Vol. 49 No. SIG 3(PRO 36) IPSJ Transactions on Programming Mar. 2008

Regular Paper

Parallel Skeletons for Sparse Matrices in SkeTo Skeleton Library

Yuki Karasawa[†1] and Hideya Iwasaki[†1]

Skeletal parallel programming makes both parallel programs development and parallelization easier. The idea is to abstract generic and recurring patterns within parallel programs as skeletons and provide them as a library whose parallel implementations are transparent to the programmer. SkeTo is a parallel skeleton library that enables programmers to write parallel programs in C++ in a sequential style. However, SkeTo's matrix skeletons assume that a matrix is dense, so they are incapable of efficiently dealing with a sparse matrix, which has many zeros, because of duplicated computations and commutations of identical values. This problem is solved by re-formalizing the matrix data type to cope with sparse matrices and by implementing a new C++ class of SkeTo with efficient sparse matrix skeletons based on this new formalization. Experimental results show that the new skeletons for sparse matrices perform well compared to existing skeletons for dense matrices.

1. Introduction

It has become quite common in many areas that need scientific computation to write parallel programs and execute them on parallel machines like PC clusters to substantially improve the performance. *Skeletal parallel programming* [7),10),20)] is a promising approach that makes this programming process easier, especially for non-experienced programmers with little knowledge of parallel programming. The idea of skeletal parallel programming is to abstract generic and recurring patterns within parallel programs as *skeletons* and to provide them as a library whose parallel implementations are transparent to the programmer. Programmers can create moderately efficient parallel programs with little effort by combining suitable skeletons. Much research [1),6),8),9)] has been devoted to skeletal parallel programming and to the development of skeleton libraries and systems [2),5),16)].

The SkeTo (Skeletons in Tokyo) library [17),21)] enables programmers to write skeletal parallel programs in C++ in a sequential style. It is implemented in standard C++ with Message Passing Interface (MPI) and has four distinguishing features.

- It is based on the theory of Constructive Algorithmics [3)], in which programs are structured in accordance with the data type. It provides a set of *data parallel* skeletons that reflect the structure of the data type of interest and that perform simultaneous computations on data distributed across processors.
- It provides parallel skeletons for lists, matrices, and trees [18)]. Since the parallel behavior of each data type or each skeleton is concealed within its implementation, programmers can write parallel programs as if they were sequential ones.
- In contrast to systems [5),9),10)] that support parallel skeletons with enhanced syntax in their base language, SkeTo introduces no special extension to the base C++ language. Programmers who can develop a standard C++ program can use SkeTo without the burden of acquiring a new syntax or new language.
- SkeTo has a systematic program optimization mechanism based on fusion transformation [12),14)]. This transformation merges two successive function calls into a single one, thereby eliminating the overhead of function calls and the generation of intermediate data structures passed between the two functions.

Focusing on SkeTo's matrix skeletons [11),17)], we see that their major premise is that a matrix is *dense*. This originates from SkeTo's inductive definition of a matrix: a single element constructs a 1×1 matrix, and two matrices of the same width or same height can be joined together to form a larger matrix. Since this definition cannot efficiently represent a *sparse* matrix, which has many zeros, skeletons based on this definition have to perform many redundant computations for the same values (elements)

†1 Department of Computer Science, The University of Electro-Communications

when they are applied to sparse matrices. This inefficiency is a big problem with SkeTo because sparse matrices frequently appear in scientific computations.

To solve this problem and make SkeTo more practical for the development of matrix-oriented parallel programs, we re-formalize the matrix data type to efficiently cope with sparse matrices and implement a new C++ class of SkeTo based on this formalization. The main contributions of this paper are as follows.

- It provides a theoretical background for handling sparse matrices by formalizing the matrix data type using a *block* of identical values and gives definitions for basic parallel skeletons based on this formalization.
- It describes the implementation of a C++ class for sparse matrices in the SkeTo library. Since the class has exactly the same interface as that of the existing matrix class, users need not change their programs to use the sparse matrix class. This compatibility is very important because a user encounters no barriers in using the new sparse skeleton class.
- It presents experimental results showing that the new implementations of matrix skeletons perform well for sparse matrices. Even for dense matrices, their overhead compared to those of the existing matrix skeletons is very small.

From the viewpoint of internal representation of matrices, the data structure for sparse matrices resembles a quadtree [22]. The most important point of this work is that we have developed a *practical* and *easy-to-use* parallel skeleton library that has good performance for sparse matrices whose sparseness is transparent to the programmer.

In this paper, we use *dense matrix class/skeleton* to represent the existing matrix class/skeleton in SkeTo and *sparse matrix class/skeleton* to represent the new matrix class/skeleton.

We also use the following functional notation of Haskell to describe the theoretical backgrounds and definitions of skeletons because of its concision and clarity. *Function application* is denoted by a space, and the argument is written without brackets. Thus, $f\,a$ means $f\,(a)$. Functions are curried, and applications associate to the left. Thus, $f\,a\,b$ means $(f\,a)\,b$. A function application binds stronger than any operator, so $f\,a \oplus b$ means $(f\,a) \oplus b$, but not $f\,(a \oplus b)$. Infix binary operators are generally denoted by $\oplus$, $\otimes$, etc. and can be *sectioned*; an infix binary operator like $\oplus$ is turned into a unary or binary function by $a \oplus b = (a \oplus)\,b = (\oplus b)\,a = (\oplus)\,a\,b$. Binary operators $\ll$ and $\gg$ are defined by $a \ll b = a$ and $a \gg b = b$.

2. Related Work

The research on Muesli [16], a skeleton library in C++ that supports distributed matrices, is the most related to our work. Muesli supports the map, zipwith, and fold data parallel skeletons. (The fold corresponds to reduce in SkeTo.) In the internal implementation of Muesli, elements distributed to each processor are stored in an array, and each data parallel skeleton accesses each element in this array in turn. This implementation of a matrix resembles that of SkeTo's dense matrix class and has the same inefficiency in handling sparse matrices.

Other skeleton library systems like Skil [5] and P3L [9] also support matrices as their basic data type, but they do not offer efficient implementation of sparse matrices. In addition, unlike SkeTo, they have enhanced syntax in their base language, which is likely to be a serious barrier for non-experienced users.

Intel's Threading Building Blocks [15] is a C++ runtime library especially for multi-core processors that abstracts the low-level threading activities. It offers generic parallel algorithms like parallel-for (which corresponds to the map skeleton in SkeTo), parallel-reduce, and parallel-scan. A matrix is abstracted in blocked_range2d class, which describes the index range of two dimensions. This class is used together with parallel algorithms such as parallel-for. Although this library can deal with matrices in parallel, whether the sparseness of a matrix is taken into account in its implementation is not clear.

As for the internal representation of matrices, Wise [22] used a quadtree, which is constructed by combining four small sub-matrices of the same size to represent a matrix. However, once a matrix is represented by a quadtree, it is difficult to re-construct the representation. This prevents one from flexibly and dynamically changing the matrix representation in accordance with the progress of computations in which a dense matrix is transformed into a sparse one.

PETSc [23] (Portable, Extensible Toolkit for Scientific Computation) is a suit of data struc-

tures and routines for parallel (and sequential) scientific applications. It provides a variety of sparse matrix implementations capable of dealing with a sparse AIJ format (also called compressed sparse row format), blocked sparse AIJ format, block diagonal format and so on. After creating a matrix, programmers are required to explicitly set an appropriate format of the matrix to have a good speed; the efficiency of the program may be awfully degraded unless an appropriate format is specified. Compared to this, SkeTo's data constructor and skeletons automatically configure an adequate internal representation for a matrix based on the tree structure described in Section 5.2, according to the degree of its sparseness. Thus the user of SkeTo needs not specify the format of a matrix and needs not worry about the transition of the degree of sparseness of a matrix.

