```
int main(int, char**) {
    int n;
    // ... initialization of the input numbers x[] and y[] ...

    int c = 0;
    for (int i = n-1; i >= 0; i--) {
        z[i] = x[i] + y[i] + c;
        if(z[i] >= b) {
            c = 1;
            z[i] -= b;
        } else {
            c = 0;
        }
    }

    // output the result
    std::cout << (c == 0 ? " ": "1");
    for (int i = 0; i < n; i++)
        std::cout << " " << z[i];
    std::cout << std::endl;
    return 0;
}
```

The above simple program is completely sequential. Therefore, we introduce some invention to parallelize it. The resulting program introduces several extra arrays **ps** (propagators), **gs** (generator), and **cs** (carries): **ps**[i] indicates a carry from the lower digit is propagated to the upper digit at i th digit; **gs**[i] indicates a carry is generated at i th digit; **cs**[i] is the resulting carry to be added at i th digit. This invention is well known in the field of electronic circuits. The resulting code is as follows:

```

int main(int, char**) {
    // ... initialization of the input numbers x[] and y[] ...

    for (int i = 0; i < n; i++) {
        z[i] = x[i] + y[i];
    }

    for (int i = 0; i < n; i++) {
        ps[i] = z[i] == base - 1;
        gs[i] = z[i] >= base;
    }

    cs[n-1] = 0; // no carry to LSD
    for (int i = n-1; i >= 1; i--) {
        cs[i-1] = (ps[i] && cs[i]) || gs[i];
    }
    int c = (ps[0] && cs[0]) || gs[0];

    for (int i = 0; i < n; i++) {
        z[i] = (z[i] + cs[i]) % base;
    }

    // output the result
    std::cout << (c == 0 ? " ": "1");
    for (int i = 0; i < n; i++)
        std::cout << " " << z[i];
    std::cout << std::endl;
    return 0;
}

```

Now we develop a parallel program for addition of multi-precision numbers using our parallel skeletons. The archive of the SkeTo library includes the complete code of this example (`samples/list/mpadd.cpp`).

Outline of Program

The outline of the program with the SkeTo library becomes as follows.

```

#include <list_skeletons.h>
using namespace sketo;
using namespace sketo::list_skeletons;

typedef std::pair<int, int> ipair;

// ... user-defined function objects

int sketo::main(int, char**) {
    // ... initialization of the input numbers x[] and y[] ...

```

```

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

```

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

Data Distribution

In this example, we use a constructor of `dist_list` that takes a sequential array.

```

sketo::dist_list<int> dx(x, n);
sketo::dist_list<int> dy(y, n);

```

The elements of `x` (`y`) are distributed to processes in this constructor.

Parallel Computation

We then manipulate distributed lists with parallel list skeletons. In this example, we use `map` and the following two parallel list skeletons:

- **zipwith**: a parallel skeleton that applies a given function to each pair of elements of the lists, and
- **scanr**: a parallel skeleton that computes a prefix-sum with a given associative binary operator.

An example code is given as follows.

```

dist_list<int> dz = zipwith(std::plus<int>(), dx, dy);
dist_list<ipair> dgspes = map(generator_propagator, dz);
dist_list<int> dcs = map(snd, scanr(timesr, e, dgspes, &all));
dist_list<int> dr = zipwith(addmod, dz, dcs);
int c = snd(all);

```

Declaration of the user-defined function objects is shown below.

```

struct randint_t : public sketo::functions::base<int (int)> {
    int operator()(int /* a */) const {
        return (int)((std::rand() * 1.0 / RAND_MAX) * base) % base;
    }
} randint;

struct generator_propagator_t : public sketo::functions::base<ipair (int)> {
    ipair operator()(int x) const {
        return ipair(x == base-1, x >= base);
    }
} generator_propagator;

struct snd_t : public sketo::functions::base<int (ipair)> {
    int operator()(ipair x) const {
        return x.second;
    }
}

```

```

    }
} snd;

struct timesr_t : public sketo::functions::base<ipair (ipair, ipair)> {
    ipair operator()(ipair x, ipair y) const {
        return ipair(x.first && y.first, (x.first && y.second) || x.second);
    }
} timesr;
ipair identity_element(const timesr_t&) {
    return ipair(1, 0);
}

struct addmod_t : public sketo::functions::base<int (int, int)> {
    int operator()(int x, int y) const {
        return (x + y) % base;
    }
} addmod;

```

The non-trivial part of the computation is the use of `scanr` skeleton followed by a `map`. This part computes the result of the following for-loop.

```

cs[n-1] = 0; // no carry to LSD
for (int i = n-1; i >= 1; i--) {
    cs[i-1] = (ps[i] && cs[i]) || gs[i];
}

```

The body of the for-loop is equivalent to the following computation with an associative operator \otimes that represents a matrix-multiplication on the Boolean semi-ring.

$$\begin{pmatrix} cs[i-1] \\ 1 \end{pmatrix} = \begin{pmatrix} ps[i] & gs[i] \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} cs[i] \\ 1 \end{pmatrix}$$

Therefore, we can use the `scanr` skeleton to compute products of the matrices $\begin{pmatrix} ps[i] & gs[i] \\ 0 & 1 \end{pmatrix} \otimes \dots \otimes \begin{pmatrix} ps[n-1] & gs[n-1] \\ 0 & 1 \end{pmatrix}$ for each `i` from 0 to `n-1` in parallel. The function object `times` of the `scanr` computes this \otimes , except that it removes redundant computation about two numbers in the lower row; they are always 1 and 0. Its definition is given as follows.

$$(a, b) \otimes' (a', b') = (a \wedge a', (a \wedge b') \vee b)$$

Output Result to Console

The output of the distributed array is carried out by the method `print` of `dist_list`.

```

sketo::cout << "x + y = " << (c == 0 ? " " : "1" ) << " "; r.print();

```

Chapter 2

Advanced Use of the SkeTo Library

2.1 Automatic Fusion Optimization

One of the most important features of the SkeTo library is that parallel skeletons are self-equipped with an automatic optimization mechanism by fusion transformation. Since the optimization mechanism is implemented with templates techniques in standard C++, the optimization works automatically at the compilation time (we can switch it off with a compilation parameter). See [?] for technical details of the implementation of the optimization mechanism. In this section, we see to which skeletons the optimization mechanism works.

