International Journal of Parallel Programming, Vol. 35, No. 6, December 2007 (© 2007)
DOI: 10.1007/s10766-007-0043-4

A Compositional Framework for Developing Parallel Programs on Two-Dimensional Arrays

Kento Emoto,[1,4] Zhenjiang Hu,[2] Kazuhiko Kakehi,[3] and Masato Takeichi[2]

Received December 29, 2005; accepted March 23, 2007

Computations on two-dimensional arrays such as matrices and images are one of the most fundamental and ubiquitous things in computational science and its vast application areas, but development of efficient parallel programs on two-dimensional arrays is known to be hard. In this paper, we propose a compositional framework that supports users, even with little knowledge about parallel machines, to develop both correct and efficient parallel programs on dense two-dimensional arrays systematically. The key feature of our framework is a novel use of the abide-tree representation of two-dimensional arrays. The presentation not only inherits the advantages of tree representations of matrices where recursive blocked algorithms can be defined to achieve better performance, but also supports transformational development of parallel programs and architecture-independent implementation owing to its solid theoretical foundation – the theory of constructive algorithmics.

KEY WORDS: Constructive algorithmics; skeletal parallel programming; matrix.

[1]Department of Creative Informatics, Graduate School of Information Science and Technology, The University of Tokyo, 7–3–1 Hongo, Bunkyo, Tokyo, 113-8656, Japan. E-mail: emoto@ipl.t.u-tokyo.ac.jp
[2]Department of Mathematical Informatics, Graduate School of Information Science and Technology, The University of Tokyo, 7–3–1 Hongo, Bunkyo, Tokyo, 113-8656, Japan. E-mails: {hu,takeichi}@mist.i.u-tokyo.ac.jp
[3]Division of University Corporate Relations, The University of Tokyo, 7–3–1 Hongo, Bunkyo, Tokyo, 113-8656, Japan. E-mail: kaz@ipl.t.u-tokyo.ac.jp
[4]To whom correspondence should be addressed.

1. INTRODUCTION

Computations on two-dimensional arrays, such as matrix computations, image processing, and relational database managements, are both fundamental and ubiquitous in computational science and its vast application areas.[1–3] Developing efficient parallel algorithms for these computations is one of the most important topics in many textbooks on parallel programming.[4,5] Then, algorithms have been designed and implemented as standard libraries. For example, for matrix computations,[6,7] we have the useful libraries like ScaLAPACK,[8] PLAPACK[9] and RECSY.[10] Though these are useful, there are some limitations when using these libraries to develop parallel programs for manipulating two-dimensional arrays.

- First, the libraries are of low abstraction, and thus difficult to be modified or adapted to solve slightly different problems. For example, we cannot easily change the operators used in matrix multiplications. In fact, the increasing popularity of parallel machines like PC clusters enables more and more users to utilize such parallel computer environments to perform parallel computations of various kinds, which can naturally be slightly different from those the libraries provide. The libraries are of no direct help for the users here, and they have to rewrite or develop the libraries for themselves. However, since (re-)building parallel libraries is much more involved than sequential algorithms due to necessity of considering the synchronization and communication between processors, not everyone can do it easily.
- Second, the libraries are not well structured, and thus hard to be efficiently composed together in the sense that we may need to convert intermediate data structures between two different libraries. Often each library is carefully designed with suitable data structures and algorithms so that it can be efficiently executed on specific parallel architectures. This may, however, prevents us from making efficient use of two libraries developed for two different parallel architectures.

This situation demands a new programming model allowing users to describe parallel computation over two-dimensional arrays in an easy, efficient, but compositional way. As one promising solution to the demand, *skeletal parallel programming* using the parallel skeleton is known.[11–13] In this model, users can build parallel programs by composing ready-made components (called *skeletons*) implemented efficiently in parallel for various parallel architectures. This compositional approach has two major advantages: (1) since low-level parallelism is concealed in skeletons,

users can obtain a comparatively efficient parallel program without needing detailed techniques of parallel computers and being conscious of parallelism explicitly, (2) since the skeletons are designed for structured programming, they can be efficiently composed to solve various problems.

Much research has been devoted to parallel skeletons on lists, a one-dimensional data structure, and it has been shown[14,15] that parallel programming with list skeletons is very powerful since we can describe many problems with these list skeletons. Moreover much research has been done on methods of deriving and optimizing parallel programs with parallel skeletons on lists,[16–18] and especially about optimization, and there is a library that can automatically optimize a program described by skeletons.[19] Similarly, for parallel skeletons on the tree data structure there is research on binary trees,[20,21] general trees and derivation of programs on these tree skeletons.

Despite the success of parallel programming with list or tree skeletons, it remains as a big challenge[22] to design a skeletal framework for developing parallel programs for manipulating two-dimensional arrays. Generally, a skeleton (compositional) framework for manipulating two-dimensional arrays should consist of the following three parts:

- *a fixed set of parallel skeletons* for manipulating two-dimensional arrays, which cannot only capture fundamental computations on two-dimensional arrays but also be efficiently implemented in parallel for various parallel architectures;
- *a systematic programming methodology*, which can support developing both efficient and correct parallel programs composed by these skeletons; and
- *an automatic optimization mechanism*, which can eliminate inefficiency due to compositional or nested uses of parallel skeletons in parallel programs.

Our idea is to use the theory of constructive algorithmics (also known as *Bird-Meertens Formalism*),[23–25] a successful theory for compositional sequential program development. In this theory, aggregate data types are formalized *constructively* as an algebra, and computations on the aggregate data are structured as *recursive* mappings between algebras while enjoying nice algebraic properties for composition with each other.

The key is to formalize two-dimensional arrays constructively so that we can describe computations on them as recursions with maximum (potential) parallelism, allowing implementation to have the maximum freedom to reorder operations for efficiency on parallel architectures. The traditional representations of two-dimensional arrays by nested one-dimensional arrays (row-majored or column-majored representations)[25,26] impose much restriction on the access order of elements. Wise et al.

represent a two-dimensional array by a quadtree[27] and show that recursive algorithms on quadtree provide better performance than existing algorithms for some matrix computations (QR factorization,[28] LU factorization[29]). More examples can be found in the survey paper.[1] However, the unique representation of two-dimensional arrays by quadtrees does not capture the whole information a two-dimensional data may have. In contrast, Bird[23] represents two-dimensional arrays by dynamic trees allowing restructuring trees when necessary.

In this paper, we propose a compositional framework that allows users, even with little knowledge about parallel machines, to describe safe and efficient parallel computation over two-dimensional arrays easily, and enables derivation and optimization of programs. The main contributions of this paper are summarized as follows.

- We propose a *novel use of the abide-tree representation of two-dimensional arrays*[23] in developing *parallel* programs for manipulating two-dimensional arrays, whose importance has not been fully recognized in parallel programming community. Our abide-tree representation of two-dimensional arrays not only inherits the advantages of tree representations of matrices where recursive blocked algorithms can be defined to achieve better performance,[1,28,29] but also supports systematic development of parallel programs and architecture independent implementation owing to its solid theoretical foundation – the theory of constructive algorithmics.[23–25]
- We provide *a strong programming support* for developing both efficient and correct parallel programs on two-dimensional arrays in a highly abstract way (without the need to be concerned with low level implementation). In our framework (Sect. 4), programmers can easily build up a complicated parallel system by defining basic components *recursively*, combining components *compositionally*, and improving efficiency *systematically*. The power of our approach can be seen from the nontrivial programming examples of matrix multiplication and QR decomposition,[27] and a successful derivation of an involved efficient parallel program for the maximum rectangle sum problem.[18,30–32]
- We demonstrate an *efficient implementation* of basic computation skeletons (in C++ and MPI) on distributed PC clusters, guaranteeing that programs composed by these parallel skeletons can be efficiently executed in parallel. So far most research focuses on showing that the recursive tree representation of matrices is suitable for parallel computation on shared memory systems,[1,28] this work shows that the recursive tree representation is also suitable for

distributed memory systems. In fact, our parallel skeletons, being of high abstraction with all potential parallelism, are architecture-independent.

Our framework can be considered as an extension of the quadtree framework of Wise et al. in the sense that it imposes no restriction on the size and the element order of two-dimensional arrays and provides an support of derivation and optimization of programs on two-dimensional arrays.

The rest of this paper is organized as follows. In Section 2, we construct a theory of the abide-trees. Then, we design a fixed set of parallel skeletons and give some examples of parallel algorithms on the abide-trees in Section 3. Section 4 demonstrates systematic development of parallel programs on two-dimensional arrays. Section 5 gives efficient implementations and shows the experimental results. We discuss the related work in Section 6, and make conclusion in Section 7.

An extended version of this paper is available in the technical report.[33]

2. THEORY OF TWO-DIMENSIONAL ARRAYS

Before explaining our compositional programming framework, we will construct a theory of two-dimensional arrays, the basis of our framework, based on the theory of constructive algorithmics.[23–25]

Notation in this paper follows that of Haskell,[34] a pure functional language that can describe both algorithms and algorithmic transformation concisely. Function application is denoted by a space and the argument may be written without brackets. Thus, $f\ a$ means $f(a)$ in ordinary notation. Functions are curried, i.e., functions take one argument and return a function or a value, and the function application associates to the left. Thus, $f\ a\ b$ means $(f\ a)\ b$. The function application binds more strongly than any other operator, so $f\ a \otimes b$ means $(f\ a) \otimes b$, but not $f\ (a \otimes b)$. Function composition is denoted by $\circ$, so $(f \circ g)\, x = f\,(g\, x)$ from its definition. Binary operators can be used as functions by sectioning as follows: $a \oplus b = (a\oplus)\, b = (\oplus b)\, a = (\oplus)\, a\, b$. For arbitrary binary operator $\otimes$, an operator in which the arguments are swapped is denoted by $\tilde{\otimes}$. Thus, $a \tilde{\otimes} b = b \otimes a$. Two binary operators $\ll$ and $\gg$ are defined by $a \ll b = a$, $a \gg b = b$. Pairs are Cartesian products of plural data, written like (x, y). A function that applies functions f and g, respectively, to the elements of a pair (x, y) is denoted by $(f \times g)$. Thus, $(f \times g)\,(x, y) = (f\, x, g\, y)$. A function that applies functions f and g separately to an element and returns a pair of the results is denoted by $(f \vartriangle g)$. Thus, $(f \vartriangle g)\, a = (f\, a, g\, a)$. A projection function π_1 extracts the first

element of a pair. Thus, $\pi_1\ (x, y) = x$. These can be extended to the case of arbitrary number of elements.

Note that we use functional style notations only for parallel algorithm development; in fact, we use the ordinary programming language C++ for practical coding.

2.1. Two-Dimensional Arrays in Abide-trees

To represent two-dimensional arrays, we define the following abide-trees, which are built up by three constructors $|\cdot|$ (singleton), $\div$ (above) and ϕ (beside).[23]

$$\begin{aligned}\textbf{data}\ \ \mathit{AbideTree}\,\alpha\ &= |\cdot|\ \alpha \\ &\mid\ (\mathit{AbideTree}\,\alpha) \div (\mathit{AbideTree}\,\alpha) \\ &\mid\ (\mathit{AbideTree}\,\alpha)\ \phi\ (\mathit{AbideTree}\,\alpha)\end{aligned}$$

Here, $|\cdot|\ a$, or abbreviated as $|a|$, means a singleton array of a, i.e. a two-dimensional array with a single element a. We define the function the to extract the element from a singleton array, i.e., $\mathsf{the}\ |a| = a$. For two-dimensional arrays x and y of the same width, $x \div y$ means that x is located above y. Similarly, for two-dimensional arrays x and y of the same height, $x\ \phi\ y$ means that x is located on the left of y. Moreover, $\div$ and ϕ are associative binary operators and satisfy the following *abide* (a coined term from above and beside) property.

Definition 2.1. (Abide Property) Two binary operators $\oplus$ and $\otimes$ are said to satisfy the abide property or to be abiding, if the following equation is satisfied:

$$(x \otimes u) \oplus (y \otimes v) = (x \oplus y) \otimes (u \oplus v).$$

In the rest of the paper, we will assume no inconsistency in height or width when ϕ and $\div$ are used.

Note that one two-dimensional array may be represented by more than one abide-trees, but these abide-trees are equivalent because of the abide property of $\div$ and ϕ. For example, we can express the 2×2 two-dimensional array

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

by the following two equivalent abide-trees.

$$(|1| \diamond |2|) \ominus (|3| \diamond |4|)$$
$$(|1| \ominus |3|) \diamond (|2| \ominus |4|)$$

This is in sharp contrast to the quadtree representation of matrices,[28] which does not allow such freedom. This freedom is important in our framework. In our framework, a program construction consists of two phases. First, we make a general program that is architecture independent. Then, we derive a good program that may be architecture dependent or may have restricted order of access. In this case, restrictive representation prevents us form describing and deriving efficient programs. For example, a list of lists representation, which restricts the access order to outer dimension to inner dimension, does not allow us to describe and to derive efficient blocked algorithms. Thus, we start with a program using the abide-tree representation that does not impose restrictions on the access order, then we transform it to a good program that may have the restricted order of accesses. Moreover, this freedom allows easy re-balancing of the tree in computations such as divide-and-conquer computations on abide-trees.

2.2. Abide-tree Homomorphism

From the theory of constructive algorithmics,[24] it follows that each constructively built-up data structure (i.e., algebraic data structure) is equipped with a powerful computation pattern called homomorphism.

Definition 2.2. ((Abide-tree) Homomorphism) A function h is said to be abide-tree homomorphism, if it is defined as follows for a function f and some binary operators $\oplus, \otimes$.

$$\begin{aligned} h\ |a| &= f\ a \\ h\ (x \ominus y) &= h\ x \oplus h\ y \\ h\ (x \diamond y) &= h\ x \otimes h\ y \end{aligned}$$

For notational convenience, we write $(\!|f, \oplus, \otimes|\!)$ to denote h. When it is clear from the context, we just call $(\!|f, \oplus, \otimes|\!)$ homomorphism.

Intuitively, a homomorphism $(\!|f, \oplus, \otimes|\!)$ is a function to replace the constructors $|\cdot|$, $\ominus$ and $\diamond$ in an input abide-tree by f, $\oplus$ and $\otimes$ respectively. We will see in Section 3 that many algorithms on two-dimensional arrays can be concisely specified by homomorphisms.

Note that $\oplus$ and $\otimes$ in $(\!|f, \oplus, \otimes|\!)$ should be associative and satisfy the abide property, inheriting the properties of $\ominus$ and $\diamond$.

Homomorphism enjoys many nice transformation rules, among which the following fusion rule is of particular importance. The fusion rule gives us a way to create a new homomorphism from composition of a function and a homomorphism. As will be seen in Section 4, it plays a key role in derivation of efficient parallel programs on abide-trees.