PETSc is not a skeleton library; it is specialized in numerical computation and provides rather large built-in operations like matrix-vector/matrix-matrix multiplications and LU factorizations that are frequently used in numerical applications. It is easy to write a program using PETSc as long as the program can be described by the combination of these built-in operations. However, since PETSc does not support small, flexible, and higher-order routines with high abstraction level like our skeletons, it seems difficult to write a program in PETSc for a problem, e.g., Maximum Rectangle Sum described in Section 6.2, which does not use common numerical operations. Thus the goal of SkeTo is different from that of PETSc; SkeTo enables users to implement general (matrix) algorithms in terms of small ready-made skeletons.

3. Matrix Skeletons in SkeTo Library

3.1 Formalization of Matrix and Skeletons

SkeTo defines a matrix as being formed by three constructors, namely $|\cdot|$, $\ominus$, and $\oplus$[4].

$$\begin{aligned}\textbf{data}\ \mathsf{Matrix}\ \alpha = {} & |\cdot|\ \alpha \\ \mid {} & \mathsf{Matrix}\ \alpha \ominus \mathsf{Matrix}\ \alpha \\ \mid {} & \mathsf{Matrix}\ \alpha \oplus \mathsf{Matrix}\ \alpha\end{aligned}$$

Here, $|\cdot|\ a$, which is abbreviated as $|a|$, forms a matrix with a single element a. Given two matrices x and y with the same width, $u \ominus v$ forms a matrix in which u is located above v. Similarly, given two matrices x and y with the same height, $x \oplus y$ forms a matrix in which x is located to the left of y. We say that a matrix formed by $\ominus$ and $\oplus$ has an *abide* structure, where *abide* is a coinage of *above* and *beside*.

In general, a matrix formed using these constructors has many possible representations. For example, a 2×2 matrix has two representations:

$$\begin{aligned}\begin{pmatrix}1 & 2\\ 3 & 4\end{pmatrix} &= (|1| \oplus |2|) \ominus (|3| \oplus |4|) \\ &= (|1| \ominus |3|) \oplus (|2| \ominus |4|).\end{aligned}$$

To avoid this ambiguity in representations, data constructors $\ominus$ and $\oplus$ are made to have two properties that ensure that all representations for a matrix can be regarded as identical.

Associativity The constructors $\ominus$ and $\oplus$ are regarded as associative.

Abide Property The constructors $\ominus$ and $\oplus$ are considered to satisfy the following equation, called the *abide property*.

$$(x \oplus y) \ominus (z \oplus w) = (x \ominus z) \oplus (y \ominus w)$$

The matrix skeletons supported by the SkeTo library are designed based on the abide structure of a matrix. Four important data parallel skeletons are map, reduce, zipwith, and scan. Their intuitive and formal definitions are shown in **Figs. 1** and **2**, respectively.

The map skeleton is the operation that applies a function to every element in a matrix.

The reduce skeleton collapses a matrix into a single value by repeated applications of $\oplus$ for the vertical direction and $\otimes$ for the horizontal direction. Both $\oplus$ and $\otimes$ have to be associative and have to satisfy the abide property, $(x \oplus y) \otimes (z \oplus w) = (x \otimes z) \oplus (y \otimes w)$.

The zipwith skeleton is an extension of map. It takes two matrices of the same shape and returns a matrix of the same shape by applying the function to their corresponding elements.

Similar to reduce, the scan skeleton takes two associative operators, $\oplus$ and $\otimes$, which satisfy the abide property. It returns a matrix that holds all values generated while reducing a matrix by reduce.

Hereafter, all binary operators like $\oplus$ and $\otimes$ are assumed to be associative and all pairs of binary operators are assumed to satisfy the abide property.

During the computation of matrices, it is often necessary to rearrange matrix data among processors. An example is matrix multiplication, where a matrix is rotated in both the row and column directions. SkeTo supports *communication skeletons* for such a rearrangement.

$$\mathsf{map}\ f \begin{pmatrix} x_{11} & x_{12} & \dots & x_{1n} \\ x_{21} & x_{22} & \dots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & \dots & x_{mn} \end{pmatrix} = \begin{pmatrix} f\,x_{11} & f\,x_{12} & \dots & f\,x_{1n} \\ f\,x_{21} & f\,x_{22} & \dots & f\,x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ f\,x_{m1} & f\,x_{m2} & \dots & f\,x_{mn} \end{pmatrix}$$

$$\mathsf{reduce}\ (\oplus,\otimes) \begin{pmatrix} x_{11} & x_{12} & \dots & x_{1n} \\ x_{21} & x_{22} & \dots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & \dots & x_{mn} \end{pmatrix} = \begin{array}{l} (x_{11} \otimes x_{12} \otimes \cdots \otimes x_{1n}) \oplus \\ (x_{21} \otimes x_{22} \otimes \cdots \otimes x_{2n}) \oplus \\ \cdots \\ (x_{m1} \otimes x_{m2} \otimes \cdots \otimes x_{mn}) \end{array}$$

$$\mathsf{zipwith}\ f \begin{pmatrix} x_{11} & x_{12} & \dots & x_{1n} \\ x_{21} & x_{22} & \dots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & \dots & x_{mn} \end{pmatrix} \begin{pmatrix} y_{11} & y_{12} & \dots & y_{1n} \\ y_{21} & y_{22} & \dots & y_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ y_{m1} & y_{m2} & \dots & y_{mn} \end{pmatrix}$$
$$= \begin{pmatrix} f\,x_{11}\,y_{11} & f\,x_{12}\,y_{12} & \dots & f\,x_{1n}\,y_{1n} \\ f\,x_{21}\,y_{21} & f\,x_{22}\,y_{22} & \dots & f\,x_{2n}\,y_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ f\,x_{m1}\,y_{m1} & f\,x_{m2}\,y_{m2} & \dots & f\,x_{mn}\,y_{mn} \end{pmatrix}$$

$$\mathsf{scan}\ (\oplus,\otimes) \begin{pmatrix} x_{11} & x_{12} & \dots & x_{1n} \\ x_{21} & x_{22} & \dots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & \dots & x_{mn} \end{pmatrix} = \begin{pmatrix} y_{11} & y_{12} & \dots & y_{1n} \\ y_{21} & y_{22} & \dots & y_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ y_{m1} & y_{m2} & \dots & y_{mn} \end{pmatrix} \text{ where } y_{ij} = \begin{array}{l} (x_{11} \otimes \cdots \otimes x_{1j}) \oplus \\ (x_{21} \otimes \cdots \otimes x_{2j}) \oplus \\ \cdots \\ (x_{i1} \otimes \cdots \otimes x_{ij}) \end{array}$$

Fig. 1 Intuitive definitions of basic data parallel skeletons.