Currently, the optimization mechanism works for parallel list skeletons and parallel matrix skeletons. Two kinds of optimizations have been implemented: one is fusion transformation and the other is overwriting a list with the result.

Fusion transformation is a program transformation that fuses consecutive calls of parallel skeletons into one to remove intermediate data. Since we often develop a program by composing several skeletons, this optimization may improve the performance of skeletal programs. Here are the conditions for the optimization by fusion transformation:

- The fusion optimization only works on parallel skeletons that are written in a single expression. For example, for the following program

```
bs = sketo::list_skeletons::map(f, as);
cs = sketo::list_skeletons::map(g, bs);
```

the fusion optimization does not work. Instead, we should write as follows.

```
cs = sketo::list_skeletons::map(g,
    sketo::list_skeletons::map(f, as));
```

- The fusion optimization works for parallel list skeletons `generate`, `map`, `wap_with_index`, `zip`, `zipwith`, `reduce`, `scan`, `scanr`, `postscan`, `postscanr`, `shiftl`, and `shiftr`.

The other optimization is to overwrite a list with the result of computation. For example, for the program

```
as = sketo::list_skeletons::map(f, as);
```

the buffer originally allocated for variable `as` is reused after the computation. In general, this optimization is not always performed since it may be unsafe to do so. Here are the conditions for the optimizing of overwriting a list.

- The variable for the resulting list has no alias, *i.e.*, there is no other variable that shares allocated space with the variable.
- When the right-hand side of the sentence has parallel skeletons applied to the variable, they are either `map`, `wap_with_index`, `zip`, `zipwith`, `scan`, `scanr`, `postscan`, or `postscanr`. Note that we can use any parallel skeletons if they are applied to other variables (with no sharing of data).
- When the right-hand side of the sentence has a `shiftl` or `shiftr` skeleton applied to the variable, then the skeleton should be called at the root of the call tree of skeletons. For example, combination of `map` followed by `shiftl`

```
as = shiftl(a, map(f, as));
```

is optimized, but combination of `shiftl` followed by `map`

```
as = map(f, shiftl(a, as));
```

is not.

For some reasons, you may not want to automatically optimize skeletal programs. In such cases, you can tell your compiler by defining macro variable `__SKETO_NO_FUSION__`. The easiest way is to add `“-D__SKETO_NO_FUSION__` into the command-line parameter, as follows.

```
sketocxx -D__SKETO_NO_FUSION__ (other parameters) (source file)
```

2.2 Function Objects

A function object is an object that can act as a function by overloading `operator()`, and is often given as a parameter to an abstract algorithm to make a concrete computation. For example, a sorting algorithm may receive a function object that is used to compare elements. The SkeTo library also asks users to specify functions objects to make concrete parallel computation with the skeletons, in which users can use both ready-made functions objects (in the SkeTo library or STL) and their defined ones. This section explains how we can define new function objects for parallel skeletons of the SkeTo library.

The SkeTo library provides template classes for nullary-, unary-, binary-, and ternary-functions (functions with zero, one, two, and three arguments, respectively) as listed in Section 4.5.1. When we want to define a function object for a parallel skeleton, we should make a class that inherits one of these base classes with suitable type information.

For example, let us define a class of function objects that take an integer and returns a real number whose value is a cube of the input. Here is program code for this.

```
class cubed_t : public sketo::functions::base<double (int)> {
public:
    double operator()(const int& x) const {
        return static_cast<double>(x * x * x);
    }
} cubed;
```

In the first line, we define the class `cubed_t` to inherit the template class `sketo::functions::base`, whose template parameter `double (int)` indicates that the input is of type `int` and the output is of type `double`. The `operator()` member function should be *public* and *constant* as we can see in the program above. It is worth noting that we can optimize the function object by using variables of type “`const X&`” for some input type X .

Some parallel skeletons require associative binary operators with their unit. For example, the binary operator \otimes defined for addition of the multi-precision numbers in Section 1.4

$$(a, b) \otimes' (a', b') = (a \wedge a', (a \wedge b') \vee b)$$

has its unit $(True, False)$. To specify the unit, we define an overloaded function `identity_element`, which takes a function object and returns the unit, as follows. Note that in the program below, `ipair` is a pair of integers and *True* and *False* are denoted by 1 and 0, respectively.

```
ipair identity_element(const timesr_t&) {
    return ipair(1, 0);
}
```

2.3 User-defined Data Structures

Usually, we need to define and use specific structures in our programs, rather than simple data types such as `int` or `double`. In this section, we explain how to use such structures. We classify the structures into the following three.

- Simple data structures: those that are put simply on stack area.
- Data structures on heap area: those that require serialization.
- More complex data structures: those that require more complicated treatment when serialization.

2.3.1 Simple Data Structures

Many simple data structures defined with `struct` or `class` are easily used with the SkeTo library. For example, if we want to use a structure for points in the 3D space we can simply define it as

```
struct point3D {
    double x, y, z;
};
```

and use it for elements of distributed data.

Note that, even if we use pointers for the member variables, we need to serialize it as seen in the following section.

2.3.2 Data structures on Heap Area

We may want to use more complicated data structure with dynamic allocation of memory from heap area or with support of containers such as `vector` in STL. These data structure includes some data allocated on heap area, and the SkeTo library asks us the way to serialize such data.