Theorem 2.3. (Fusion) Let h and $(\!|f, \oplus, \otimes|\!)$ be given. If there exist $\odot$ and $\ominus$ such that for any x and y,

$$\begin{cases} h\ (x \oplus y) = h\ x \odot h\ y \\ h\ (x \otimes y) = h\ x \ominus h\ y \end{cases}$$

hold, then

$$h \circ (\!|f, \oplus, \otimes|\!) = (\!|h \circ f, \odot, \ominus|\!).$$

Proof. The theorem is proved by induction on the structure of abide-trees. See the technical report[33] for details. □

Because of the flexibility of the abide-tree representation, a homomorphism $(\!|f, \oplus, \otimes|\!)$ can be implemented *efficiently* in parallel, which will be shown in Section 5. Therefore, we will design our parallel skeletons based on the homomorphisms.

3. PARALLEL SKELETONS AND ALGORITHMS ON TWO-DIMENSIONAL ARRAYS

In this section, we design a set of basic parallel skeletons for manipulating two-dimensional arrays, and show composition of the skeletons is powerful enough to describe useful parallel algorithms. Since homomorphisms are suitable for manipulating and developing parallel programs, which has been argued in the previous section, we will use the homomorphisms for the basis of the design of our parallel skeletons. The power of composition of the skeletons will be shown by describing nontrivial problems such as matrix multiplication and QR decomposition. One important fact here is that any homomorphism can be written as a composition of two basic skeletons. This fact is motivating the systematic derivation of parallel programs in Section 4.

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

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

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

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

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

$$\textbf{where}\ y_{ij} = \begin{array}{c} (x_{11} \otimes x_{12} \otimes \cdots \otimes x_{1j}) \oplus \\ (x_{21} \otimes x_{22} \otimes \cdots \otimes x_{2j}) \oplus \\ \vdots \\ (x_{i1} \otimes x_{i2} \otimes \cdots \otimes x_{ij}) \end{array}$$

Fig. 1. Intuitive definition of four primitive skeletons on two-dimensional arrays.

3.1. Data Parallel Skeletons

We define four primitive functions map, reduce, zipwith and scan on the data type *AbideTree*. In the theory of Constructive Algorithmics,[23–25] these functions are known to be the most fundamental computation components for manipulating algebraic data structures and for being glued together to express complicated computations. We call them *parallel skeletons* because they have potential parallelism and can be implemented efficiently in parallel (see Section 5.) Intuitive definitions of the skeletons are shown in Fig. 1.

3.1.1. Map and Reduce

The skeletons map and reduce are two special cases of homomorphism. The skeleton map applies a function f to each element of a two-dimensional array while keeping the structure, and is defined by

$$\begin{aligned}
\mathsf{map}\ f\ |a| &= |f\ a| \\
\mathsf{map}\ f\ (x \mathbin{\ominus} y) &= (\mathsf{map}\ f\ x) \mathbin{\ominus} (\mathsf{map}\ f\ y) \\
\mathsf{map}\ f\ (x \mathbin{\oplus} y) &= (\mathsf{map}\ f\ x) \mathbin{\oplus} (\mathsf{map}\ f\ y),
\end{aligned}$$

that is, $\mathsf{map}\ f = (\!|\ |\cdot| \circ f, \ominus, \oplus\ |\!)$. For example, a scalar product of an array is performed with map as follows.

$$\mathsf{map}\ (2\times) \begin{pmatrix} 6 & 2 & 1 \\ 4 & 3 & 5 \end{pmatrix} = \begin{pmatrix} 12 & 4 & 2 \\ 8 & 6 & 10 \end{pmatrix}$$

The skeleton reduce collapses a two-dimensional array to a value using two abiding binary operators $\oplus$, $\otimes$, and is defined by

$$\begin{aligned}
\mathsf{reduce}(\oplus, \otimes)\ |a| &= a \\
\mathsf{reduce}(\oplus, \otimes)\ (x \mathbin{\ominus} y) &= (\mathsf{reduce}(\oplus, \otimes)\ x) \oplus (\mathsf{reduce}(\oplus, \otimes)\ y) \\
\mathsf{reduce}(\oplus, \otimes)\ (x \mathbin{\oplus} y) &= (\mathsf{reduce}(\oplus, \otimes)\ x) \otimes (\mathsf{reduce}(\oplus, \otimes)\ y),
\end{aligned}$$

that is, $\mathsf{reduce}(\oplus, \otimes) = (\!|\ id, \oplus, \otimes\ |\!)$. For example, a summation of an array is calculated with reduce as follows.

$$\mathsf{reduce}(+, +) \begin{pmatrix} 6 & 2 & 1 \\ 4 & 3 & 5 \end{pmatrix} = 21$$

Interestingly, any homomorphism can be written as a composition of map and reduce.

Lemma 3.1. (Homomorphism) A homomorphism $(\!|\ f, \oplus, \otimes\ |\!)$ can be written as a composition of map and reduce:

$$(\!|\ f, \oplus, \otimes\ |\!) = \mathsf{reduce}(\oplus, \otimes) \circ \mathsf{map}\ \ f.$$

Proof. The lemma is proved by induction on the structure of abide-trees. □

This lemma implies that if we have efficient parallel implementations for reduce and map, we get an efficient implementation for homomorphism. Since it is shown in the Sect. 5 that parallel skeletons have efficient parallel implementations, a homomorphism has an efficient parallel

implementations. This fact is motivating the systematic derivation of parallel programs in Section 4.

3.1.2. Zipwith

The two skeletons defined above are primitive skeletons. We define other skeletons that are extensions of these primitive skeletons. The skeleton zipwith, an extension of map, takes two two-dimensional arrays of the same shape, applies a function f to corresponding elements of the arrays and returns a new array of the same shape.

$$\begin{aligned}
&\mathsf{zipwith}\ f\ |a|\ |b| &&= |f\ a\ b| \\
&\mathsf{zipwith}\ f\ (x \oplus y)\ (u \oplus v) &&= (\mathsf{zipwith}\ f\ x\ u) \oplus (\mathsf{zipwith}\ f\ y\ v) \\
&\mathsf{zipwith}\ f\ (x \otimes y)\ (u \otimes v) &&= (\mathsf{zipwith}\ f\ x\ u) \otimes (\mathsf{zipwith}\ f\ y\ v)
\end{aligned}$$

Note that in the above definition two-dimensional arrays that are the arguments of the function should be divided in the way that the sizes of x and u are the same. For example, an addition of two arrays is calculated with reduce as follows.

$$\mathsf{zipwith}\ (+) \begin{pmatrix} 6 & 2 & 1 \\ 4 & 3 & 5 \end{pmatrix} \begin{pmatrix} 0 & 8 & 2 \\ 9 & 1 & 7 \end{pmatrix} = \begin{pmatrix} 6 & 10 & 3 \\ 13 & 4 & 12 \end{pmatrix}$$

Function zip is a specialization of zipwith, making a two-dimensional array of pairs of corresponding elements.

$$\mathsf{zip}\ (u, v) = \mathsf{zipwith}\ (\lambda\, xy.\ (x, y))\ u\ v$$

We may define similar zip and zipwith for the case when the number of input arrays is three or more, and those that take k arrays are denoted by zip_k and $\mathsf{zipwith}_k$. Also, we define unzip to be the inverse of zip.

Composing these skeletons defined above, we can describe many useful functions.

$$
\begin{array}{ll}
id & = \mathsf{reduce}(\oplus, \phi) \circ \mathsf{map}\,|\cdot| \\
tr & = \mathsf{reduce}(\phi, \oplus) \circ \mathsf{map}\,|\cdot| \\
rev & = \mathsf{reduce}(\tilde{\oplus}, \tilde{\phi}) \circ \mathsf{map}\,|\cdot| \\
flatten & = \mathsf{reduce}(\oplus, \phi) \\
height & = \mathsf{reduce}(+, \ll) \circ \mathsf{map}\,(\lambda\, x.\, 1) \\
width & = \mathsf{reduce}(\ll, +) \circ \mathsf{map}\,(\lambda\, x.\, 1) \\
cols & = \mathsf{reduce}(\mathsf{zipwith}(\oplus), \phi) \circ \mathsf{map}\,||\cdot|| \\
rows & = \mathsf{reduce}(\oplus, \mathsf{zipwith}(\phi)) \circ \mathsf{map}\,||\cdot|| \\
\mathsf{reduce}_c(\oplus) & = \mathsf{map}(\mathsf{reduce}(\oplus, \ll)) \circ cols \\
\mathsf{reduce}_r(\otimes) & = \mathsf{map}(\mathsf{reduce}(\ll, \otimes)) \circ rows \\
\mathsf{map}_c\ f & = \mathsf{reduce}(\ll, \phi) \circ \mathsf{map}\ f \circ cols \\
\mathsf{map}_r\ f & = \mathsf{reduce}(\oplus, \ll) \circ \mathsf{map}\ f \circ rows \\
add & = \mathsf{zipwith}(+) \\
sub & = \mathsf{zipwith}(-)
\end{array}
$$

Here, $||\cdot||$ is abbreviation of $|\cdot| \circ |\cdot|$. The function id is the identity function of *AbideTree*, and tr is the matrix-transposing function. The function rev takes a two-dimensional array and returns the array reversed in the vertical and the horizontal direction, and $flatten$ flattens a nested *AbideTree*. The functions $height$ and $width$ return the number of rows and columns, respectively, and $cols$ and $rows$ return an array of which elements are columns and rows of the array of the argument, respectively. The functions reduce_c and reduce_r that are specializations of reduce reduce a two-dimensional array in each column and row direction, respectively, and return a row-vector (an array of which height is one) and a column-vector (an array of which width is one). The functions map_c and map_r that are specializations of map apply a function to each column and row, respectively (i.e., the function of the argument takes column-vector or row-vector). The functions add and sub denote matrix addition and subtraction, respectively.

3.1.3. Scan

The skeleton scan, an extension of reduce, holds all values generated in reducing a two-dimensional array by reduce.

$$
\begin{aligned}
\mathsf{scan}(\oplus, \otimes)\,|a| &= |a| \\
\mathsf{scan}(\oplus, \otimes)(x \mathbin{\ominus} y) &= (\mathsf{scan}(\oplus, \otimes)\,x) \oplus' (\mathsf{scan}(\oplus, \otimes)\,y) \\
\mathsf{scan}(\oplus, \otimes)(x \mathbin{\phi} y) &= (\mathsf{scan}(\oplus, \otimes)\,x) \otimes' (\mathsf{scan}(\oplus, \otimes)\,y)
\end{aligned}
$$

Here operators $\oplus'$ and $\otimes'$ are defined as follows.

$$
\begin{aligned}
sx \oplus' sy &= sx \mathbin{\ominus} sy' \\
&\quad \textbf{where}\ \ sy' = \mathsf{map}_r\,(\mathsf{zipwith}(\oplus)(bottom\,sx))\,sy \\
sx \otimes' sy &= sx \mathbin{\phi} sy' \\
&\quad \textbf{where}\ \ sy' = \mathsf{map}_c\,(\mathsf{zipwith}(\otimes)(last\,sx))\,sy \\
bottom &= \mathsf{reduce}(\gg, \phi) \circ \mathsf{map}\,|\cdot| \\
last &= \mathsf{reduce}(\ominus, \gg) \circ \mathsf{map}\,|\cdot|
\end{aligned}
$$

For example, a prefix-um (an upper-left prefix-sum) of an array is calculated with scan as follows.

$$
\mathsf{scan}(+, +) \begin{pmatrix} 6 & 2 & 1 \\ 4 & 3 & 5 \end{pmatrix} = \begin{pmatrix} 6 & 8 & 9 \\ 10 & 15 & 21 \end{pmatrix}
$$

It should be noted that reduce can be expressed by reduce_c and reduce_r when two binary operators $\oplus$ and $\otimes$ are abiding.

$$
\begin{aligned}
\mathsf{reduce}(\oplus, \otimes) &= \mathsf{the} \circ \mathsf{reduce}_c(\oplus) \circ \mathsf{reduce}_r(\otimes) \\
\mathsf{reduce}(\oplus, \otimes) &= \mathsf{the} \circ \mathsf{reduce}_r(\otimes) \circ \mathsf{reduce}_c(\oplus)
\end{aligned} \tag{1}
$$

Like reduce, we may define $\mathsf{scan}_\downarrow$ and $\mathsf{scan}_\rightarrow$ that are specialization of scan and scan a two-dimensional array in column and row direction, respectively:

$$
\begin{aligned}
\mathsf{scan}_\downarrow(\oplus) &= \mathsf{scan}(\oplus, \gg) \\
\mathsf{scan}_\rightarrow(\otimes) &= \mathsf{scan}(\gg, \otimes);
\end{aligned}
$$

scan can be expressed by $\mathsf{scan}_\downarrow$ and $\mathsf{scan}_\rightarrow$ when two binary operators $\oplus$ and $\otimes$ are abiding.

$$
\begin{aligned}
\mathsf{scan}(\oplus, \otimes) &= \mathsf{scan}_\downarrow(\oplus) \circ \mathsf{scan}_\rightarrow(\otimes) \\
\mathsf{scan}(\oplus, \otimes) &= \mathsf{scan}_\rightarrow(\otimes) \circ \mathsf{scan}_\downarrow(\oplus)
\end{aligned} \tag{2}
$$

Using the skeleton scan, we can define scanr that executes scan reversely (i.e., from bottom to top, from right to left), $allred_r$ and $allred_c$ that broadcast the results in each row and column after reduce_r and reduce_c, respectively. These functions are used in later section.

$$
\begin{aligned}
\mathsf{scanr}(\oplus, \otimes) &= rev \circ \mathsf{scan}(\tilde{\oplus}, \tilde{\otimes}) \circ rev \\
allred_c(\oplus) &= \mathsf{scanr}(\gg, \ll) \circ \mathsf{scan}(\oplus, \gg) \\
allred_r(\otimes) &= \mathsf{scanr}(\ll, \gg) \circ \mathsf{scan}(\gg, \otimes)
\end{aligned}
$$

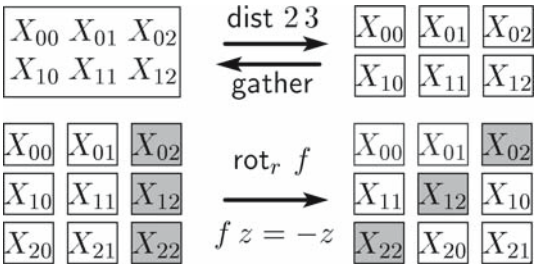

Fig. 2. An image of communication skeletons (each rectangle corresponds to each processor; X_{ij} represents a subarray.).

