MATHEMATICAL ENGINEERING
TECHNICAL REPORTS

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

Kento EMOTO, Zhenjiang HU,
Kazuhiko KAKEHI and Masato TAKEICHI

METR 2005–09 April 2005

DEPARTMENT OF MATHEMATICAL INFORMATICS
GRADUATE SCHOOL OF INFORMATION SCIENCE AND TECHNOLOGY
THE UNIVERSITY OF TOKYO
BUNKYO-KU, TOKYO 113-8656, JAPAN

WWW page: http://www.i.u-tokyo.ac.jp/mi/mi-e.htm

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

Kento EMOTO, Zhenjiang HU, Kazuhiko KAKEHI and Masato TAKEICHI

Department of Mathematical Informatics
Graduate School of Information Science and Technology
The University of Tokyo
{emoto,hu,kaz,takeichi}@ipl.t.u-tokyo.ac.jp

April 12th, 2005

Abstract

Computations on two-dimensional arrays such as matrix computations are one of the most fundamental and ubiquitous 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 which supports users, even with little knowledge about parallel machines, to systematically develop both correct and efficient parallel programs on two-dimensional arrays. The key feature of our framework is a novel use of the abide-tree representation of two-dimensional arrays, which not only inherits the advantages of tree representations of matrix 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.

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 [11, 26, 17]. And developing efficient parallel algorithms for these computations is one of the most important topics in many textbooks on parallel programming [16, 28]. Many algorithms have been designed and implemented as standard libraries. For example, for matrix computations [14, 32], we have the useful libraries like ScaLAPACK[10], PLAPACK [1] and RECSY [23]. Though being 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. 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 libraries provide. The libraries are no direct help for the users in this case, and they have to rewrite or develop the libraries for themselves to serve their purpose. However, since (re-)building parallel libraries is not an easy task, much more involved than sequential algorithm due to necessity of taking the synchronization and communication between processors into consideration, not everyone can do it easily.
- Second, the libraries are not well structured, and thus hard to be efficiently composed together. 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 [7, 27, 9]. In this model, users can build parallel programs by composing ready-made components (called *skeletons*) that are 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 unconscious of parallelism explicitly, (2) since the skeletons are designed for structured programming, they can be efficiently composed to solve various problems.

There is much research devoted to parallel skeletons on lists, which is a one-dimensional data structure, and it has been shown [21, 19] that parallel programming with list skeletons is very powerful since we can describe many problems in terms of a few skeletons. Moreover many researches have been done on methods of deriving and optimizing parallel programs by means of parallel skeletons on lists [15, 6, 18], and especially about optimization, and there is a library which can automatically optimize a program described by skeletons [24]. Similarly, for parallel skeletons on the tree data structure there is research on binary trees [31, 13], general trees and derivation of programs on these tree skeletons. Unfortunately, it has proved to be a challenge [25] 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 make use of the theory of constructive algorithmics (also known as *Bird-Meertens Formalism*) [4, 30, 2], a successful theory for compositional sequential program development, where 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) [30, 22] impose much restriction on the access order of elements. Wise et al. represent a two-dimensional array by a quadtree [33] and show that recursive algorithms on quadtree provide better performance than existing algorithms for some matrix computations (QR factorization [12], LU factorization [34]). More examples can be found in [11]. 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 [4] represents two-dimensional arrays by dynamic trees allowing restructuring trees when necessary.

In this paper, we propose a compositional framework which allows users, even with little knowledge about parallel machines, to easily describe safe and efficient parallel computation over two-dimensional arrays, and enables discussion of methods of 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* [4] in developing *parallel* programs for manipulating two-dimensional arrays, whose importance has not been well 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 [11, 12, 34], but also supports systematic development of parallel programs and architecture independent implementation owing to its solid theoretical foundation - the theory of constructive algorithmics [4, 2, 30].

- 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 (Section 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 [12], and a successful derivation of an involved efficient parallel program for the maximum rectangle sum problem [18].

- 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. So far most research focuses on showing that the recursive tree representation of matrices is suitable for parallel computation on shared memory systems [12, 11], 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 our framework imposes no restriction on the size and the element order of two-dimensional arrays and provides an additional 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 abide tree. In Section 3, we give some examples of parallel algorithms on the abide tree. In Section 4, we demonstrate development of parallel programs on two-dimensional arrays. In Section 5, we give efficient implementations and show their experiments. In Section 6, we remark on the related work and finally in Section 7, we make conclusion.

2. Theory of Two-Dimensional Arrays

Before explaining our compositional programming framework, we shall construct a theory of two-dimensional arrays, the basis of our framework, according to the theory of constructive algorithmics [4, 30, 2].

Notation in this paper follows that of Haskell [5], a pure functional language that is able to 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 stronger 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 which 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 which applies functions f and g separately to an element and returns a pair of the results is denoted by $(f \triangle g)$. Thus $(f \triangle g)\,a = (f\,a, g\,a)$. 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 without loss of information, we define the following abide trees, which are built up by three constructors $|\cdot|$ (singleton), $\ominus$ (above) and ϕ (beside) [4].

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{AbideTree}\,\alpha & = & |\cdot|\ \alpha \\
 & | & (\mathit{AbideTree}\,\alpha) \ominus (\mathit{AbideTree}\,\alpha) \\
 & | & (\mathit{AbideTree}\,\alpha) \phi (\mathit{AbideTree}\,\alpha)
\end{array}
$$

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 function the to extract the element from a singleton array, i.e. $\mathsf{the}\ |a| = a$. For two-dimensional arrays x and y which have the same width, $x \ominus y$ means that x is located above y. Similarly, for two-dimensional arrays x and y which have the same height, $x \phi y$ means that x is located on the left of y. Moreover, $\ominus$ and ϕ are associative binary operators and satisfy following *abide* (a coined term from <u>ab</u>ove and bes<u>ide</u>) property.

$$(x \phi u) \ominus (y \phi v) = (x \ominus y) \phi (u \ominus v)$$

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

$$
\begin{array}{c}
(|1| \phi |2|) \ominus (|3| \phi |4|) \\
(|1| \ominus |3|) \phi (|2| \ominus |4|)
\end{array}
$$

This is in sharp contrast to the quadtree representation of matrix [12], which does not allow such freedom.

2.2 Abide Tree Homomorphism

It follows from the theory of constructive algorithmics [2] that each constructively built-up data structure (i.e., algebraic data structure) is equipped with a powerful computation pattern called homomorphism.

Definition 2.1 ((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 $\oplus, \otimes$.

$$
\begin{array}{lcl}
h\ |a| & = & f\ a \\
h\ (x \ominus y) & = & h\ x \oplus h\ y \\
h\ (x \phi y) & = & h\ x \otimes h\ y
\end{array}
$$

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 $|\cdot|$, $\mathbin{-\!\!\!\!\circ}$ and $\mathbin{\phi}$ 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.

It is worth noting that $\oplus$ and $\otimes$ in $(\!| f, \oplus, \otimes |\!)$ should be associative and satisfy the abide property, inheriting the properties of $\mathbin{-\!\!\!\!\circ}$ and $\mathbin{\phi}$.

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 2.1 (Fusion) Let h and $(\!| f, \oplus, \otimes |\!)$ be given. If there exist $\odot$ and $\ominus$ such that for all 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 the induction on the structure of abide trees.

Base case:

$$\begin{array}{cl} & (h \circ (\!| f, \oplus, \otimes |\!))\ |a| \\ = & \quad \{ \text{ Definition of } (\!| f, \oplus, \otimes |\!) \ \} \\ & h\ (f\ a) \\ = & \quad \{ \text{ Definition of } (\!| h \circ f, \odot, \ominus |\!) \ \} \\ & (\!| h \circ f, \odot, \ominus |\!)\ |a| \end{array}$$

Induction for $\mathbin{-\!\!\!\!\circ}$:

$$\begin{array}{cl} & (h \circ (\!| f, \oplus, \otimes |\!))\ (x \mathbin{-\!\!\!\!\circ} y) \\ = & \quad \{ \text{ Definition of } (\!| f, \oplus, \otimes |\!) \ \} \\ & h\ ((\!| f, \oplus, \otimes |\!)\ x \oplus (\!| f, \oplus, \otimes |\!)\ y) \\ = & \quad \{ \text{ Definition of } h \ \} \\ & h\ ((\!| f, \oplus, \otimes |\!)\ x) \odot h\ ((\!| f, \oplus, \otimes |\!)\ y) \\ = & \quad \{ \text{ Hypothesis of induction } \} \\ & (\!| h \circ f, \odot, \ominus |\!)\ x \odot (\!| h \circ f, \odot, \ominus |\!)\ y \\ = & \quad \{ \text{ Definition of } (\!| h \circ f, \odot, \ominus |\!) \ \} \\ & (\!| h \circ f, \odot, \ominus |\!)\ (x \mathbin{-\!\!\!\!\circ} y) \end{array}$$

Induction for $\mathbin{\phi}$ is proved similarly. $\square$

A homomorphism $(\!| f, \oplus, \otimes |\!)$ can be implemented *efficiently* in parallel, which will be shown in Section 5. Let N be the number of elements in a two-dimensional array, $T_f, T_\oplus, T_\otimes$ be the parallel time cost for computing f, $\oplus$ and $\otimes$ respectively. Then, $(\!| f, \oplus, \otimes |\!)$ takes parallel time of $T_f \times \mathrm{O}(\log N) \times max(T_\oplus, T_\otimes)$ with enough number of processors.

2.3 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*, as discussed in [8], 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. Let k be a nonhomomorphic function, and consider a new function g such that $g\ x = (k\ x, x)$. The tupled function g is a homomorphism.

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

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 Section 4.

3. Parallel Algorithms on Two-Dimensional Arrays

Homomorphisms are suitable for parallel implementation, which has been argued in the previous section and will be detailed in Section 5. In this section, we show that homomorphisms are powerful enough to describe many useful parallel algorithms for manipulating two-dimensional arrays. We will start by demonstrating that basic parallel computation components, namely basic data parallel skeletons and basic communication skeletons, can be specified by either homomorphisms or recursions on the abide trees, and then we show that composition of these basic parallel skeletons is powerful enough to solve nontrivial problems such as matrix multiplication and QR decomposition.

3.1 Data Parallel Skeletons

We define four primitive functions map, reduce, zipwith and scan on the data type $\mathit{AbideTree}$. In the theory of Constructive Algorithmics [4, 30, 2], 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 *data parallel skeletons* because they have potential parallelism and can be implemented efficiently in parallel (see Section 5.)

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

$$
\begin{array}{lcl}
\mathsf{map}\, f\, |a| & = & |f\, a| \\
\mathsf{map}\, f\, (x \ominus y) & = & (\mathsf{map}\, f\, x) \ominus (\mathsf{map}\, f\, y) \\
\mathsf{map}\, f\, (x \phi\ y) & = & (\mathsf{map}\, f\, x) \phi\ (\mathsf{map}\, f\, y) \ ,
\end{array}
$$

that is, $\mathsf{map}\, f = (\!| |\cdot| \circ f, \ominus, \phi |\!)$.

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

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

that is, $\mathsf{reduce}(\oplus, \otimes) = (\!| id, \oplus, \otimes |\!)$.

Interestingly, any homomorphism $(\!|f, \oplus, \otimes|\!)$ can be written as a composition of map and reduce, i.e.

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

which implies that if we have efficient parallel implementations for reduce and map, we get an efficient implementation for homomorphism.

Zipwith

The two skeletons defined above are primitive skeletons. We define other skeletons which are extensions of these primitive skeletons. The skeleton $\mathsf{zipwith}$, 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.

$$\begin{array}{lcl}
\mathsf{zipwith}\, f\, |a|\, |b| & = & |f\, a\, b| \\
\mathsf{zipwith}\, f\, (x \ominus y)\, (u \ominus v) & = & (\mathsf{zipwith}\, f\, x\, u) \ominus (\mathsf{zipwith}\, f\, y\, v) \\
\mathsf{zipwith}\, f\, (x \phi\, y)\, (u \phi\, v) & = & (\mathsf{zipwith}\, f\, x\, u) \phi\, (\mathsf{zipwith}\, f\, y\, v)
\end{array}$$

Note that in the above definition two-dimensional arrays which are the arguments of the function should be divided in the way that the sizes of x and u are the same. Function zip is a specialization of $\mathsf{zipwith}$, 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 $\mathsf{zipwith}$ for the case when the number of input arrays is three or more, and those which take k arrays are denoted by zip_k and $\mathsf{zipwith}_k$. Also we define unzip to be the inverse of zip.

With these three skeletons defined above, we are able to describe many useful functions.

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

Note that $||\cdot||$ is abbreviation of $|\cdot| \circ |\cdot|$; *id* is the identity function of *AbideTree*; *tr* is the matrix-transposing function; *rev* takes a two-dimensional array and returns the array reversed in the vertical and the horizontal direction; *flatten* flattens a nested *AbideTree*; *height* and *width* return the number of rows and columns respectively, *cols* and *rows* return an array of which elements are columns and rows of the array of the argument respectively; reduce_c and reduce_r which 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); map_c and map_r which 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); and *add* and *sub* denote matrix addition and subtraction respectively.

Scan

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

$$\begin{array}{lcl}
\mathsf{scan}(\oplus,\otimes)\,|a| & = & |a| \\
\mathsf{scan}(\oplus,\otimes)(x \ominus y) & = & (\mathsf{scan}(\oplus,\otimes)\,x) \oplus' (\mathsf{scan}(\oplus,\otimes)\,y) \\
\mathsf{scan}(\oplus,\otimes)(x \obar y) & = & (\mathsf{scan}(\oplus,\otimes)\,x) \otimes' (\mathsf{scan}(\oplus,\otimes)\,y)
\end{array}$$

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

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

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

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

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

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

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

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

Using the skeleton scan, we can define scanr which executes scan reversely, allred_r and allred_c which broadcast the results in each row and column after reduce_r and reduce_c respectively. These functions are used in later section.

$$\begin{array}{lcl}
\mathsf{scanr}(\oplus,\otimes) = \mathit{rev} \circ \mathsf{scan}(\tilde{\oplus}, \tilde{\otimes}) \circ \mathit{rev} \\
\mathit{allred}_c(\oplus) & = & \mathsf{scanr}(\gg, \ll) \circ \mathsf{scan}(\oplus, \gg) \\
\mathit{allred}_r(\otimes) & = & \mathsf{scanr}(\ll, \gg) \circ \mathsf{scan}(\gg, \otimes)
\end{array}$$

3.2 Data Communication Skeletons

We show how to define data communication skeletons dist, gather, rot_r and rot_c which 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

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

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

$$\begin{array}{lcll}
\mathit{grp}_r\,k\,(x \ominus y) & = & |x| \ominus (\mathit{grp}_r\,k\,y) & \text{if } \mathit{height}\,x = k \\
\mathit{grp}_r\,k\,x & = & |x| & \text{if } \mathit{height}\,x < k
\end{array}$$

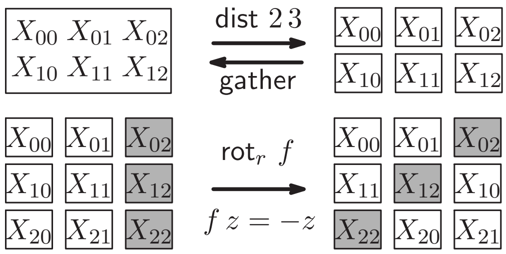

Figure 1: An image of communication skeletons (each rectangle corresponds to each processor; X_{ij} represents a subarray.)

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.

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

Although definitions of these skeletons may seem complicated, actual operations are rather simple as illustrated in Figure 1. 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 which takes a function f and rotates i-th row (the index begins from 0) by $f\ i$, is defined as follows:

$$\begin{aligned}
\mathsf{rot}_r\, f \ &= \ \mathit{flatten} \circ \mathsf{map}\, \mathit{shift}_r \circ \mathit{addidx}_r \circ \mathit{rows} \\
&\mathbf{where} \\
&\qquad \mathit{addidx}_r\, u \ = \ \mathsf{zip}(\mathsf{map}\, f\ (\mathit{idx}_r\, u), u) \\
&\qquad \mathit{idx}_r = \mathsf{map}(-1) \circ \mathsf{scan}_{\downarrow}(+) \circ \mathsf{map}(\lambda\, x.\, 1) \quad ;
\end{aligned}$$

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

$$\begin{aligned}
\mathit{shift}_r\ (0, x) \quad &= \ x \\
\mathit{shift}_r\ (i, x \phi y) \quad &= \ y \phi x \quad \text{if } \mathit{width}\, y = i \\
\mathit{shift}_r\ (-i, x \phi y) &= \ y \phi x \quad \text{if } \mathit{width}\, x = i
\end{aligned}$$

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

3.3 Matrix Multiplication

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

The first description is of Cannon's Algorithm [16]:

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

where p is a natural number indicating the number of division of matrices in column and row direction, and lmm is a function which 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.