$$\begin{aligned}
&\mathsf{map} :: (a \to b) \to \mathsf{Matrix}\ a \to \mathsf{Matrix}\ b \\
&\mathsf{map}\ f\ |a| \quad\ = |f\ a| \\
&\mathsf{map}\ f\ (u \circ v) = \mathsf{map}\ f\ u\ \circ\ \mathsf{map}\ f\ v \\
&\mathsf{map}\ f\ (x \diamond y) = \mathsf{map}\ f\ x\ \diamond\ \mathsf{map}\ f\ y
\end{aligned}$$

$$\begin{aligned}
&\mathsf{reduce} :: (a \to a \to a,\ a \to a \to a) \to \mathsf{Matrix}\ a \to a \\
&\mathsf{reduce}\ (\oplus,\otimes)\ |a| \quad\ = a \\
&\mathsf{reduce}\ (\oplus,\otimes)\ (u \circ v) = \mathsf{reduce}\ (\oplus,\otimes)\ u \oplus \mathsf{reduce}\ (\oplus,\otimes)\ v \\
&\mathsf{reduce}\ (\oplus,\otimes)\ (x \diamond y) = \mathsf{reduce}\ (\oplus,\otimes)\ x \otimes \mathsf{reduce}\ (\oplus,\otimes)\ y
\end{aligned}$$

$$\begin{aligned}
&\mathsf{zipwith} :: (a \to b \to c) \to \mathsf{Matrix}\ a \to \mathsf{Matrix}\ b \to \mathsf{Matrix}\ c \\
&\mathsf{zipwith}\ f\ |a|\ |b| \quad\ = |f\ a\ b| \\
&\mathsf{zipwith}\ f\ (u \circ v)\ (s \circ t) \ = \mathsf{zipwith}\ f\ u\ s \circ \mathsf{zipwith}\ f\ v\ t \\
&\mathsf{zipwith}\ f\ (x \diamond y)\ (z \diamond w) = \mathsf{zipwith}\ f\ x\ z \diamond \mathsf{zipwith}\ f\ y\ w
\end{aligned}$$

$$\begin{aligned}
&\mathsf{scan} :: (a \to a \to a,\ a \to a \to a) \to \mathsf{Matrix}\ a \to \mathsf{Matrix}\ a \\
&\mathsf{scan}\ (\oplus,\otimes)\ |a| \quad\ = |a| \\
&\mathsf{scan}\ (\oplus,\otimes)\ (u \circ v) = \mathsf{scan}\ (\oplus,\otimes)\ u \oplus' \mathsf{scan}\ (\oplus,\otimes)\ v \\
&\mathsf{scan}\ (\oplus,\otimes)\ (x \diamond y) = \mathsf{scan}\ (\oplus,\otimes)\ x \otimes' \mathsf{scan}\ (\oplus,\otimes)\ y \\
&\mathsf{where}\ sx \oplus' sy \quad\ = sx \circ (\mathsf{map}_r\ (\mathsf{zipwith}\ (\oplus)\ (\mathit{bottom}\ sx))\ sy) \\
&\qquad sx \otimes' sy \quad\ = sx \diamond (\mathsf{map}_c\ (\mathsf{zipwith}\ (\otimes)\ (\mathit{last}\ sx))\ sy) \\
&\qquad \mathit{bottom} \quad\ = \mathsf{reduce}\ (\gg, \diamond) \circ \mathsf{map}\ |\cdot| \\
&\qquad \mathit{last} \quad\ = \mathsf{reduce}\ (\circ, \gg) \circ \mathsf{map}\ |\cdot| \\
&\qquad \mathsf{map}_r\ f \quad\ = \mathsf{reduce}\ (\circ, \ll) \circ \mathsf{map}\ f \circ \mathit{rows} \\
&\qquad \mathsf{map}_c\ f \quad\ = \mathsf{reduce}\ (\ll, \diamond) \circ \mathsf{map}\ f \circ \mathit{cols} \\
&\qquad \mathit{rows} \quad\ = \mathsf{reduce}\ (\circ, \mathsf{zipwith}\ (\diamond)) \circ \mathsf{map}\ (|\cdot| \circ |\cdot|) \\
&\qquad \mathit{cols} \quad\ = \mathsf{reduce}\ (\mathsf{zipwith}\ (\circ), \diamond) \circ \mathsf{map}\ (|\cdot| \circ |\cdot|)
\end{aligned}$$

Fig. 2 Formal definitions of basic data parallel skeletons.

An instance of a communication skeleton is rot_r which takes a function f and rotates the i-th sub-matrix in the row direction by $f\ i$. For example, by letting $f\ z = -z$ and X_{ij} be a sub-matrix assigned to each processor, we get

$$\mathsf{rot}_r\ f \begin{pmatrix} X_{00} & X_{01} & X_{02} \\ X_{10} & X_{11} & X_{12} \\ X_{20} & X_{21} & X_{22} \end{pmatrix}$$

$$= \begin{pmatrix} X_{00} & X_{01} & X_{02} \\ X_{11} & X_{12} & X_{10} \\ X_{22} & X_{20} & X_{21} \end{pmatrix}.$$

Similarly, rot_c rotates the specified matrix in the column direction. Their formal definitions are omitted here.

3.2 Current Implementation of Matrices

In SkeTo, parallel skeletons for matrices are designed and implemented on the basis of the formalization described in Section 3.1. A matrix is divided into rectangular areas in accordance with its abide structure in such a way that the number of rectangular areas is equal to the number of processors and each area has almost the same number of elements. Each rectangular area is then sent to the corresponding processor. This distribution process is completely concealed within the constructor of the dense matrix class. At each processor, all elements in the assigned rectangular area are stored in a one-dimensional array with an index table that enables skeletons to access a specified element directly.

The current implementation of a matrix does not give any special consideration to sparse matrices. It has two inefficiencies when treating a sparse matrix.

- The data parallel skeletons repeatedly perform the same computation on the zeros. These computations are redundant and waste CPU power.
- Communication skeletons like rot_r send/receive many zeros to/from other processors. This increases the amount of data exchanged between processors and causes superfluous communication.

Considering that many matrix-oriented practical programs are likely to transform a matrix into a sparse one, it is important to improve the efficiency of the SkeTo library when dealing with sparse matrices.

4. Design of Sparse Matrices in SkeTo

4.1 New Formalization of Sparse Matrices

As described in Section 3.2, the current dense matrix class is problematic for sparse matrices. To solve this problem, we revised the formalization of matrices so as to deal with a sub-rectangle of an arbitrary size whose elements are identical. We call this sub-rectangle within a matrix a *block*.

This revised definition of matrices, coined SMatrix to distinguish it from Matrix, is

$$\begin{aligned} \textbf{data}\ \mathsf{SMatrix}\ \alpha = {} & |\cdot|_{Int\times Int}\ \alpha \\ \mid {} & \mathsf{SMatrix}\ \alpha \ \ominus\ \mathsf{SMatrix}\ \alpha \\ \mid {} & \mathsf{SMatrix}\ \alpha \ \oplus\ \mathsf{SMatrix}\ \alpha. \end{aligned}$$

In this definition, $|\cdot|_{p\times q}\ a$, abbreviated as $|a|_{p\times q}$, is a block with the value a in each element and with the size $p \times q$, where p and q are not fixed values. Note that the singleton $|a|$ in Matrix can be regarded as a special case of $|a|_{p\times q}$ in SMatrix, where $p = q = 1$. With this definition, we can represent a sparse matrix using 'large' blocks of zeros.