3.2. Data Communication Skeletons

We show how to define data communication skeletons dist, gather, rot_r and rot_c that abstract distribution, collection and rearrangement of a two-dimensional array among processors. The idea is to use nested two-dimensional arrays to represent distributed two-dimensional arrays.

The skeleton dist abstracts distribution of a two-dimensional array to processors, and is defined as

$$\begin{array}{l}\mathsf{dist}\ p\ q\ x = (\mathit{flatten} \circ \mathsf{map}(\mathit{grp}_c\, n) \circ \mathit{grp}_r\, m)\ x \\ \textbf{where}\ m = \lceil \mathit{height}\ x/p \rceil,\ \ n = \lceil \mathit{width}\ x/q \rceil\end{array}$$

where grp_r is defined as follows and grp_c is defined similarly.

$$\begin{array}{lll}\mathit{grp}_r\ k\ (x \oplus y) = |x| \oplus (\mathit{grp}_r\ k\ y) & \text{if} & \mathit{height}\ x = k \\ \mathit{grp}_r\ k\ x \qquad\quad = |x| & \text{if} & \mathit{height}\ x < k\end{array}$$

The distribution dist $p\ q\ x$ means that the two-dimensional array x will be divided into $p \times q$ subarrays (i.e., x is divided into p subarrays in vertical direction, then each subarray is divided into q subarrays in horizontal direction), and each subarray is distributed to a processor.

The skeleton gather, the inverse operator of dist, abstracts gathering of two-dimensional arrays distributed to the processors into a two-dimensional array on the root processor.

$$\mathsf{gather} = \mathsf{reduce}(\oplus, \phi)$$

Although definitions of these skeletons may seem complicated, actual operations are simple as illustrated in Fig. 2. What is significant here is that these skeletons satisfy the relation of $id = \mathsf{gather} \circ \mathsf{dist}\ p\ q$.

The rotation skeleton rot_r that takes a function f and rotates the i-th row (the index begins from 0) by $f\ i$, is defined as follows:

$$\begin{aligned}&\mathsf{rot}_r\ f = flatten \circ \mathsf{map}\ shift_r \circ addidx_r \circ rows\\ &\quad \textbf{where}\\ &\qquad addidx_r\ u = \mathsf{zip}(\mathsf{map}\ f\ (idx_r\ u), u)\\ &\qquad idx_r = \mathsf{map}(-1) \circ \mathsf{scan}_{\downarrow}(+) \circ \mathsf{map}(\lambda x.\,1)\,;\end{aligned}$$

here $shift_r$ is defined under the condition $i > 0$.

$$\begin{aligned}shift_r\ (0, x) &= x\\ shift_r\ (i, x \Phi y) &= y \Phi x \quad \text{if } width\ y = i\\ shift_r\ (-i, x \Phi y) &= y \Phi x \quad \text{if } width\ x = i\end{aligned}$$

Similarly, we can define the skeleton rot_c that takes a function f and rotates the i-th column by $f\ i$. An image of the above communication skeletons is depicted in Fig. 2. In the figure, since the rotation skeleton rot_r takes a negation function, the 0th row does not rotate (rotates by 0), the first row rotates to the left by 1 (to the right by -1) and the second row rotates to the left by 2 (to the right by -2).

3.3. Matrix Multiplication

As an involved example, we describe two known parallel algorithms for matrix multiplication, which is a primitive operation of matrices, with the above defined parallel skeletons on two-dimensional arrays.

The first description is Cannon's Algorithm:[4]

$$\begin{aligned}&mm_C = \mathsf{gather} \circ (\mathsf{map}\ thd) \circ (iter\ p\ step) \circ arrange \circ distribute \circ init\\ &\quad \textbf{where}\\ &\qquad init\ (A, B) = (A, B, \mathsf{map}(\lambda x.\,0)\ A)\\ &\qquad distribute = (\mathsf{dist}\ p\ p \times \mathsf{dist}\ p\ p \times \mathsf{dist}\ p\ p)\\ &\qquad arrange = \mathsf{zip}_3 \circ (\mathsf{rot}_r\ neg \times \mathsf{rot}_c\ neg \times id)\\ &\qquad step = rearrange \circ \mathsf{unzip}_3 \circ \mathsf{map}\ lmm\\ &\qquad rearrange = \mathsf{zip}_3 \circ (\mathsf{rot}_r(\lambda x.1) \times \mathsf{rot}_c(\lambda x.1) \times id)\\ &\qquad neg\ x = -x\\ &\qquad thd\ (x, y, z) = z\end{aligned}$$

where p is a natural number indicating the number of divisions of matrices in column and row direction, and lmm is a function that executes locally

matrix multiplication on matrices on each processor, i.e., $lmm\,(A, B, C) = (A, B, C + A \times B)$. The function $iter$ is defined as follows.

$$\begin{array}{ll} iter\,k\,f\,x = x & \text{if} \quad k = 0 \\ iter\,k\,f\,x = iter\,(k-1)\,f\,(f\,x) & \text{if} \quad k > 0 \end{array}$$

Explicit distribution of matrices by the data communication skeletons makes this description looking complicated. However, it should be noted that even non-intuitive complicated Cannon's Algorithm can be described by composition of the skeletons.

The second description is an intuitively understandable description using only data parallel skeletons (an element of the resulting matrix is an inner product of a row vector and a column vector of the input matrices.) This description describes just a definition of matrix multiplication. Although users do not need to consider parallelism at all, this program can be executed in parallel due to parallelism of each skeleton.

$$\begin{array}{l} mm = \mathsf{zipwith}_P\ iprod \circ (allrows \times allcols) \\ \quad \textbf{where} \\ \quad\quad allrows = allred_r(\Phi) \circ \mathsf{map}\,|\cdot| \\ \quad\quad allcols = allred_c(\ominus) \circ \mathsf{map}\,|\cdot| \\ \quad\quad iprod = (\mathsf{reduce}(+,+)\circ) \circ \mathsf{zipwith}(\times) \circ tr \\ \quad\quad \mathsf{zipwith}_P\,(\otimes)\,(x, y) = \mathsf{zipwith}\,(\otimes)\,x\,y \end{array}$$

Although this definition seems to use $O(n^3)$ memory space for $n \times n$ matrices due to duplications of rows and columns with $allred_r$ and $allred_c$, we can execute this multiplication using $O(n^2)$ memory space. This is because we can use references instead of duplications in the implementation of $allred_r$ and $allred_c$ due to the properties of the operators in their definitions. This kind of optimization is currently done by hand. However, we think it will be automatically done by compilers when it takes the properties of the operators into account.

3.4. QR Factorization

As the final nontrivial example, we show descriptions of two parallel algorithms for QR factorization.(1) We will not explain the details, but we hope to show that these algorithms can be dealt with in our framework.

We give the recursive description of a QR factorization algorithm based on Householder transform. This function returns Q and R which satisfy $A = QR$ where A is a matrix of $m \times n$, Q an orthogonal matrix of $m \times m$ and R an upper-triangular matrix of $m \times n$.

$$
\begin{aligned}
&qr\ ((A_{11} \ominus A_{21}) \oplus (A_{12} \ominus A_{22})) \\
&\qquad = \textbf{let}\ (Q_1, R_{11} \ominus 0) = qr\ (A_{11} \ominus A_{21}) \\
&\qquad\qquad (R_{12} \ominus \hat{A_{22}}) = mm\ (tr\, Q_1)\ (A_{12} \ominus A_{22}) \\
&\qquad\qquad (\hat{Q_2}, R_{22}) = qr\ \hat{A_{22}} \\
&\qquad\qquad Q = mm\ Q_1\ ((I \oplus 0) \ominus (0 \oplus \hat{Q_2})) \\
&\qquad\quad \textbf{in}\ (Q, (R_{11} \oplus R_{12}) \ominus (0 \oplus R_{22})) \\
&qr\ (|a| \ominus x) = hh\ (|a| \ominus x) \\
&hh\ v \qquad = \textbf{let}\ v' = add\ v\ e \\
&\qquad\qquad a = \sqrt{\mathsf{reduce}(+,+)\ (\mathsf{zipwith}(\times)\ v'\ v')} \\
&\qquad\qquad u = \mathsf{map}\ (/a)\ v' \\
&\qquad\qquad Q = sub\ I\ (\mathsf{map}\ (\times 2)\ (mm\ u\ (tr\ u))) \\
&\qquad\quad \textbf{in}\ (Q, e)
\end{aligned}
$$

Here e is a vector (a matrix of which width is 1) whose first element is 1 and the other elements are 0, and I and 0 represent an identity matrix and a zero matrix of suitable size, respectively.

Furthermore, we give the recursive description of QR factorization algorithm on quadtree;[28] transforming algorithms on quadtrees to those on abide-trees is always possible because abide-trees is more flexible than quadtrees. This function qr_q is mutual recursively defined with an extra function e, and returns Q and R that satisfy $A = QR$ where A is a matrix of $n \times n$ ($n = 2^k$ for a natural number k), Q is an orthogonal matrix of $n \times n$ and R is an upper-triangular matrix of $n \times n$.

$$
\begin{aligned}
&qr_q\ |a| = (|1|, |a|) \\
&qr_q\ ((A_{11} \ominus A_{21}) \oplus (A_{12} \ominus A_{22})) \\
&\qquad = \textbf{let}\ (Q_1, R_1) = qr_q\ A_{11} \\
&\qquad\qquad (Q_2, R_2) = qr_q\ A_{21} \\
&\qquad\qquad Q_{12} = (Q_1 \oplus 0) \ominus (0 \oplus Q_2) \\
&\qquad\qquad (Q_3, R_3) = e\ (R_1, R_2) \\
&\qquad\qquad Q_4 = mm\ Q_{12}\ Q_3 \\
&\qquad\qquad (U_n \ominus U_s) = mm\ (tr\ Q_4)\ (A_{12} \ominus A_{22}) \\
&\qquad\qquad (Q_6, R_6) = qr_q\ U_s \\
&\qquad\qquad Q = mm\ Q_4\ ((I \oplus 0) \ominus (0 \oplus Q_6)) \\
&\qquad\qquad R = (R_3 \oplus U_n) \ominus (0 \oplus R_6) \\
&\qquad\quad \textbf{in}\ (Q, R)
\end{aligned}
$$

Note that A_{ij} $(i, j \in \{1, 2\})$ have the same shape. A definition of the involved extra function e is as follows.

$$
\begin{aligned}
e\ (N, O) &= (I, N) \\
e\ (|n|, |s|) &= \textbf{let}\ \ Q = g(n, s) \\
&\qquad\qquad (N, O) = mm(tr\ \ Q)\ \ (|n| \ominus |s|) \\
&\qquad \textbf{in}\ \ (Q, N) \\
e\ ((N_{11} \ominus N_{21}) \oplus (N_{12} \ominus N_{22}), (S_{11} \ominus S_{21}) \oplus (S_{12} \ominus S_{22})) \\
&\quad = \textbf{let} \\
&\qquad\qquad ((Q_1^{11} \ominus Q_1^{21}) \oplus (Q_1^{12} \ominus Q_1^{22}), N_1) = e\ \ (N_{11}, S_{11}) \\
&\qquad\qquad ((Q_2^{11} \ominus Q_2^{21}) \oplus (Q_2^{12} \ominus Q_2^{22}), N_2) = e\ \ (N_{22}, S_{22}) \\
&\qquad\qquad Q_{12} = (Q_1^{11} \oplus O \oplus Q_1^{12} \oplus O) \ominus (O \ominus Q_2^{11} \oplus O \oplus Q_2^{12}) \\
&\qquad\qquad\qquad \ominus (Q_1^{21} \oplus O \oplus Q_1^{22} \oplus O) \ominus (O \ominus Q_2^{21} \oplus O \oplus Q_2^{22}) \\
&\qquad\qquad Q_1 = (Q_1^{11} \ominus Q_1^{21}) \oplus (Q_1^{12} \ominus Q_1^{22}) \\
&\qquad\qquad (U_n \ominus U_s) = mm\ \ (tr\ \ Q_1)\ \ (N_{12} \ominus S_{12}) \\
&\qquad\qquad (Q_4, R_4) = qr_q\ \ U_s \\
&\qquad\qquad Q_4' = (I \oplus O \oplus O \oplus O) \ominus (O \oplus I \oplus O \oplus O) \\
&\qquad\qquad\qquad \ominus (O \oplus O \oplus Q_4 \oplus O) \ominus (O \oplus O \oplus O \oplus I) \\
&\qquad\qquad Q_5 = mm\ \ Q_{12}\ \ Q4' \\
&\qquad\qquad ((Q_6^{11} \ominus Q_6^{21}) \oplus (Q_6^{12} \ominus Q_6^{22}), N_6) = e\ \ (N_2, R_4) \\
&\qquad\qquad Q_6' = (I \oplus O \oplus O \oplus O) \ominus (O \oplus Q_6^{11} \oplus Q_6^{12} \oplus O) \\
&\qquad\qquad\qquad \ominus (O \oplus Q_6^{21} \oplus Q_6^{22} \oplus O) \ominus (O \oplus O \oplus O \oplus I) \\
&\qquad \textbf{in}\ \ (mm\ \ Q_5\ \ Q_6', (N_1 \oplus U_n) \ominus (O \oplus N_6)) \\
g\ \ (a, b) &= (|c| \oplus |s|) \ominus (|-s| \oplus |c|) \\
&\qquad\qquad \textbf{where}\ \ c = \frac{a}{\sqrt{a^2 + b^2}},\ \ s = \frac{-b}{\sqrt{a^2 + b^2}}
\end{aligned}
$$

Note that N_{ij} and S_{ij} $(i, j \in \{1, 2\})$ have the same shape and Q_k^{ij} $(i, j, k \in \{1, 2\})$ have the same shape.

We can efficiently parallelize some parts of these complicated recursive functions, such as matrix multiplication and independent calculations like $(Q_1, R_1) = qr_q\ \ A_{11}$ and $(Q_2, R_2) = qr_q\ \ A_{21}$. These independent calculations are explicitly parallelized by describing them with the map skeleton as follows.

$$\left\{ \begin{array}{l} (Q_1, R_1) = qr_q\ A_{11} \\ (Q_2, R_2) = qr_q\ A_{21} \end{array} \right\}$$

$$\Downarrow$$

$$|(Q_1, R_1)| \ominus |(Q_2, R_2)| = \mathsf{map}\ \ apply\ (|(qr_q, A_{11})| \ominus |(qr_q, A_{21})|)$$

Here, the function *apply* is defined as $apply\ (f, x) = f\ x$. It is, however, still an open problem whether the complicated recursive functions can be parallelized with our defined skeletons, although we can introduce other skeletons such as divide-and-conquer skeleton to parallelize them. This is our future work.