For example, if we want to use two variable-length arrays for the elements of a list (the following tutorial of the bracket-matching problem is such a case), we may define the following structure.


```
struct vect_element {
    std::vector<int> vec1;
    std::vector<double> vec2;
};
```

Since the actual data of `vec1` and `vec2` are allocated in another place from `vect_element`, the SkeTo library asks how to access and serialize such data for background communication. Therefore, we need to define the structure as follows when we use the SkeTo library.

```
struct vect_element : sketo::serialization::require_serialization {
    std::vector<int> vec1;
    std::vector<double> vec2;

private:
    friend class sketo::serialization::access;
    template <class Archive>
    void serialize(Archive& ar, const unsigned int /* version */) {
        ar & vec1;
        ar & vec2;
    }
};
```

In the program above, the structure inherits `sketo::serialization::require_serialization` to let the compiler know that serialization of this structure is necessary during communication. The friend declaration `friend class sketo::serialization::access;` is just a charm. We specify how to serialize data in the template function `serialize`. The way to specify serialization is very similar to that of `boost::serialization` in the boost library. We just place the data to be serialized one by one, with the `&` operator. It is worth noting that we need not to place the contents of `std::vector`, `std::string`, and `std::pair` since they are already defined properly in the SkeTo library.

2.3.3 More Complex Data structures

Sometimes, we may need to use more complicated data structures, for example, ones that require dynamic allocation before copying them. Among the samples of list skeletons, such a program is `cgstab_sketo_3D_ng.cpp` that solves systems of linear equations, in which we use complex data structures that implement reference counting to avoid unwilling copies.

In such cases, we need to implement the serialization function more carefully. In fact, the `serialization` function will be called for the following three purposes.

- Size counting: In this phase, the serialization function just counts the size of data and does not get nor put the value of data.
- Serialization: In this phase, the serialization function copies the data to the buffer prepared for communication.
- Deserialization: In this phase, the serialization function copies the data back from the buffer given after communication.

We can distinguish these three phases with the member variable `ar_type` of `Archive`:

- `sketo::serialization::AR_SIZE_COUNT`: size counting phase,

- `sketo::serialization::AR_SERIALIZE`: serialization phase, and
- `sketo::serialization::AR_DESERIALIZE`: deserialization phase.

In the sample program `cgstab_sketo_3D_ng.cpp`, we allocate a new buffer in the deserialization phase. Please see the program for details.

Chapter 3

Experimental Features in SkeTo

This chapter shows experimental features of SkeTo library to provide better programmability for skeleton programs. These features are realized by a successful fusion of the current template-based implementation of SkeTo and the new features of C++0x (the next C++ specification).

Currently, the following experimental features of SkeTo only work on g++ version 4.5. So, SkeTo has to be configured with an MPI that uses g++ version 4.5 as its base compiler. For example, if you have mpich2 compiled with g++ version 4.5 in directory `/usr/local/mpich2-1.2.1p1-g++4.5`, your SkeTo should be configured as follows.

```
./configure --with-mpi=/usr/local/mpich2-1.2.1p1-g++4.5
```

Also, to use the following experimental features, we have to specify option `-std=gnu++0x`. For example, to compile sample program `var-lambda.cpp`, we need to use a command like below.

```
sketocxx var-lambda.cpp -o var-lambda -O3 -std=gnu++0x
```

3.1 Use of Lambda Expressions

This experimental feature supports the use of lambda expressions in skeletons. Lambda expressions are very useful to build concise programs using higher-order functions (i.e., computation patterns) in functional programming. Skeletons in SkeTo are in fact higher-order functions, so lambda expressions are also very useful for concise programming with SkeTo.

Users can use lambda expressions of C++0x (the next C++ specification) in calls for skeletons. For example, we can write the following program to compute variance.

```
using namespace sketo::list_skeletons;
using namespace sketo::functions;
auto plus = [](double a, double b){return a + b;};
double ave = reduce(plus, da) / size;
auto db = map([ave](double a){return a - ave;}, da);
auto dc = map([](double a){return a * a;}, db);
double var = reduce(plus, dc) / size;
```

A lambda expression can be supplied to a skeleton directly as its argument. For example, the second `map` directly receives the lambda expression `[](double a){return a * a;}`. It is also possible to store a lambda expression to a variable and supply it to skeletons, which is useful if the lambda expression is used many times. In the above program, the variable `plus` holds a lambda expression `[](double a, double b){return a + b;}`, and is used twice.

3.2 Fusion of Separate Expressions

Owing to the `auto`-variables feature of C++0x, the use of SkeTo's fusion mechanism becomes more reasonable: Users need not to write a sequence of skeleton calls in a single expression to fuse them.

Here is the program to compute the variance of a given distributed list `da`. Here, `auto` is the new word of C++0x to declare auto-variables. The type of an auto-variable is inferred by a compiler from its initializing expression.

```
using namespace sketo::list_skeletons;
using namespace sketo::functions;
auto plus = [](double a, double b){return a + b;};
double ave = reduce(plus, da) / size;
auto db = map([ave](double a){return a - ave;}, da);
auto dc = map([](double a){return a * a;}, db);
double var = reduce(plus, dc) / size;
```

Two `maps` and a `reduce` to compute the variance are now fused into one loop, although they are used in separate expressions. This is owing to the use of `auto`.

If you want to control the region of the fusion, you can do it by specifying types explicitly instead of using `auto`. For example, the following program fuses two `maps`, but they are not fused into the last `reduce`, because the explicit type `dist_list<double>` of `dc` finalizes the fusion.

```
auto db = map([ave](double a){return a - ave;}, da);
dist_list<double> dc = map([](double a){return a * a;}, db);
double var = reduce(plus, dc) / size;
```

3.3 Use of Any Function Objects

This experimental feature frees users from the restriction that their defined function objects have to inherit the specific super classes.