For example, by letting E_n be the $n \times n$ unit matrix, we can represent E_8 using the SMatrix data structure:

$$\begin{aligned} E_8 = {} & (((E_2 \ominus |0|_{2\times 2}) \oplus (|0|_{2\times 2} \ominus E_2)) \ominus |0|_{4\times 4}) \\ & \oplus (|0|_{4\times 4} \ominus ((E_2 \ominus |0|_{2\times 2}) \oplus (|0|_{2\times 2} \ominus E_2))) \\ & \textbf{where} \\ & E_2 = (|1|_{1\times 1} \ominus |0|_{1\times 1}) \oplus (|0|_{1\times 1} \ominus |1|_{1\times 1}) \end{aligned}$$

For the purpose of representing a sparse matrix that has many zeros, one could fix the value of a block to zero and specify only the size of the block. However, we did not take this approach because it lacks flexibility; the block must be partitioned even though all its elements are equally updated to some non-zero value. It is more efficient to retain the block shape as long as all elements in the block are identical (their value does not need to be zero) and to delay partitioning until there are different values in the block.

Similar to the case of a dense matrix, we assume the following identifications to avoid ambiguities in the representation.

$$\begin{aligned} |a|_{m\times n} &= |a|_{m_1\times n} \ominus |a|_{m_2\times n} \\ &= |a|_{m\times n_1} \oplus |a|_{m\times n_2} \\ &\quad (m = m_1 + m_2,\ n = n_1 + n_2) \end{aligned}$$

4.2 New Definitions of Matrix Skeletons

Using this new formalization of sparse matrices, we re-defined the basic data parallel skeletons, as shown in **Fig. 3**.

In the map skeleton, since all elements in a block are identical and thus have the same resulting value of a function application, the result of map is a block of the same size as the input. Note that for a block of size $p \times q$, it is not necessary to apply the function pq times; once is enough.

In the reduce skeleton, for a block of value a and of size $p \times q$, it is sufficient to compute

$$
\begin{aligned}
&\mathsf{map} :: (a \to b) \to \mathsf{SMatrix}\ a \to \mathsf{SMatrix}\ b\\
&\mathsf{map}\ f\ |a|_{p\times q} = |f\ a|_{p\times q}\\
&\mathsf{map}\ f\ (u \ominus v) = \mathsf{map}\ f\ u \ominus \mathsf{map}\ f\ v\\
&\mathsf{map}\ f\ (x \oplus y) = \mathsf{map}\ f\ x \oplus \mathsf{map}\ f\ y
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{reduce} :: (a \to a \to a,\ a \to a \to a) \to \mathsf{SMatrix}\ a \to a\\
&\mathsf{reduce}\ (\oplus, \otimes)\ |a|_{p\times q} = r\ (\oplus)\ p\ (r\ (\otimes)\ q\ a)\\
&\qquad\qquad \mathsf{where}\ r\ (\odot)\ 1\ x = x\\
&\qquad\qquad\qquad r\ (\odot)\ (k+1)\ x = x \odot r\ (\odot)\ k\ x\\
&\mathsf{reduce}\ (\oplus, \otimes)\ (u \ominus v) = \mathsf{reduce}\ (\oplus, \otimes)\ u \oplus \mathsf{reduce}\ (\oplus, \otimes)\ v\\
&\mathsf{reduce}\ (\oplus, \otimes)\ (x \oplus y) = \mathsf{reduce}\ (\oplus, \otimes)\ x \otimes \mathsf{reduce}\ (\oplus, \otimes)\ y
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{zipwith} :: (a \to b \to c) \to \mathsf{SMatrix}\ a \to \mathsf{SMatrix}\ b \to \mathsf{SMatrix}\ c\\
&\mathsf{zipwith}\ f\ |a|_{p\times q}\ |b|_{p\times q} = |f\ a\ b|_{p\times q}\\
&\mathsf{zipwith}\ f\ (u \ominus v)\ (s \ominus t) = \mathsf{zipwith}\ f\ u\ s \ominus \mathsf{zipwith}\ f\ v\ t\\
&\mathsf{zipwith}\ f\ (x \oplus y)\ (z \oplus w) = \mathsf{zipwith}\ f\ x\ z \oplus \mathsf{zipwith}\ f\ y\ w\\
&\mathsf{zipwith}\ f\ |a|_{p\times q}\ (u \ominus v) = \mathsf{map}\ (f\ a)\ u \ominus \mathsf{map}\ (f\ a)\ v\\
&\mathsf{zipwith}\ f\ |a|_{p\times q}\ (x \oplus y) = \mathsf{map}\ (f\ a)\ x \oplus \mathsf{map}\ (f\ a)\ y\\
&\mathsf{zipwith}\ f\ (u \ominus v)\ |a|_{p\times q} = \mathsf{map}\ g\ u \ominus \mathsf{map}\ g\ v \quad \mathsf{where}\ g\ x = f\ x\ a\\
&\mathsf{zipwith}\ f\ (x \oplus y)\ |a|_{p\times q} = \mathsf{map}\ g\ x \oplus \mathsf{map}\ g\ y \quad \mathsf{where}\ g\ x = f\ x\ a
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{scan} :: (a \to a \to a,\ a \to a \to a) \to \mathsf{SMatrix}\ a \to \mathsf{SMatrix}\ a\\
&\mathsf{scan}\ (\oplus, \otimes)\ |a|_{p\times q} = r\ (\oplus')\ p\ (r\ (\otimes')\ q\ |a|_{1\times 1})\\
&\qquad\qquad \mathsf{where}\quad \text{definition of } r \text{ is the same as that of } \mathsf{reduce}\\
&\mathsf{scan}\ (\oplus, \otimes)\ (u \ominus v) = \mathsf{scan}\ (\oplus, \otimes)\ u \oplus' \mathsf{scan}\ (\oplus, \otimes)\ v\\
&\mathsf{scan}\ (\oplus, \otimes)\ (x \oplus y) = \mathsf{scan}\ (\oplus, \otimes)\ x \otimes' \mathsf{scan}\ (\oplus, \otimes)\ y\\
&\mathsf{where}\quad \text{definitions of } \oplus' \text{ and } \otimes' \text{ are the same as those in Fig. 2}
\end{aligned}
$$

Fig. 3 New formal definitions of basic data parallel skeletons.

$a \otimes a \otimes \cdots \otimes a$ only once. Local function r is used for the repeated application of the binary operator $\oplus$ or $\otimes$ to the same value. Though r is defined to take linear time for simplicity, it can be implemented in a divide-and-conquer manner as will be mentioned in Section 5.3.

The zipwith skeleton is almost the same as map except that it takes two matrices. If one is a block and the other is not, the result of zipwith cannot be a block; the result has to be constructed using the same constructor, $\ominus$ or $\oplus$, as that of the non-block.

In the scan skeleton, the reduction part is almost the same as with the reduce skeleton. The same local function, r, as that of reduce is used in scan.

5. Implementation of Sparse Matrix Skeletons

5.1 Data Structures

Using the formalization described in Section 4.1, we implemented the sparse matrix class as a library of SkeTo following two design principles. First, even if the sparse matrix skeletons are applied to dense matrices, their overhead must be within a permissible range, i.e., small enough compared to that of the existing dense matrix skeletons. Second, the sparse matrix skeletons must have exactly the same interfaces as those of existing dense skeletons. This is very important from the viewpoint of compatibility; programmers should encounter no barrier in using the new sparse skeleton class.

In this implementation, we separated the method for holding the element values in a matrix from that for managing the abide structure of the matrix (how a matrix is constructed). That is, each processor has the following data structures for the assigned rectangular area within a matrix.