4. Developing Efficient Parallel Programs

It has been shown so far that compositions of recursive functions on abide-trees including homomorphisms and skeletons provide us with a powerful mechanism to describe parallel algorithms on two-dimensional arrays, where parallelism in the original parallel algorithms can be well captured. In this section, we move on from issues of parallelism to the issues of efficiency. We will illustrate a strategy to guide programmers to develop *efficient* parallel algorithms systematically through program transformation.

Recall homomorphisms have efficient parallel implementation as composition of our parallel skeletons. Thus, a goal of this derivation may be to write a program by a homomorphism. However, not all functions can be specified by a single homomorphism. Therefore, we first introduce a more powerful tool called almost-homomorphism.[35] Then, we demonstrate the strategy with an example.

4.1. Almost-Homomorphism

Not all functions can be specified by a single homomorphism, but we can always tuple these functions with some extra functions so that the tupled functions can be specified by a homomorphism. An *almost homomorphism* is a composition of a projection function and a homomorphism. Since projection functions are simple, almost homomorphisms are suitable for parallel computation as homomorphisms are.

In fact, every function can be represented in terms of an almost homomorphism at the cost of redundant computation. Let k be a non-homomorphic function, and consider a new function g such that $gx = (k\ x, x)$. The tupled function g is a homomorphism.

$$
\begin{aligned}
g\ |a| \quad &= (k\ |a|, |a|) \\
g\ (x \ominus y) &= g\ x \oplus g\ y \\
&\quad \textbf{where}\ (k_1, x_1) \oplus (k_2, x_2) = (k\ (x_1 \ominus x_2), x_1 \ominus x_2) \\
g\ (x \oplus y) &= g\ x \otimes g\ y \\
&\quad \textbf{where}\ (k_1, x_1) \otimes (k_2, x_2) = (k\ (x_1 \oplus x_2), x_1 \oplus x_2)
\end{aligned}
$$

Then, k is written as an almost homomorphism:

$$k = \pi_1 \circ g = \pi_1 \circ (\!| g \circ |\cdot|, \oplus, \otimes |\!).$$

However, the definition above is not efficient because binary operators $\oplus$ and $\otimes$ do not use the previously computed values k_1 and k_2. In order to derive a good almost homomorphism, we should carefully define a suitable tupled function, making full use of previously computed values. We will see this in our parallel program development in the next section. However, to determine the class of problems that have good (efficient) almost-homomorphic implementation is an open problem.

4.2. A Strategy for Deriving Efficient Parallel Programs

Our strategy for deriving efficient parallel programs on two-dimensional arrays consists of the following four steps, extending the result of lists.[18] The goal of the strategy is to write a given program by an efficient almost-homomorphism that has an efficient implementation in parallel.

Step 1. Define the target program p as a composition of $p_1, \ldots, p_n$ that are already defined, i.e., $p = p_n \circ \ldots \circ p_1$. Each of $p_1, \ldots, p_n$ may be defined as a composition of small functions or a recursive function (see Sections3.3 and 3.4).

Step 2. Derive an almost homomorphism (Sect. 4.1) from the recursive definition of p_1.

Step 3. Fuse p_2 into the derived almost homomorphism to obtain a new almost homomorphism for $p_2 \circ p_1$, and repeat this derivation until p_n is fused.

Step 4. Let $\pi_1 \circ (\!| f, \oplus, \otimes |\!)$ be the resulting almost homomorphism for $p_n \circ \ldots \circ p_1$ obtained at Step 3. For the functions inside the homomorphism, namely f, $\oplus$ and $\otimes$, try to repeat Steps 2 and 3 to find efficient parallel implementations for them.

In the following, we explain this strategy through a derivation of an efficient program for the maximum rectangle sum problem: compute the

maximum of sums of all the rectangle data areas in a two-dimensional data. This problem was originated by Bentley[30,31] and improved by Takaoka.[32] The solution can be used in a sort of data mining and pattern matching of two dimensional data. For example, for the following two-dimensional data

$$\begin{pmatrix} 3 & -1 & \mathbf{4} & \mathbf{-1} & -5 \\ 1 & -4 & \mathbf{-1} & \mathbf{5} & -3 \\ -4 & 1 & \mathbf{5} & \mathbf{3} & 1 \end{pmatrix}$$

the result should be 15, which denotes the maximum sum contributed by the sub-rectangular area with bolded numbers above. To appreciate difficulty of this problem, we ask the reader to pause for a while to think of how to solve it.

We will omit some portions of the description in the following derivation for the readability. Please see the technical report[33] for the full description.

4.2.1. Step 1. Defining a Clear Parallel Program

A clear and straightforward solution to the maximum rectangle sum problem is as follows: enumerating all possible rectangles, then computing sums for all rectangles, and finally returning the maximum value as the result.

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

Here *rects* is a function that takes a two-dimensional array and returns all possible rectangles of the array. The returned value of *rects* is an array of arrays of arrays, and the (k, l)-element of the (i, j)-element of the resulting array is a sub-rectangle having rows from the ith to the jth and columns from the kth to the lth of the original array. An example of *rects* is shown below. Note that we think that the special value is contained in the blank portion of the above-mentioned array, and we write the blank of arbitrary size by *NIL* for brevity. Here, *NIL* may be an array of which element is $-\infty$ or an array of it.

$$rects\ \begin{pmatrix}1\,2\,3\\5\,6\,7\end{pmatrix} = \begin{pmatrix} \begin{pmatrix}(1)&(1\,2)&(1\,2\,3)\\&(2)&(2\,3)\\&&(3)\end{pmatrix} & \begin{pmatrix}\binom{1}{5}&\binom{1\,2}{5\,6}&\binom{1\,2\,3}{5\,6\,7}\\&\binom{2}{6}&\binom{2\,3}{6\,7}\\&&\binom{3}{7}\end{pmatrix} \\ & \begin{pmatrix}(5)&(5\,6)&(5\,6\,7)\\&(6)&(6\,7)\\&&(7)\end{pmatrix} \end{pmatrix}$$

The function $rects$ is mutual recursively defined as follows.

$$\begin{aligned} rects\,|a| &= |||a||| \\ rects\,(x \oplus y) &= (rects\,x \mathbin{\phi} gemm(\_, \mathsf{zipwith}(\oplus))\ (bottoms\,x)\,(tops\,y)) \\ &\qquad \oplus (NIL \mathbin{\phi} rects\,y) \\ rects\,(x \mathbin{\phi} y) &= \mathsf{zipwith}_4\ f_s\,(rects\,x)\,(rects\,y)\,(rights\,x)\,(lefts\,y) \\ &\quad \textbf{where}\ f_s\,s_1\,s_2\,r_1\,l_2 = (s_1 \mathbin{\phi} gemm(\_, \phi)\,r_1\,l_2) \oplus (NIL \mathbin{\phi} s_2) \end{aligned}$$

where '$\_$' indicates "don't care" and generalized matrix multiplication $gemm$ is defined as follows.

$$\begin{aligned} &gemm(\oplus, \otimes) = g \\ &\textbf{where} \\ &\quad g\,(X_1 \mathbin{\phi} X_2)\,(Y_1 \oplus Y_2) = \mathsf{zipwith}(\oplus)\,(g\,X_1\,Y_1)\,(g\,X_2\,Y_2) \\ &\quad g\,(X_1 \oplus X_2)\,Y = (g\,X_1\,Y) \oplus (g\,X_2\,Y) \\ &\quad g\,X\,(Y_1 \mathbin{\phi} Y_2) = (g\,X\,Y_1) \mathbin{\phi} (g\,X\,Y_2) \\ &\quad g\,|a|\,|b| = |a \otimes b| \end{aligned}$$

Figure 3 shows recursive computation of $rects$ for the $\oplus$ case. Since indices i and j of the resulting array mean the top row and the bottom row of the generated rectangles within the argument array, the upper-left block ($i \leq m_x$ and $j \leq m_x$) of the resulting array contains rectangles included in the upper block of the argument, i.e., x. Thus, the upper-left block is $rects\ x$, and the lower-right block is $rects\ y$ similarly (Fig. 3(a)). Each rectangle in the upper-right block ($i \leq m_x$ and $j > m_x$) consists of a rectangle at the bottom of x and a rectangle at the top of y as shown in Fig. 3(b). Since there are all combinations of i and j, this block is computed by a general matrix multiplication of $bottoms\ x$ and $tops\ y$ as shown in Fig. 3(c). The computation of the $\oplus$ case is similar.

Functions $bottoms$, $tops$, $rights$ and $lefts$ are similarly defined as mutual recursive functions shown in Fig. 4, and their examples are shown in Fig. 5. Each of these functions is the partial result of $rects$ in the sense that

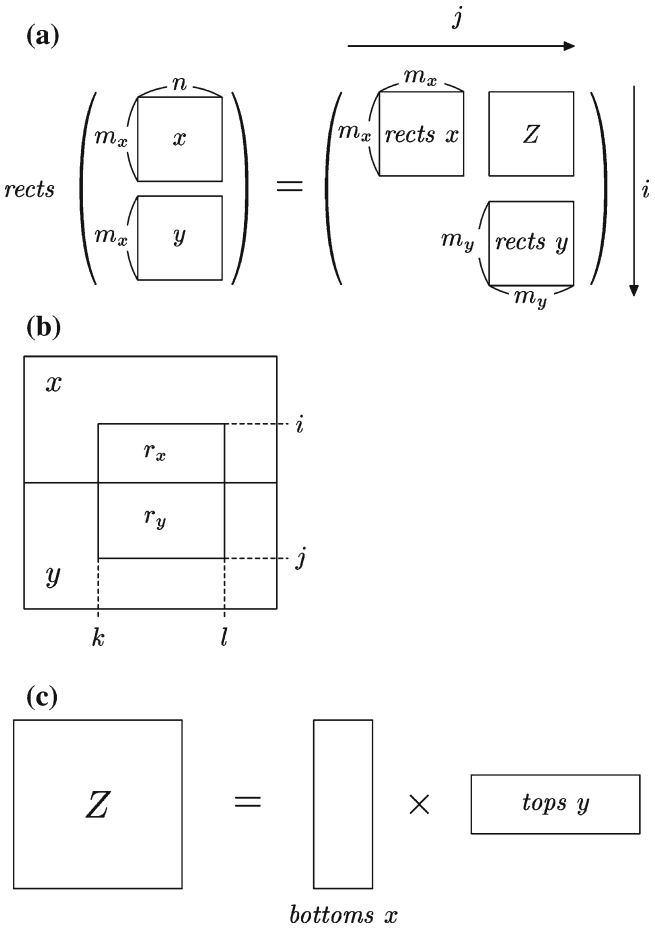

Fig. 3. Recursive computation of *rects*.

it returns the rectangles that are restricted to include segments of some edges of the input as shown in Fig. 6. For example, the *tops* returns the rectangles that include segments of the top edge (the top-most row) of the input array. Similarly, *bottoms*, *rights* and *lefts* return the rectangles that include segments of the bottom edge, the right edge and the left edge, respectively. The other functions returns the rectangles that include segments of two edges of the input. Usually, users do not need to understand well these functions because these functions often used are provided by experts.

Although this initial program is clear and has all its parallelism specified in terms of our parallel skeletons, it is inefficient in the sense that it

$$
\begin{array}{l}
tops\ |a| \quad = \ |||a||| \\
tops\ (x \circ y) \ = \ tops\ x \diamond \mathsf{map}\ (\mathsf{zipwith}(\circ)\ (cols'\ x))\ (tops\ y) \\
tops\ (x \diamond y) \ = \ \mathsf{zipwith}_4\ f_t\ (tops\ x)\ (tops\ y)\ (toprights\ x)\ (toplefts\ y) \\
\qquad \textbf{where}\ f_t\ t_1\ t_2\ tr_1\ tl_2 = (t_1 \diamond gemm\ (\_, \diamond)\ tr_1\ tl_2) \circ (NIL \diamond t_2)
\end{array}
$$

$$
\begin{array}{l}
bottoms\ |a| \quad = \ |||a||| \\
bottoms\ (x \circ y) \ = \ \mathsf{map}\ (\lambda z \to \mathsf{zipwith}(\circ)\ z\ (cols'\ y))\ (bottoms\ x) \circ bottoms\ y \\
bottoms\ (x \diamond y) \ = \ \mathsf{zipwith}_4\ f_b\ (bottoms\ x)\ (bottoms\ y)\ (bottomrights\ x)\ (bottomlefts\ y) \\
\qquad \textbf{where}\ f_b\ b_1\ b_2\ br_1\ bl_2 = (b_1 \diamond gemm\ (\_, \diamond)\ br_1\ bl_2) \circ (NIL \diamond b_2)
\end{array}
$$

$$
\begin{array}{l}
rights\ |a| \quad = \ |||a||| \\
rights\ (x \circ y) \ = \ (rights\ x \diamond gemm\ (\_, \mathsf{zipwith}(\circ))\ (bottomrights\ x)\ (toprights\ y)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \circ (NIL \diamond rights\ y) \\
rights\ (x \diamond y) \ = \ \mathsf{zipwith}_3\ f_r\ (rights\ x)\ (rights\ y)\ (rows'\ y) \\
\qquad \textbf{where}\ f_r\ r_1\ r_2\ ro_2 = \mathsf{map}\ (\diamond ro_2)\ r_1 \circ r_2
\end{array}
$$

$$
\begin{array}{l}
lefts\ |a| \quad = \ |||a||| \\
lefts\ (x \circ y) \ = \ (lefts\ x \diamond gemm\ (\_, \mathsf{zipwith}(\circ))\ (bottomlefts\ x)\ (toplefts\ y)) \circ (NIL \diamond lefts\ y) \\
lefts\ (x \diamond y) \ = \ \mathsf{zipwith}_3\ f_l\ (lefts\ x)\ (lefts\ y)\ (rows'\ x) \\
\qquad \textbf{where}\ f_l\ l_1\ l_2\ ro_1 = l_1 \diamond \mathsf{map}\ (ro_1 \diamond)\ l_2
\end{array}
$$

$$
\begin{array}{l}
toprights\ |a| \ = \ |||a||| \\
toprights\ (x \circ y) = toprights\ x \diamond \mathsf{map}\ (\mathsf{zipwith}(\circ)\ (right'\ (toprights\ x)))\ (toprights\ y) \\
toprights\ (x \diamond y) = \mathsf{zipwith}\ f_{tr}\ (toprights\ x)\ (toprights\ y) \\
\qquad \textbf{where}\ f_{tr}\ tr_1\ tr_2 = \mathsf{map}\ (\diamond top'\ tr_2 \diamond)\ tr_1 \circ tr_2
\end{array}
$$