For example, we can use the following a class of function objects that take an integer and returns a real number whose value is a cube of the input.

```
class cubed_t {
public:
    double operator()(const int& x) const {
        return static_cast<double>(x * x * x);
    }
} cubed;
```

This class `cubed_t` can be used in calls of skeletons like `map(cubed, da)`, although it does not extend the specific super class like `sketo::functions::base<double (int)>`.

Chapter 4

Reference Manual

4.1 Class `dist_list`: Distributed List Structure

iwasaki: ♠ プログラムにおいて , `int/const` などイタリックでない方が良いところがある . ユーザが自由に与えるところだけイタリックに ♠

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 4.2.

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 among 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 different semantics 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, in 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);
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.
```

4.1.1 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 in the allocated distributed list 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 whose 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. iwasaki: ♠ これは上には書かないの？ ♠

4.1.2 clone

Type Signature:

```
template <typename A>
dist_list<A> dist_list<A> :: clone() const;
```

This method generates another copy of a 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 the other.

4.1.3 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.

4.1.4 get

Type Signature:

```
template <typename A>
A dist_list<A> :: get(int index) const;
```

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

4.1.5 set

Type Signature:

```
template <typename A>
void dist_list(A) :: set(int index,
                        A value);
```

This method sets the *index*-th element's value with *value* in the distributed list.

4.1.6 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*. The user has to allocate sufficient space (i.e., the size given by `get_global_size`) for the array *buffer*.

iwasaki: ♠Detail:は必要ですか？♠

Detail: For reasons of performance, the 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.

4.2 Namespace `list_skeletons`: Skeletons for Distributed Lists

The namespace `list_skeletons` provides parallel list skeletons that manipulate distributed lists in parallel. The list skeletons 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: `shifl`, `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 definition. 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 $\text{Fobj}\langle C(A, B) \rangle$ for the type of function objects which denotes the type of a function object that takes two arguments of types *A* and *B* and returns a value of type *A*.

4.2.1 generate

Type Signature:

```
template <typename A>
dist_list<A> generate(int n,
                    Fobj<A (int)> f);
```

Mathematical Definition:

$$[a_0, a_1, \dots, a_{n-1}] = \text{generate}(n, f)$$

$$\implies a_i = f(i)$$

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, $[0, 1, \dots, n-1]$. 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>());
```

4.2.2 map

Type Signature:

```
template <typename A, typename B>
dist_list<B> map(Fobj<B (A)> f,
               dist_list<A> as);
```

Mathematical Definition:

$$[b_0, b_1, \dots, b_{n-1}] = \text{map}(f, [a_0, a_1, \dots, a_{n-1}])$$

$$\implies b_i = f(a_i)$$

C-like Definition:

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

This function applies a function f to every element of the distributed list as . iwasaki: ♠ 結果のリストは (fusion されなければ) allocate されることを言う. ♠

4.2.3 map_with_index

Type Signature:

```
template <typename A, typename B>
dist_list<B> map_with_index(Fobj<B (int, A)> f,
                           dist_list<A> as);
```

Mathematical Definition:

$$[b_0, b_1, \dots, b_{n-1}] = \text{map_with_index}(f, [a_0, a_1, \dots, a_{n-1}])$$

$$\implies b_i = f(i, a_i)$$

C-like Definition:

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

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

4.2.4 zip**Type Signature:**

```
template <typename A, typename B>
dist_list<std::pair<A, B>> zip(dist_list<A> as,
                             dist_list<B> bs);
```

Mathematical Definition:

$$[c_0, c_1, \dots, c_{n-1}] = \text{zip}([a_0, a_1, \dots, a_{n-1}], [b_0, b_1, \dots, b_{n-1}]) \\ \implies c_i = (a_i, b_i)$$
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 returns a distributed list whose element is a pair of the values for each index. This skeleton is a special case of the following `zipwith` skeleton.

4.2.5 zipwith**Type Signature:**

```
template <typename A, typename B, typename C>
dist_list<C> zipwith(Fobj<C (A, B)> f,
                    dist_list<A> as,
                    dist_list<B> bs);
```

Mathematical Definition:

$$[c_0, c_1, \dots, c_{n-1}] = \text{zipwith}(f, [a_0, a_1, \dots, a_{n-1}], [b_0, b_1, \dots, b_{n-1}]) \\ \implies c_i = f(a_i, b_i)$$
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 returns a distributed list whose element is the result of the application of the argument function f to each pair of the values of the same index.

4.2.6 reduce

Type Signature:

```
template <typename A>
A reduce(Fobj<A (A, A)> ⊕,
        dist_list<A> as);
```

Mathematical Definition:

$$b = \text{reduce}(\oplus, [a_0, a_1, \dots, a_{n-1}]) \\ \implies b = a_0 \oplus a_1 \oplus \dots \oplus a_{n-1}$$

C-like Definition:

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

Condition: An 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 an associative binary operator \oplus .

4.2.7 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:

$$[b_0, b_1, \dots, b_{n-1}] = \text{scan}(\oplus, e, [a_0, a_1, \dots, a_{n-1}], last) \\ \implies b_i = e \oplus a_0 \oplus a_1 \oplus \dots \oplus a_{i-1} \quad last = e \oplus a_0 \oplus a_1 \oplus \dots \oplus a_{n-1}$$

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: An 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 from 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 $last$, unless $last$ is NULL (parameter $last$ can be omitted).

4.2.8 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:

$$[b_0, b_1, \dots, b_{n-1}] = \text{scanr}(\oplus, e, [a_0, a_1, \dots, a_{n-1}], \text{first})$$

$$\implies b_i = a_{i+1} \oplus a_{i+2} \oplus \dots \oplus a_{n-1} \oplus e$$

$$\text{first} = a_0 \oplus a_1 \oplus \dots \oplus a_{n-1} \oplus e$$

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: An 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 from *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*, unless *first* is 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. iwasaki: ♠ なぜこれが必要？ ♠

4.2.9 postscan

Type Signature:

```
template <typename A>
dist_list<A> postscan(Fobj<A (A, A)> ⊕,
                     dist_list<A> as);
```

Mathematical Definition:

$$[b_0, b_1, \dots, b_{n-1}] = \text{postscan}(\oplus, [a_0, a_1, \dots, a_{n-1}])$$

$$\implies b_i = a_0 \oplus a_1 \oplus \dots \oplus a_i$$

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. 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 is the same as the result of reduction, $\text{reduce}(\oplus, as)$.