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

Explicit distribution of matrices by 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. This description describes just a definition of matrix multiplication. Although users do not need to take parallelism into consideration at all, this program can be executed in parallel due to parallelism of each skeleton.

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

3.4 QR Factorization

As the final nontrivial example, we show descriptions of two parallel algorithms for QR factorization [11]. We will not explain the details, rather we hope to show that these algorithms can be dealt with in our framework.

We give the recursive description of 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 $m\times n$.

$$
\begin{array}{lcl}
\multicolumn{3}{l}{qr\ \ ((A_{11}\ominus A_{21})\ \phi\ (A_{12}\ominus A_{22}))} \\
 & = & \textbf{let}\ (Q_1, R_{11}\ominus 0) = qr\ \ (A_{11}\ominus A_{21}) \\
 & & \qquad (R_{12}\ominus \hat{A_{22}}) = mm\ \ (trQ_1)\ \ (A_{12}\ominus A_{22}) \\
 & & \qquad (\hat{Q_2}, R_{22}) = qr\ \ \hat{A_{22}} \\
 & & \qquad Q = mm\ Q_1\ \ ((I\ \phi\ 0)\ominus(0\ \phi\ \hat{Q_2})) \\
 & & \textbf{in}\ (Q, (R_{11}\ \phi\ R_{12})\ominus(0\ \phi\ R_{22})) \\
qr\ \ (|a|\ominus x) & = & hh\ \ (|a|\ominus x) \\
hh\ \ v & = & \textbf{let}\ v' = add\ \ v\ \ e \\
 & & \qquad a = \sqrt{\mathsf{reduce}(+,+)\ \ (\mathsf{zipwith}(\times)\ \ v'\ \ v')} \\
 & & \qquad u = \mathsf{map}\ \ (/a)\ \ v' \\
 & & \qquad Q = sub\ \ I\ \ (\mathsf{map}\ \ (\times 2)\ \ (mm\ \ u\ \ (tr\ \ u))) \\
 & & \textbf{in}\ (Q, e)
\end{array}
$$

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 [12]; 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 which satisfy $A = QR$ where A is a matrix of $n\times n$ ($n = 2^k$ for a natural number k), Q an orthogonal matrix of $n\times n$ and R an upper-triangular matrix of $n \times n$.

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

Note that A_{ij} $(i, j \in \{1, 2\})$ have the same shape.

$$
\begin{array}{lll}
e\ (N, 0) & = & (I, N) \\
e\ (|n|, |s|) & = & \textbf{let}\ \ Q = g(n, s) \\
& & \qquad (N, 0) = mm(tr\ Q)\ (|n| \ominus |s|) \\
& & \textbf{in}\ (Q, N) \\
\multicolumn{3}{l}{e\ ((N_{11} \ominus N_{21}) \phi (N_{12} \ominus N_{22}), (S_{11} \ominus S_{21}) \phi (S_{12} \ominus S_{22}))} \\
\multicolumn{3}{l}{\qquad = \textbf{let}} \\
& & ((Q_1^{11} \ominus Q_1^{21}) \phi (Q_1^{12} \ominus Q_1^{22}), N_1) = e\ (N_{11}, S_{11}) \\
& & ((Q_2^{11} \ominus Q_2^{21}) \phi (Q_2^{12} \ominus Q_2^{22}), N_2) = e\ (N_{22}, S_{22}) \\
& & Q_{12} = (Q_1^{11} \phi 0 \phi Q_1^{12} \phi 0) \ominus (0 \ominus Q_2^{11} \phi 0 \phi Q_2^{12}) \\
& & \qquad \ominus (Q_1^{21} \phi 0 \phi Q_1^{22} \phi 0) \ominus (0 \ominus Q_2^{21} \phi 0 \phi Q_2^{22}) \\
& & Q_1 = (Q_1^{11} \ominus Q_1^{21}) \phi (Q_1^{12} \ominus Q_1^{22}) \\
& & (U_n \ominus U_s) = mm\ (tr\ Q_1)\ (N_{12} \ominus S_{12}) \\
& & (Q_4, R_4) = qr_q\ U_s \\
& & Q_4' = (I \phi 0 \phi 0 \phi 0) \ominus (0 \phi I \phi 0 \phi 0) \ominus (0 \phi 0 \phi Q_4 \phi 0) \ominus (0 \phi 0 \phi 0 \phi I) \\
& & Q_5 = mm\ Q_{12}\ Q4' \\
& & ((Q_6^{11} \ominus Q_6^{21}) \phi (Q_6^{12} \ominus Q_6^{22}), N_6) = e\ (N_2, R_4) \\
& & Q_6' = (I \phi 0 \phi 0 \phi 0) \ominus (0 \phi Q_6^{11} \phi Q_6^{12} \phi 0) \ominus (0 \phi Q_6^{21} \phi Q_6^{22} \phi 0) \ominus (0 \phi 0 \phi 0 \phi I) \\
& \multicolumn{2}{l}{\textbf{in}\ (mm\ Q_5\ Q_6', (N_1 \phi U_n) \ominus (0 \phi N_6))} \\
g\ (a, b) & = & (|c| \phi |s|) \ominus (|-s| \phi |c|) \\
& & \textbf{where}\ \ c = \dfrac{a}{\sqrt{a^2 + b^2}},\ \ s = \dfrac{-b}{\sqrt{a^2 + b^2}}
\end{array}
$$

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.

Like in [12], we can efficiently parallelize some parts of these complicated recursive functions, such as matrix multiplication in the recursion. It is, however, still an open problem whether the complicated recursive functions can be parallelized, which is one of our future work.

4. DEVELOPING EFFICIENT PARALLEL PROGRAMS

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

Our strategy for deriving efficient parallel programs on two-dimensional arrays consists of the following four steps, extending the result of [18].

Step 1. Define the target program p as a composition of $p_1, \ldots, p_n$ which are already defined, i.e. $p = p_n \circ \cdots \circ p_1$. Each of $p_1, \ldots, p_n$ may be defined as a composition of small functions or a recursive function (see Section 3.3 and Section 3.4).

Step 2. Derive an almost homomorphism (Section 2.3) 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 $p_n \circ \cdots \circ p_1$ 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. 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 how you solve it.

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 which 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 (k, l)-element of (i, j)-element of the resulting array is a sub-rectangle having rows from i-th to j-th and columns from k-th to l-th 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. In this case, *NIL* may be an array of which element is $-\infty$ or an array of it.

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

The function *rects* is mutual recursively defined as follows:

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

where '_' indicates "don't care" and generalized matrix multiplication *gemm* is defined as follows:

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

Functions *bottoms*, *tops*, *rights* and *lefts* are similarly defined as mutual recursive functions with some extra functions:

$$
\begin{array}{lcl}
\mathit{tops}\ |a| & = & |||a||| \\
\mathit{tops}\ (x \ominus y) & = & \mathit{tops}\ x \phi\ \mathsf{map}\ (\mathsf{zipwith}(\ominus)\ (\mathit{cols}'\ x))\ (\mathit{tops}\ y) \\
\mathit{tops}\ (x \phi\ y) & = & \mathsf{zipwith}_4\ f_t\ (\mathit{tops}\ x)\ (\mathit{tops}\ y)\ (\mathit{toprights}\ x)\ (\mathit{toplefts}\ y) \\
& & \quad \textbf{where}\ f_t\ t_1\ t_2\ tr_1\ tl_2 = (t_1 \phi\ \mathit{gemm}\ (\_, \phi)\ tr_1\ tl_2) \ominus (\mathit{NIL} \phi\ t_2)
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{bottoms}\ |a| & = & |||a||| \\
\mathit{bottoms}\ (x \ominus y) & = & \mathsf{map}\ (\lambda z \rightarrow \mathsf{zipwith}(\ominus)\ z\ (\mathit{cols}'\ y))\ (\mathit{bottoms}\ x) \ominus \mathit{bottoms}\ y \\
\mathit{bottoms}\ (x \phi\ y) & = & \mathsf{zipwith}_4\ f_b\ (\mathit{bottoms}\ x)\ (\mathit{bottoms}\ y)\ (\mathit{bottomrights}\ x)\ (\mathit{bottomlefts}\ y) \\
& & \quad \textbf{where}\ f_b\ b_1\ b_2\ br_1\ bl_2 = (b_1 \phi\ \mathit{gemm}\ (\_, \phi)\ br_1\ bl_2) \ominus (\mathit{NIL} \phi\ b_2)
\end{array}
$$

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

$$
\begin{array}{lcl}
\mathit{lefts}\ |a| & = & |||a||| \\
\mathit{lefts}\ (x \ominus y) & = & (\mathit{lefts}\ x \phi\ \mathit{gemm}\ (\_, \mathsf{zipwith}(\ominus))\ (\mathit{bottomlefts}\ x)\ (\mathit{toplefts}\ y)) \ominus (\mathit{NIL} \phi\ \mathit{lefts}\ y) \\
\mathit{lefts}\ (x \phi\ y) & = & \mathsf{zipwith}_3\ f_l\ (\mathit{lefts}\ x)\ (\mathit{lefts}\ y)\ (\mathit{rows}'\ x) \\
& & \quad \textbf{where}\ f_l\ l_1\ l_2\ ro_1 = l_1 \phi\ \mathsf{map}\ (ro_1 \phi)\ l_2
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{toprights}\ |a| & = & |||a||| \\
\mathit{toprights}\ (x \ominus y) & = & \mathit{toprights}\ x \phi\ \mathsf{map}\ (\mathsf{zipwith}(\ominus)\ (\mathit{right}'\ (\mathit{toprights}\ x)))\ (\mathit{toprights}\ y) \\
\mathit{toprights}\ (x \phi\ y) & = & \mathsf{zipwith}\ f_{tr}\ (\mathit{toprights}\ x)\ (\mathit{toprights}\ y) \\
& & \quad \textbf{where}\ f_{tr}\ tr_1\ tr_2 = \mathsf{map}\ (\phi\, \mathit{top}'\ tr_2 \phi)\ tr_1 \ominus tr_2
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{bottomrights}\ |a| & = & |||a||| \\
\mathit{bottomrights}\ (x \ominus y) & = & \mathsf{map}\ (\lambda z \rightarrow \mathsf{zipwith}(\ominus)\ z\ (\mathit{top}'\ (\mathit{bottomrights}\ y)))\ (\mathit{bottomrights}\ x) \\
& & \qquad \ominus \mathit{bottomrights}\ y \\
\mathit{bottomrights}\ (x \phi\ y) & = & \mathsf{zipwith}\ f_{br}\ (\mathit{bottomrights}\ x)\ (\mathit{bottomrights}\ y) \\
& & \quad \textbf{where}\ f_{br}\ br_1\ br_2 = \mathsf{map}\ (\phi\, \mathit{top}'\ br_2 \phi)\ br_1 \ominus br_2
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{toplefts}\ |a| & = & |||a||| \\
\mathit{toplefts}\ (x \ominus y) & = & \mathit{toplefts}\ x \varphi\ \mathsf{map}\ (\mathsf{zipwith}(\ominus)\ (\mathit{right}'\ (\mathit{toplefts}\ x)))\ (\mathit{toplefts}\ y) \\
\mathit{toplefts}\ (x \varphi y) & = & \mathsf{zipwith}\ f_{tl}\ (\mathit{toplefts}\ x)\ (\mathit{toplefts}\ y) \\
 & & \mathbf{where}\ f_{tl}\ tl_1\ tl_2 = tl_1 \varphi\ \mathsf{map}\ (\mathit{right}'\ tl_1 \varphi)\ tl_2
\end{array}
$$

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

$$
\begin{array}{lcl}
\mathit{cols}'\ |a| & = & ||a|| \\
\mathit{cols}'\ (x \ominus y) & = & \mathsf{zipwith}(\ominus)\ (\mathit{cols}'\ x)\ (\mathit{cols}'\ y) \\
\mathit{cols}'\ (x \varphi y) & = & (\mathit{cols}'\ x \varphi\ \mathit{gemm}\ (\_, \varphi)\ (\mathit{right}\ (\mathit{cols}'\ x))\ (\mathit{top}\ (\mathit{cols}'\ y))) \ominus (\mathit{NIL} \varphi\ \mathit{cols}'\ y)
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{rows}'\ |a| & = & ||a|| \\
\mathit{rows}'\ (x \ominus y) & = & (\mathit{rows}'\ x \varphi\ \mathit{gemm}\ (\_, \ominus)\ (\mathit{right}\ (\mathit{rows}'\ x))\ (\mathit{top}\ (\mathit{rows}'\ y))) \ominus (\mathit{NIL} \varphi\ \mathit{rows}'\ y) \\
\mathit{rows}'\ (x \varphi y) & = & \mathsf{zipwith}(\varphi)\ (\mathit{rows}'\ x)\ (\mathit{rows}'\ y)
\end{array}
$$

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

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

Examples of these functions are listed in Appendix A.

Step 2. Deriving Almost Homomorphism

First of all, we propose a way of deriving almost homomorphism from mutual recursive definitions. For notational convenience, we define

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

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

$$
\left\{
\begin{array}{lcl}
h_i\ |a| & = & f_i\ a \\
h_i\ (x \ominus y) & = & ((\triangle_1^n h_i)\ x) \oplus_i ((\triangle_1^n h_i)\ y) \\
h_i\ (x \varphi y) & = & ((\triangle_1^n h_i)\ x) \otimes_i ((\triangle_1^n h_i)\ y)
\end{array}
\right. \tag{3}
$$

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. According to the definition of array homomorphisms, it is sufficient to prove that

$$
\begin{array}{lcl}
(\triangle_1^n h_i)\ |a| & = & (\triangle_1^n f_i)\ a \\
(\triangle_1^n h_i)\ (x \ominus y) & = & ((\triangle_1^n h_i)\ x)\ (\triangle_1^n \oplus_i)\ ((\triangle_1^n h_i)\ y) \\
(\triangle_1^n h_i)\ (x \varphi y) & = & ((\triangle_1^n h_i)\ x)\ (\triangle_1^n \otimes_i)\ ((\triangle_1^n h_i)\ y)\ .
\end{array}
$$

The first equation is proved by the following calculation.

$$
\begin{array}{ll}
 & (\triangle_1^n h_i)\ |a| \\
= & \quad \{ \text{ Definition of } \triangle \ \} \\
 & (h_1\ |a|, \ldots, h_n\ |a|) \\
= & \quad \{ \text{ Definition of } h_i \ \} \\
 & (f_1\ a, \ldots, f_n\ a) \\
= & \quad \{ \text{ Definition of } \triangle \ \} \\
 & (\triangle_1^n f_i)\ a
\end{array}
$$

The second is proved as follows.

$$
\begin{array}{ll}
 & (\triangle_1^n h_i)\ (x \ominus y) \\
= & \quad \{ \text{ Definition of } \triangle \ \} \\
 & (h_1\ (x \ominus y), \ldots, h_n\ (x \ominus y)) \\
= & \quad \{ \text{ Definition of } h_i \ \} \\
 & (((\triangle_1^n h_i)\ x) \oplus_1 ((\triangle_1^n h_i)\ y), \\
 & \qquad \ldots, ((\triangle_1^n h_i)\ x) \oplus_n ((\triangle_1^n h_i)\ y)) \\
= & \quad \{ \text{ Definition of } \triangle \ \} \\
 & ((\triangle_1^n h_i)\ x)\ (\triangle_1^n \oplus_i)\ ((\triangle_1^n h_i)\ y)
\end{array}
$$

The third is proved similarly. $\square$

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*. In fact *rects* is mutually defined with some other functions such as *tops* and *bottoms*, and these functions are in the form of Eq. (3). Thus, letting $h_1 = \mathit{rects}$, $h_2 = \mathit{tops}$, $h_3 = \mathit{bottoms}$, $h_4 = \mathit{rights}$, $h_5 = \mathit{lefts}$, $h_6 = \mathit{toprights}$, $h_7 = \mathit{bottomrights}$, $h_8 = \mathit{toplefts}$, $h_9 = \mathit{bottomlefts}$, $h_{10} = \mathit{cols}'$, $h_{11} = \mathit{rows}'$, we can obtain an almost homomorphism for *rects* by tupling these functions as follows.