$$
\begin{array}{l}
bottomrights\ |a| \ = \ |||a||| \\
bottomrights\ (x \circ y) = \mathsf{map}\ (\lambda z \to \mathsf{zipwith}(\circ)\ z\ (top'\ (bottomrights\ y)))\ (bottomrights\ x) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \circ bottomrights\ y \\
bottomrights\ (x \diamond y) = \mathsf{zipwith}\ f_{br}\ (bottomrights\ x)\ (bottomrights\ y) \\
\qquad \textbf{where}\ f_{br}\ br_1\ br_2 = \mathsf{map}\ (\diamond top'\ br_2 \diamond)\ br_1 \circ br_2
\end{array}
$$

$$
\begin{array}{l}
toplefts\ |a| \quad = \ |||a||| \\
toplefts\ (x \circ y) \ = \ toplefts\ x \diamond \mathsf{map}\ (\mathsf{zipwith}(\circ)\ (right'\ (toplefts\ x)))\ (toplefts\ y) \\
toplefts\ (x \diamond y) \ = \ \mathsf{zipwith}\ f_{tl}\ (toplefts\ x)\ (toplefts\ y) \\
\qquad \textbf{where}\ f_{tl}\ tl_1\ tl_2 = tl_1 \diamond \mathsf{map}\ (right'\ tl_1 \diamond)\ tl_2
\end{array}
$$

$$
\begin{array}{l}
bottomlefts\ |a| \ = \ |||a||| \\
bottomlefts\ (x \circ y) = \mathsf{map}\ (\lambda z \to \mathsf{zipwith}(\circ)\ z\ (top'\ (bottomlefts\ y)))\ (bottomlefts\ x) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \circ bottomlefts\ y \\
bottomlefts\ (x \diamond y) = \mathsf{zipwith}\ f_{bl}\ (bottomlefts\ x)\ (bottomlefts\ y) \\
\qquad \textbf{where}\ f_{bl}\ bl_1\ bl_2 = bl_1 \diamond \mathsf{map}\ (right'\ bl_1 \diamond)\ bl_2
\end{array}
$$

$$
\begin{array}{l}
cols'\ |a| \quad = \ ||a|| \\
cols'\ (x \circ y) \ = \ \mathsf{zipwith}(\circ)\ (cols'\ x)\ (cols'\ y) \\
cols'\ (x \diamond y) \ = \ (cols'\ x \diamond gemm\ (\_, \diamond)\ (right\ (cols'\ x))\ (top\ (cols'\ y))) \circ (NIL \diamond cols'\ y)
\end{array}
$$

$$
\begin{array}{l}
rows'\ |a| \quad = \ ||a|| \\
rows'\ (x \circ y) \ = \ (rows'\ x \diamond gemm\ (\_, \circ)\ (right\ (rows'\ x))\ (top\ (rows'\ y))) \circ (NIL \diamond rows'\ y) \\
rows'\ (x \diamond y) \ = \ \mathsf{zipwith}(\diamond)\ (rows'\ x)\ (rows'\ y)
\end{array}
$$

$$
\begin{array}{lcl}
top & = & \mathsf{reduce}(\ll, \diamond) \circ \mathsf{map}\ |\cdot| \\
bottom & = & \mathsf{reduce}(\gg, \diamond) \circ \mathsf{map}\ |\cdot| \\
right & = & \mathsf{reduce}(\circ, \gg) \circ \mathsf{map}\ |\cdot| \\
left & = & \mathsf{reduce}(\circ, \ll) \circ \mathsf{map}\ |\cdot| \\
top' & = & \mathsf{the} \circ top \\
bottom' & = & \mathsf{the} \circ bottom \\
right' & = & \mathsf{the} \circ right \\
left' & = & \mathsf{the} \circ left
\end{array}
$$

Fig. 4. Extra functions for *rects*.

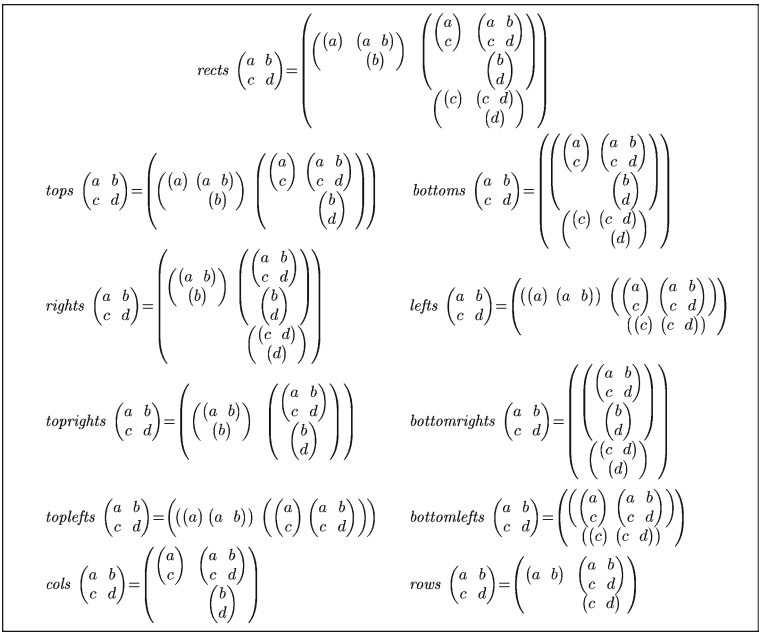

Fig. 5. Examples of extra functions for *rects*.

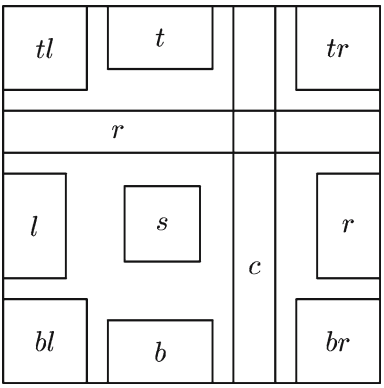

Fig. 6. Corresponding parts of elements in the tuple.

needs to execute $O(n^6)$ addition operations for the input of $n \times n$ array. We will show how to develop a more efficient parallel program.

4.2.2. Step 2. Driving Almost Homomorphism
First, we propose a way of deriving almost homomorphism from mutual recursive definitions. For notational convenience, we define

$$\begin{aligned} \triangle_1^n f_i &= f_1 \triangle f_2 \triangle \cdots \triangle f_n \\ x(\triangle_1^n \oplus_i) y &= (x \oplus_1 y, x \oplus_2 y, \ldots, x \oplus_n y). \end{aligned}$$

Our main idea is based on the following theorem.

Theorem 4.1. (Tupling) Let $h_1, h_2, \ldots, h_n$ be mutual recursively defined by

$$\begin{cases} h_i\ |a| &= f_i\ a, \\ h_i\ (x \circledcirc y) &= ((\triangle_1^n h_i)\ x) \oplus_i ((\triangle_1^n h_i)\ y), \\ h_i\ (x \circledast y) &= ((\triangle_1^n h_i)\ x) \otimes_i ((\triangle_1^n h_i)\ y). \end{cases} \tag{3}$$

Then $\triangle_1^n h_i$ is a homomorphism $(\!|\ \triangle_1^n\ f_i, \triangle_1^n \oplus_i, \triangle_1^n\ \otimes_i\ |\!)$.

Proof. The theorem is proven based on the definition of homomorphisms. See the technical report[33] for details. □

Theorem 4.1 says that if h_1 is mutually defined with other functions (i.e., $h_2, \ldots, h_n$) which traverse over the same array in the specific form of Eq. (3), then tupling $h_1, \ldots, h_n$ will give a homomorphism. It follows that every h_i is an almost homomorphism. Thus, this theorem gives us a systematic way to execute Step 2 of the strategy.

We apply this theorem to derive an almost homomorphism for *rects*. The definition of *rects* and the extra functions are in the form of Eq. (3). Thus, we can obtain the following almost homomorphism by tupling these functions as follows (we show only operators corresponding to *rects* for readability.)

$$
\begin{array}{l}
rects = \pi_1 \circ (\!| \triangle_1^{11}\ f_i, \triangle_1^{11} \oplus_i, \triangle_1^{11} \otimes_i |\!) \\
\textbf{where} \\
\quad f_1\ |a| = |||a||| \\
\quad (s_1, t_1, b_1, r_1, l_1, tr_1, br_1, tl_1, bl_1, c_1, ro_1) \\
\qquad \oplus_1\ (s_2, t_2, b_2, r_2, l_2, tr_2, br_2, tl_2, bl_2, c_2, ro_2) \\
\qquad\quad = (s_1 \diamond gemm\ (\_, \mathsf{zipwith}(\ominus))\ b_1\ t_2) \ominus (NIL \diamond s_2) \\
\quad (s_1, t_1, b_1, r_1, l_1, tr_1, br_1, tl_1, bl_1, c_1, ro_1) \\
\qquad \otimes_1\ (s_2, t_2, b_2, r_2, l_2, tr_2, br_2, tl_2, bl_2, c_2, ro_2) \\
\qquad\quad = \mathsf{zipwith}_4\ \ f_s\ s_1\ s_2\ r_1\ l_2 \\
\quad \textbf{where}\ f_s\, s_1\, s_2\, r_1\, l_2 = (s_1 \diamond gemm(\_, \diamond)\, r_1\, l_2) \ominus (NIL \diamond s_2)
\end{array}
$$

These operators are straightforward rewriting of the definition of *rects*.

4.2.3. Step 3. Fusing with Almost Homomorphisms

We aim to derive an efficient almost homomorphism for *mrs*. To this end, we give the following theorem showing how to fuse a function with an almost homomorphism to get new another almost homomorphism.

Theorem 4.2. (Almost Fusion) Let h and $(\!| \triangle_1^n\ f_i, \triangle_1^n \oplus_i, \triangle_1^n \otimes_i |\!)$ be given. If there exist $\odot_i, \ominus_i\ (i = 1, \ldots, n)$ and $H = h_1 \times h_2 \times \cdots \times h_n\ (h_1 = h)$ such that for any i, x and y,

$$
\begin{aligned}
h_i\ (x \oplus_i y) &= H\ x \odot_i H\ y \\
h_i\ (x \otimes_i y) &= H\ x \ominus_i H\ y
\end{aligned}
$$

hold, then

$$
h \circ (\pi_1 \circ (\!| \triangle_1^n\ f_i, \triangle_1^n \oplus_i, \triangle_1^n \otimes_i |\!)) = \pi_1 \circ (\!| \triangle_1^n\ (h_i \circ f_i), \triangle_1^n \odot_i, \triangle_1^n \ominus_i |\!). \quad (4)
$$

Proof. The theorem is proven by some calculation and Theorem 2.3. See the technical report[33] for details. □

Theorem 4.2 says that we can fuse a function with an almost homomorphism to get another almost homomorphism by finding $h_2, \ldots, h_n$ together with $\odot_1, \ldots, \odot_n$, $\ominus_1, \ldots, \ominus_n$ that satisfy Eq. (4). Thus, this theorem gives us a systematic way to execute Step 3 of the strategy.

Returning to our example, we apply this theorem to *mrs*. The second function p_2 of our example is map (map *sum*), so $h_1 =$ map (map *sum*). Then, we calculate $h_1\ (x \oplus_1 y)$ to find other functions and operators.

$$
\begin{aligned}
& h_1\ (x \oplus_1 y) \\
= \quad & \{ \text{ Expand } x,\ y \text{ and } h_1 \ \} \\
& \mathsf{map}\,(\mathsf{map}\ sum)\,((s_1 \Phi\ gemm(\_, \mathsf{zipwith}(\ominus))\ b_1\ t_2) \ominus (NIL \Phi\ s_2)) \\
= \quad & \{ \text{ Definition of } \mathsf{map} \ \} \\
& (\mathsf{map}\,(\mathsf{map}\ sum) s_1 \Phi\ \mathsf{map}\,(\mathsf{map}\ sum)\ (gemm(\_, \mathsf{zipwith}(\ominus))\ b_1\ t_2)) \\
& \qquad\qquad \ominus (NIL \Phi\ \mathsf{map}\,(\mathsf{map}\ sum) s_2) \\
= \quad & \{ \text{ Promotion of } \mathsf{map}, \text{ folding } \} \\
& (h_1\, s_1 \Phi\ gemm(\_, \mathsf{zipwith}(+))(\mathsf{map}\,(\mathsf{map}\ sum)\ b_1)\ (\mathsf{map}\,(\mathsf{map}\ sum)\ t_2)) \\
& \qquad\qquad \ominus (NIL \Phi\ h_1\, s_2)
\end{aligned}
$$

In the last formula, functions applied to t_1 and b_1 should be h_2 and h_3, respectively, which suggests us to define h_2, h_3 and $\odot_1$ as follows.

$$
\begin{aligned}
& h_1 = h_2 = h_3 = \mathsf{map}\,(\mathsf{map}\ sum) \\
& (s_1, t_1, b_1, r_1, l_1, tr_1, br_1, tl_1, bl_1, c_1, ro_1) \\
& \odot_1\ \ (s_2, t_2, b_2, r_2, l_2, tr_2, br_2, tl_2, bl_2, c_2, ro_2) \\
& = (s_1 \Phi\ gemm\ (\_, \mathsf{zipwith}(+))\ b_1\ t_2) \ominus (NIL \Phi\ s_2)
\end{aligned}
$$

Similarly, we can derive $\ominus_1$ by calculating $h_1\ (x \otimes_1 y)$ as follows:

$$
\begin{aligned}
& (s_1, t_1, b_1, r_1, l_1, tr_1, br_1, tl_1, bl_1, c_1, ro_1) \\
& \ominus_1\ \ (s_2, t_2, b_2, r_2, l_2, tr_2, br_2, tl_2, bl_2, c_2, ro_2) \\
& \qquad = \mathsf{zipwith}_4\ \ f_s\ \ s_1\ \ s_2\ \ r_1\ \ l_2 \\
& \textbf{where}\ \ f_s\, s_1\, s_2\, r_1\, l_2 = (s_1 \Phi\ gemm(\_, +)\, r_1\, l_2) \ominus (NIL \Phi\ s_2)
\end{aligned}
$$

and derive other functions and operators by doing similarly about $\oplus_i$ and $\otimes_i$. Finally, we get the following.

$$
\begin{aligned}
& h_i = \mathsf{map}\,(\mathsf{map}\ sum)\ \ (i = 1, \ldots, 9) \\
& h_{10} = h_{11} = \mathsf{map}\ sum \\
& \mathsf{map}\,(\mathsf{map}\ sum) \circ rects = \pi_1 \circ (\!|\ \triangle_1^{11}\ f_i', \triangle_1^{11} \odot_i, \triangle_1^{11}\ \ominus_i\ |\!) \\
& \quad \textbf{where} \\
& \qquad f_1'\ |a| = ||a||
\end{aligned}
$$