4.2.10 postscanr

Type Signature:

```
template <typename A>
dist_list<A> postscanr(Fobj<A (A, A)> &oplus,
                      dist_list<A> as);
```

Mathematical Definition:

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

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. 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 is the same as the result of reduction, $\text{reduce}(\oplus, as)$.

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

4.2.11 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)> &oplus,
                  Fobj<B (B, C)> &ominus,
                  B* last = NULL);
```

Mathematical Definition:

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

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).
iwasaki: ♠ a, b, c ではなく, x, y, z を使う方が良い. ♠

$$\begin{aligned}
f(b, a) &= b \ominus g(a) \\
(b \ominus c_1) \ominus c_2 &= b \ominus (c_1 \oplus c_2) \\
c_1 \oplus (c_2 \oplus c_3) &= (c_1 \oplus c_2) \oplus c_3
\end{aligned}$$

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 of `gscanl`, 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 a function f from left to right starting from the value c . 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 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 above condition form a sufficient condition for enabling parallel computation.

4.2.12 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:

$$\begin{aligned}
[b_0, b_1, \dots, b_{n-1}] &= \text{gscanr}(f, c, [a_0, a_1, \dots, a_{n-1}], g, \oplus, \ominus, first) \\
\implies b_i &= f(\dots f(f(c, a_{n-1}), a_{n-2}), \dots, a_{i+1}) \\
first &= f(\dots f(f(c, a_{n-1}), a_{n-2}), \dots, a_0)
\end{aligned}$$

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).

iwasaki: ♠xyz ∧ ♠

$$\begin{aligned} f(b, a) &= b \ominus g(a) \\ (b \ominus c_1) \ominus c_2 &= b \ominus (c_1 \oplus c_2) \\ c_1 \oplus (c_2 \oplus c_3) &= (c_1 \oplus c_2) \oplus c_3 \end{aligned}$$

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 of `gscanr`, 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 a function *f* from right to left starting from the value *c*. 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.

iwasaki: ♠ これは必要？ ♠

4.2.13 shiftl

Type Signature:

```
template <typename A>
dist_list<A> shiftl(A      fromR,
                   dist_list<A> as,
                   A*      toL = NULL);
```

Mathematical Definition:

$$\begin{aligned} [b_0, b_1, \dots, b_{n-1}] &= \text{shiftl}(\text{fromR}, [a_0, a_1, \dots, a_{n-1}], \text{toL}) \\ \implies b_i &= a_{i+1} \\ b_{n-1} &= \text{fromR} \\ \text{toL} &= a_0 \end{aligned}$$

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* to the left direction. The rightmost element of the returned list becomes *fromR*. The leftmost value of the original distributed list will be given through the pointer *toL*, if *toL* is not NULL (parameter *toL* can be omitted.)

4.2.14 shiftr

Type Signature:

```
template <typename A>
dist_list<A> shiftr(A      fromL,
                   dist_list<A> as,
                   A*      toR = NULL);
```

Mathematical Definition:

$$\begin{aligned}
[b_0, b_1, \dots, b_{n-1}] &= \text{shiftr}(\text{fromL}, [a_0, a_1, \dots, a_{n-1}], \text{toR}) \\
&\implies b_i = a_{i-1} \\
b_0 &= \text{fromL} \\
\text{toR} &= a_{n-1}
\end{aligned}$$

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* to the right direction. The leftmost element of the returned list becomes *fromL*. The rightmost value of the original distributed list will be given through the pointer *toR*, if *toR* is not NULL (parameter *toR* can be omitted.)

4.3 Class `dist_matrix`: Distributed Matrix Structure

The class `dist_matrix` provides containers for distributed matrices. This class is a template class `dist_matrix<A>` where *A* denotes the type of elements of a distributed matrix. This class provides only a small number of methods for its manipulation. Instead, users can manipulate distributed matrices by using parallel matrix skeletons defined in the namespace `sketo :: matrix_skeletons`, which are listed in Section 4.4.

The following code shows some methods provided in the `dist_matrix` class.

```

int array[] = {10, 11, 12, 13, 14, 15};
sketo::dist_matrix<int> as(msize(2, 3), array);
sketo::cout << as.get(mindex(0, 2)) << std::endl; // This outputs 12.

as.set(mindex(1,1), 20);
sketo::cout << as.get(mindex(1,1)) << std::endl; // This outputs 20.

```

In the background of the `dist_matrix` class, the elements are distributed among 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 need not to know how the data are distributed. You can use the `gather` method to restore a sequential array from a distributed matrix. The methods `get` and `set` implicitly do communications among processes.

One important thing to be noted is that the distributed matrices by the `dist_matrix` class have different semantics from usual `vector` or other containers in the STL. The semantics of `dist_matrix` is similar to that of an array in Java. An instance of the `dist_matrix` class has a reference to a concrete distributed matrix. 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 matrix that is allocated independently from another you should use the `clone` method explicitly. For example, look at the following code; the distributed matrices `as` and `bs` point the same concrete distributed matrix while the distributed matrix `cs` points a different one.

```

sketo::dist_matrix<int> as(msize(1,1));
as.set(mindex(0,0), 10);          // Generate a distributed matrix of one element.