$$
\begin{array}{l}
\mathit{rects} = \pi_1 \circ (\triangle_1^{11} h_i) = \pi_1 \circ (\!|\ \triangle_1^{11}\ f_i, \triangle_1^{11} \oplus_i, \triangle_1^{11} \otimes_i \ |\!) \\
\textbf{where} \\
\triangle_1^{11} f_i\ |a| \ = \ (|||a|||, |||a|||, |||a|||, |||a|||, |||a|||, |||a|||, |||a|||, |||a|||, |||a|||, ||a||, ||a||) \\
(s_1, t_1, b_1, r_1, l_1, tr_1, br_1, tl_1, bl_1, c_1, ro_1)\ (\triangle_1^{11} \oplus_i)\ (s_2, t_2, b_2, r_2, l_2, tr_2, br_2, tl_2, bl_2, c_2, ro_2) \\
\quad = (s_0, t_0, b_0, r_0, l_0, tr_0, br_0, tl_0, bl_0, c_0, ro_0) \\
\quad \textbf{where} \\
\qquad \begin{array}{lcl}
s_0 & = & (s_1 \mathbin{\phi} \mathit{gemm}\ (\_, \mathsf{zipwith}(\ominus))\ b_1\ t_2) \ominus (\mathit{NIL} \mathbin{\phi} s_2) \\
t_0 & = & t_1 \mathbin{\phi} \mathsf{map}\ (\mathsf{zipwith}(\ominus)\ c_1)\ t_2 \\
b_0 & = & \mathsf{map}\ (\lambda z \rightarrow \mathsf{zipwith}(\ominus)\ z\ c_2)\ b_1 \ominus b_2 \\
r_0 & = & (r_1 \mathbin{\phi} \mathit{gemm}\ (\_, \mathsf{zipwith}(\ominus))\ br_1\ tr_2) \ominus (\mathit{NIL} \mathbin{\phi} r_2) \\
l_0 & = & (l_1 \mathbin{\phi} \mathit{gemm}\ (\_, \mathsf{zipwith}(\ominus))\ bl_1\ tl_2) \ominus (\mathit{NIL} \mathbin{\phi} l_2) \\
tr_0 & = & tr_1 \mathbin{\phi} \mathsf{map}\ (\mathsf{zipwith}(\ominus)\ (\mathit{right}'\ tr_1))\ tr_2 \\
br_0 & = & \mathsf{map}\ (\lambda z \rightarrow \mathsf{zipwith}(\ominus)\ z\ (\mathit{top}'\ br_2))\ br_1 \ominus br_2 \\
tl_0 & = & tl_1 \mathbin{\phi} \mathsf{map}\ (\mathsf{zipwith}(\ominus)\ (\mathit{right}'\ tl_1))\ tl_2 \\
bl_0 & = & \mathsf{map}\ (\lambda z \rightarrow \mathsf{zipwith}(\ominus)\ z\ (\mathit{top}'\ bl_2))\ bl_1 \ominus bl_2 \\
c_0 & = & \mathsf{zipwith}(\ominus)\ c_1\ c_2 \\
ro_0 & = & (ro_1 \mathbin{\phi} \mathit{gemm}\ (\_, \ominus)\ (\mathit{right}\ ro_1)\ (\mathit{top}\ ro_2)) \ominus (\mathit{NIL} \mathbin{\phi} ro_2)
\end{array}
\end{array}
$$

$$
\begin{array}{l}
(s_1, t_1, b_1, r_1, l_1, tr_1, br_1, tl_1, bl_1, c_1, ro_1)\ (\triangle_1^{11} \otimes_i)\ (s_2, t_2, b_2, r_2, l_2, tr_2, br_2, tl_2, bl_2, c_2, ro_2) \\
\quad = (s_0, t_0, b_0, r_0, l_0, tr_0, br_0, tl_0, bl_0, c_0, ro_0) \\
\quad \mathbf{where} \\
\qquad \begin{array}{lcl}
s_0 & = & \mathsf{zipwith}_4\ f_s\ s_1\ s_2\ r_1\ l_2 \\
 & & \mathbf{where}\ f_s\ s_1\ s_2\ r_1\ l_2 = (s_1 \phi\ gemm\ (\_, \phi)\ r_1\ l_2) \ominus (NIL \phi\ s_2) \\
t_0 & = & \mathsf{zipwith}_4\ f_t\ t_1\ t_2\ tr_1\ tl_2 \\
 & & \mathbf{where}\ f_t\ t_1\ t_2\ tr_1\ tl_2 = (t_1 \phi\ gemm\ (\_, \phi)\ tr_1\ tl_2) \ominus (NIL \phi\ t_2) \\
b_0 & = & \mathsf{zipwith}_4\ f_b\ b_1\ b_2\ br_1\ bl_2 \\
 & & \mathbf{where}\ f_b\ b_1\ b_2\ br_1\ bl_2 = (b_1 \phi\ gemm\ (\_, \phi)\ br_1\ bl_2) \ominus (NIL \phi\ b_2) \\
r_0 & = & \mathsf{zipwith}_3\ f_r\ r_1\ r_2\ ro_2 \\
 & & \mathbf{where}\ f_r\ r_1\ r_2\ ro_2 = \mathsf{map}\ (\phi ro_2)\ r_1 \ominus r_2 \\
l_0 & = & \mathsf{zipwith}_3\ f_l\ l_1\ l_2\ ro_1 \\
 & & \mathbf{where}\ f_l\ l_1\ l_2\ ro_1 = l_1 \phi\ \mathsf{map}\ (ro_1 \phi)\ l_2 \\
tr_0 & = & \mathsf{zipwith}\ f_{tr}\ tr_1\ tr_2 \\
 & & \mathbf{where}\ f_{tr}\ tr_1\ tr_2 = \mathsf{map}\ (\phi\, top'\ tr_2)\ tr_1 \ominus tr_2 \\
br_0 & = & \mathsf{zipwith}\ f_{br}\ br_1\ br_2 \\
 & & \mathbf{where}\ f_{br}\ br_1\ br_2 = \mathsf{map}\ (\phi\, top'\ br_2)\ br_1 \ominus br_2 \\
tl_0 & = & \mathsf{zipwith}\ f_{tl}\ tl_1\ tl_2 \\
 & & \mathbf{where}\ f_{tl}\ tl_1\ tl_2 = tl_1 \phi\ \mathsf{map}\ (right'\ tl_1 \phi)\ tl_2 \\
bl_0 & = & \mathsf{zipwith}\ f_{bl}\ bl_1\ bl_2 \\
 & & \mathbf{where}\ f_{bl}\ bl_1\ bl_2 = bl_1 \phi\ \mathsf{map}\ (right'\ bl_1 \phi)\ bl_2 \\
c_0 & = & (c_1 \phi\ gemm\ (\_, \phi)\ (right\ c_1)\ (top\ c_2)) \ominus (NIL \phi\ c_2) \\
ro_0 & = & \mathsf{zipwith}(\phi)\ ro_1\ ro_2
\end{array}
\end{array}
$$

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 $\forall i$, $\forall x, y$

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

then

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

Proof. The theorem is proven by some calculation and Theorem 2.1.

$$
\begin{array}{cl}
 & h \circ (\pi_1 \circ (\!|\ \triangle_1^n\ f_i, \triangle_1^n \oplus_i, \triangle_1^n \otimes_i\ |\!)) \\
= & \qquad \{ \text{ Definition of } H \text{ and } \pi_1 \ \} \\
 & \pi_1 \circ H \circ (\!|\ \triangle_1^n\ f_i, \triangle_1^n \oplus_i, \triangle_1^n \otimes_i\ |\!) \\
= & \qquad \{ \text{ Theorem 2.1 and the proofs below } \} \\
 & \pi_1 \circ (\!|\ \triangle_1^n\ (h_i \circ f_i), \triangle_1^n \odot_i, \triangle_1^n \ominus_i\ |\!)
\end{array}
$$

To complete the above proof, we need to show

$$
\left\{
\begin{array}{lcl}
H \circ (\triangle_1^n f_i) & = & \triangle_1^n (h_i \circ f_i) \\
H(x\ (\triangle_1^n \oplus_i)\ y) & = & (H\ x)\ (\triangle_1^n \odot_i)\ (H\ y) \\
H(x\ (\triangle_1^n \otimes_i)\ y) & = & (H\ x)\ (\triangle_1^n \ominus_i)\ (H\ y)\ .
\end{array}
\right.
$$

These equations are proved as follows.

$$
\begin{array}{ll}
 & (H \circ (\triangle_1^n f_i))\ a \\
= & \qquad \{ \text{ Definition of } \triangle \text{ and } H \ \} \\
 & ((h_1 \circ f_1)\ a, \ldots, (h_n \circ f_n)\ a) \\
= & \qquad \{ \text{ Definition of } \triangle \ \} \\
 & (\triangle_1^n (h_i \circ f_i))\ a
\end{array}
$$

$$
\begin{array}{ll}
 & H\ (x\ (\triangle_1^n \oplus_i)\ y) \\
= & \qquad \{ \text{ Definition of } \triangle \text{ and } H \ \} \\
 & (h_1\ (x \oplus_1 y), \ldots, h_n\ (x \oplus_n y)) \\
= & \qquad \{ \text{ Assumption of } h_i \ \} \\
 & ((H\ x) \odot_1 (H\ y), \ldots, (H\ x) \odot_n (H\ y)) \\
= & \qquad \{ \text{ Definition of } \triangle \ \} \\
 & (H\ x)\ (\triangle_1^n \odot_i)\ (H\ y)
\end{array}
$$

The third is similar to the second. □

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 $h_1 = \mathsf{map}\ (\mathsf{map}\ \ sum)$. Then, we calculate $h_1\ (x \oplus_1 y)$ to find other functions and operators.

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

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.

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

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

$$
\begin{array}{l}
\quad (s_1, t_1, b_1, r_1, l_1, tr_1, br_1, tl_1, bl_1, c_1, ro_1) \\
\qquad \otimes_1\ \ (s_2, t_2, b_2, r_2, l_2, tr_2, br_2, tl_2, bl_2, c_2, ro_2) \\
= \mathsf{zipwith}_4\ \ f_s\ s_1\ s_2\ r_1\ l_2 \\
\quad \textbf{where}\ f_s\, s_1\, s_2\, r_1\, l_2 = (s_1 \phi\ gemm(\_, +)\, r_1\, l_2) \ominus (NIL \phi\ s_2)
\end{array}
$$

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

$$
\begin{array}{l}
\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\ |\!) \\
\textbf{where} \\
\triangle_1^{11} f_i'\ |a| \quad = \quad (||a||, ||a||, ||a||, ||a||, ||a||, ||a||, ||a||, ||a||, ||a||, |a|, |a|) \\
(s_1, t_1, b_1, r_1, l_1, tr_1, br_1, tl_1, bl_1, c_1, ro_1)\ (\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) \\
\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} \\
\qquad \begin{array}{lcl}
s_0 & = & (s_1 \phi\ gemm\ (\_, \mathsf{zipwith}(+))\ b_1\ t_2) \ominus (NIL \phi\ s_2) \\
t_0 & = & t_1 \phi\ \mathsf{map}\ (\mathsf{zipwith}(+)\ c_1)\ t_2 \\
b_0 & = & \mathsf{map}\ (\lambda z \rightarrow \mathsf{zipwith}(\ominus)\ z\ c_2)\ b_1 \ominus b_2 \\
r_0 & = & (r_1 \phi\ gemm\ (\_, \mathsf{zipwith}(+))\ br_1\ tr_2) \ominus (NIL \phi\ r_2) \\
l_0 & = & (l_1 \phi\ gemm\ (\_, \mathsf{zipwith}(+))\ bl_1\ tl_2) \ominus (NIL \phi\ l_2) \\
tr_0 & = & tr_1 \phi\ \mathsf{map}\ (\mathsf{zipwith}(+)\ (right'\ tr_1))\ tr_2 \\
br_0 & = & \mathsf{map}\ (\lambda z \rightarrow \mathsf{zipwith}(+)\ z\ (top'\ br_2))\ br_1 \ominus br_2 \\
tl_0 & = & tl_1 \phi\ \mathsf{map}\ (\mathsf{zipwith}(+)\ (right'\ tl_1))\ tl_2 \\
bl_0 & = & \mathsf{map}\ (\lambda z \rightarrow \mathsf{zipwith}(+)\ z\ (top'\ bl_2))\ bl_1 \ominus bl_2 \\
c_0 & = & \mathsf{zipwith}(+)\ c_1\ c_2 \\
ro_0 & = & (ro_1 \phi\ gemm\ (\_, +)\ (right\ ro_1)\ (top\ ro_2)) \ominus (NIL \phi\ ro_2)
\end{array} \\
(s_1, t_1, b_1, r_1, l_1, tr_1, br_1, tl_1, bl_1, c_1, ro_1)\ (\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) \\
\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} \\
\qquad \begin{array}{lcl}
s_0 & = & \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) \\
t_0 & = & \mathsf{zipwith}_4\ \ f_t\ t_1\ t_2\ tr_1\ tl_2 \\
 & & \textbf{where}\ f_t\ t_1\ t_2\ tr_1\ tl_2 = (t_1 \phi\ gemm\ (\_, +)\ tr_1\ tl_2) \ominus (NIL \phi\ t_2) \\
b_0 & = & \mathsf{zipwith}_4\ \ f_b\ b_1\ b_2\ br_1\ bl_2 \\
 & & \textbf{where}\ f_b\ b_1\ b_2\ br_1\ bl_2 = (b_1 \phi\ gemm\ (\_, +)\ br_1\ bl_2) \ominus (NIL \phi\ b_2) \\
r_0 & = & \mathsf{zipwith}_3\ \ f_r\ r_1\ r_2\ ro_2 \\
 & & \textbf{where}\ f_r\ r_1\ r_2\ ro_2 = \mathsf{map}\ (+ro_2)\ r_1 \ominus r_2 \\
l_0 & = & \mathsf{zipwith}_3\ \ f_l\ l_1\ l_2\ ro_1 \\
 & & \textbf{where}\ f_l\ l_1\ l_2\ ro_1 = l_1 \phi\ \mathsf{map}\ (ro_1 +)\ l_2 \\
tr_0 & = & \mathsf{zipwith}\ \ f_{tr}\ tr_1\ tr_2 \\
 & & \textbf{where}\ f_{tr}\ tr_1\ tr_2 = \mathsf{map}\ (+top'\ tr_2)\ tr_1 \ominus tr_2 \\
br_0 & = & \mathsf{zipwith}\ \ f_{br}\ br_1\ br_2 \\
 & & \textbf{where}\ f_{br}\ br_1\ br_2 = \mathsf{map}\ (+top'\ br_2)\ br_1 \ominus br_2 \\
tl_0 & = & \mathsf{zipwith}\ \ f_{tl}\ tl_1\ tl_2 \\
 & & \textbf{where}\ f_{tl}\ tl_1\ tl_2 = tl_1 \phi\ \mathsf{map}\ (right'\ tl_1 +)\ tl_2 \\
bl_0 & = & \mathsf{zipwith}\ \ f_{bl}\ bl_1\ bl_2 \\
 & & \textbf{where}\ f_{bl}\ bl_1\ bl_2 = bl_1 \phi\ \mathsf{map}\ (right'\ bl_1 +)\ bl_2 \\
c_0 & = & (c_1 \phi\ gemm\ (\_, +)\ (right\ c_1)\ (top\ c_2)) \ominus (NIL \phi\ c_2) \\
ro_0 & = & \mathsf{zipwith}(+)\ ro_1\ ro_2
\end{array}
\end{array}
$$

In this case, the function H appeared in Theorem 4.2 is as follows:

$$\begin{array}{rcl} H & = & h \times h \times h \times h \times h \times h \times h \times h \times h \times (\mathsf{map}\ \ sum) \times (\mathsf{map}\ \ sum) \\ & & \mathbf{where}\ \ h = \mathsf{map}\ (\mathsf{map}\ \ sum)\ . \end{array}$$

Some calculation rules used in this derivation are listed in Appendix B.