Repeating such fusion with $\mathsf{map}\ max$ and max will yield the result shown in Fig. 7. In this final program, each of the nine elements of the resulting tuple is the partial answer of its counterpart in Fig. 6. Some parts are calculated with a general matrix multiplication, and others are updated with map and zipwith. For example, the element s_0, which is the solution of the maximum rectangle sum, is the maximum of the solutions

$$
\begin{array}{l}
mrs = \pi_1 \circ (\!| \triangle_1^{11}\ f_i''', \triangle_1^{11} \odot_i'', \triangle_1^{11} \ominus_i'' |\!) \\
\textbf{where} \\
\quad (\triangle_1^{11} f_i''')\ |a| = (a, |a|, |a|, |a|, |a|, |a|, |a|, |a|, |a|, |a|, |a|, |a|) \\
\quad (s_1, t_1, b_1, r_1, l_1, tr_1, br_1, tl_1, bl_1, c_1, ro_1) \\
\qquad (\triangle_1^{11} \odot_i'')\ (s_2, t_2, b_2, r_2, l_2, tr_2, br_2, tl_2, bl_2, c_2, ro_2) \\
\qquad\quad = (s_0, t_0, b_0, r_0, l_0, tr_0, br_0, tl_0, bl_0, c_0, ro_0) \\
\quad \textbf{where} \\
\quad\ s_0 \ = \ (s_1 \uparrow max(\mathsf{zipwith}(+)\ b_1\ t_2) \uparrow s_2 \\
\quad\ t_0 \ = \ \mathsf{zipwith}_3\ f_t\ t_1\ c_1\ t_2 \\
\qquad \textbf{where}\ f_t\ t_1\ c_1\ t_2 = t_1 \uparrow (c_1 + t_2) \\
\quad\ b_0 \ = \ \mathsf{zipwith}_3\ f_b\ b_1\ c_2\ b_2 \\
\qquad \textbf{where}\ f_b\ b_1\ c_2\ b_2 = (b_1 + c_2) \uparrow b_2 \\
\quad\ r_0 \ = \ (r_1 \Phi\ gemm\ (\uparrow, +)\ (tr\ br_1)\ tr_2) \Theta (NIL \Phi\ r_2) \\
\quad\ l_0 \ = \ (l_1 \Phi\ gemm\ (\uparrow, +)\ bl_1\ (tr\ tl_2)) \Theta (NIL \Phi\ l_2) \\
\quad\ tr_0 \ = \ tr_1 \Phi\ \mathsf{map}_c\ (\mathsf{zipwith}(+)\ (right\ tr_1))\ tr_2 \\
\quad\ br_0 \ = \ \mathsf{map}_c\ (\mathsf{zipwith}(+)\ (left\ br_2))\ br_1 \Phi\ br_2 \\
\quad\ tl_0 \ = \ tl_1 \Theta \mathsf{map}_r\ (\mathsf{zipwith}(+)\ (bottom\ tl_1))\ tl_2 \\
\quad\ bl_0 \ = \ \mathsf{map}_r\ (\mathsf{zipwith}(+)\ (top\ bl_2))\ bl_1 \Theta bl_2 \\
\quad\ c_0 \ = \ \mathsf{zipwith}(+)\ c_1\ c_2 \\
\quad\ ro_0 \ = \ (ro_1 \Phi\ gemm(\_, +)\ (right\ ro_1)\ (top\ ro_2)) \Theta (NIL \Phi\ ro_2) \\
\\
\quad (s_1, t_1, b_1, r_1, l_1, tr_1, br_1, tl_1, bl_1, c_1, ro_1) \\
\qquad (\triangle_1^{11} \ominus_i'')\ (s_2, t_2, b_2, r_2, l_2, tr_2, br_2, tl_2, bl_2, c_2, ro_2) \\
\qquad\quad = (s_0, t_0, b_0, r_0, l_0, tr_0, br_0, tl_0, bl_0, c_0, ro_0) \\
\quad \textbf{where} \\
\quad\ s_0 \ = \ s_1 \uparrow max(\mathsf{zipwith}(+)\ r_1\ l_2) \uparrow s_2 \\
\quad\ t_0 \ = \ (t_1 \Phi\ gemm\ (\uparrow, +)\ tr_1\ tl_2) \Theta (NIL \Phi\ t_2) \\
\quad\ b_0 \ = \ (b_1 \Phi\ gemm\ (\uparrow, +)\ br_1\ bl_2) \Theta (NIL \Phi\ b_2) \\
\quad\ r_0 \ = \ \mathsf{zipwith}_3\ f_r\ r_1\ r_2\ ro_2 \\
\qquad \textbf{where}\ f_r\ r_1\ r_2\ ro_2 = (r_1 + ro_2) \uparrow r_2 \\
\quad\ l_0 \ = \ \mathsf{zipwith}_3\ f_l\ l_1\ l_2\ ro_1 \\
\qquad \textbf{where}\ f_l\ l_1\ l_2\ ro_1 = l_1 \uparrow (ro_1 + l_2) \\
\quad\ tr_0 \ = \ \mathsf{map}_r\ (\mathsf{zipwith}(+)(top\ tr_2))\ tr_1 \Theta tr_2 \\
\quad\ br_0 \ = \ \mathsf{map}_r\ (\mathsf{zipwith}(+)(top\ br_2))\ br_1 \Theta br_2 \\
\quad\ tl_0 \ = \ tl_1 \Phi\ \mathsf{map}_c\ (\mathsf{zipwith}(+)\ (right\ tl_1))\ tl_2 \\
\quad\ bl_0 \ = \ bl_1 \Phi\ \mathsf{map}_c\ (\mathsf{zipwith}(+)\ (right\ bl_1))\ bl_2 \\
\quad\ c_0 \ = \ (c_1 \Phi\ gemm(\_, +)\ (right\ c_1)\ (top\ c_2)) \Theta (NIL \Phi\ c_2) \\
\quad\ ro_0 \ = \ \mathsf{zipwith}(+)\ ro_1\ ro_2
\end{array}
$$

Fig. 7. Derived efficient program of maximum rectangle sum.

of upper and lower subarray or the maximum of the solutions generated by combining partial answers of the top and the bottom rectangles as shown in Fig. 8(a). Here, the rectangles that share the same edge are combined by zipwith(+). Similarly, some elements of the array r_0, which is the partial solutions of the rectangles on the right edge, are calculated with a general matrix multiplication as shown in Fig. 8(b). Here, (i, j) element in the block is the maximum of $br_1(i, k) + br_1(k, j)$ for all k.

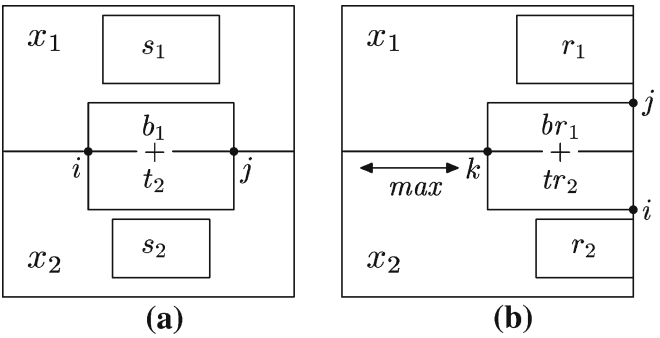

Fig. 8. Computation of the operator for derived *mrs*.

Provided that we divide the input array into two parts evenly, this final parallel program uses only $O(n^3)$ addition operations as follows. For an $n \times n$ input array, the program's cost $T(n, n)$ satisfies the next equation with $gemm$'s cost $T_{gemm}(n, n)$ and some constants c_1 and c_2.

$$T(n, n) = 4T(n/2, n/2) + c_1 T_{gemm}(n/2, n/2) + c_2 n^2$$

Since the cost of the general matrix multiplication ($gemm$) is $O(n^3)$, the answer of the above equation is $T(n, n) = O(n^3)$. This is much better than the initial one. Moreover, since the general matrix multiplication $gemm(\uparrow, +)$ used in the program is a distance matrix multiplication, we can achieve subcubic cost with the special implementation of the $gemm(\uparrow, +)$ used in Takaoka's algorithm.[32] However, the implementation is somewhat tricky, so that we cannot describe it with our skeletons.

4.2.4. Step 4. Optimizing Inner Functions

For our example, we may proceed to optimize the operators and functions such as f_i''', $\odot_i''$ and $\ominus_i''$ in the program of Step 3. Since they cannot be made efficient any more, we finish our derivation of an efficient parallel program.

5. Implementation

In this section, we will give an efficient parallel implementation (on PC clusters) of the parallel skeletons, which are primitive operations on two-dimensional arrays defined in Sections 3.1 and 3.2. Since a homomorphism can be specified as a composition of the reduce and map skeletons, homomorphisms have efficient parallel implementations. Our parallel skeletons are implemented as a C++ library with MPI. We will report some

experimental results, showing programs described with skeletons can be executed efficiently in parallel.

5.1. Implementation of Data Parallel Skeletons

The four basic data parallel skeletons of map, zipwith, reduce and scan can be efficiently implemented on distributed memory systems. To illustrate this, we separate computations of a skeleton into two parts: local computations within a processor and global computations crossing processors.

For map skeleton, we can separate its computation as follows.

$$\begin{aligned}\mathsf{map}\ f &= \mathsf{map}\ f \circ \mathsf{gather} \circ \mathsf{dist}\ p\ q \\ &= \mathsf{map}\ f \circ \mathsf{reduce}(\leftarrow, \phi) \circ \mathsf{dist}\ p\ q \\ &= \mathsf{reduce}(\leftarrow, \phi) \circ \mathsf{map}\ (\mathsf{map}\ f) \circ \mathsf{dist}\ p\ q \\ &= \mathsf{gather} \circ \mathsf{map}\ (\mathsf{map}\ f) \circ \mathsf{dist}\ p\ q\end{aligned}$$

The last formula indicates that we can compute map f by distributing a two-dimensional array of the argument to the processors by dist $p\ q$, applying map f to each local array independently on each processor, and finally gathering the results onto the root processor by gather. Thus, for a two-dimensional array of $n \times n$ size we can compute map f in $O(n^2/P)$ parallel time, using $P = pq$ processors and ignoring distribution and collection provided that the function f can be computed in O(1) time. This is the same also about zipwith.

For reduce skeleton, we can separate its computation as follows.

$$\begin{aligned}\mathsf{reduce}(\oplus, \otimes) &= \mathsf{reduce}(\oplus, \otimes) \circ \mathsf{gather} \circ \mathsf{dist}\ p\ q \\ &= \mathsf{reduce}(\oplus, \otimes) \circ \mathsf{reduce}(\leftarrow, \phi) \circ \mathsf{dist}\ p\ q \\ &= \mathsf{reduce}(\oplus, \otimes) \circ \mathsf{map}\ (\mathsf{reduce}(\oplus, \otimes)) \circ \mathsf{dist}\ p\ q\end{aligned}$$

The last formula indicates that we can compute reduce$(\oplus, \otimes)$ by distributing a two-dimensional array of the argument to the processors by dist $p\ q$, applying reduce$(\oplus, \otimes)$ to each local array independently on each processor, and finally reducing the results into the root processor by reduce$(\oplus, \otimes)$ described in the last formula. From the property of Eq. (1), the last reduction over the results of all processors can be computed by using tree-like computation in column and row directions respectively like parallel computation of reduction on one-dimensional lists. Thus, for a two-dimensional array of $n \times n$ size we can compute reduce$(\oplus, \otimes)$ in $O(n^2/P + \log P)$ parallel time, using $P = pq$ processors and ignoring distribution provided that the binary operators $\oplus$ and $\otimes$ can be computed

in O(1) time. Note that we can also execute the tree-like computation in both of column and row directions simultaneously.

For scan skeleton, we can separate its computation as follows.

$$\begin{aligned}
\mathsf{scan}(\oplus, \otimes) &= \mathsf{reduce}(\oplus', \otimes') \circ \mathsf{map}\,|\cdot| \circ \mathsf{gather} \circ \mathsf{dist}\ p\ q \\
&= \mathsf{reduce}(\oplus', \otimes') \circ \mathsf{map}\,|\cdot| \circ \mathsf{reduce}(\ominus, \phi) \circ \mathsf{dist}\ p\ q \\
&= \mathsf{reduce}(\oplus', \otimes') \circ \mathsf{map}\,(\mathsf{reduce}(\oplus', \otimes') \circ \mathsf{map}\,|\cdot|) \circ \mathsf{dist}\ p\ q \\
&= \mathsf{reduce}(\oplus', \otimes') \circ \mathsf{map}\,(\mathsf{scan}(\oplus, \otimes)) \circ \mathsf{dist}\ p\ q \\
&= \mathsf{gather} \circ \underline{\mathsf{dist}\ p\ q \circ \mathsf{reduce}(\oplus', \otimes')} \circ \mathsf{map}\,(\mathsf{scan}(\oplus, \otimes)) \circ \mathsf{dist}\ p\ q
\end{aligned}$$

The second last formula indicates we can compute $\mathsf{scan}(\oplus, \otimes)$ by distributing a two-dimensional array of the argument to the processors by dist $p\ q$, applying $\mathsf{scan}(\oplus, \otimes)$ to each local array independently on each processor, and finally reducing the results into the root processor by $\mathsf{reduce}(\oplus', \otimes')$. However, since the result of $\mathsf{scan}(\oplus, \otimes)$ is a two-dimensional array, we want that the last operation of computing $\mathsf{scan}(\oplus, \otimes)$ is gather like the case of map f. Thus, we compute underlined dist $p\ q \circ \mathsf{reduce}(\oplus', \otimes')$ instead of the last reduction $\mathsf{reduce}(\oplus', \otimes')$. Although under our notation the underlined computation cannot be written in simpler form, we can compute it in sequence in column and row direction like the case of reduce. The computation in each direction can be done like those of lists.[17] From the property of Eq. (2), we can also compute $\mathsf{scan}(\oplus, \otimes)$ by computing $\mathsf{scan}_{\downarrow}(\oplus)$ after $\mathsf{scan}_{\rightarrow}(\otimes)$. Note that $\mathsf{scan}_{\downarrow}(\oplus)$ and $\mathsf{scan}_{\rightarrow}(\otimes)$ can be computed in the same way of scan on lists although it performs to two or more lists simultaneously. Thus, for a two-dimensional array of $n \times n$ size we can compute $\mathsf{scan}(\oplus, \otimes)$ in $O(n^2/P + \sqrt{n^2/P} \log P)$ parallel time, using $P = pq$ processors and ignoring distribution and collection provided that the binary operators $\oplus$ and $\otimes$ can be computed in O(1) time. Note that we can also execute the global computation in both of column and row directions simultaneously.