- A one-dimensional array holds all the elements in the assigned area.
- A binary tree whose internal nodes are either 'above' or 'beside' holds the abide structure of the area. A leaf of this tree is called a *sub-matrix* and is either a block of identical values ($|a|_{p\times q}$) or a sub-area with a rectangular shape that contains non-identical values. Each internal node and each leaf has the index information for the upper-left element and the size information. In addition, each leaf has a flag value indicating whether it is a block or not and an index table that enables skeletons to access a specific element directly.

Figure 4 shows a fragment of the new definition of the `dist_matrix` sparse matrix class with accompanying classes for the tree structure. The tree structure is represented by an abstract class `ab_tree` and its child classes.

For example, the 8×8 unit matrix has the

```
template <typename A>
class dist_matrix {           // new definition for sparse matrix
  friend class matrix_skeletons; // class that defines matrix skeletons
  int myrank;                 // rank (processor id) of MPI
  int n;                      // number of rows of the entire matrix
  int m;                      // number of columns of the entire matrix
  int localRows;              // number of rows of the assigned area
  int localCols;              // number of columns of the assigned area
  int bir;                    // number of blocks in rows
  int bic;                    // number of blocks in columns
  A *ele;                     // one-dimensional array of elements
  ab_tree<A> *t;              // tree structure
  ...
};

template <typename A>
class ab_tree {               // abstract class for tree structure
  int rows, cols;             // number of rows and cols of the tree
  int rowidx, colidx;         // index information of upper-left corner
  ...
};

template <typename A>
class above_node : public ab_tree<A> {
  ab_tree<A> *up, *down;      // two subtrees
  ...
};

// class beside_node is defined in almost the same manner as above_node

template <typename A>
class sub_matrix : public ab_tree<A> {
  bool isBlock;               // flag indicating whether it is a block or not
  A **idxEle;                 // index table
  ...
};
```

Fig. 4 New class definitions for sparse matrices.

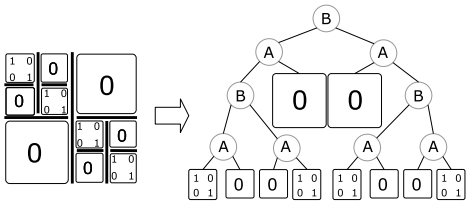


Fig. 5 Tree structure for 8×8 unit matrix (A: above node, B: beside node).

internal representation shown in **Fig. 5**. There are three main advantages of this internal representation. First, when we separate a sub-matrix into two smaller sub-matrices or merge two adjacent sub-matrices into a larger sub-matrix, we do not have to allocate a memory space for the elements in the new sub-matrix and copy the elements into this memory area. Second, although this representation does not save the memory space for elements of a sparse matrix, it is quite easy to implement the separation of a sub-matrix and the merging of two sub-matrices with much flexibility. Third, this representation substantially matches the SMatrix data type; skeletons can be naturally implemented in accordance with their definitions (Fig. 3).

5.2 Tree Structure Construction

When we make a new instance of the sparse matrix class, elements of a matrix are automatically distributed to each processor by the constructor of the class. This automatic distribution is exactly the same as with the existing dense matrix class. The construction of a tree structure on each processor consists of three steps.

Step 1 Vertically and horizontally cut the assigned rectangular area an appropriate number of times to separate it into sub-matrices and build a tree structure in which 'above' and 'beside' nodes alternately appear on each path from the root to a leaf.

Step 2 Determine whether each leaf (sub-matrix) is a block of identical values or not and store the result in the flag (`isBlock` of `sub_matrix` class in Fig. 4) of the leaf.

Step 3 To avoid waste, recursively merge two

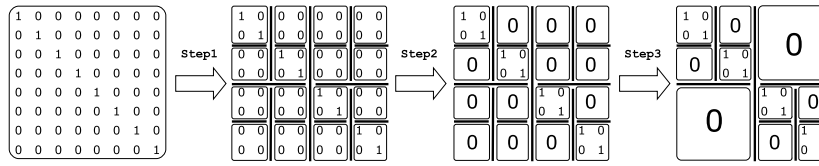


Fig. 6 Constructing a tree structure for 8×8 unit matrix.

```
// skeleton functions are defined in the matrix_skeletons class
template <typename A, typename F>
void matrix_skeletons::map_i(const F &f, dist_matrix<A> *mat)
{
  mat->t->map_i(f);
}

template <typename A> template <typename F>
void above_node<A>::map_i(const F &f)
{
  up->map_i(f);
  down->map_i(f);
}

// beside_node<A>::map_i is defined almost the same manner as above_node<A>::map_i

template <typename A> template <typename F>
void sub_matrix<A>::map_i(const F &f)
{
  if (isBlock) apply f to the representative element;
  else apply f to all elements;
}
```

Fig. 7 Parts of a new implementation of map skeleton.

adjacent leaves with the same parent node into a larger leaf if one of the following conditions is satisfied.

- The two adjacent leaves are blocks and their values are equal.
- Neither of the two adjacent leaves is a block.

Figure 6 shows an example of this process.

In the current implementation, the number of times of vertically or horizontally cutting is empirically determined to be $k^{1/4}$, where k is the vertical or horizontal size. For example, if the size of the area assigned to some processor is $10{,}240 \times 10{,}240$, the area is cut $10(\approx 10{,}240^{1/4})$ times for each direction. Thus, the smallest sub-matrix is 10×10 because $10{,}240/2^{10} = 10$.

There are several points to be noted. First, even if a sub-matrix seems to be too small after Step 1, many sub-matrices will likely be merged in Step 3 because it will be a rare case in which blocks and non-blocks are alternately located so as to prevent merging. Thus, we will probably be able to construct a tree structure whose sub-matrices have appropriate sizes. Second, if these steps are applied to a dense matrix that contains no block after Step 2, non-blocks are eventually integrated into a single sub-matrix in Step 3. Although this separation and integration generates an overhead when given a dense matrix, the experimental results (Section 6) showed that this overhead is small.

5.3 Skeleton Implementation

We implemented skeletons for sparse matrices using the formalization described in Section 4.1 and the data structures described in Section 5.1.

In the implementation of the map skeleton, although the specified function has to be applied to all elements in each non-block leaf, it is applied only once to a representative element for each block. Thus, for a block of size $p \times q$, the cost of this skeleton is $O(1)$ while map for a non-block of the same size has a cost of $O(pq)$. **Figure 7** shows a C++ function, `map_i`, an implementation of the map skeleton that overwrites an input matrix. The sparse matrix skeletons called from a user's program are defined in the `matrix_skeletons` class.