Then, applying the theorem again with the third function $p_3 = \mathsf{map}\ max$, we obtain another almost-homomorphism with $H = (\mathsf{map}\ \ max) \times id \times id \times id \times id \times id \times id \times id \times id \times id \times id$ as follows:

$$\mathsf{map}\ max \circ \mathsf{map}\,(\mathsf{map}\ \ sum) \circ rects = \pi_1 \circ (\!|\ \triangle_1^{11}\ f_i'', \triangle_1^{11} \odot_i', \triangle_1^{11}\ \ominus_i'\ |\!)$$

$$\begin{array}{l}
\mathbf{where} \\
\quad \triangle_1^{11} f_i''\ |a| \quad = \quad (|a|, ||a||, ||a||, ||a||, ||a||, ||a||, ||a||, ||a||, ||a||, |a|, |a|) \\
\quad (s_1, t_1, b_1, r_1, l_1, tr_1, br_1, tl_1, bl_1, c_1, ro_1)\ (\triangle_1^{11} \odot_i')\ (s_2, t_2, b_2, r_2, l_2, tr_2, br_2, tl_2, bl_2, c_2, ro_2) \\
\qquad = (s_0, t_0, b_0, r_0, l_0, tr_0, br_0, tl_0, bl_0, c_0, ro_0) \\
\qquad \mathbf{where}
\end{array}$$

$$\begin{array}{rcl}
s_0 & = & (s_1 \phi \ \mathsf{map}\, max(gemm\ (_, \mathsf{zipwith}(+))\ b_1\ t_2)) \ominus (NIL \phi\ s_2) \\
t_0 & = & t_1 \phi\ \mathsf{map}\ (\mathsf{zipwith}(+)\ c_1)\ t_2 \\
b_0 & = & \mathsf{map}\ (\lambda z \rightarrow \mathsf{zipwith}(\ominus)\ z\ c_2)\ b_1 \ominus b_2 \\
r_0 & = & (r_1 \phi\ gemm\ (_, \mathsf{zipwith}(+))\ br_1\ tr_2) \ominus (NIL \phi\ r_2) \\
l_0 & = & (l_1 \phi\ gemm\ (_, \mathsf{zipwith}(+))\ bl_1\ tl_2) \ominus (NIL \phi\ l_2) \\
tr_0 & = & tr_1 \phi\ \mathsf{map}\ (\mathsf{zipwith}(+)\ (right'\ tr_1))\ tr_2 \\
br_0 & = & \mathsf{map}\ (\lambda z \rightarrow \mathsf{zipwith}(+)\ z\ (top'\ br_2))\ br_1 \ominus br_2 \\
tl_0 & = & tl_1 \phi\ \mathsf{map}\ (\mathsf{zipwith}(+)\ (right'\ tl_1))\ tl_2 \\
bl_0 & = & \mathsf{map}\ (\lambda z \rightarrow \mathsf{zipwith}(+)\ z\ (top'\ bl_2))\ bl_1 \ominus bl_2 \\
c_0 & = & \mathsf{zipwith}(+)\ c_1\ c_2 \\
ro_0 & = & (ro_1 \phi\ gemm\ (_, +)\ (right\ ro_1)\ (top\ ro_2)) \ominus (NIL \phi\ ro_2)
\end{array}$$

$$\begin{array}{l}
\quad (s_1, t_1, b_1, r_1, l_1, tr_1, br_1, tl_1, bl_1, c_1, ro_1)\ (\triangle_1^{11} \ominus_i')\ (s_2, t_2, b_2, r_2, l_2, tr_2, br_2, tl_2, bl_2, c_2, ro_2) \\
\qquad = (s_0, t_0, b_0, r_0, l_0, tr_0, br_0, tl_0, bl_0, c_0, ro_0) \\
\qquad \mathbf{where}
\end{array}$$

$$\begin{array}{rcl}
s_0 & = & \mathsf{zipwith}_4\ \ f_s\ s_1\ s_2\ r_1\ l_2 \\
 & & \mathbf{where}\ \ f_s\ s_1\ s_2\ r_1\ l_2 = s_1 \uparrow max(gemm\ (_, +)\ r_1\ l_2) \uparrow s_2 \\
t_0 & = & \mathsf{zipwith}_4\ \ f_t\ t_1\ t_2\ tr_1\ tl_2 \\
 & & \mathbf{where}\ \ f_t\ t_1\ t_2\ tr_1\ tl_2 = (t_1 \phi\ gemm\ (_, +)\ tr_1\ tl_2) \ominus (NIL \phi\ t_2) \\
b_0 & = & \mathsf{zipwith}_4\ \ f_b\ b_1\ b_2\ br_1\ bl_2 \\
 & & \mathbf{where}\ \ f_b\ b_1\ b_2\ br_1\ bl_2 = (b_1 \phi\ gemm\ (_, +)\ br_1\ bl_2) \ominus (NIL \phi\ b_2) \\
r_0 & = & \mathsf{zipwith}_3\ \ f_r\ r_1\ r_2\ ro_2 \\
 & & \mathbf{where}\ \ f_r\ r_1\ r_2\ ro_2 = \mathsf{map}\ (+ro_2)\ r_1 \ominus r_2 \\
l_0 & = & \mathsf{zipwith}_3\ \ f_l\ l_1\ l_2\ ro_1 \\
 & & \mathbf{where}\ \ f_l\ l_1\ l_2\ ro_1 = l_1 \phi\ \mathsf{map}\ (ro_1+)\ l_2 \\
tr_0 & = & \mathsf{zipwith}\ \ f_{tr}\ tr_1\ tr_2 \\
 & & \mathbf{where}\ \ f_{tr}\ tr_1\ tr_2 = \mathsf{map}\ (+top'\ tr_2)\ tr_1 \ominus tr_2 \\
br_0 & = & \mathsf{zipwith}\ \ f_{br}\ br_1\ br_2 \\
 & & \mathbf{where}\ \ f_{br}\ br_1\ br_2 = \mathsf{map}\ (+top'\ br_2)\ br_1 \ominus br_2 \\
tl_0 & = & \mathsf{zipwith}\ \ f_{tl}\ tl_1\ tl_2 \\
 & & \mathbf{where}\ \ f_{tl}\ tl_1\ tl_2 = tl_1 \phi\ \mathsf{map}\ (right'\ tl_1+)\ tl_2 \\
bl_0 & = & \mathsf{zipwith}\ \ f_{bl}\ bl_1\ bl_2 \\
 & & \mathbf{where}\ \ f_{bl}\ bl_1\ bl_2 = bl_1 \phi\ \mathsf{map}\ (right'\ bl_1+)\ bl_2 \\
c_0 & = & (c_1 \phi\ gemm\ (_, +)\ (right\ c_1)\ (top\ c_2)) \ominus (NIL \phi\ c_2) \\
ro_0 & = & \mathsf{zipwith}(+)\ ro_1\ ro_2
\end{array}$$

Finally, applying such fusion with max will yield the result shown below. This final parallel program uses only $\mathrm{O}(n^3)$ addition operations, which is much better than the initial one. .

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

The function H for the final fusion is as follows:

$$
\begin{array}{l}
H = max \times (\mathsf{reduce}(\_, \mathsf{zipwith}(\uparrow))) \times (\mathsf{reduce}(\mathsf{zipwith}(\uparrow), \_)) \times (\mathsf{map}\ (\mathsf{reduce}(\uparrow, \_))) \\
\qquad \times (\mathsf{map}\ (\mathsf{reduce}(\_, \uparrow))) \times (\mathsf{reduce}(\_, \phi)) \times (\mathsf{reduce}(\phi, \_)) \\
\qquad \times (\mathsf{reduce}(\_, \ominus)) \times (\mathsf{reduce}(\ominus, \_)) \times id \times id
\end{array}
$$

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 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 Section 3.1 and Section 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 in terms of 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.

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

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 $\mathrm{O}(n^2/P)$ parallel time using $P = pq$ processors and ignoring distribution and collection provided that the function f can be computed in $\mathrm{O}(1)$ time. This is the same also about zipwith.

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

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

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 $\mathrm{O}(n^2/P + \log P)$ parallel time using $P = pq$ processors and ignoring distribution provided that the binary operators $\oplus$ and $\otimes$ can be computed in $\mathrm{O}(1)$ time.

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 scan$(\oplus, \otimes)$ by distributing a two-dimensional array of the argument to the processors by dist $p\ q$, applying scan$(\oplus, \otimes)$ to each local array independently on each processor, and finally reducing the results into the root processor by reduce$(\oplus', \otimes')$. However, since the result of scan$(\oplus, \otimes)$ is a two-dimensional array, we want that the last operation of computing scan$(\oplus, \otimes)$ is gather like the case of map f. Thus, we compute underlined $\underline{\mathsf{dist}\,p\,q \circ \mathsf{reduce}(\oplus', \otimes')}$ 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 [15]. Or, from the property of Eq. (2), we can 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 list 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 $\mathrm{O}(n^2/P + \sqrt{n^2/P}\log P)$ parallel time using $P = pq$ processors and ignoring distribution and collection provided that the binary operators $\oplus$ and $\otimes$ can be computed in $\mathrm{O}(1)$ time.

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 $\mathrm{O}(n^2)$ 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 i-th 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 $0 < r \le n/2$ 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 r-th continue behind the last processor, making groups of $2r$ processors from the r-th 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 which have not transmitted their subarrays yet, considering the processors which 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 $\mathrm{O}(n^2/P)$ for P processors, $\mathsf{rot}_r\ f$ can be executed in $\mathrm{O}(n^2/P)$ parallel time. Similarly, $\mathsf{rot}_c\ f$ can be executed in $\mathrm{O}(n^2/P)$ parallel time.

5.3 Experiments

We implemented the parallel skeletons as a library with C++ and MPI, and did our experiments on a small-scale cluster of four Pentium 4 Xeon 2.0-GHz dualprocessor PCs with 1 GB of memory, connected through a Gigabit Ethernet. The OS was FreeBSD 4.10 and we used gcc 2.95 for the compiler, MPICH 1.1.2 for MPI.

Figures 2 and 3 show speedup of the following parallel skeletons and matrix multiplication described in terms of parallel skeletons with $\mathit{square}\ x = x^2$:

(1) $\mathsf{map}\ \mathit{square}$, (2) $\mathsf{reduce}(+,+)$,
(3) $\mathsf{zipwith}(\lambda xy \rightarrow \sqrt{x^2+y^2})$, (4) $\mathsf{scan}(+,+)$,
(5) mm (composition of skeletons, see Section 3.3) .

The inputs for the first five parallel programs are 8000×8000 matrices, and 1800×1800 matrices for mm. The computation times of the above programs on one processor are 4.72sec, 0.32sec, 4.85sec, 0.36sec and 135.3sec respectively.

The result shows programs described in terms of skeletons can be executed efficiently in parallel, and proves the success of our framework. The speedup of matrix multiplication is super-linear. This can happen in large matrix operations where the matrix on a single processor

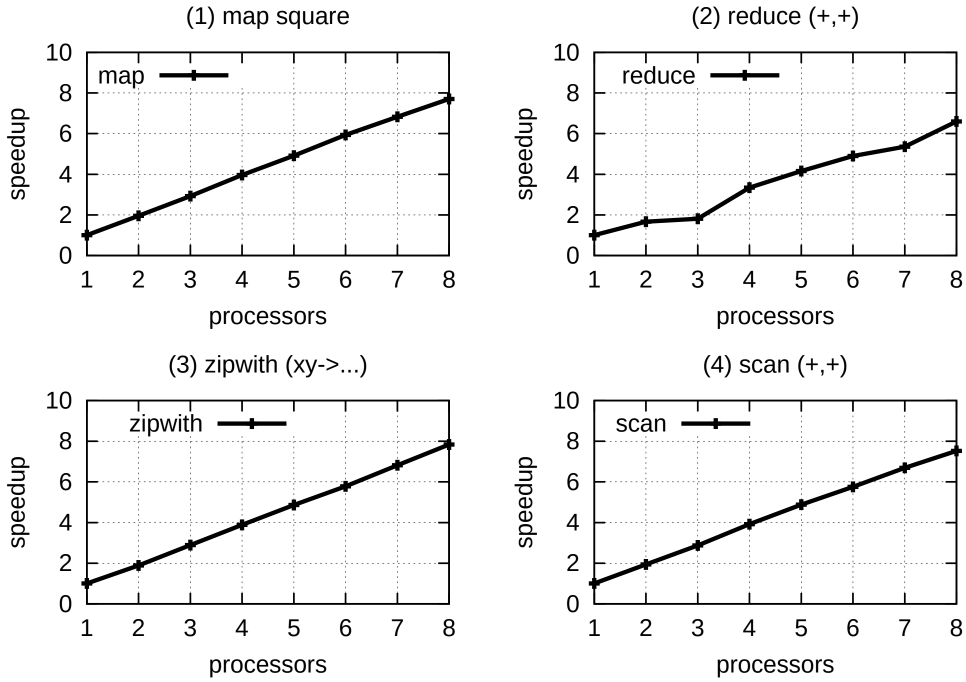

Figure 2: Speedup of Parallel Skeletons

is large with respect to the cache size. It is reasonable that super-linear speedup is achieved here.

Finally, we list part of the C++ code of *mm* written with the skeleton library in Figure 4, 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 the active researches on matrix representation for parallel computations and the compositional approach to parallel program development.

Recursive Matrix Representations

Wise et al. [33] propose 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 [11], which can provide better performance than existing algorithms for some matrix computations such as LU and QR factorizations [12, 34]. However, the quadtree representation requires the size of two-dimensional arrays to be the power of 2. 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.

A more natural representation of a two-dimensional array is to use nested one-dimensional arrays (lists) [4, 30, 22]. 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 [4], as an extension of one-dimensional join list. 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.

Compositional Parallel Programming

This work were greatly inspired by the success of compositional (skeletal) parallel programming on one-dimensional arrays (lists) [27], and our initial motivation was to import the results so

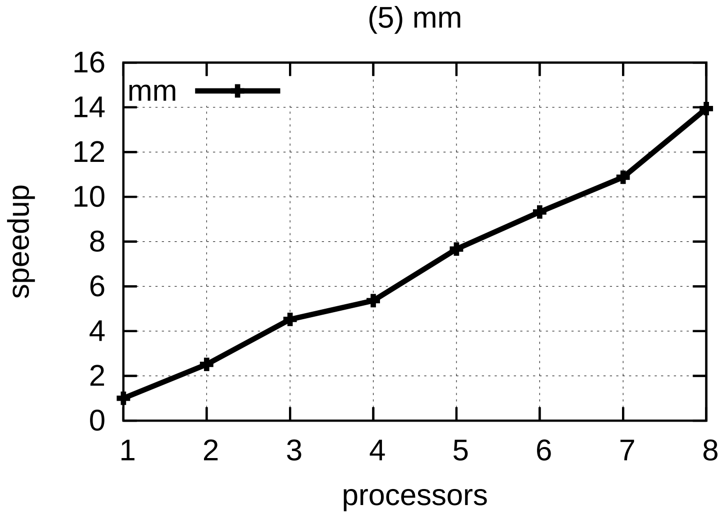

Figure 3: Speedup of Matrix Multiplication

```
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::map_i(Tri< matrix <B> >(), *B2);
    m_skeletons::zipwith(Iprod<C>(), *A2, *B2, Z2);
    delete B2;
    delete A2;
}
```

Figure 4: C++ Code of *mm*

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 [3] was designed as a calculus for deriving (sequential) efficient programs on lists. Skillicorn [29] 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 [21] gives a powerful method to obtain suitable composition of skeletons for a program recursively defined on lists and trees. Chin et al. [20, 6] have studied a systematic method to derive an associative operator which plays an important role in parallelization, based on which Xu et al. [35] build an automatic derivation system for parallelizing recursive linear functions with normalization rules.

7. Conclusion

In this paper, we propose a compositional framework which allows users, even with little knowledge about parallel machines, to easily describe safe and efficient parallel computation over two-dimensional arrays. In our framework, two-dimensional arrays are represented by the abide-tree which 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 programs 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.

This work is still in an early stage, and there are at least two things to do. One is to construct more powerful theories for a systematic programming methodology, in which we can develop efficient and correct parallel programs by parallel skeletons from their recursive specifications. Another is to study an automatic optimization mechanism, which can eliminate inefficiency due to compositional or nested uses of parallel skeletons in parallel programs. It is also our future work to compare our matrix computation algorithms with existing routines (e.g. BLAS).

REFERENCES

[1] 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.

[2] R. Bird and O. de Moor. *Algebras of Programming.* Prentice Hall, 1996.

[3] R. S. Bird. An Introduction to the Theory of Lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume 36 of *NATO ASI Series F*, pages 5–42. Springer-Verlag, 1987.

[4] R. S. Bird. Lectures on Constructive Functional Programming. Technical Report Technical Monograph PRG-69, Oxford University Computing Laboratory, 1988.

[5] R. S. Bird. *Introduction to Functional Programming using Haskell.* Prentice Hall, 1998.

[6] 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)*, pages 153–162. IEEE Press., 1998.