Finally, we note that gathering and re-distribution between successive calls of skeletons can be canceled when dist $p\ q \circ$ gather $= id$. This condition is satisfied when the argument arrays of the successive calls of skeletons are distributed in the same way. That is, once argument arrays are distributed, we do not need extra communication caused by re-distribution of arrays. Since this situation occurs very often, we omit communication cost of distribution of argument arrays in the cost of skeletons. However, we sometimes need to change distribution of argument arrays in some computations. For example, iterated matrix-vector multiplication $x_n = Ax_{n-1}$, which is the kernel of many important algorithms, needs such re-distribution between steps. In the computation, we distribute the matrix A row-wisely and duplicate x_{n-1} onto each processor. Each processor computes a slice of x_n and then

broadcasts it to all other processors for the next iteration. In this case, the communication cost of the re-distribution (the broadcast) cannot be factored out by the condition $\mathsf{dist}\ p\ q \circ \mathsf{gather} = id$. To take the communication cost into account, we represent the communication via computation with our skeletons. To illustrate it, we formalize the example as follows.

$$
\begin{aligned}
&mvmul\ n\ A\ x_0 = iter\ n\ step\ x_0 \\
&\quad \textbf{where}\ \ step = (\mathsf{zipwith}\ \ iprod\ rA) \circ duplicate \\
&\qquad\qquad duplicate = \mathsf{scanr}(\gg, \gg) \circ \mathsf{scan}(\ominus, \oplus) \circ \mathsf{map}\ \ |\cdot| \\
&\qquad\qquad rA = rows\ A \\
&\qquad\qquad rows = \mathsf{reduce}(\ominus, \mathsf{zipwith}(\oplus)) \circ \mathsf{map}\ ||\cdot||
\end{aligned}
$$

In each iteration step, the argument vector x_{n-1} is redistributed (broadcasted) by $duplicate$, which is a composition of skeletons, and then each copy is used to produce an inner product with a row of the matrix A, which is distributed row-wisely and named rA. This $duplicate$ seems not to do actual computation but does computation of re-distribution, i.e., it copies a partial result on other processor and concatenates it to its own partial result. Since the cost of the concatenating operators (i.e., $\ominus$ and $\oplus$) for arrays of N elements is considered to be $O(N)$ in the worst case, the cost of $duplicate$ is $O(N \log N)$ and this is equal to the cost of the usual communication. This is the idea to take the communication cost into account. Note that gathering and re-distribution between successive steps in the iteration can be canceled since the distribution of the argument vector x_n is the same at beginning of each step.

Besides $duplicate$, we can represent several communications such as broadcasting and shifting via our skeletons.

$$
\begin{aligned}
bcast &= \mathsf{scan}(\ll, \ll) \\
shift_C &= \mathsf{map}(\pi_1) \circ \mathsf{scan}(\gg, \odot) \circ \mathsf{map}\ \ init \\
shift_R &= \mathsf{map}(\pi_1) \circ \mathsf{scan}(\odot, \gg) \circ \mathsf{map}\ \ init \\
&\qquad \textbf{where}\ \ init\ a = (\epsilon, a) \\
&\qquad\qquad\qquad (x, a) \odot (\epsilon, b) = (a, b) \\
&\qquad\qquad\qquad (x, a) \odot (y, b) = (y, b) \quad (y \neq \epsilon)
\end{aligned}
$$

The function $bcast$ broadcasts the element on the root processor to the other processors. The column-shifting function $shift_C$ shifts each column to the right and fills the first column with special value ϵ. The row-shifting function $shift_R$ shifts each row to the bottom. As illustrated above, we represent explicit communications by computations with skeletons. However, sometimes those representations have more costs than usual implementations and we need to consider special implementations. Some of such communications are discussed in the next section.

5.2. Implementation of Data Communication Skeletons

We have efficient parallel implementations for the data communication skeletons defined in Section 3.2.

Since dist distributes all elements of a two-dimensional array at the root processor to all other processors and gather does the inverse, we can compute dist and gather in $O(n^2)$ parallel time for a two-dimensional array of $n \times n$ size.

Although the definition of $\mathsf{rot}_r\ f$ given in Section 3 is complicated, the actual operation of $\mathsf{rot}_r\ f$ is simple. Function $\mathsf{rot}_r\ f$ merely rotates independently the ith row by $f\ i$, and rotation of each row can be done by four parallel communications. Without losing generality we can assume that the amount of rotation $r = f\ i$ satisfies $0 < r \leq n/2$ where n is the length of the row because we just reverse the direction of rotation in the case of $n/2 < r$. The operations are followings: (1) making groups of $2r$ processors from the first processor of the row (i.e., $n/(2r)$ groups are made) and transmitting subarrays of first r processors to the rest r processors in each group, (2) considering that processors from the 0th to the rth continue behind the last processor, making groups of $2r$ processors from the rth processor of the row and transmitting subarrays of first r processors to the rest r processors in each group (i.e., processors in the first $n/(2r)$ groups have transmitted their subarrays), (3) doing the former two operations on the rest processors that have not transmitted their subarrays yet, considering the processors that have done continue behind the processors. Since more than the half processors have transmitted their subarrays by the end of the former two operations, all processors can transmit their subarrays by the end of third operation. Thus, since the amount of one communication is $O(n^2/P)$ for P processors, $\mathsf{rot}_r\ f$ can be executed in $O(n^2/P)$ parallel time. Similarly, $\mathsf{rot}_c\ f$ can be executed in $O(n^2/P)$ parallel time.

5.3. Optimization by Fusion

Although it is easy to make a parallel program by composing parallel skeletons, a simply composed skeletal program has overheads due to redundant intermediate data structures, communications and synchronizations. Thus, we perform optimization on user-written simple skeleton programs by fusing successive skeletons to eliminate redundant intermediate data structures, communications and synchronizations.

The main rules to perform optimization by fusion in our current system are as follows.

$$\begin{aligned}
&\mathsf{map}\ f \circ \mathsf{map}\ g &&\Rightarrow \mathsf{map}\ (f \circ g)\\
&\mathsf{reduce}(\oplus, \otimes) \circ \mathsf{map}\ f &&\Rightarrow (\!|f, \oplus, \otimes|\!)\\
&(\!|f, \oplus, \otimes|\!) \circ \mathsf{map}\ g &&\Rightarrow (\!|f \circ g, \oplus, \otimes|\!)
\end{aligned}$$

These rules are instances of the fusion rule of Theorem 2.3. Successive skeletons on the left hand side are fused into one skeleton on the right hand side. Then, intermediate data structures, communications and synchronizations between skeletons on the left hand side are eliminated in the right hand side. Here, a homomorphism $(\!|f, \oplus, \otimes|\!)$ is just used as an internal representation for the optimization, and users do not need to consider homomopphisms explicitly. Since a homomorphism clearly has an efficient implementation in parallel, we use the implementation for homomorphisms produced by the optimization.

This optimization can be done automatically by the library, in the similar way to automatic optimization for lists.[19] It is our future work to formalize further rules for optimization by fusion.

5.4. Experimental Results

We implemented the parallel skeletons as part of the skeleton library *SkeTo*[36] [5] with C++ and MPI, and did our experiment on a cluster (distributed memory). Each of the nodes connected with Gigabit Ethernet has a CPU of Intel® Xeon®2.80 GHz and 2 GB memory, with Linux 2.4.21 for the OS, gcc 4.1.1 for the compiler, and mpich 1.2.7 for the MPI.

We measured execution times and speedups for the following parallel programs described by the parallel skeletons:

Frobenius Norm
$fnorm = \mathsf{reduce}(+, +) \circ \mathsf{map}\ square$,
Matrix Multiplication
mm (composition of skeletons; see Sect. 3.3),
Maximum Rectangle Sum
mrs (derived program written by skeletons; see Sect. 4.2).

Figures 9–11 show measured speedups of the programs and their optimized versions. Optimized programs are named XXX_opt. The speedup is a

[5]It is available at http://www.ipl.t.u-tokyo.ac.jp/sketo/

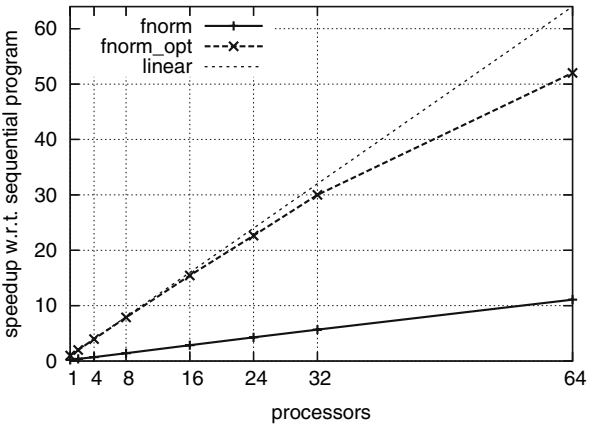

Fig. 9. Speedup of F-norm.

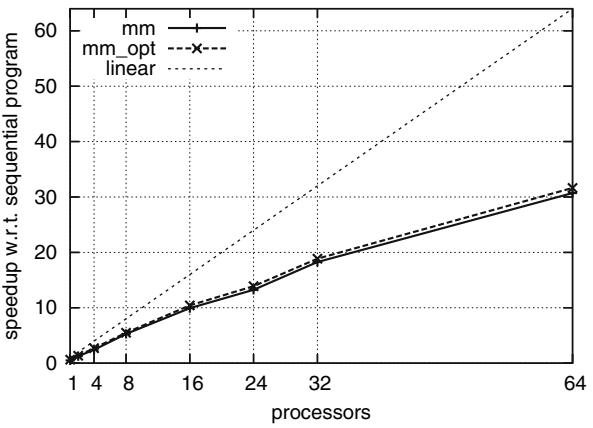

Fig. 10. Speedup of matrix multiplication.

ratio of running time of the program on p processors to that of hand-written sequential program on one processor, without first distribution of arrays. Tables I–III show measured running times and speedups. Besides speedups with respect to hand-written sequential programs, speedups with respect to the skeleton programs on one processor are listed in the tables. The inputs are an 8000×8000 matrix for $fnorm$, a 3000×3000 matrix for mm, and a 1000×1000 matrix for mrs. The running time of mrs is relatively big because its operators in its calculation are more complicated than those of the others.

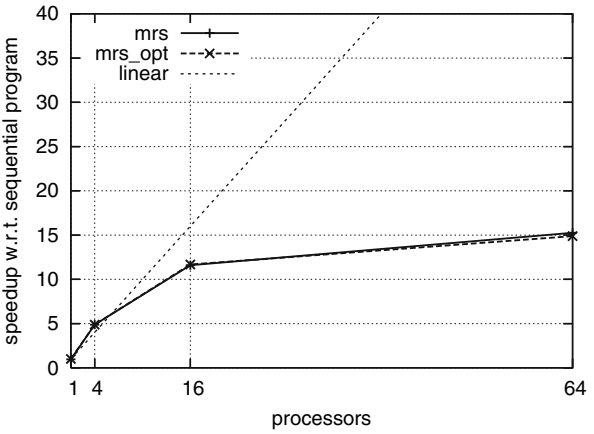

Fig. 11. Speedup of maximum rectangle sum.

Table I. Experimental Results of F-norm for 8000 × 8000 Matrices

#Processors	1	2	4	8	16	24	32	64
fnorm								
time (s)	1.40	0.70	0.39	0.19	0.097	0.065	0.049	0.025
speedup	1.00	1.99	3.59	7.21	14.45	21.55	28.65	55.86
†speedup^s	0.20	0.40	0.71	1.43	2.87	4.27	5.68	11.08
fnorm_opt								
time (s)	0.28	0.14	0.070	0.035	0.018	0.012	0.009	0.005
speedup	1.00	1.99	3.98	7.88	15.46	22.65	30.00	52.01
†speedup^s	1.00	1.99	3.98	7.88	15.46	22.65	30.00	52.01

†speedup^s is a speedup with respect to hand-written sequential program.

The result shows programs described with skeletons can be executed efficiently in parallel, and proves the success of our framework. Both simply composed skeleton programs and optimized programs achieve good speedups. The optimized programs achieve better running time than the simple programs. Since *mrs* requires a large amount of communication data and nested parallelism, speedup of *mrs* on 64 processors for a 1000 × 1000 matrix is not so good.

To examine sequential performances of our skeleton programs, we implemented sequential programs for the above programs. We compared their calculation times with those of the skeleton programs on one processor. The program *fnorm* simply written with our skeletons is five times slower than the hand-made sequential program. This is caused by

Table II. Experimental Results of Matrix Multiplication for 3000 x 3000 Matrices

#Processors	1	2	4	8	16	24	32	64
mm								
time (s)	247.02	96.26	49.26	22.66	12.09	9.05	6.56	3.91
speedup	1.00	2.57	5.02	10.90	20.44	27.30	37.64	63.23
†speedup^s	0.49	1.25	2.43	5.29	9.92	13.25	18.27	30.70
mm_opt								
time (s)	193.34	89.86	44.17	21.89	11.51	8.64	6.36	3.79
speedup	1.00	2.15	4.38	8.83	16.80	22.39	30.42	50.99
†speedup^s	0.62	1.33	2.71	5.48	10.42	13.89	18.87	31.63

†speedup^s is a speedup with respect to hand-written sequential program.

Table III. Experimental Results of Maximum Rectangle Sum for 1000 x 1000 Matrices

#Processors	1	4	16	64
mrs				
time (s)	157.01	31.04	13.05	9.92
speedup	1.00	5.06	12.03	15.84
†speedup^s	0.96	4.88	11.59	15.26
mrs_opt				
time (s)	151.32	31.09	12.97	10.17
speedup	1.00	4.87	11.67	14.88
†speedup^s	1.00	4.87	11.67	14.88

†speedup^s is a speedup with respect to hand-written sequential program.

overhead due to creation of intermediate data between map $square$ and reduce$(+, +)$. Thus, the program optimized by fusion achieves the same running time as the sequential program due to elimination of the intermediate data. The program mm written with our skeletons is two times slower than the hand-made sequential program. This is mainly caused by overhead of methods for accessing elements of matrices, and some compilers can eliminate this overhead. Thus, the program optimized by fusion achieves better running time than the simple program, but it is still 1.6 times slower than the sequential program. Since operators in the calculation of mrs are complicated, the overhead of skeletons are relatively small. Thus, the running times of sequential program, simple program and optimized program are almost the same.

```
template <class C, class A, class B>
void mm(dist_matrix<C> &Z2, const dist_matrix<A> &X2, const dist_matrix<B> &Y2)
{
    dist_matrix < matrix < int > > *A2;
    dist_matrix < matrix < int > > *B2;
    A2 = all_rows2(X2);
    B2 = all_cols2(Y2);
    m_skeletons::zipwith(Iprod<C>(), *A2, *B2, Z2);
    delete B2;
    delete A2;
}
```

Fig. 12. C++ code of matrix multiplication by parallel skeletons.