The reduce skeleton first performs reduction for each leaf (sub-matrix) of the tree structure and then combines them using two specified operators (functions) on the basis of the

```
template <typename F, typename G>
void matrix_skeletons::reduce(const F &oplus, const G &otimes, dist_matrix<A> *mat, A* res)
{
  A val = mat->t->reduce(oplus, otimes);                // locally compute the reduce
  int bicfloor = (mat->myrank / mat->bic) * mat->bic;   // rank of the leftmost processor
  // communication in the row direction
  for( int stage = 1; stage < mat->bic; stage <<= 1 ) { // communication based on the tree structure
    int target = ((mat->myrank % mat->bic) ^ stage) + bicfloor; // target of the communication
    if ( target < mat->myrank ) {
      send a value;
      break;
    } else if ( ((mat->myrank % mat->bic) ^ stage) < mat->bic ) { // target exists
      A rval;
      receive a value into rval;
      A tmp = val;
      otimes(tmp, rval, &val);     // combine rval and value using otimes
    }
  }
  // communication in the column direction
  if ( mat->myrank % mat->bic == 0 )
    for(int stage = 1; stage < mat->bir; stage <<= 1) {
      int target = ((mat->myrank / mat->bic) ^ stage) * mat->bic;
      if( target < mat->myrank ) {
        send a value;
        break;
      } else if ( ((mat->myrank / mat->bic) ^ stage) < mat->bir ) {
        A rval;
        receive a value into rval;
        A tmp = val;
        oplus(tmp, rval, &val);
      }
    }
  }
  *res = val;
}

template <typename A> template <typename F, typename G>
A above_node<A>::reduce(const F &oplus, const G &otimes)
{
  A tmp;
  oplus(up->reduce(oplus, otimes), down->reduce(oplus, otimes), &tmp);
  return tmp;
}

template <typename A> template <typename F, typename G>
A beside_node<A>::reduce(const F &oplus, const G &otimes)
{
  A tmp;
  otimes(left->reduce(oplus, otimes), right->reduce(oplus, otimes), &tmp);
  return tmp;
}

template <typename A> template <typename F, typename G>
A sub_matrix<A>::reduce(const F &oplus, const G &otimes)
{
  if (isBlock){
    A res = sum(otimes, localCols, get());  // get returns the value of the block
    return sum(oplus, localRows, res);
  } else{
    reduce the values using the code for dense matrix;
  }
}

template <typename A> template <typename F>
A sub_matrix<A>::sum(const F &op, int count, const A &val)
{
  if( count <= 1 ) return val;
  A tmp1 = sum(op, count / 2, val);
  A tmp2;
  op(tmp1, tmp1, &tmp2);
  if ( count % 2 == 0 ) return tmp2;
  op(tmp2, val, &tmp1);
  return tmp1;
}
```

Fig. 8 Parts of a new implementation of reduce skeleton.

abide structure of the tree. For the reduction of a block of size $p \times q$, computation time is $O(\log p + \log q)$ because the value can be computed in a divide-and-conquer manner. **Figure 8** shows a C++ function for the reduce skeleton.

The implementation of the zipwith skeleton is almost the same as that of the map one. For two blocks of the same size, the specified function is applied only once to their representative elements, so the cost is O(1) for a block.

The implementation of the scan skeleton consists of six steps: (1) the local reduction of the row direction, (2) the communication of the resulting value of the local reduction to a neighboring processor in the row direction, (3) the local scan of the row direction, (4) the local reduction of the column direction, (5) the communication of the resulting value of the local reduction to a neighboring processor in the column direction, and (6) the local scan of the column direction. Steps (1) and (4) can be efficiently implemented for a block.

Because the map and zipwith skeletons update only the representative element of a block, they cause a somewhat 'inconsistent' situation in which the block's representative and other elements differ in the one-dimensional array that holds element values. If we eagerly copy the representative's correct value to the other elements, the copying overhead is not negligible. This overhead is minimized by having each skeleton defined for sparse matrices manage a flag value that indicates whether the representative value has already been copied or not. A skeleton can then determine the necessity of the copying and, if necessary, perform the copying lazily in a demand-driven manner.

In the implementation of the rot_r communication skeleton, we coded a serializer of a tree structure, which enables each processor to efficiently send/receive the abide structure of the assigned rectangular area without duplicated communications for a block having identical values. With this serializer, both the amount of exchanged data and the time for constructing a tree structure are reduced.

6. Evaluation

We experimentally evaluated the effectiveness of the new implementations. The parallel environment used was a PC cluster system with 16 processors connected by a gigabit Ethernet. Each PC had an Intel Pentium 4 (3.0 GHz) CPU and 1 GB of main memory. The operating system was Linux kernel 2.6.8., the C compiler was GCC 3.3.5 with optimizing level O2, and the MPI implementation was MPICH 1.2.6.

In some experiments, we used six sparse matrices taken from the University of Florida database [19]. **Figure 9** shows views of these matrices, where black dots mean non-zero values.

In the tables in this section, P is the number of processors, 'sparse' means the skeleton for sparse matrices, and 'dense' means the existing skeleton for dense matrices. In addition, E_n stands for the $n \times n$ unit matrix and D_n stands for the $n \times n$ dense matrix whose (i, j)-element is $i - j$.

6.1 Micro Benchmark

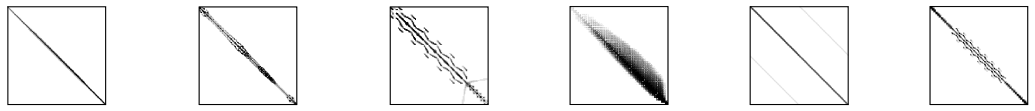
Data structures for representing a sparse matrix are created in the data constructor of the sparse matrix class. Tree construction described in Section 5.2 is an extra work of the constructor compared to that of the dense matrix class. To evaluate this overhead, we measured the execution times of data constructors, each of which is the time needed to distribute the matrix elements in an array on the root processor among processors and to construct the data structures on each processor in parallel. The results are shown in **Table 1**. We can see that the overhead of tree construction is not noticeable because it is much less than the data communication time.

Next, to determine whether each of the four basic data skeletons for a sparse matrix had sufficient performance as a component used in a large parallel program, we measured the performance of each skeleton for the sparse matrix class and compared it with that for the existing dense matrix class.

First, we gave as input a unit matrix, an instance of an ideal sparse matrix, to evaluate the speedup of the sparse matrix skeletons. Next, we gave as input a dense matrix to determine the worst case performances of the sparse matrix skeletons compared to those of the dense matrix skeletons.

To clarify the basic performance of each sparse matrix skeleton, we set three experimental conditions.

- Each C++ function for each skeleton does not newly allocate the resulting matrix, which eliminates the data allocation time; the input matrix or a pre-allocated matrix is overwritten with the results of map, zip-



S_1 ($7{,}055 \times 7{,}055$) S_2 ($10{,}974 \times 10{,}974$) S_3 (656×656) S_4 (765×765) S_5 ($2{,}000 \times 2{,}000$) S_6 ($2{,}283 \times 2{,}283$)

Fig. 9 Views of sparse matrices used in experiments.

Table 1 Results of micro benchmark for constructors (data distribution).

	distribution of S_1			distribution of S_4			distribution of D_{8192}		
P	sparse (sec)	dense (sec)	ratio (%)	sparse (msec)	dense (msec)	ratio (%)	sparse (sec)	dense (sec)	ratio (%)
2	3.33	3.28	102	42.2	42.1	100	5.15	4.85	106
4	2.89	2.63	110	23.3	23.4	99	3.67	3.64	101
8	3.19	3.09	103	30.0	27.4	109	4.14	4.22	98
16	3.34	3.31	101	30.1	29.4	102	4.43	4.45	100

Table 2 Results of micro benchmark for data parallel skeletons ($n = 8{,}192$).

	map $(99*)$ E_n			reduce $(+,+)$ E_n			zipwith $(+)$ E_n E_n			scan $(+,+)$ E_n		
P	sparse (msec)	dense (msec)	ratio (%)	sparse (msec)	dense (msec)	ratio (%)	sparse (msec)	dense (msec)	ratio (%)	sparse (msec)	dense (msec)	ratio (%)
2	1.48	81.9	1.80	2.08	48.9	4.25	3.35	122	2.73	4,200	6,080	69.1
4	0.784	40.9	1.92	1.85	25.9	7.15	2.56	61.0	4.20	1,780	2,670	66.7
8	0.632	20.8	3.04	2.07	13.8	15.0	1.39	30.9	4.49	905	1,340	67.6
16	0.152	11.7	1.31	2.42	7.39	32.7	1.05	15.9	6.59	340	536	63.5
	map $(99*)$ D_n			reduce $(+,+)$ D_n			zipwith $(+)$ D_n D_n			scan $(+,+)$ D_n		
P	sparse (msec)	dense (msec)	ratio (%)	sparse (msec)	dense (msec)	ratio (%)	sparse (msec)	dense (msec)	ratio (%)	sparse (msec)	dense (msec)	ratio (%)
2	84.2	81.9	103	44.0	49.0	89.7	119	121	98.7	6,110	6,080	101
4	42.4	40.9	104	22.6	26.0	86.9	60.0	61.2	98.0	2,660	2,670	99.4
8	21.0	20.9	100	12.0	13.4	89.5	30.1	30.7	98.0	1,340	1,340	99.9
16	10.5	10.4	101	6.88	7.34	93.7	15.7	15.7	99.9	538	536	100

with and scan.