[7] M. Cole. *Algorithmic Skeletons : A Structured Approach to the Management of Parallel Computation.* Research Monographs in Parallel and Distributed Computing, Pitman, London, 1989.

[8] M. Cole. Parallel Programming with List Homomorphisms. *Parallel Processing Letters*, 5(2), 1995.

[9] M. Cole. eSkel Home Page. `http://homepages.inf.ed.ac.uk/mic/eSkel/`, 2002.

[10] 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.

[11] 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.

[12] J. D. Frens and D. S. Wise. QR Factorization with Morton-Ordered Quadtree Matrices for Memory Re-use and Parallelism. In *Proc. 2003 ACM Symp. on Principles and Practice of Parallel Programming*, pages 144–154, 2003.

[13] J. Gibbons, W. Cai, and D. B. Skillicorn. Efficient Parallel Algorithms for Tree Accumulations. *Science of Computer Programming*, 23(1):1–18, 1994.

[14] G. H. Golub and C. F. V. Loan. *Matrix Computations (3rd ed.)*. Johns Hopkins University Press, 1996.

[15] S. Gorlatch. Systematic Efficient Parallelization of Scan and Other List Homomorphisms. In *Proceedings of EuroPar'96, LNCS 1124*, pages 401–408. Springer-Verlag, 1996.

[16] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing*. Addison-Wesley, second edition, 2003.

[17] G. Hains. Programming with Array Structures. In A. Kent and J. G. Williams, editors, *Encyclopedia of Computer Science and Technology*, volume 14, pages 105–119. M. Dekker inc, New-York, 1994. Appears also in *Encyclopedia of Microcomputers*.

[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] 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*, pages 83–97. Springer-Verlag, Apr. 2002.

[20] Z. Hu, M. Takeichi, and W. N. Chin. Parallelization in Calculational Forms. In *Proceedings of 25th ACM Symposium on Principles of Programming Languages*, pages 316–328, San Diego, California, USA, Jan. 1998.

[21] 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)*, pages 85–94, 1999.

[22] J. Jeuring. *Theories for Algorithm Calculation*. PhD thesis, Utrecht University, 1993. Parts of the thesis appeared in the Lecture Notes of the STOP 1992 Summerschool on Constructive Algorithmics.

[23] I. Jonsson and B. Kagstroom. RECSY – A High Performance Library for Sylvester-Type Matrix Equations. In *Proceedings of EuroPar'03, LNCS 2790*, pages 810–819. Springer-Verlag, 2003.

[24] K. Matsuzaki, K. Kakehi, H. Iwasaki, Z. Hu, and Y. Akashi. A Fusion-Embedded Skeleton Library. In *Proceedings of EuroPar'04, LNCS 3149*, pages 644–653. Springer-Verlag, 2004.

[25] R. Miller. *Two Approaches to Architecture-Independent Parallel Computation*. PhD thesis, Computing Laboratory, Oxford University, 1994.

[26] L. Mullin, editor. *Arrays, Functional Languages, and Parallel Systems*. Kluwer Academic Publishers, 1991.

[27] F. A. Rabhi and S. Gorlatch, editors. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer-Verlag, 2002.

[28] J. Reif and J. H. Reif, editors. *Synthesis of Parallel Algorithms*. Morgan Kaufmann, 1993.

[29] D. B. Skillicorn. The Bird-Meertens Formalism as a Parallel Model. In *NATO ARW "Software for Parallel Computation"*, June 1992.

[30] D. B. Skillicorn. *Foundations of Parallel Programming*. Cambridge University Press, 1994.

[31] D. B. Skillicorn. Parallel Implementation of Tree Skeletons. *Journal of Parallel and Distributed Computing*, 39(2):115–125, 1996.

[32] G. W. Stewart. *Matrix Algorithms*. Society for Industrial and Applied Mathematics, 2001.

[33] D. S. Wise. Representing Matrices as Quadtrees for Parallel Processors. *Information Processing Letters*, 20(4):195–199, 1984.

[34] D. S. Wise. Undulant Block Elimination and Integer-Preserving Matrix Inversion. *Science of Computer Programming*, 22(1):29–85, 1999.

[35] 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 2004), LNCS 3302*, pages 197–212. Springer-Verlag, November 2004.

A. Examples of *rects* and so on.

We give example values of eleven functions which constructs the mutually defined function *rects* .

$$segs \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} \begin{pmatrix} (a) & \begin{pmatrix} a & b \end{pmatrix} \\ & (b) \end{pmatrix} & \begin{pmatrix} \begin{pmatrix} a \\ c \end{pmatrix} & \begin{pmatrix} a & b \\ c & d \end{pmatrix} \\ & \begin{pmatrix} b \\ d \end{pmatrix} \end{pmatrix} \\ & \begin{pmatrix} (c) & \begin{pmatrix} c & d \end{pmatrix} \\ & (d) \end{pmatrix} \end{pmatrix} ,$$

$$tops \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} \begin{pmatrix} (a) & \begin{pmatrix} a & b \end{pmatrix} \\ & (b) \end{pmatrix} & \begin{pmatrix} \begin{pmatrix} a \\ c \end{pmatrix} & \begin{pmatrix} a & b \\ c & d \end{pmatrix} \\ & \begin{pmatrix} b \\ d \end{pmatrix} \end{pmatrix} \end{pmatrix} , \ bottoms \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} \begin{pmatrix} \begin{pmatrix} a \\ c \end{pmatrix} & \begin{pmatrix} a & b \\ c & d \end{pmatrix} \\ & \begin{pmatrix} b \\ d \end{pmatrix} \end{pmatrix} \\ \begin{pmatrix} (c) & \begin{pmatrix} c & d \end{pmatrix} \\ & (d) \end{pmatrix} \end{pmatrix} ,$$

$$rights \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} \begin{pmatrix} \begin{pmatrix} a & b \end{pmatrix} \\ (b) \end{pmatrix} & \begin{pmatrix} \begin{pmatrix} a & b \\ c & d \end{pmatrix} \\ \begin{pmatrix} b \\ d \end{pmatrix} \end{pmatrix} \\ & \begin{pmatrix} \begin{pmatrix} c & d \end{pmatrix} \\ (d) \end{pmatrix} \end{pmatrix} , \ lefts \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} \begin{pmatrix} (a) & \begin{pmatrix} a & b \end{pmatrix} \end{pmatrix} & \begin{pmatrix} \begin{pmatrix} a \\ c \end{pmatrix} & \begin{pmatrix} a & b \\ c & d \end{pmatrix} \end{pmatrix} \\ & \begin{pmatrix} (c) & \begin{pmatrix} c & d \end{pmatrix} \end{pmatrix} \end{pmatrix} ,$$

$$toprights \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} \begin{pmatrix} \begin{pmatrix} a & b \end{pmatrix} \\ (b) \end{pmatrix} & \begin{pmatrix} \begin{pmatrix} a & b \\ c & d \end{pmatrix} \\ \begin{pmatrix} b \\ d \end{pmatrix} \end{pmatrix} \end{pmatrix} , \ bottomrights \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} \begin{pmatrix} \begin{pmatrix} a & b \\ c & d \end{pmatrix} \\ \begin{pmatrix} b \\ d \end{pmatrix} \end{pmatrix} \\ \begin{pmatrix} \begin{pmatrix} c & d \end{pmatrix} \\ (d) \end{pmatrix} \end{pmatrix} ,$$

$$toplefts \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} \begin{pmatrix} (a) & \begin{pmatrix} a & b \end{pmatrix} \end{pmatrix} & \begin{pmatrix} \begin{pmatrix} a \\ c \end{pmatrix} & \begin{pmatrix} a & b \\ c & d \end{pmatrix} \end{pmatrix} \end{pmatrix} , \ bottomlefts \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} \begin{pmatrix} \begin{pmatrix} a \\ c \end{pmatrix} & \begin{pmatrix} a & b \\ c & d \end{pmatrix} \end{pmatrix} \\ \begin{pmatrix} (c) & \begin{pmatrix} c & d \end{pmatrix} \end{pmatrix} \end{pmatrix} ,$$

$$cols \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} \begin{pmatrix} a \\ c \end{pmatrix} & \begin{pmatrix} a & b \\ c & d \end{pmatrix} \\ & \begin{pmatrix} b \\ d \end{pmatrix} \end{pmatrix} \quad , \; rows \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} \begin{pmatrix} a & b \end{pmatrix} & \begin{pmatrix} a & b \\ c & d \end{pmatrix} \\ & \begin{pmatrix} c & d \end{pmatrix} \end{pmatrix} \quad .$$

B. Some Calculation Rules

We summarize the calculation rules used in Section 4 for derivation of the efficient parallel program for solving the maximum rectangle sum problems.

B.1 Rule I

$$\begin{aligned} &\mathsf{map}\ f\ (\mathsf{zipwith}(\oplus)\ x\ y) = \mathsf{zipwith}(\oplus')\ (\mathsf{map}\ f\ x)\ (\mathsf{map}\ f\ y) \\ &\qquad\qquad \Leftarrow \forall a, b\ \ f\ (a \oplus b) = f\ a \oplus' f\ b \end{aligned}$$

Proof. The induction on the structure of abide trees.

$$\begin{array}{ll} & \mathsf{map}\ f\ (\mathsf{zipwith}(\oplus)\ |a|\ |b|) \\ = & \quad \{\ \text{def. of } \mathsf{zipwith}, \mathsf{map}\ \} \\ & |f\ (a \oplus b)| \\ = & \quad \{\ \text{hypo.}\ \} \\ & |f\ a \oplus' f\ b| \\ = & \quad \{\ \text{def. of } \mathsf{zipwith}, \mathsf{map}\ \} \\ & \mathsf{zipwith}(\oplus')\ (\mathsf{map}\ f\ |a|)\ (\mathsf{map}\ f\ |b|) \end{array}$$

$$\begin{array}{ll} & \mathsf{map}\ f\ (\mathsf{zipwith}(\oplus)\ (x \ominus y)\ (u \ominus v)) \\ = & \quad \{\ \text{def. of } \mathsf{zipwith}, \mathsf{map}\ \} \\ & \mathsf{map}\ f\ (\mathsf{zipwith}(\oplus)\ x\ u) \ominus \mathsf{map}\ f\ (\mathsf{zipwith}(\oplus)\ y\ v) \\ = & \quad \{\ \text{hypo. of induction}\ \} \\ & \mathsf{zipwith}(\oplus')\ (\mathsf{map}\ f\ x)\ (\mathsf{map}\ f\ u) \ominus \mathsf{zipwith}(\oplus')\ (\mathsf{map}\ f\ y)\ (\mathsf{map}\ f\ v) \\ = & \quad \{\ \text{def. of } \mathsf{zipwith}, \mathsf{map}\ \} \\ & \mathsf{zipwith}(\oplus')\ (\mathsf{map}\ f\ (x \ominus y))\ (\mathsf{map}\ f\ (u \ominus v)) \end{array}$$

$$\begin{array}{ll} & \mathsf{map}\ f\ (\mathsf{zipwith}(\oplus)\ (x \obar y)\ (u \obar v)) \\ = & \quad \{\ \text{similar to } \ominus\ \} \\ & \mathsf{zipwith}(\oplus')\ (\mathsf{map}\ f\ (x \obar y))\ (\mathsf{map}\ f\ (u \obar v)) \end{array}$$

B.2 Rule II

$$\mathsf{map}\ (\mathsf{reduce}(\oplus, \otimes))\ (\mathsf{zipwith}(\ominus)\ x\ y) = \mathsf{zipwith}(\oplus)\ (\mathsf{map}\ (\mathsf{reduce}(\oplus, \otimes))\ x)\ (\mathsf{map}\ (\mathsf{reduce}(\oplus, \otimes))\ y)$$

Proof. Rule I and the following calculation with $f = \mathsf{reduce}(\oplus, \otimes), \oplus = \ominus, \oplus' = \oplus$.

$$\mathsf{reduce}(\oplus, \otimes)\ (a \ominus b) = \mathsf{reduce}(\otimes, \oplus)\ a \oplus \mathsf{reduce}(\otimes, \oplus)\ b$$

B.3 Rule III

$$
\begin{aligned}
\mathsf{map}\ f\ (gemm(\oplus,\otimes)\ x\ y) &= gemm(\oplus',\otimes')\ (\mathsf{map}\ f\ x)\ (\mathsf{map}\ f\ y) \\
&\Leftarrow \forall a,b\ \ f\ (a\oplus b) = f\ a \oplus' f\ b,\ \ f\ (a\otimes b) = f\ a \otimes' f\ b
\end{aligned}
$$

Proof. The induction on the structure of abide trees.

$$
\begin{array}{ll}
& \mathsf{map}\ f\ (gemm(\oplus,\otimes)\ |a|\ |b|) \\
= & \quad \{\ \text{def. of } gemm, \mathsf{map}\ \} \\
& |f\ (a\otimes b)| \\
= & \quad \{\ \text{hypo.}\ \} \\
& |f\ a \otimes' f\ b| \\
= & \quad \{\ \text{def. of } gemm, \mathsf{map}\ \} \\
& gemm(\oplus',\otimes')\ (\mathsf{map}\ f\ |a|)\ (\mathsf{map}\ f\ |b|)
\end{array}
$$

$$
\begin{array}{ll}
& \mathsf{map}\ f\ (gemm(\oplus,\otimes)\ (x \ominus y)\ z) \\
= & \quad \{\ \text{def. of } gemm, \mathsf{map}\ \} \\
& \mathsf{map}\ f\ (gemm(\oplus,\otimes)\ x\ z) \ominus \mathsf{map}\ f\ (gemm(\oplus,\otimes)\ y\ z) \\
= & \quad \{\ \text{hypo. of induction}\ \} \\
& gemm(\oplus',\otimes')\ (\mathsf{map}\ f\ x)\ (\mathsf{map}\ f\ z) \ominus gemm(\oplus',\otimes')\ (\mathsf{map}\ f\ y))\ (\mathsf{map}\ f\ z) \\
= & \quad \{\ \text{def. of } gemm, \mathsf{map}\ \} \\
& gemm(\oplus',\otimes')\ (\mathsf{map}\ f\ (x \ominus y))\ (\mathsf{map}\ f\ z)
\end{array}
$$

$$
\begin{array}{ll}
& \mathsf{map}\ f\ (gemm(\oplus,\otimes)\ x\ (y \obar z)) \\
= & \quad \{\ \text{similar to above}\ \} \\
& gemm(\oplus',\otimes')\ (\mathsf{map}\ f\ x)\ (\mathsf{map}\ f\ (y \obar z))
\end{array}
$$

$$
\begin{array}{ll}
& \mathsf{map}\ f\ (gemm(\oplus,\otimes)\ (x \obar y)\ (u \ominus v)) \\
= & \quad \{\ \text{def. of } gemm, \mathsf{map}\ \} \\
& \mathsf{map}\ f\ (\mathsf{zipwith}(\oplus)\ (gemm(\oplus,\otimes)\ x\ u)\ (gemm(\oplus,\otimes)\ y\ v)) \\
= & \quad \{\ \text{I}\ \} \\
& \mathsf{zipwith}(\oplus')(\mathsf{map}\ f\ (gemm(\oplus,\otimes)\ x\ u))\ (\mathsf{map}\ f\ (gemm(\oplus,\otimes)\ y\ v)) \\
= & \quad \{\ \text{hypo. of induction}\ \} \\
& \mathsf{zipwith}(\oplus')(gemm(\oplus',\otimes')\ (\mathsf{map}\ f\ x)\ (\mathsf{map}\ f\ u))\ (gemm(\oplus',\otimes')\ (\mathsf{map}\ f\ y)\ (\mathsf{map}\ f\ v)) \\
= & \quad \{\ \text{def. of } gemm, \mathsf{map}\ \} \\
& gemm(\oplus',\otimes')\ (\mathsf{map}\ f\ (x \ominus y))\ (\mathsf{map}\ f\ z)
\end{array}
$$

B.4 Rule IV

$$\begin{aligned}\mathsf{map}\ \ f\ (\mathsf{map}(\oplus x)\ y) = \mathsf{map}\ (\otimes'(f\ x))(\mathsf{map}\ \ f\ y)\\ \Leftarrow \forall a, b\ \ f\ (a \oplus b) = f\ a \oplus' f\ b\end{aligned}$$

Proof. The induction on the structure of abide trees.

$$\begin{aligned}&\mathsf{map}\ \ f\ (\mathsf{map}(\oplus x)\ |a|)\\ = &\quad \{\ \text{def. of}\ \mathsf{map}\ \}\\ &|f\ (x \oplus a)|\\ = &\quad \{\ \text{hypo.}\ \}\\ &|f\ x \oplus' f\ a)|\\ = &\quad \{\ \text{def. of}\ \mathsf{map}\ \}\\ &\mathsf{map}\ (\otimes'(f\ x))(\mathsf{map}\ \ f\ y)\end{aligned}$$

$$\begin{aligned}&\mathsf{map}\ \ f\ (\mathsf{map}(\oplus x)\ (y \phi z))\\ = &\quad \{\ \text{def. of}\ \mathsf{map}\ \}\\ &\mathsf{map}\ \ f\ (\mathsf{map}(\oplus x)\ y) \phi \mathsf{map}\ \ f\ (\mathsf{map}(\oplus x)\ z)\\ = &\quad \{\ \text{hypo. of induction}\ \}\\ &\mathsf{map}\ (\otimes'(f\ x))(\mathsf{map}\ \ f\ y) \phi \mathsf{map}\ (\otimes'(f\ x))(\mathsf{map}\ \ f\ z)\\ = &\quad \{\ \text{def. of}\ \mathsf{map}\ \}\\ &\mathsf{map}\ (\otimes'(f\ x))(\mathsf{map}\ \ f\ (y \phi z))\end{aligned}$$

The incuntion case for $\ominus$ is proved similarly.