Finally, we list part of the C++ code of mm written with the skeleton library in Fig. 12, to give a concrete impression of the conciseness our library provides.

6. RELATED WORKS

Besides the related work as in the introduction, our work is closely related with the active researches on matrix representation for parallel computations and the compositional approach to parallel program development.

6.1. Recursive Matrix Representations

Wise et al.[27] proposed representation of a two-dimensional array by a quadtree, i.e., a two-dimensional array recursively constructed by four small sub-arrays of the same size. This representation is suitable for describing recursive blocked algorithms,[1] which can provide better performance than existing algorithms for some matrix computations such as LU and QR factorizations.[28,29] However, the quadtree representation requires the size of two-dimensional arrays to be the power of two. Moreover, once a two-dimensional array is represented by a quadtree, we cannot reblock the array by restructuring the quadtree, which would prevent us from developing more parallelism in the recursive blocked algorithms on them.

Bikshandi et al.[37] proposed representation of a two-dimensional array by a hierarchically tiled array (HTA). An HTA is an array partitioned into tiles, and these tiles can be either conventional arrays or lower level HTAs. The outermost tiles are distributed across processors for parallelism and the inner tiles are utilized for locality. In the HTA programming, users are allowed to use recursive index accessing according to the structure of HTAs, so that they can easily transform conventional programs onto HTA programs. Communication and synchronization are

explicitly expressed by index accessing to remote tiles. Thus HTA programs can control relatively low-level parallelism and can be efficient implementation. However, it is not presented how to derive efficient HTA programs. We think it is good that we derive an efficient algorithm on the abide-tree then implement it on HTAs.

A more natural representation of a two-dimensional array is to use nested one-dimensional arrays (lists).[23,25,26] The advantage is that many results developed for lists can be reused. However, this representation imposes much restriction on the access order of elements.

The abide-tree representation, as used in this paper, was first proposed by Bird,[23] as an extension of one-dimensional join lists. However, the focus there is on derivation of sequential programs for manipulating two-dimensional arrays, and there is little study on the framework for developing efficient parallel programs. Our work provides a good complement.

6.2. Compositional Parallel Programming

This work was greatly inspired by the success of compositional (skeletal) parallel programming on one-dimensional arrays (lists),[13] and our initial motivation was to import the results so far to two-dimensional arrays. This turns out to be more difficult than we had expected.

Compositional parallel Programming using Bird-Meertens Formalism (BMF) has been attracting many researchers. The initial BMF[38] was designed as a calculus for deriving (sequential) efficient programs on lists. Skillicorn[39] showed that BMF could also provide an architecture independent parallel model for parallel programming because a small fixed set of higher order functions (skeletons) in BMF such as map and reduce can be mapped efficiently to a wide range of parallel architectures.

Systematic programming methods have actively been studied in the framework of skeletal (compositional) parallel programming on lists. The diffusion theorem[15] gives a powerful method to obtain suitable composition of skeletons for a program recursively defined on lists and trees. Chin et al.[16,40] have studied a systematic method to derive an associative operator that plays an important role in parallelization, based on which Xu et al.[41] build an automatic derivation system for parallelizing recursive linear functions with normalization rules.

Our parallel skeletons can be seen as an instance of design-patterns in object-oriented programmings. The difference between our approach and usual design-pattern approaches is that we use a small number of skeletons carefully designed rather than heedlessly increasing the number of patterns. This enables us to make a clear automatic optimization mechanism. We also give a methodology to derive an efficient program described with skeletons.

APL[42–44] is a pioneer language that supports operators to manipulate arrays, namely array operators. APL's array operators are extensions of scalar operations and functions to multi-dimensional arrays in an element-wise manner, manipulations of the layout of arrays such as shift and rotate, and reduction operators to collapse arrays with binary operators. More complex computations on arrays are constructed by composition of these array operators. The ideas of APL and our skeletal approach are the same in the sense that both provide users with a set of basic patterns of array computations and let users make sophisticated computations by compositions of these basic patterns. Also, both array operators and skeletons conceal complicated parallelism form users. Main difference between APL and our skeletal approach is as follows. We give a systematic methodology to derive (make) correct, safe and efficient programs, i.e., how to derive efficient programs from correct and safe programs that are not so efficient. Our skeletons treat 2D arrays as '2D' in reductions since they can use two binary operators for horizontal direction and vertical direction, while treatment of multidimensional arrays in reductions of APL is rather like '1D' because they use only one operator in one reduction. Our skeletons are designed to consider derivations and optimizations for parallel programs of compositions of skeletons based on the mathematical theory.

7. CONCLUSION

In this paper, we propose a compositional framework that allows users, even with little knowledge about parallel machines, to describe safe and efficient parallel computation over two-dimensional arrays easily. In our framework, two-dimensional arrays are represented by the abide-tree that supports systematic development of parallel programs and architecture-independent implementation, and programmers can easily build up a complicated parallel system by defining basic components recursively, putting components compositionally, and improving efficiency systematically. The power of our approach is seen from the nontrivial programming examples of matrix multiplication and QR decomposition, and a successful derivation of an involved efficient parallel program for the maximum rectangle sum problem.[18] A demonstration of an efficient implementation of basic computation skeletons (in C++ and MPI) on distributed PC clusters guarantees that programs composed by these parallel skeletons can be efficiently executed in parallel.

Currently, we have not succeeded in developing efficient and correct parallel programs with parallel skeletons from their recursive specifications such as nested recursive functions on nested lists and divide-and-conquer algorithm on quad-trees. Thus, it is our future work to construct more

powerful theories for a systematic programming methodology that supports such derivations, and to introduce control skeletons such as divide-and-conquer skeleton if needed. It is also our future work to study an automatic optimization mechanism. The mechanism should be able to eliminate inefficiency due to compositional or nested uses of parallel skeletons in parallel programs. Moreover, it should be able to introduce low-level parallelism that currently our skeletons cannot describe. Some of the optimizations will be performed based on the properties of operators in the skeletons. Finally, we mention about higher-dimensional arrays. To extend our framework to higher-dimensional arrays, we introduce constructors for new dimensions and the condition that any pair of the constructors satisfies the abide-property. Then, definitions of parallel skeletons and programming methodology for higher-dimensional arrays are almost the same as two-dimensional.

ACKNOWLEDGMENTS

The authors would like to thank Kiminori Matsuzaki for valuable discussions with him, and the reviewers for their useful comments. This work was partially supported by Japan Society for the Promotion of Science, Grant-in-Aid for Scientific Research (B) 17300005.

REFERENCES

1. E. Elmroth, F. Gustavson, I. Jonsson, and B. Kagstroom, Recursive Blocked Algorithms and Hybrid Data Structures for Dense Matrix Library Software, *SIAM Review*, **46**(1):3–45 (2004).
2. G. Hains, Programming with Array Structures, in A. Kent and J. G. Williams (eds.), *Encyclopedia of Computer Science and Technology*, Vol. 14. M. Dekker inc, New-York, pp. 105–119 (1994). Appears also in *Encyclopedia of Microcomputers*.
3. L. Mullin, (ed.), *Arrays, Functional Languages, and Parallel Systems*. Kluwer Academic Publishers (1991).
4. A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*, 2 ed., Addison-Wesley, (2003).
5. J. Reif and J. H. Reif (eds.), *Synthesis of Parallel Algorithms*. Morgan Kaufmann (1993).
6. G. H. Golub and C. F. V. Loan, *Matrix Computations*, (3rd ed.), Johns Hopkins University Press (1996).
7. G. W. Stewart, *Matrix Algorithms*. Society for Industrial and Applied Mathematics (2001).
8. J. J. Dongarra, L. S. Blackford, J. Choi, A. Cleary, E. D'Azeuedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK User's Guide*, Society for Industrial and Applied Mathematics (1997).
9. P. Alpatov, G. Baker, C. Edwards, J. Gunnels, G. Morrow, J. Overfelt, R. van de Geijn, and Y. J. Wu, PLAPACK: Parallel Linear Algebra Package, in *Proceedings of the SIAM Parallel Processing Conference* (1997).

10. I. Jonsson and B. Kagstroom, RECSY – A High Performance Library for Sylvester-Type Matrix Equations, in *Proceedings of 9th International Euro-Par Conference (Euro-Par'03)*, Vol. 2790 of Lecture Notes in Computer Science, pp. 810–819 (2003).
11. M. Cole, *Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computation*, Pitman, London: Research Monographs in Parallel and Distributed Computing (1989).
12. M. Cole, eSkel Home Page, 2002 **http://homepages.inf.ed.ac.uk/mic/eSkel/.**
13. F. A. Rabhi, and S. Gorlatch (eds.), *Patterns and Skeletons for Parallel and Distributed Computing*. Springer-Verlag (2002).
14. Z. Hu, H. Iwasaki, and M. Takeichi, An Accumulative Parallel Skeleton for All, in *Proceedings of 11st European Symposium on Programming (ESOP 2002), LNCS 2305*, pp. 83–97 (2002).
15. Z. Hu, M. Takeichi, and H. Iwasaki, Diffusion: Calculating Efficient Parallel Programs, in *Proceedings of 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99)*, pp. 85–94 (1999).
16. W. N. Chin, A. Takano, and Z. Hu, Parallelization via Context Preservation, in *Proceedings of IEEE Computer Society International Conference on Computer Languages (ICCL'98)*, pp. 153–162 (1998).
17. S. Gorlatch, Systematic Efficient Parallelization of Scan and Other List Homomorphisms, in Proceedings of 2nd International Euro-Par Conference (Euro-Par'96), Vol. 1124 of Lecture Notes in Computer Science, pp. 401–408 (1996).
18. Z. Hu, H. Iwasaki, and M. Takeichi, Formal Derivation of Efficient Parallel Programs by Construction of List Homomorphisms, *ACM Transactions on Programming Langauges and Systems* **19**(3):444–461 (1997).
19. K. Matsuzaki, K. Kakehi, H. Iwasaki, Z. Hu, and Y. Akashi, A Fusion-Embedded Skeleton Library, in *Proceedings of 10th International Euro-Par Conference (Euro-Par'04)*, Vol. 3149 of Lecture Notes in Computer Science, pp. 644–653 (2004).
20. J. Gibbons, W. Cai, and D. B. Skillicorn, Efficient Parallel Algorithms for Tree Accumulations, *Science of Computer Programming*, **23**(1):1–18 (1994).
21. D. B. Skillicorn, Parallel Implementation of Tree Skeletons, *Journal of Parallel and Distributed Computing*, **39**(2):115–125 (1996).
22. R. Miller, Two Approaches to Architecture-Independent Parallel Computation. Ph.D. thesis, Computing Laboratory, Oxford University (1994).
23. R. S. Bird, Lectures on Constructive Functional Programming. Technical Report Technical Monograph PRG-69, Oxford University Computing Laboratory (1988).
24. R. S. Bird, and O. de Moor, *Algebras of Programming*, Prentice Hall (1996).
25. D. B. Skillicorn, *Foundations of Parallel Programming*, Cambridge University Press (1994).
26. J. Jeuring, Theories for Algorithm Calculation. Ph.D. thesis, Utrecht University. Parts of the thesis appeared in the Lecture Notes of the STOP 1992 Summerschool on Constructive Algorithmics (1993).
27. D. S. Wise, Representing Matrices as Quadtrees for Parallel Processors, *Information Processing Letters*, **20**(4):195–199 (1984).
28. J. D. Frens, and D. S. Wise, QR Factorization with Morton-Ordered Quadtree Matrices for Memory Re-use and Parallelism, in *Proceedings of 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'03)*, pp. 144–154 (2003).
29. D. S. Wise, Undulant Block Elimination and Integer-Preserving Matrix Inversion, *Science of Computer Programming*, **22**(1):29–85 (1999).

30. J. Bentley, Programming Pearls: Algorithm Design Techniques, *Communications of the ACM*, **27**(9):865–873 (1984a).
31. J. Bentley, Programming Pearls: Perspective on Performance, *Communications of the ACM*, **27**(11):1087–1092 (1984b).
32. T. Takaoka, Efficient Algorithms for the Maximum Subarray Problem by Distance Matrix Multiplication, in *Proceedings of Computing: The Australasian Theory Symposium (CATS'02)*, pp. 189–198 (2002).
33. K. Emoto, Z. Hu, K. Kakehi, and M. Takeichi, A Compositional Framework for Developing Parallel Programs on Two Dimensional Arrays, Technical Report METR2005-09, Department of Mathematical Informatics, University of Tokyo (2005).
34. R. S. Bird, *Introduction to Functional Programming using Haskell*, Prentice Hall (1998).
35. M. Cole, Parallel Programming with List Homomorphisms, *Parallel Processing Letters*, **5**(2):191–203 (1995).
36. K. Matsuzaki, K. Emoto, H. Iwasaki, and Z. Hu, A Library of Constructive Skeletons for Sequential Style of Parallel Parogramming (Invited Paper), in *Proceedings of the 1st International Conference on Scalable Information Systems (INFOSCALE 2006)*, Vol. 152 of ACM International Conference Proceeding Series, p. 13 (2006).
37. G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, B. B. Fraguela, M. J. Garzaran, D. Padua, and C. von Praun, Programming for Parallelism and Locality with Hierarchically Tiled Arrays, in *Proceedings of 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'06)*. New York, NY, USA, pp. 48–57 (2006).
38. R. S. Bird, An Introduction to the Theory of Lists, in M. Broy (ed.), *Logic of Programming and Calculi of Discrete Design*, Vol. 36 of NATO ASI Series F. pp. 5–42 (1987).
39. D. B. Skillicorn, The Bird-Meertens Formalism as a Parallel Model, in *NATO ARW "Software for Parallel Computation"* (1992).
40. Z. Hu, M. Takeichi, and W. N. Chin, Parallelization in Calculational Forms, in *Proceedings of 25th ACM Symposium on Principles of Programming Languages*. San Diego, California, USA, pp. 316–328 (1998).
41. D. N. Xu, S.-C. Khoo, and Z. Hu, PType System: A Featherweight Parallelizability Detector, in *Proceedings of Second Asian Symposium on Programming Languages and Systems (APLAS'04)*, Vol. 3302 of Lecture Notes in Computer Science. pp. 197–212 (2004).
42. R. Bernecky, The Role of APL and J in High-performance Computation, *APL Ouote Ouad* **24**(1):17–32 (1993).
43. A. D. Falkoff, and K. E. Iverson, The design of APL, *IBM Journal of Research and Development* **17**(4):324–334 (1973).
44. K. E. Iverson, *A Programming Language*. John Wiley and Sons (1962).