- The input matrix for each skeleton is distributed beforehand to eliminate the time of the constructor.
- The functions (operators) given to each skeleton are simple ones such as addition or multiplication.

The results are shown in **Table 2**. From this table, we can see that the map and zipwith skeletons, given unit matrix E_{8192}, showed good speedups of less than 10% compared to those for dense matrices. For the reduce skeleton, the speedup was not as good as that for map or reduce, but it was also satisfactory. For the scan skeleton, the speedup was almost 2/3. The reason for this moderate speedup is that, as described in Section 5.3 only two of the six computation steps are likely to be improved.

For a dense matrix, D_{8192}, the new skeletons for sparse matrices have almost the same performance as the existing ones. Even in the worst case, the overhead is within the amount permissible, 10%. For the reduce, scan, and zipwith skeletons, most of the experimental results show that the sparse matrix skeletons were faster, which means that the overhead, i.e., the management cost, of the tree structure is negligible.

There are two points to be noted. First, a super-linear effect is observed in the results of scan, because its computation time is mainly governed by the control structure in its implementation due to the simplicity of the + operation. However, this simple operation suffices for the purpose of this experiment, i.e., the evaluation of the new scan. Second, the parallelization effects of the sparse matrix skeletons for E_n are not as clear as those of the dense matrix skeletons. This is because the effect of the sparse matrix skeletons mainly depends on how a target matrix is separated and formed using blocks of identical values. Typically, if a zero matrix were given to map, we would have almost the same execution time independently of the number of processors because each processor is assigned a smaller zero matrix represented by a single block.

For the rot_r communication skeleton, we mea-

Table 3 Results of micro benchmark for communication skeleton (n = 8,192, $f\,z = -1$).

P	$\mathsf{rot}_r\ f\ E_n$			$\mathsf{rot}_r\ f\ D_n$		
	sparse (sec)	dense (sec)	ratio (%)	sparse (sec)	dense (sec)	ratio (%)
2	0.103	4.95	2.09	5.41	4.98	109
4	0.0469	2.50	1.88	2.73	2.52	108
8	0.0303	1.50	2.02	1.61	1.51	107
16	0.0159	0.753	2.12	0.817	0.760	108

```
double fnorm(dist_matrix<double> &mat) {
  matrix_skeletons::map_i(Sqr<double>(), &mat);
  double ss;
  matrix_skeletons::reduce(Add<double>(), Add<double>(), &mat, &ss);
  return sqrt(ss);
}
```

Fig. 10 Parts of SkeTo code for FNORM.

sured the execution times of a program that called rot_r skeleton many times and calculated the time needed for one rotation. The results of this experiment are shown in **Table 3**. For the sparse matrix (E_{8192}), we can see the effect of the serialization; the time needed to rotate is under 3%. For the dense matrix, the overhead of the serialization was not large, i.e., within 10%.

6.2 Macro Benchmark

To examine the performance of the proposed skeletons in practical problems, we measured the execution times for the following programs using the SkeTo library.

Frobenius Norm (FNORM) Frobenius Norm is a norm of a matrix that can be easily calculated using

$$fnorm\ A \equiv \left(\sum_{i=1}^{n}\sum_{j=1}^{m}|a_{ij}|^2\right)^{1/2},$$

where $A = (a_{ij})$ is an $n \times m$ matrix. A function that computes this value can be defined in Haskell as follows.

$$fnorm = \mathsf{reduce}\ (+,+) \circ \mathsf{map}\ sqr,$$

where sqr is a squaring function. SkeTo's code for this function is shown in **Fig. 10**. This program uses the map and reduce skeletons.

Maximum Rectangle Sum (MRS) This problem is to compute the maximum of the sums of all the rectangular areas in a matrix. For example, for

$$\begin{pmatrix} -3 & 5 & -4 & -8 & 3 & -3 \\ -6 & -8 & 2 & -5 & 4 & 1 \\ 9 & -9 & \mathbf{3} & \mathbf{6} & -5 & 2 \\ 5 & -7 & \mathbf{8} & \mathbf{-2} & 2 & -6 \end{pmatrix},$$

the answer is 15, which is taken from the sub-rectangular area with the numbers in bold. A naive function to solve MRS problem is defined as

$$\begin{aligned} mrs &= max \circ \mathsf{map}\ max \circ \\ &\qquad \mathsf{map}\ (\mathsf{map}\ sum) \circ rects \\ \mathsf{where}& \\ &\qquad max = \mathsf{reduce}(\uparrow,\uparrow) \\ &\qquad sum = \mathsf{reduce}(+,+), \end{aligned}$$

where $rects$ generates a list of all possible sub-rectangles and $x \uparrow y$ returns the bigger of x and y. An efficient program can be derived from this naive specification using program calculation; the detailed process is described elsewhere [11]. The derived program uses a tuple of ten elements and is very complicated. It uses the map, reduce, and zipwith skeletons. The SkeTo program used in this benchmark is the result of this optimization, which is also very complicated and thus is omitted here.

Matrix Multiplication (MM) This program for matrix multiplication uses Cannon's algorithm [13], and is also complex. **Figure 11** shows the SkeTo code for MM used in this benchmark. The main loop of this program consists of communications between processors by matrix rotations and local computations for sub-matrix (rectangular area) multiplication. For the communications, this program calls communication skeletons rot_r and rot_c. For the local computations, this program uses nested for-loops and does not use data parallel skeletons. Thus the speedup is generated by the efficient implementation of rot_r and rot_c skeletons for sparse matrices.

The results without the times for data distribution (by constructors) are shown in **Table 4**. For the programs that combine suitable skeletons, our sparse matrix skeletons can reduce ex-

```
template <typename C, typename A, typename B>
void MatrixMultiply(dist_matrix<C>& Z, dist_matrix<A>& X, dist_matrix<B>& Y)
{
  matrix_skeletons::rotateCols(Neg(), &X);
  matrix_skeletons::rotateRows(Neg(), &Y);
  int q = X.getBlocksInRow();
  for ( int i = 0; i < q; i++ ) {
    MatrixMultiplyLocal(Z, X, Y);
    matrix_skeletons::rotateColsConst(1, &X);
    matrix_skeletons::rotateRowsConst(1, &Y);
  }
}
```

Fig. 11 Parts of SkeTo code for MM.

Table 4 Results of macro benchmark.