The following is an instance of this rule:

$$\mathsf{map}\ \ sum\ (\mathsf{zipwith}\ (\ominus)\ a\ b) = \mathsf{zipwith}(+)\ (\mathsf{map}\ \ sum\ a)\ (\mathsf{map}\ \ sum\ b)$$

B.5 Rule V

$$\mathsf{map}\ \ f\ (right'\ x) = right'(\mathsf{map}\ (\mathsf{map}\ \ f)\ x)$$

Proof.

$$\begin{aligned}&\mathsf{map}\ \ f \circ right'\\ = &\quad \{\ \text{def. of}\ right'\ \}\\ &\mathsf{map}\ \ f \circ \mathsf{the} \circ right\\ = &\quad \{\ \text{def. of}\ right\ \}\\ &\mathsf{map}\ \ f \circ \mathsf{the} \circ \mathsf{reduce}(\ominus, \gg) \circ \mathsf{map}\ \ |\cdot|\\ = &\quad \{\ \text{def. of}\ \mathsf{the}, \mathsf{map}\ \}\\ &\mathsf{the} \circ \mathsf{map}\ (\mathsf{map}\ \ f) \circ \mathsf{reduce}(\ominus, \gg) \circ \mathsf{map}\ \ |\cdot|\\ = &\quad \{\ \text{VI}\ \}\\ &\mathsf{the} \circ \mathsf{reduce}(\ominus, \gg) \circ \mathsf{map}\ (\mathsf{map}\ (\mathsf{map}\ \ f))\ \ \mathsf{map}\ \ |\cdot|\\ = &\quad \{\ \text{def. of}\ |\cdot|, \mathsf{map}\ \}\\ &\mathsf{the} \circ \mathsf{reduce}(\ominus, \gg) \circ \mathsf{map}\ \ |\cdot| \circ \mathsf{map}\ (\mathsf{map}\ \ f))\\ = &\quad \{\ \text{def. of}\ right'\ \}\\ &right' \circ \mathsf{map}\ (\mathsf{map}\ \ f)\end{aligned}$$

This rule for top' holds similarly.

B.6 Rule VI

$$
\begin{aligned}
\mathsf{map}\ \ f\ \circ \mathsf{reduce}(\oplus,\otimes) = \mathsf{reduce}(\oplus,\otimes) \circ \mathsf{map}\ (\mathsf{map}\ \ f)\\
\Leftarrow \oplus,\otimes \in \{\ominus, \phi, \ll, \gg\}
\end{aligned}
$$

Proof.

$$
\begin{aligned}
&\mathsf{map}\ \ f\ (\mathsf{reduce}(\oplus,\otimes)\ |a|)\\
=&\qquad \{\text{ def. of } \mathsf{reduce} \}\\
&\mathsf{map}\ \ f\ a\\
=&\qquad \{\text{ def. of } \mathsf{reduce} \}\\
&\mathsf{reduce}(\oplus,\otimes)\ |\,\mathsf{map}\ \ f\ a|\\
=&\qquad \{\text{ def. of } \mathsf{map} \}\\
&\mathsf{reduce}(\oplus,\otimes)\ (\mathsf{map}\ (\mathsf{map}\ \ f)\ |a|)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{map}\ \ f\ (\mathsf{reduce}(\oplus,\otimes)\ (x \phi y))\\
=&\qquad \{\text{ def. of } \mathsf{reduce} \}\\
&\mathsf{map}\ \ f\ (\mathsf{reduce}(\oplus,\otimes)\ x \otimes \mathsf{reduce}(\oplus,\otimes) y)\\
=&\qquad \{\text{ below } \}\\
&\mathsf{map}\ \ f\ (\mathsf{reduce}(\oplus,\otimes)\ x) \otimes \mathsf{map}\ \ f\ (\mathsf{reduce}(\oplus,\otimes) y)\\
=&\qquad \{\text{ hypo. of induction } \}\\
&\mathsf{reduce}(\oplus,\otimes)\ (\mathsf{map}\ (\mathsf{map}\ \ f)\ x) \otimes \mathsf{reduce}(\oplus,\otimes)\ (\mathsf{map}\ (\mathsf{map}\ \ f)\ y)\\
=&\qquad \{\text{ def. of } \mathsf{map}, \mathsf{reduce} \}\\
&\mathsf{reduce}(\oplus,\otimes)\ (\mathsf{map}\ (\mathsf{map}\ \ f)\ (x \phi y))
\end{aligned}
$$

The incuntion case for $\ominus$ is proved similarly.

$$
\mathsf{map}\ \ f\ (x \oplus y) = \mathsf{map}\ \ f\ x \oplus \mathsf{map}\ \ f\ y \Leftarrow \oplus \in \{\ominus, \phi, \ll, \gg\}
$$

Proof.

$$
\begin{aligned}
&\mathsf{map}\ \ f\ (x \phi y) = \mathsf{map}\ \ f\ x \phi \mathsf{map}\ \ f\ y\\
&\mathsf{map}\ \ f\ (x \ominus y) = \mathsf{map}\ \ f\ x \ominus \mathsf{map}\ \ f\ y\\
&\mathsf{map}\ \ f\ (x \gg y) = \mathsf{map}\ \ f\ y = \mathsf{map}\ \ f\ x \gg \mathsf{map}\ \ f\ y\\
&\mathsf{map}\ \ f\ (x \ll y) = \mathsf{map}\ \ f\ x = \mathsf{map}\ \ f\ x \ll \mathsf{map}\ \ f\ y
\end{aligned}
$$

B.7 Rule VII

$$
\begin{aligned}
\mathsf{map}\ \ f\ (\mathsf{zipwith}_4\ \ g\ x\ u\ w\ a) = \mathsf{zipwith}_4\ \ g'\ (\mathsf{map}\ \ f_1\ x)\ (\mathsf{map}\ \ f_2\ u)\ (\mathsf{map}\ \ f_3\ w)\ (\mathsf{map}\ \ f_4\ a)\\
\Leftarrow f\ (g\ x\ u\ w\ a) = g'\ (f_1\ x)\ (f_2\ u)\ (f_3\ w)\ (f_4\ a)
\end{aligned}
$$

Proof. The induction on the structure of abide trees.

$$
\begin{aligned}
& \mathsf{map}\ f\ (\mathsf{zipwith}_4\ g\ |a|\ |b|\ |c|\ |d|) \\
= & \quad \{ \text{ def. of } \mathsf{zipwith}, \mathsf{map} \} \\
& |f\ (g\ a\ b\ c\ d)| \\
= & \quad \{ \text{ hypo. } \} \\
& |g'\ (f_1\ a)\ (f_2\ b)\ (f_3\ c)\ (f_4\ d)| \\
= & \quad \{ \text{ def. of } \mathsf{zipwith}, \mathsf{map} \} \\
& \mathsf{zipwith}_4\ g'\ (\mathsf{map}\ f_1\ |a|)\ (\mathsf{map}\ f_2\ |b|)\ (\mathsf{map}\ f_3\ |c|)\ (\mathsf{map}\ f_4\ |d|)
\end{aligned}
$$

$$
\begin{aligned}
& \mathsf{map}\ f\ (\mathsf{zipwith}_4\ g\ (a \phi x)\ (b \phi y)\ (c \phi z)\ (d \phi w)) \\
= & \quad \{ \text{ def. of } \mathsf{zipwith}, \mathsf{map} \} \\
& map\ f\ (\mathsf{zipwith}_4\ g\ a\ b\ c\ d) \phi \mathsf{map}\ f\ (\mathsf{zipwith}_4\ g\ x\ y\ z\ w) \\
= & \quad \{ \text{ hypo. of induction } \} \\
& \mathsf{zipwith}_4\ g'\ (\mathsf{map}\ f_1\ a)\ (\mathsf{map}\ f_2\ b)\ (\mathsf{map}\ f_3\ c)\ (\mathsf{map}\ f_4\ d) \\
& \phi\ \mathsf{zipwith}_4\ g'\ (\mathsf{map}\ f_1\ x)\ (\mathsf{map}\ f_2\ y)\ (\mathsf{map}\ f_3\ z)\ (\mathsf{map}\ f_4\ w) \\
= & \quad \{ \text{ def. of } \mathsf{zipwith}, \mathsf{map} \} \\
& \mathsf{zipwith}_4\ g'\ (\mathsf{map}\ f_1\ (a \phi x))\ (\mathsf{map}\ f_2\ (b \phi y))\ (\mathsf{map}\ f_3\ (c \phi z))\ (\mathsf{map}\ f_4\ (d \phi w))
\end{aligned}
$$

The incuntion case for $\ominus$ is proved similarly.

B.8 Rule VIII

$$sum \circ (\phi x) = (+(sum\ x)) \circ sum$$

Proof.

$$(sum \circ (\phi x))\ y = sum\ (y \phi x) = sum\ y + sum\ x = ((+(sum\ x)) \circ sum)\ y$$

B.9 Rule IX

$$
\begin{aligned}
& \mathsf{map}\ \ sum(\mathsf{map}\ (\phi\ \ top'\ tr_2)\ tr_1 \ominus tr_2) \\
= & \quad \{ \text{ def. of } \mathsf{map} \} \\
& \mathsf{map}\ \ sum(\mathsf{map}\ (\phi\ \ top'\ tr_2)\ tr_1) \ominus \mathsf{map}\ \ sumtr_2 \\
= & \quad \{ \text{ def. of } \mathsf{map} \} \\
& \mathsf{map}\ (sum \circ (\phi\ \ top'\ tr_2))\ tr_1 \ominus \mathsf{map}\ \ sumtr_2 \\
= & \quad \{ \text{ V, VIII } \} \\
& \mathsf{map}\ (+\ top'\ (\mathsf{map}\ \ sum\ tr_2))\ (\mathsf{map}\ \ sum\ tr_1) \ominus \mathsf{map}\ \ sumtr_2
\end{aligned}
$$

B.10 Rule X

$$
\begin{aligned}
& \mathsf{reduce}(\oplus, \oplus)(\mathsf{zipwith}(\otimes)\ (\mathsf{zipwith}(\oplus)\ a\ b)\ \ (\mathsf{zipwith}(\oplus)\ c\ d)) \\
& = \mathsf{reduce}(\oplus, \oplus)(\mathsf{zipwith}(\otimes)\ a\ c) \oplus \mathsf{reduce}(\oplus, \oplus)(\mathsf{zipwith}(\otimes)\ a\ d) \\
& \quad \oplus \mathsf{reduce}(\oplus, \oplus)(\mathsf{zipwith}(\otimes)\ b\ c) \oplus \mathsf{reduce}(\oplus, \oplus)(\mathsf{zipwith}(\otimes)\ b\ d) \\
& \Leftarrow (a \oplus b) \otimes (c \oplus d) = (a \otimes c) \oplus (a \otimes d) \oplus (b \otimes c) \oplus (b \otimes d)
\end{aligned}
$$

Proof. The induction on the structure of abide trees.

$$
\begin{aligned}
& \quad \textsf{reduce}(\oplus,\oplus)(\textsf{zipwith}(\otimes)\ (\textsf{zipwith}(\oplus)\ |a|\ |b|))\ (\textsf{zipwith}(\otimes)\ (\textsf{zipwith}(\oplus)\ |c|\ |d|)) \\
= & \qquad \{\ \text{def. of } \textsf{zipwith}, \textsf{reduce}\ \} \\
& \quad (a \oplus b) \otimes (c \oplus d) \\
= & \qquad \{\ \text{hypo.}\ \} \\
& \quad (a \otimes c) \oplus (a \otimes d) \oplus (b \otimes c) \oplus (b \otimes d) \\
= & \qquad \{\ \text{def. of } \textsf{zipwith}, \textsf{reduce}\ \} \\
& \quad \textsf{reduce}(\oplus,\oplus)(\textsf{zipwith}(\otimes)\ |a|\ |c|) \oplus \textsf{reduce}(\oplus,\oplus)(\textsf{zipwith}(\otimes)\ |a|\ |d|) \\
& \qquad \oplus \textsf{reduce}(\oplus,\oplus)(\textsf{zipwith}(\otimes)\ |b|\ |c|) \oplus \textsf{reduce}(\oplus,\oplus)(\textsf{zipwith}(\otimes)\ |b|\ |d|)
\end{aligned}
$$

$$
\begin{aligned}
& \quad \textsf{reduce}(\oplus,\oplus)(\textsf{zipwith}(\otimes)\ (\textsf{zipwith}(\oplus)\ (a_1 \mathbin{\phi} a_2)\ (b_1 \mathbin{\phi} b_2)))\ (\textsf{zipwith}(\otimes)\ (\textsf{zipwith}(\oplus)\ (c_1 \mathbin{\phi} c_2)\ (d_1 \mathbin{\phi} d_2))) \\
= & \qquad \{\ \text{def. of } \textsf{zipwith}, \textsf{reduce}\ \} \\
& \quad \textsf{reduce}(\oplus,\oplus)(\textsf{zipwith}(\otimes)\ (\textsf{zipwith}(\oplus)\ a_1\ b_1))\ (\textsf{zipwith}(\otimes)\ (\textsf{zipwith}(\oplus)\ c_1\ d_1)) \\
& \qquad \oplus \textsf{reduce}(\oplus,\oplus)(\textsf{zipwith}(\otimes)\ (\textsf{zipwith}(\oplus)\ a_2\ b_2))\ (\textsf{zipwith}(\otimes)\ (\textsf{zipwith}(\oplus)\ c_2\ d_2)) \\
= & \qquad \{\ \text{hypo. of induction}\ \} \\
& \quad \textsf{reduce}(\oplus,\oplus)(\textsf{zipwith}(\otimes)\ a_1\ c_1) \oplus \textsf{reduce}(\oplus,\oplus)(\textsf{zipwith}(\otimes)\ a_1\ d_1) \\
& \qquad \oplus \textsf{reduce}(\oplus,\oplus)(\textsf{zipwith}(\otimes)\ b_1\ c_1) \oplus \textsf{reduce}(\oplus,\oplus)(\textsf{zipwith}(\otimes)\ b_1\ d_1) \\
& \qquad\quad \oplus \textsf{reduce}(\oplus,\oplus)(\textsf{zipwith}(\otimes)\ a_2\ c_2) \oplus \textsf{reduce}(\oplus,\oplus)(\textsf{zipwith}(\otimes)\ a_2\ d_2) \\
& \qquad\qquad \oplus \textsf{reduce}(\oplus,\oplus)(\textsf{zipwith}(\otimes)\ b_2\ c_2) \oplus \textsf{reduce}(\oplus,\oplus)(\textsf{zipwith}(\otimes)\ b_2\ d_2) \\
= & \qquad \{\ \text{def. of } \textsf{zipwith}, \textsf{reduce}\ \} \\
& \quad \textsf{reduce}(\oplus,\oplus)(\textsf{zipwith}(\otimes)\ (a_1 \mathbin{\phi} a_2)\ (c_1 \mathbin{\phi} c_2)) \oplus \textsf{reduce}(\oplus,\oplus)(\textsf{zipwith}(\otimes)\ (a_1 \mathbin{\phi} a_2)\ (d_1 \mathbin{\phi} d_2)) \\
& \qquad \oplus \textsf{reduce}(\oplus,\oplus)(\textsf{zipwith}(\otimes)\ (b_1 \mathbin{\phi} b_2)\ (c_1 \mathbin{\phi} c_2)) \oplus \textsf{reduce}(\oplus,\oplus)(\textsf{zipwith}(\otimes)\ (b_1 \mathbin{\phi} b_2)\ (d_1 \mathbin{\phi} d_2))
\end{aligned}
$$

The incuntion case for $\ominus$ is proved similarly.