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

as.set(mindex(0, 0), 20);

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

```

4.3.1 Matrix Indices and Sizes

The `dist_matrix` class uses two classes to receive (return) indices and sizes of matrices.

Class `mindex` is used to represent an index of a matrix; `mindex(nr, nc)` indicates the index of nr -th row and nc -th column.

Similarly, class `msize` is used to represent a size of a matrix; `msize(nr, nc)` indicates the size of a matrix that has nr rows and nc columns.

4.3.2 Constructors

Type Signature:

```

template <typename A>
dist_matrix<A> :: dist_matrix();

template <typename A>
dist_matrix<A> :: dist_matrix(msize size);

template <typename A>
dist_matrix<A> :: dist_matrix(msize size,
                             const A* seq_data,
                             int org = -1);

```

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

The second constructor takes an `msize` (size of matrix) `size` and allocates a distributed matrix of the size. The values of the elements will not be initialized by this constructor.

The third constructor takes an `msize` (size of matrix) `size` and an array `seq_data`. It also takes an optional integer `org`. This constructor allocates a distributed matrix of the size and the elements are initialized with those of `seq_data` (row-major). When the third argument `org` is negative, the constructor assumes that all processes have the same `seq_data` and carries out no communication. Otherwise it distributes the elements of `seq_data` on processor `org` to other processors.

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. iwasaki: ♠ これは上の定義に含めるべき? ♠

4.3.3 clone

Type Signature:

```

template <typename A>
dist_matrix<A> dist_matrix<A> :: clone() const;

```


This method generates another copy of a distributed matrix.

As we stated above, the semantics of the `dist_matrix` is similar to that of the array in Java. We need to use this method if we want an independent distributed matrix from the other.

4.3.4 `get_global_size`

Type Signature:

```
template <typename A>
msize dist_matrix<A> :: get_global_size() const;
```

This method returns the (whole) size of the distributed matrix.

4.3.5 `get`

Type Signature:

```
template <typename A>
A dist_matrix<A> :: get(mindex index) const;
```

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

4.3.6 `set`

Type Signature:

```
template <typename A>
void dist_matrix<A> :: set(mindex index,
                          A          value);
```

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

4.3.7 `gather`

Type Signature:

```
template <typename A>
void dist_matrix<A> :: gather(A* buffer) const;
```

This method copies the value of the elements of the distributed matrix to the array *buffer*. The user has to allocate sufficient space (i.e., the size given by `get_global_size`) for the array *buffer*.

iwasaki: ♠Detail:は必要ですか？♠

Detail: For reasons of performance, the 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 matrix 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 matrix from the root process.

4.4 Namespace `matrix_skeletons`: Skeletons for Distributed Matrices

The namespace `matrix_skeletons` provides parallel matrix skeletons that manipulate distributed matrices in parallel. The matrix skeletons 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): `scan`
- Suffix-sums (scanr): `scanr`
- Shift: `shift`

In this section, we use two notations for showing informal definition of the parallel matrix skeletons: one is mathematical definition and the other is C-like definition. 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 $\text{Fobj}\langle C(A, B) \rangle$ for the type of function objects which denotes the type of a function object that takes two arguments of types A and B and returns a value of type A .

4.4.1 `generate`

Type Signature:

```
template <typename A>
dist_matrix<A> generate(msize      size,
                      Fobj<A (mindex)> f);
```

Mathematical Definition:

$$\begin{bmatrix} a_{(0,0)} & \cdots & a_{(0,n-1)} \\ a_{(1,0)} & \cdots & a_{(1,n-1)} \\ \vdots & \ddots & \vdots \\ a_{(m-1,0)} & \cdots & a_{(m-1,n-1)} \end{bmatrix} = \text{generate}((m, n), f)$$

$$\implies a_{(i,j)} = f((i, j))$$

C-like Definition:

```
for (int i = 0; i < m; i++)
    for (int j = 0; j < n; j++)
        as[i][j] = f(mindex(i, j));
return as;
```

This function returns a distributed matrix of size `size` whose elements are generated by a generator function `f`.

An important use of this function is to generate a matrix of indices. In the following code, `sketo::functions::identity<mindex>()` is a function object for the identity function on matrix indices.

```
dist_matrix<mindex> as
    = sketo::matrix_skeletons::generate(n, sketo::functions::identity<mindex>());
```

4.4.2 map

Type Signature:

```
template <typename A, typename B>
dist_matrix<B> map(Fobj<B (A)> f,
                  dist_matrix<A> as);
```

Mathematical Definition:

$$\begin{bmatrix} b_{(0,0)} & \cdots & b_{(0,n-1)} \\ b_{(1,0)} & \cdots & b_{(1,n-1)} \\ \vdots & \ddots & \vdots \\ b_{(m-1,0)} & \cdots & b_{(m-1,n-1)} \end{bmatrix} = \text{map}(f, \begin{bmatrix} a_{(0,0)} & \cdots & a_{(0,n-1)} \\ a_{(1,0)} & \cdots & a_{(1,n-1)} \\ \vdots & \ddots & \vdots \\ a_{(m-1,0)} & \cdots & a_{(m-1,n-1)} \end{bmatrix})$$

$$\implies b_{(i,j)} = f(a_{(i,j)})$$

C-like Definition:

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

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

4.4.3 map_with_index

Type Signature:

```
template <typename A, typename B>
dist_matrix<B> map_with_index(Fobj<B (mindex, A)> f,
                              dist_matrix<A> as);
```

Mathematical Definition:

$$\begin{bmatrix} b_{(0,0)} & \cdots & b_{(0,n-1)} \\ b_{(1,0)} & \cdots & b_{(1,n-1)} \\ \vdots & \ddots & \vdots \\ b_{(m-1,0)} & \cdots & b_{(m-1,n-1)} \end{bmatrix} = \text{map_with_index}(f, \begin{bmatrix} a_{(0,0)} & \cdots & a_{(0,n-1)} \\ a_{(1,0)} & \cdots & a_{(1,n-1)} \\ \vdots & \ddots & \vdots \\ a_{(m-1,0)} & \cdots & a_{(m-1,n-1)} \end{bmatrix})$$

$$\implies b_{(i,j)} = f((i, j), a_{(i,j)})$$

C-like Definition:

```
for (int i = 0; i < m; i++)
    for (int j = 0; j < n; j++)
        bs[i][j] = f(mindex(i, j), as[i][j]);
return bs;
```

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

4.4.4 zip

Type Signature:

```
template <typename A, typename B>
dist_matrix<std :: pair<A, B>> zip(dist_matrix<A> as,
                                   dist_matrix<B> bs);
```

Mathematical Definition:

$$\begin{bmatrix} c_{(0,0)} & \cdots & c_{(0,n-1)} \\ c_{(1,0)} & \cdots & c_{(1,n-1)} \\ \vdots & \ddots & \vdots \\ c_{(m-1,0)} & \cdots & c_{(m-1,n-1)} \end{bmatrix} = \text{zip}\left(\begin{bmatrix} a_{(0,0)} & \cdots & a_{(0,n-1)} \\ a_{(1,0)} & \cdots & a_{(1,n-1)} \\ \vdots & \ddots & \vdots \\ a_{(m-1,0)} & \cdots & a_{(m-1,n-1)} \end{bmatrix}, \begin{bmatrix} b_{(0,0)} & \cdots & b_{(0,n-1)} \\ b_{(1,0)} & \cdots & b_{(1,n-1)} \\ \vdots & \ddots & \vdots \\ b_{(m-1,0)} & \cdots & b_{(m-1,n-1)} \end{bmatrix}\right)$$

$$\implies c_{(i,j)} = (a_{(i,j)}, b_{(i,j)})$$

C-like Definition:

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

Condition: The sizes of two argument matrices must be the same.

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

4.4.5 zipwith

Type Signature:

```
template <typename A, typename B, typename C>
dist_matrix<C> zipwith(Fobj<C (A, B)> f,
                      dist_matrix<A> as,
                      dist_matrix<B> bs);
```

Mathematical Definition:

$$\begin{bmatrix} c_{(0,0)} & \cdots & c_{(0,n-1)} \\ c_{(1,0)} & \cdots & c_{(1,n-1)} \\ \vdots & \ddots & \vdots \\ c_{(m-1,0)} & \cdots & c_{(m-1,n-1)} \end{bmatrix} = \text{zipwith}\left(f, \begin{bmatrix} a_{(0,0)} & \cdots & a_{(0,n-1)} \\ a_{(1,0)} & \cdots & a_{(1,n-1)} \\ \vdots & \ddots & \vdots \\ a_{(m-1,0)} & \cdots & a_{(m-1,n-1)} \end{bmatrix}, \begin{bmatrix} b_{(0,0)} & \cdots & b_{(0,n-1)} \\ b_{(1,0)} & \cdots & b_{(1,n-1)} \\ \vdots & \ddots & \vdots \\ b_{(m-1,0)} & \cdots & b_{(m-1,n-1)} \end{bmatrix}\right)$$

$$\implies c_{(i,j)} = f(a_{(i,j)}, b_{(i,j)})$$

C-like Definition:

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

Condition: The sizes of two argument matrices must be the same.

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

4.4.6 reduce

Type Signature:

```
template <typename A>
A reduce(Fobj<A (A, A)> ⊕,
        Fobj<A (A, A)> ⊗,
        dist_matrix<A> as);
```

Mathematical Definition:

$$b = \text{reduce}(\oplus, \otimes, \begin{bmatrix} a_{(0,0)} & \cdots & a_{(0,n-1)} \\ a_{(1,0)} & \cdots & a_{(1,n-1)} \\ \vdots & \ddots & \vdots \\ a_{(m-1,0)} & \cdots & a_{(m-1,n-1)} \end{bmatrix})$$

$$\implies b = (a_{(0,0)} \otimes a_{(0,1)} \otimes \cdots \otimes a_{(0,n-1)})$$

$$\oplus (a_{(1,0)} \otimes \cdots \otimes a_{(1,n-1)}) \oplus \cdots$$

$$\oplus (a_{(m-1,0)} \otimes \cdots \otimes a_{(m-1,n-1)})$$

C-like Definition:

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

Condition: Operators \oplus and \otimes should be associative: \oplus satisfies the equation $a \oplus (b \oplus c) = (a \oplus b) \oplus c$ for any values a , b , and c , and so does \otimes . Also, they should satisfy the following abide property: $(a \oplus b) \otimes (c \oplus d) = (a \otimes c) \oplus (b \otimes d)$ for any values a , b , c , and d . It should be noted that any commutative, associative operator \odot , such as $+$, $*$, and the maximum operator, satisfies the abide property with itself: $(a \odot b) \odot (c \odot d) = a \odot b \odot c \odot d = a \odot c \odot b \odot d = (a \odot c) \odot (b \odot d)$.

This function collapses the distributed matrix as into a single value by using \oplus for vertical direction and \otimes for horizontal direction.

4.4.7 scan

Type Signature:

```
template <typename A>
dist_matrix<A> scan(Fobj<A (A, A)> ⊕,
                  Fobj<A (A, A)> ⊗,
                  dist_matrix<A> as);
```

Mathematical Definition:

$$\begin{bmatrix} b_{(0,0)} & \cdots & b_{(0,n-1)} \\ b_{(1,0)} & \cdots & b_{(1,n-1)} \\ \vdots & \ddots & \vdots \\ b_{(m-1,0)} & \cdots & b_{(m-1,n-1)} \end{bmatrix} = \text{scan}(\oplus, \otimes, \begin{bmatrix} a_{(0,0)} & \cdots & a_{(0,n-1)} \\ a_{(1,0)} & \cdots & a_{(1,n-1)} \\ \vdots & \ddots & \vdots \\ a_{(m-1,0)} & \cdots & a_{(m-1,n-1)} \end{bmatrix})$$

$$\begin{aligned}
\Rightarrow b_{(i,j)} &= (a_{(0,0)} \otimes a_{(0,1)} \otimes \cdots \otimes a_{(0,j)}) \\
&\quad \oplus (a_{(1,0)} \otimes \cdots \otimes a_{(1,j)}) \oplus \cdots \\
&\quad \oplus (a_{(i,0)} \otimes \cdots \otimes a_{(i,j)})
\end{aligned}$$

Condition: The operators \oplus and \otimes should be associative: \oplus satisfies the equation $a \oplus (b \oplus c) = (a \oplus b) \oplus c$ for any values a , b , and c , and so does \otimes . Also, they should satisfy the following abide property: $(a \oplus b) \otimes (c \oplus d) = (a \otimes c) \oplus (b \otimes d)$ for any values a , b , c , and d . It should be noted that any commutative, associative operator \odot , such as $+$, $*$, and the maximum operator, satisfies the abide property with itself: $(a \odot b) \odot (c \odot d) = a \odot b \odot c \odot d = a \odot c \odot b \odot d = (a \odot c) \odot (b \odot d)$.

This function computes accumulation of the values of as from top-left to bottom-right by using \oplus for vertical direction and \otimes for horizontal direction.

4.4.8 scanr**Type Signature:**

```
template <typename A>
dist_matrix<A> scanr(Fobj<A (A, A)> &oplus,
                    Fobj<A (A, A)> &otimes,
                    dist_matrix<A> &as);
```

Type Signature:

```
template <typename A>
dist_matrix<A> scanr(Fobj<A (A, A)> &oplus,
                    Fobj<A (A, A)> &otimes,
                    dist_matrix<A> &as);
```

Mathematical Definition:

$$\begin{bmatrix} b_{(0,0)} & \cdots & b_{(0,n-1)} \\ b_{(1,0)} & \cdots & b_{(1,n-1)} \\ \vdots & \ddots & \vdots \\ b_{(m-1,0)} & \cdots & b_{(m-1,n-1)} \end{bmatrix} = \text{scanr}(\oplus, \otimes, \begin{bmatrix} a_{(0,0)} & \cdots & a_{(0,n-1)} \\ a_{(1,0)} & \cdots & a_{(1,n-1)} \\ \vdots & \ddots & \vdots \\ a_{(m-1,0)} & \cdots & a_{(m-1,n-1)} \end{bmatrix})$$

$$\begin{aligned}
\Rightarrow b_{(i,j)} &= (a_{(i,j)} \otimes a_{(i,j+1)} \otimes \cdots \otimes a_{(i,n-1)}) \\
&\quad \oplus (a_{(i+1,j)} \otimes \cdots \otimes a_{(i+1,n-1)}) \oplus \cdots \\
&\quad \oplus (a_{(m-1,j)} \otimes \cdots \otimes a_{(m-1,n-1)})
\end{aligned}$$

Condition: The operators \oplus and \otimes should be associative: \oplus satisfies the equation $a \oplus (b \oplus c) = (a \oplus b) \oplus c$ for any values a , b , and c , and so does \otimes . Also, they should satisfy the following abide property: $(a \oplus b) \otimes (c \oplus d) = (a \otimes c) \oplus (b \otimes d)$ for any values a , b , c , and d . It should be noted that any commutative, associative operator \odot , such as $+$, $*$, and the maximum operator, satisfies the abide property with itself: $(a \odot b) \odot (c \odot d) = a \odot b \odot c \odot d = a \odot c \odot b \odot d = (a \odot c) \odot (b \odot d)$.

This function computes accumulation of the values of as from bottom-right to top-left.

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

4.4.9 shift

Type Signature:

```
template <typename A>
dist_matrix<A> shift<dr, dc>(Fobj<A (mindex, A)> f, dist_matrix<A> as);
```

Mathematical Definition:

$$\begin{bmatrix} b_{(0,0)} & \cdots & b_{(0,n-1)} \\ b_{(1,0)} & \cdots & b_{(1,n-1)} \\ \vdots & \ddots & \vdots \\ b_{(m-1,0)} & \cdots & b_{(m-1,n-1)} \end{bmatrix} = \text{shift}_{(dr, dc)}(f, \begin{bmatrix} a_{(0,0)} & \cdots & a_{(0,n-1)} \\ a_{(1,0)} & \cdots & a_{(1,n-1)} \\ \vdots & \ddots & \vdots \\ a_{(m-1,0)} & \cdots & a_{(m-1,n-1)} \end{bmatrix})$$

$$\Rightarrow b_{(i,j)} = \begin{cases} \text{if } 0 \leq i' < m \wedge 0 \leq j' < n \text{ then} & a_{(i',j')} \\ \text{otherwise} & f(\text{mindex}(i', j'), a_{(i' \bmod m, j' \bmod n)}) \\ \text{where } i' = i - dr \text{ and } j' = j - dc \end{cases}$$

This function shifts (rotates) the elements of the distributed matrix *as* by (dr, dc) . If the index (i', j') is out of the original matrix, then the value of the element is computed by the given function *f*. For example, you can carry out a simple rotation by letting *f* to be $f(mi, a) = a$. Also, if you want to fill up the border with a boundary value *b*, you can use $f(mi, a) = b$, because the function *f* is applied to only the elements moved beyond the border.

4.5 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 4.5.1 to inform the compiler about the type of the function object.

4.5.1 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 absolute value of their difference can be defined as follows.

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

4.5.2 identity, caster

Type Signature:

```
template <typename A>
class identity {
    A operator()(A x)
};
```

This function object takes a value x and simply returns the value¹.

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 .

4.5.3 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 its first element, i.e., 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 its second element, i.e., x_2 .

¹In the following of this section, we may omit “public” in the definition of a class.

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 (x, y) .

Type Signature:

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

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 .

4.5.4 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. The unit of this operator is provided for the cases where A is either `int` or `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. The unit of this operator is provided for the cases where A is either `int` or `double`.