	FNORM S_1			FNORM S_2		
P	sparse (msec)	dense (msec)	ratio (%)	sparse (msec)	dense (msec)	ratio (%)
4	13.3	56.8	23.5	43.5	147	29.5
8	4.58	30.2	15.2	13.9	72.4	19.1
16	4.12	15.0	27.8	13.1	37.9	34.5
	MRS S_3			MRS S_4		
P	sparse (sec)	dense (sec)	ratio (%)	sparse (sec)	dense (sec)	ratio (%)
4	8.92	429	2.08	34.3	1,040	3.31
8	8.30	218	3.82	32.0	404	7.91
16	7.26	25.5	28.5	30.4	51.3	59.3
	MM S_5			MM S_6		
P	sparse (sec)	dense (sec)	ratio (%)	sparse (sec)	dense (sec)	ratio (%)
4	5.66	6.69	84.6	8.43	9.76	86.4
9	2.69	3.42	78.6	4.09	4.93	83.0
16	1.60	1.91	83.8	2.52	2.78	90.5

ecution times, in most cases under 35%, compared to the use of existing skeletons for dense matrices. The reason of the super-linear results of MRS is that this program needs $O(n^4)$ time for an $n \times n$ matrix.

7. Conclusion

We have formalized the sparse matrix data type by using a block of identical values. We have also implemented a new C++ class of efficient sparse matrix skeletons in the SkeTo library that are based on the new formalization. These new implementations overcome the inefficiency of existing matrix skeletons when they are applied to sparse matrices. Experimental results show that the new skeletons for sparse matrices perform well compared with existing skeletons for dense matrices.

Currently, the elements in a sparse matrix are distributed among the processors in such a way that each processor is assigned almost the same number of elements. However, the processor loads are not balanced if one processor is assigned a large block of zeros while another is assigned a large non-block with various values. We thus plan to examine the distribution of matrix elements from the viewpoint of load balancing. We also plan to work on reducing the memory space for a block of identical values.

We will incorporate the result of this work into a new version of the SkeTo library and release it on the SkeTo web page [21].

References

1) Bacci, B., Danelutto, M., Orlando, S., Pelagatti, S. and Vanneschi, M.: P3L: A Structured High Level Programming Language and its Structured Support, *Concurrency: Practice and Experience*, Vol.7, No.3, pp.225–255 (1995).
2) Benoit, A., Cole, M., Hillston, J. and Gilmore, S.: Flexible Skeletal Programming with eSkel, *Proc. 11th International Euro-Par Conference* (*Euro-Par2005*), *Lecture Notes in Computer Science*, Vol.3648, pp.761–770, Springer-Verlag (2005).
3) Bird, R.: An Introduction to the Theory of

Lists, *Proc. NATO Advanced Study Institute on Logic of Programming and Calculi of Discrete Design*, pp.5–42 (1987).
4) Bird, R.: *Lectures on Constructive Functional Programming*, Technical Monograph PRG-69, Oxford University Computing Laboratory (1988).
5) Botorog, G.H. and Kuchen, H.: Skil: An Imperative Language with Algorithmic Skeletons, *Proc. 5th International Symposium on High Performance Distributed Computing* (*HPDC'96*), pp.243–252 (1996).
6) Botorog, G.H. and Kuchen, H.: Efficient High-Level Parallel Programming, *Theoretical Computer Science*, Vol.196, No.1–2, pp.71–107 (1998).
7) Cole, M.: *Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computation*, Research Monographs in Parallel and Distributed Computing, Pitman (1989).
8) Cold, M.: Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming, *Parallel Computing*, Vol.30, No.3, pp.389–406 (2004).
9) Danelutto, M., Pasqualetti, F. and Pelagatti, S.: Skeletons for Data Parallelism in P3L, *Proc. 3rd International Euro-Par Conference* (*Euro-Par'97*), *Lecture Notes in Computer Science*, Vol.1300, pp.619–628, Springer-Verlag (1997).
10) Darlington, J., Field, A.J., Harrison, P.G., Kelly, P.H.J., Sharp, D.W.N. and Wu, Q.: Parallel Programming using Skeleton Functions, *Proc. Conference on Parallel Architectures and Reduction Languages Europe* (*PARLE'93*), *Lecture Notes in Computer Science*, Vol.694, pp.146–160, Springer-Verlag (1993).
11) Emoto, K., Hu, Z., Kakehi, K. and Takeichi, M.: A Compositional Framework for Developing Parallel Programs on Two-Dimensional Arrays, *International Journal of Parallel Programming*, Vol.35, No.6, pp.615–658 (2007).
12) Gill, A., Launchbury, J. and Jones, S.P.: A Short Cut to Deforestation, *Proc. 1993 Conference on Functional Programming Languages and Computer Architecture*, pp.223–232 (1993).
13) Grama, A., Gupta, A., Karypis, G. and Kuma, V.: *Introduction to Parallel Computing*, Addison-Wesley (2003).
14) Hu, Z., Iwasaki, H. and Takeichi, M.: An accumulative parallel skeleton for all, *Proc. 11st European Symposium on Programming* (*ESOP 2002*), *Lecture Notes in Computer Science*, Vol.2305, pp.83–97 (2002).
15) Intel Threading Building Blocks 1.1. http://www.intel.com/cd/software/products/asmo-na/eng/294797.htm
16) Kuchen, H.: A Skeleton Library, *Proc. 8th International Euro-Par Conference* (*Euro-Par2002*), *Lecture Notes in Computer Science*, Vol.2400, pp.620–629, Springer-Verlag (2002).
17) Matsuzaki, K., Emoto, K., Iwasaki, H. and Hu, Z.: A Library of Constructive Skeletons for Sequential Style of Parallel Programming, *Proc. 1st International Conference on Scalable Information Systems* (*InfoScale 2006*) (2006).
18) Matsuzaki, K., Hu, Z. and Takeichi, M.: Parallel Skeletons for Manipulating General Trees, *Parallel Computing*, Vol.32, No.7–8, pp.590–603 (2006).
19) University of Florida Sparse Matrix Collection. http://www.cise.ufl.edu/research/sparse/matrices/
20) Rabhi, F.A. and Gorlatch S.G. (Eds): *Patterns and Skeletons for Parallel and Distributed Computing*, Springer-Verlag (2002).
21) SkeTo Project. http://www.ipl.t.u-tokyo.ac.jp/sketo/
22) Wise, D.S.: Representing Matrices as Quadtrees for Parallel Processors, *Information Processing Letters*, Vol.20, No.4, pp.195–199 (1984).
23) Portable, Extensible Toolkit for Scientific Computation. http://www.mcs.anl.gov/petsc/

(Received September 12, 2007)
(Accepted December 10, 2007)

Yuki Karasawa is currently enrolled in the Department of Computer Science, Graduate school of Electro-Communications at the University of Electro-Communications. He was born in 1984, and recieved his B.S. degree from the University of Electro-Communications in March, 2007. His current research interests include skeletal parallel programming and network programming.

Hideya Iwasaki is a professor at the Department of Computer Science, the University of Electro-Communications in Japan. He received degrees of B. Eng. in 1985 and Dr. Eng. in 1988. both from the University of Tokyo. After working as a research associate in the Department of Mathematical Engineering in the University of Tokyo, he joined the Educational Computer Centre in the University of Tokyo as an associate professor in 1993. He later served as an associate professor at Tokyo University of Agriculture and Technology and the University of Tokyo. In 2001, he joined the University of Electro-Communications and was appointed a professor in 2004. His research interests include programming language systems, system software, and parallel and distributed systems.