B.11 Rule XI

$$
\begin{aligned}
& \mathit{max}\ (\textsf{map}\ \ \mathit{max}\ (\mathit{gemm}(\_, \textsf{zipwith}(+))\ b\ t)) \\
& = \mathit{max}\ (\textsf{zipwith}(+)\ (\textsf{reduce}\ (\textsf{zipwith}(\uparrow), \_)\ b)\ (\textsf{reduce}\ (\_, \textsf{zipwith}(\uparrow))\ t)) \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \Leftarrow \mathit{width}\ b = 1, \mathit{height}\ t = 1
\end{aligned}
$$

Proof.The induction on the structure of abide trees.

$$
\begin{aligned}
& \quad \mathit{max}\ (\textsf{map}\ \ \mathit{max}\ (\mathit{gemm}(\_, \textsf{zipwith}(+))\ |b|\ |t|)) \\
= & \qquad \{\ \text{def. of } \mathit{gemm}\ \} \\
& \quad \mathit{max}\ (\textsf{map}\ \ \mathit{max}\ (|\,\textsf{zipwith}(+)\ b\ t|)) \\
= & \qquad \{\ \text{def. of } \textsf{zipwith}, \textsf{map}, \mathit{max}\ \} \\
& \quad \mathit{max}\ (\textsf{zipwith}(+)\ b\ t) \\
= & \qquad \{\ \text{def. of } \textsf{reduce}\ \} \\
& \quad \mathit{max}\ (\textsf{zipwith}(+)\ (\textsf{reduce}\ (\textsf{zipwith}(\uparrow), \_)\ |b|)\ (\textsf{reduce}\ (\_, \textsf{zipwith}(\uparrow))\ |t|))
\end{aligned}
$$

$$
\begin{aligned}
& max\ (\mathsf{map}\ \ max\ (gemm(\_, \mathsf{zipwith}(+))\ (b_1 \ominus b_2)\ (t_1 \phi\ t_2))) \\
= \quad & \{ \text{ def. of } gemm \ \} \\
& max\ (\mathsf{map}\ \ max\ (((gemm(\_, \mathsf{zipwith}(+))\ b_1\ t_1) \phi\ (gemm(\_, \mathsf{zipwith}(+))\ b_1\ t_2)) \\
& \qquad \ominus ((gemm(\_, \mathsf{zipwith}(+))\ b_2\ t_1) \phi\ (gemm(\_, \mathsf{zipwith}(+))\ b_2\ t_2)))) \\
= \quad & \{ \text{ def. of } max \ \} \\
& max\ (\mathsf{map}\ \ max\ (gemm(\_, \mathsf{zipwith}(+))\ b_1\ t_1)) \uparrow \\
& \quad max\ (\mathsf{map}\ \ max\ (gemm(\_, \mathsf{zipwith}(+))\ b_1\ t_2)) \\
& \qquad \uparrow max\ (\mathsf{map}\ \ max\ (gemm(\_, \mathsf{zipwith}(+))\ b_2\ t_1)) \\
& \qquad\quad \uparrow max\ (\mathsf{map}\ \ max\ (gemm(\_, \mathsf{zipwith}(+))\ b_2\ t_2)) \\
= \quad & \{ \text{ hypo. of induction } \} \\
& max\ (\mathsf{zipwith}(+)\ (\mathsf{reduce}\ (\mathsf{zipwith}(\uparrow), \_)\ b_1)\ (\mathsf{reduce}\ (\_, \mathsf{zipwith}(\uparrow))\ t_1)) \\
& \qquad \uparrow max\ (\mathsf{zipwith}(+)\ (\mathsf{reduce}\ (\mathsf{zipwith}(\uparrow), \_)\ b_1)\ (\mathsf{reduce}\ (\_, \mathsf{zipwith}(\uparrow))\ t_2)) \\
& \qquad\quad \uparrow max\ (\mathsf{zipwith}(+)\ (\mathsf{reduce}\ (\mathsf{zipwith}(\uparrow), \_)\ b_2)\ (\mathsf{reduce}\ (\_, \mathsf{zipwith}(\uparrow))\ t_1)) \\
& \qquad\qquad \uparrow max\ (\mathsf{zipwith}(+)\ (\mathsf{reduce}\ (\mathsf{zipwith}(\uparrow), \_)\ b_2)\ (\mathsf{reduce}\ (\_, \mathsf{zipwith}(\uparrow))\ t_2)) \\
= \quad & \{ \text{ X with } \oplus = \uparrow, \otimes = + \ \} \\
& max\ (\mathsf{zipwith}(+)\ (\mathsf{reduce}\ (\mathsf{zipwith}(\uparrow), \_)\ (b_1 \ominus b_2))\ (\mathsf{reduce}\ (\_, \mathsf{zipwith}(\uparrow))\ (t_1 \phi\ t_2)))
\end{aligned}
$$

B.12 Rule XII

$$
\begin{aligned}
& max(\mathsf{zipwith}_4\ \ f_s\ s_1\ s_2\ r_1\ l_2) \\
& \ = max\ s_1 \uparrow max\ s_2 \uparrow\ \ max\ (\mathsf{zipwith}(+)\ (\mathsf{map}\ \ \mathsf{reduce}(\uparrow, \_)\ r_1)\ (\mathsf{map}\ \ \mathsf{reduce}(\_, \uparrow)\ l_2)) \\
& \qquad \textbf{where}\ \ f_s\ s_1\ s_2\ r_1\ l_2 = s_1 \uparrow max\ (gemm(\_, +)\ r_1\ l_2) \uparrow s_2 \\
& \quad \Leftarrow width \text{ of elements of } r_1 = 1, height\ \text{ of elements of } l_2 = 1
\end{aligned}
$$

Proof. First, we prove the following equation by the induction on the structure of abide trees.

$$
\begin{aligned}
max(\mathsf{zipwith}_4\ \ f_s\ s_1\ s_2\ r_1\ l_2) = max\ s_1 \uparrow max\ s_2 \uparrow max\ (\mathsf{zipwith}\ \ f'_s\ r_1\ l_2) \\
\textbf{where}\ \ f'_s\ r_1\ l_2 = max\ (gemm(\_, +)\ r_1\ l_2)
\end{aligned}
$$

Proof.

$$
\begin{aligned}
& max(\mathsf{zipwith}_4\ \ f_s\ |s_1|\ |s_2|\ |r_1|\ |l_2|) \\
= \quad & \{ \text{ def. of } f_s, \mathsf{zipwith} \ \} \\
& s_1 \uparrow max\ (gemm(\_, +)\ r_1\ l_2) \uparrow s_2 \\
= \quad & \{ \text{ def. of } f'_s, max, \text{ associativity of } \uparrow \ \} \\
& max\ |s_1| \uparrow max\ |s_2| \uparrow max\ (\mathsf{zipwith}\ \ f'_s\ |r_1|\ |l_2|)
\end{aligned}
$$

$$
\begin{aligned}
& max(\mathsf{zipwith}_4\ \ f_s\ (s_1^1 \phi\ s_1^2)\ (s_2^1 \phi\ s_2^2)\ (r_1^1 \phi\ r_1^2)\ (l_2^1 \phi\ l_2^2)) \\
= \quad & \{ \text{ def. of } max, \mathsf{zipwith} \ \} \\
& max(\mathsf{zipwith}_4\ \ f_s\ s_1^1\ s_2^1\ r_1^1\ l_2^1) \uparrow max(\mathsf{zipwith}_4\ \ f_s\ s_1^2\ s_2^2\ r_1^2\ l_2^2) \\
= \quad & \{ \text{ hypo. of induction } \} \\
& max\ s_1^1 \uparrow max\ s_2^1 \uparrow max\ (\mathsf{zipwith}\ \ f'_s\ r_1^1\ l_2^1) \uparrow max\ s_1^2 \uparrow max\ s_2^2 \uparrow max\ (\mathsf{zipwith}\ \ f'_s\ r_1^2\ l_2^2) \\
= \quad & \{ \text{ def. of } f'_s, max, \text{ associativity of } \uparrow \ \} \\
& max\ (s_1^1 \phi\ s_1^2) \uparrow max\ (s_2^1 \phi\ s_2^2) \uparrow max\ (\mathsf{zipwith}\ \ f'_s\ (r_1^1 \phi\ r_1^2)\ (l_2^1 \phi\ l_2^2))
\end{aligned}
$$

The incuntion case for $\ominus$ is proved similarly.

Then, we prove the following equation by the induction on the structure of abide trees.

$$max\ (\mathsf{zipwith}\ \ f'_s\ r_1\ l_2) = max\ (\mathsf{zipwith}(+)\ (\mathsf{map}\ \ \mathsf{reduce}(\uparrow,\_)\ r_1)\ (\mathsf{map}\ \ \mathsf{reduce}(\_,\uparrow)\ l_2))$$

Proof.

$$\begin{array}{ll} & max(\mathsf{zipwith}\ \ f'_s\ |r_1|\ |l_2|) \\ = & \quad \{\text{ def. of } max, \mathsf{zipwith}, f'_s \} \\ & max\ |\max\ (gemm(\_,+)\ r_1\ l_2)| \\ = & \quad \{\text{ below }\} \\ & max\ (|\ \mathsf{reduce}(\uparrow,\_)\ r_1 + \mathsf{reduce}(\uparrow,\_)\ l_2|) \\ = & \quad \{\text{ def. of } \mathsf{zipwith}, \mathsf{map} \} \\ & max\ (\mathsf{zipwith}(+)\ (\mathsf{map}\ \ \mathsf{reduce}(\uparrow,\_)\ |r_1|)\ (\mathsf{map}\ \ \mathsf{reduce}(\_,\uparrow)\ |l_2|)) \end{array}$$

$$\begin{array}{ll} & max(\mathsf{zipwith}\ \ f'_s\ (r_1^1 \phi r_1^2)\ (l_2^1 \phi l_2^2)) \\ = & \quad \{\text{ def. of } max, \mathsf{zipwith} \} \\ & max(\mathsf{zipwith}\ \ f'_s\ r_1^1\ l_2^1) \uparrow max(\mathsf{zipwith}\ \ f'_s\ r_1^2\ l_2^2) \\ = & \quad \{\text{ hypo. of induction }\} \\ & max\ (\mathsf{zipwith}(+)\ (\mathsf{map}\ \ \mathsf{reduce}(\uparrow,\_)\ r_1^1)\ (\mathsf{map}\ \ \mathsf{reduce}(\_,\uparrow)\ l_2^1)) \\ & \qquad \uparrow max\ (\mathsf{zipwith}(+)\ (\mathsf{map}\ \ \mathsf{reduce}(\uparrow,\_)\ r_1^2)\ (\mathsf{map}\ \ \mathsf{reduce}(\_,\uparrow)\ l_2^2)) \\ = & \quad \{\text{ def. of } max, \mathsf{zipwith}, \mathsf{map} \} \\ & max\ (\mathsf{zipwith}(+)\ (\mathsf{map}\ \ \mathsf{reduce}(\uparrow,\_)\ (r_1^1 \phi r_1^2))\ (\mathsf{map}\ \ \mathsf{reduce}(\_,\uparrow)\ (l_2^1 \phi l_2^2))) \end{array}$$

To complete the proof of the base case, we prove the next equation by the induction on the structure of abide trees.

$$\begin{array}{r} max\ (gemm(\_,+)\ r_1\ l_2) = \mathsf{reduce}(\uparrow,\_)\ r_1 + \mathsf{reduce}(\uparrow,\_)\ l_2 \\ \Leftarrow width\ r_1 = 1, height\ l_2 = 1 \end{array}$$

Proof.

$$\begin{array}{ll} & max\ (gemm(\_,+)\ |r_1|\ |l_2|) \\ = & \quad \{\text{ def. of } gemm, max \} \\ & r_1 + l_2 \\ = & \quad \{\text{ def. of } \mathsf{reduce} \} \\ & \mathsf{reduce}(\uparrow,\_)\ |r_1| + \mathsf{reduce}(\uparrow,\_)\ |l_2| \end{array}$$

$$
\begin{aligned}
& \mathit{max}\ (\mathit{gemm}(\_, +)\ (r_1^1 \ominus r_1^2)\ (l_2^1 \phi\, l_2^2)) \\
= \quad & \{ \text{ def. of } \mathit{gemm}, \mathit{max} \ \} \\
& \mathit{max}\ (\mathit{gemm}(\_, +)\ r_1^1\ l_2^1) \uparrow \mathit{max}\ (\mathit{gemm}(\_, +)\ r_1^1\ l_2^2) \\
& \qquad \uparrow \mathit{max}\ (\mathit{gemm}(\_, +)\ r_1^2\ l_2^1) \uparrow \mathit{max}\ (\mathit{gemm}(\_, +)\ r_1^2\ l_2^2) \\
= \quad & \{ \text{ hypo. of induction } \} \\
& (\mathsf{reduce}(\uparrow, \_)\ r_1^1 + \mathsf{reduce}(\uparrow, \_)\ l_2^1) \uparrow (\mathsf{reduce}(\uparrow, \_)\ r_1^1 + \mathsf{reduce}(\uparrow, \_)\ l_2^2) \\
& \qquad \uparrow (\mathsf{reduce}(\uparrow, \_)\ r_1^2 + \mathsf{reduce}(\uparrow, \_)\ l_2^1) \uparrow (\mathsf{reduce}(\uparrow, \_)\ r_1^2 + \mathsf{reduce}(\uparrow, \_)\ l_2^2) \\
= \quad & \{ \text{ associativity and distributivity } \} \\
& (\mathsf{reduce}(\uparrow, \_)\ r_1^1 \uparrow \mathsf{reduce}(\uparrow, \_)\ r_1^2) + (\mathsf{reduce}(\uparrow, \_)\ l_2^1 \uparrow \mathsf{reduce}(\uparrow, \_)\ l_2^2) \\
= \quad & \{ \text{ def. of } \mathsf{reduce} \ \} \\
& \mathsf{reduce}(\uparrow, \_)\ (r_1^1 \ominus r_1^2) + \mathsf{reduce}(\uparrow, \_)\ (l_2^1 \phi\, l_2^2)
\end{aligned}
$$

B.13 Rule XIII

$$
\begin{aligned}
& \mathsf{reduce}(\oplus, \otimes)\ (\mathsf{map}\ \ f\ x) \\
& = f\ (\mathsf{reduce}(\oplus, \otimes)\ x) \Leftarrow f\ a \otimes f\ b = f\ (a \otimes b), f\ a \oplus f\ b = f\ (a \oplus\ b)
\end{aligned}
$$

Proof. The induction on the structure of abide trees.

$$
\begin{aligned}
& \mathsf{reduce}(\oplus, \otimes)\ (\mathsf{map}\ \ f\ |x|) \\
= \quad & \{ \text{ def. of } \mathsf{reduce}, \mathsf{map} \ \} \\
& f\ x \\
= \quad & \{ \text{ def. of } \mathsf{reduce} \ \} \\
& f\ (\mathsf{reduce}(\oplus, \otimes)\ |x|)
\end{aligned}
$$

$$
\begin{aligned}
& \mathsf{reduce}(\oplus, \otimes)\ (\mathsf{map}\ \ f\ (x \ominus y)) \\
= \quad & \{ \text{ def. of } \mathsf{reduce}, \mathsf{map} \ \} \\
& \mathsf{reduce}(\oplus, \otimes)\ (\mathsf{map}\ \ f\ x)) \oplus \mathsf{reduce}(\oplus, \otimes)\ (\mathsf{map}\ \ f\ y) \\
= \quad & \{ \text{ hypo. of induction } \} \\
& f\ (\mathsf{reduce}(\oplus, \otimes)\ x) \oplus f\ (\mathsf{reduce}(\oplus, \otimes)\ y) \\
= \quad & \{ \text{ hypo. } \} \\
& f\ ((\mathsf{reduce}(\oplus, \otimes)\ x) \oplus (\mathsf{reduce}(\oplus, \otimes)\ y)) \\
= \quad & \{ \text{ def. of } \mathsf{reduce} \ \} \\
& f\ (\mathsf{reduce}(\oplus, \otimes)\ (x \phi\, y))
\end{aligned}
$$

The incuntion case for $\ominus$ is proved similarly.

For instance, $\oplus = \_$(don't care), $\otimes = \mathsf{zipwith}(\uparrow)$ and $f = \mathsf{zipwith}(+)\ c_1$ satisfy the condition $f\ a \otimes f\ b = f\ (a \otimes b)$.

B.14 Rule XIV

$$
\begin{aligned}
&\mathsf{reduce}(\oplus,\otimes)\ (\mathsf{zipwith}_4\quad f\ x\ y\ z\ w)\\
&\qquad = f'\ (\mathsf{reduce}(\oplus_1,\otimes_1)\ x)\ (\mathsf{reduce}(\oplus_2,\otimes_2)\ y)\ (\mathsf{reduce}(\oplus_3,\otimes_3)\ z)\ (\mathsf{reduce}(\oplus_4,\otimes_4)\ w)\\
&\quad \Leftarrow f\ a\ b\ c\ d = f'\ a\ b\ c\ d,\\
&\qquad\quad f'\ a\ b\ c\ d \oplus f'\ x\ y\ z\ w = f'\ (a \oplus_1 x)\ (b \oplus_2 y)\ (c \oplus_3 z)\ (d \oplus_4 w)\\
&\qquad\quad f'\ a\ b\ c\ d \otimes f'\ x\ y\ z\ w = f'\ (a \otimes_1 x)\ (b \otimes_2 y)\ (c \otimes_3 z)\ (d \otimes_4 w)
\end{aligned}
$$

Proof. The induction on the structure of abide trees.

$$
\begin{aligned}
&\quad \mathsf{reduce}(\oplus,\otimes)\ (\mathsf{zipwith}_4\quad f\ |x|\ |y|\ |z|\ |w|)\\
&= \quad \{\ \text{def. of } \mathsf{reduce}, \mathsf{zipwith}\ \}\\
&\quad f\ x\ y\ z\ w\\
&= \quad \{\ \text{hypo.}\ \}\\
&\quad f'\ x\ y\ z\ w\\
&= \quad \{\ \text{def. of } \mathsf{reduce}\ \}\\
&\quad f'\ (\mathsf{reduce}(\oplus_1,\otimes_1)\ |x|)\ (\mathsf{reduce}(\oplus_2,\otimes_2)\ |y|)\ (\mathsf{reduce}(\oplus_3,\otimes_3)\ |z|)\ (\mathsf{reduce}(\oplus_4,\otimes_4)\ |w|)
\end{aligned}
$$

$$
\begin{aligned}
&\quad \mathsf{reduce}(\oplus,\otimes)\ (\mathsf{zipwith}_4\quad f\ (x_1 \ominus x_2)\ (y_1 \ominus y_2)\ (z_1 \ominus z_2)\ (w_1 \ominus w_2))\\
&= \quad \{\ \text{def. of } \mathsf{reduce}, \mathsf{zipwith}\ \}\\
&\quad \mathsf{reduce}(\oplus,\otimes)\ (\mathsf{zipwith}_4\quad f\ x_1\ y_1\ z_1\ w_1) \oplus \mathsf{reduce}(\oplus,\otimes)\ (\mathsf{zipwith}_4\quad f\ x_2\ y_2\ z_2\ w_2)\\
&= \quad \{\ \text{hypo. of induction}\ \}\\
&\quad f'\ (\mathsf{reduce}(\oplus_1,\otimes_1)\ x_1)\ (\mathsf{reduce}(\oplus_2,\otimes_2)\ y_1)\ (\mathsf{reduce}(\oplus_3,\otimes_3)\ z_1)\ (\mathsf{reduce}(\oplus_4,\otimes_4)\ w_1)\\
&\qquad \oplus f'\ (\mathsf{reduce}(\oplus_1,\otimes_1)\ x_2)\ (\mathsf{reduce}(\oplus_2,\otimes_2)\ y_2)\ (\mathsf{reduce}(\oplus_3,\otimes_3)\ z_2)\ (\mathsf{reduce}(\oplus_4,\otimes_4)\ w_2)\\
&= \quad \{\ \text{hypo.}\ \}\\
&\quad f'\ (\mathsf{reduce}(\oplus_1,\otimes_1)\ x_1 \oplus_1 \mathsf{reduce}(\oplus_1,\otimes_1)\ x_2)\ (\mathsf{reduce}(\oplus_2,\otimes_2)\ y_1 \oplus_2 \mathsf{reduce}(\oplus_2,\otimes_2)\ y_2)\\
&\qquad\quad (\mathsf{reduce}(\oplus_3,\otimes_3)\ z_1 \oplus_3 \mathsf{reduce}(\oplus_3,\otimes_3)\ z_3)\ (\mathsf{reduce}(\oplus_4,\otimes_4)\ w_1 \oplus_4 \mathsf{reduce}(\oplus_4,\otimes_4)\ w_2)\\
&= \quad \{\ \text{def. of } \mathsf{reduce}\ \}\\
&\quad f'\ (\mathsf{reduce}(\oplus_1,\otimes_1)\ (x_1 \ominus x_2))\ (\mathsf{reduce}(\oplus_2,\otimes_2)\ (y_1 \ominus y_2))\\
&\qquad\qquad\qquad (\mathsf{reduce}(\oplus_3,\otimes_3)\ (z_1 \ominus z_2))\ (\mathsf{reduce}(\oplus_4,\otimes_4)\ (w_1 \ominus w_2))
\end{aligned}
$$

The incuntion case for ϕ is proved similarly.

For instance, $f'\ a\ b\ c\ d = (a \phi\ gemm(\uparrow,+)\ c\ d) \ominus (NIL \phi\ b)$, $\otimes_1 = \mathsf{zipwith}(\uparrow)$, $\otimes_2 = \mathsf{zipwith}(\uparrow)$, $\otimes_3 = \phi$ and $\otimes_4 = \ominus$ satisfy the condition for $f\ a\ b\ c\ d = (a \phi\ gemm(\_,+)\ c\ d) \ominus (NIL \phi\ b)$, $\otimes = \mathsf{zipwith}(\uparrow)$ and $\oplus = \_$.

B.15 Rule XV

$$
\begin{aligned}
&\mathsf{map}\ (\mathsf{reduce}(\oplus,\_))\ (gemm(\_,\mathsf{zipwith}(\otimes))\ x\ y)\\
&= gemm(\oplus,\otimes)\ (tr\ (\mathsf{reduce}\ (\phi,\_))\ x)\ (\mathsf{reduce}\ (\_,\phi)\ y)\\
&\Leftarrow width \text{ of } x \text{ and its elements} = 1, width \text{ of } y\text{'s elements} = 1, height\ y = 1
\end{aligned}
$$

Proof. The induction on the structure of abide trees.

$$
\begin{array}{ll}
& \mathsf{map}\ (\mathsf{reduce}(\oplus,\_))\ (\mathit{gemm}(\_,\mathsf{zipwith}(\otimes))\ |x|\ |y|) \\
= & \quad \{\ \text{def. of } \mathsf{map}, \mathit{gemm}\ \} \\
& |\,\mathsf{reduce}(\oplus,\_)\ x\ y| \\
= & \quad \{\ \text{below}\ \} \\
& \mathit{gemm}\ (\oplus,\otimes)\ (\mathit{tr}\ x)\ y \\
= & \quad \{\ \text{def. of } \mathsf{reduce}\ \} \\
& \mathit{gemm}\ (\oplus,\otimes)\ (\mathit{tr}\ (\mathsf{reduce}\ (\phi,\_)\ |x|))\ (\mathsf{reduce}\ (\_,\phi)\ |y|)
\end{array}
$$

$$
\begin{array}{ll}
& \mathsf{map}\ (\mathsf{reduce}(\oplus,\_))\ (\mathit{gemm}(\_,\mathsf{zipwith}(\otimes))\ (x_1 \ominus x_2)\ (y_1 \phi\ y_2)) \\
= & \quad \{\ \text{def. of } \mathsf{map}, \mathit{gemm}\ \} \\
& (\mathsf{map}\ (\mathsf{reduce}(\oplus,\_))\ (\mathit{gemm}(\_,\mathsf{zipwith}(\otimes))\ x_1\ y_1) \phi\ \mathsf{map}\ (\mathsf{reduce}(\oplus,\_))\ (\mathit{gemm}(\_,\mathsf{zipwith}(\otimes))\ x_1\ y_2)) \\
& \qquad \ominus (\mathsf{map}\ (\mathsf{reduce}(\oplus,\_))\ (\mathit{gemm}(\_,\mathsf{zipwith}(\otimes))\ x_2\ y_1) \phi\ \mathsf{map}\ (\mathsf{reduce}(\oplus,\_))\ (\mathit{gemm}(\_,\mathsf{zipwith}(\otimes))\ x_2\ y_2)) \\
= & \quad \{\ \text{hypo. of induction}\ \} \\
& (\mathit{gemm}\ (\oplus,\otimes)\ (\mathit{tr}\ (\mathsf{reduce}\ (\phi,\_)\ x_1))\ (\mathsf{reduce}\ (\_,\phi)\ y_1) \\
& \quad \phi\ \mathit{gemm}\ (\oplus,\otimes)\ (\mathit{tr}\ (\mathsf{reduce}\ (\phi,\_)\ x_1))\ (\mathsf{reduce}\ (\_,\phi)\ y_2)) \\
& \qquad \ominus (\mathit{gemm}\ (\oplus,\otimes)\ (\mathit{tr}\ (\mathsf{reduce}\ (\phi,\_)\ x_2))\ (\mathsf{reduce}\ (\_,\phi)\ y_1) \\
& \qquad\quad \phi\ \mathit{gemm}\ (\oplus,\otimes)\ (\mathit{tr}\ (\mathsf{reduce}\ (\phi,\_)\ x_2))\ (\mathsf{reduce}\ (\_,\phi)\ y_2)) \\
= & \quad \{\ \text{def. of } \mathit{gemm}\ \} \\
& \mathit{gemm}\ (\oplus,\otimes)\ (\mathit{tr}\ (\mathsf{reduce}\ (\phi,\_)\ x_1) \phi\ \mathit{tr}\ (\mathsf{reduce}\ (\phi,\_)\ x_2))\ (\mathsf{reduce}\ (\_,\phi)\ y_1 \phi\ \mathsf{reduce}\ (\_,\phi)\ y_2)) \\
= & \quad \{\ \text{def. of } \mathit{tr}, \mathsf{reduce}\ \} \\
& \mathit{gemm}\ (\oplus,\otimes)\ (\mathit{tr}\ (\mathsf{reduce}\ (\phi,\_)\ (x_1 \ominus x_2)))\ (\mathsf{reduce}\ (\_,\phi)\ (y_1 \phi\ y_2))
\end{array}
$$

To complete the proof, we prove the following equation by the induction on the structure of abide trees.

$$
\begin{array}{l}
|\,\mathsf{reduce}(\oplus,\_)\ (\mathsf{zipwith}(\otimes)\ x\ y)| = \mathit{gemm}(\oplus,\otimes)\ (\mathit{tr}\ x)\ y \\
\Leftarrow \mathit{width}\ x = 1, \mathit{width}\ y = 1
\end{array}
$$

Proof.

$$
\begin{array}{ll}
& |\,\mathsf{reduce}(\oplus,\_)\ (\mathsf{zipwith}(\otimes)\ |x|\ |y|)| \\
= & \quad \{\ \text{def. of } \mathsf{zipwith}, \mathsf{reduce}\ \} \\
& |x \otimes y| \\
= & \quad \{\ \text{def. of } \mathit{gemm}, \mathit{tr}\ \} \\
& \mathit{gemm}(\oplus,\otimes)\ (\mathit{tr}\ |x|)\ |y|
\end{array}
$$

$$
\begin{aligned}
& |\,\mathsf{reduce}(\oplus,\_)\ (\mathsf{zipwith}(\otimes)\ (x_1 \mathbin{⦵} x_2)\ (y_1 \mathbin{⦵} y_2))| \\
= & \quad \{ \text{ def. of } \mathsf{zipwith}, \mathsf{reduce} \ \} \\
& |\,\mathsf{reduce}(\oplus,\_)\ (\mathsf{zipwith}(\otimes)\ x_1\ y_1) \oplus \mathsf{reduce}(\oplus,\_)\ (\mathsf{zipwith}(\otimes)\ x_2\ y_2)| \\
= & \quad \{ \text{ def. of } \mathsf{zipwith} \ \} \\
& \mathsf{zipwith}(\oplus)\ |\,\mathsf{reduce}(\oplus,\_)\ (\mathsf{zipwith}(\otimes)\ x_1\ y_1)|\ |\,\mathsf{reduce}(\oplus,\_)\ (\mathsf{zipwith}(\otimes)\ x_2\ y_2)| \\
= & \quad \{ \text{ hypo. of induction } \} \\
& \mathsf{zipwith}(\oplus)\ (\mathit{gemm}(\oplus,\otimes)\ (\mathit{tr}\ x_1)\ y_1)\ (\mathit{gemm}(\oplus,\otimes)\ (\mathit{tr}\ x_2)\ y_2) \\
= & \quad \{ \text{ def. of } \mathit{gemm}, \mathit{tr} \ \} \\
& \mathit{gemm}(\oplus,\otimes)\ (\mathit{tr}\ (x_1 \mathbin{⦵} x_2))\ (y_1 \mathbin{⦵} y_2)
\end{aligned}
$$

B.16 Rule XVI

$$
\begin{aligned}
& \mathsf{map}\ (\mathsf{reduce}(\_,\oplus))\ (\mathit{gemm}(\_,\mathsf{zipwith}(\otimes))\ x\ y) \\
& = \mathit{gemm}(\oplus,\otimes)\ (\mathsf{reduce}\ (⦵,\_)\ x)\ (\mathit{tr}(\mathsf{reduce}\ (\_,⦵)\ y)) \\
& \Leftarrow \mathit{width}\ x = 1, \mathit{height} \text{ of } x\text{'s elements} = 1, \mathit{height} \text{ of } y \text{ and its elements} = 1
\end{aligned}
$$

Proof. Silimar to Rule XV.

B.17 Rule XVII

$$
\begin{aligned}
& \mathsf{map}(\mathsf{reduce}(\uparrow,\_))\ (\mathsf{zipwith}_3\ \ f_r\ r_1\ r_2\ ro_2) = \mathsf{zipwith}_3\ \ f'_r\ (\mathsf{reduce}\ (\uparrow)\ r_1)\ ro_2\ (\mathsf{reduce}\ (\uparrow,\_) r_2) \\
& \textbf{where}\ \ f_r\ r_1\ r_2\ ro_2 = \mathsf{map}\ (+ro_2)\ r_1 \mathbin{⦶} r_2 \\
& \qquad\quad f'_r\ r_1\ ro_2\ r_2 = (r_1 + ro_2) \uparrow\ r_2
\end{aligned}
$$

Proof. Rule VII and follwing calculation.

$$
\begin{aligned}
& \mathsf{reduce}(\uparrow,\_)\ (f_r\ r_1\ r_2\ ro_2) \\
= & \quad \{ \text{ def. of } f_r \ \} \\
& \mathsf{reduce}(\uparrow,\_)\ ((\mathsf{map}\ (+ro_2) r_1) \mathbin{⦶} r_2) \\
= & \quad \{ \text{ def. of } \mathsf{reduce} \ \} \\
& \mathsf{reduce}(\uparrow,\_)\ (\mathsf{map}\ (+ro_2) r_1) \uparrow r_2 \\
= & \quad \{ + \text{ distributes over } \uparrow \ \} \\
& ((\mathsf{reduce}(\uparrow,\_)\ r_1) + ro_2) \uparrow r_2
\end{aligned}
$$

B.18 Rule XVIII

$$
\begin{aligned}
& \mathsf{reduce}(\_,⦶)\ (\mathsf{map}\ (\mathsf{zipwith}(+)\ (\mathit{right}'\ x))\ y) \\
& = \mathsf{map}_c\ (\mathsf{zipwith}(+)\ (\mathit{right}\ (\mathsf{reduce}(\_,⦶)\ x)))\ (\mathsf{reduce}(\_,⦶)\ y) \\
& \Leftarrow \mathit{height}\ x = 1, \mathit{width} \text{ of } x\text{'s elements} = 1
\end{aligned}
$$

Proof. First, we prove the next equation by the induction on the structure of abide trees.

$$
\begin{aligned}
& \mathsf{reduce}(\_,⦶)\ (\mathsf{map}\ \ f\ \ x) = \mathsf{map}_c\ \ f\ \ (\mathsf{reduce}(\_,⦶)\ x) \\
& \Leftarrow \mathit{height}\ x = 1, \mathit{width} \text{ of } x\text{'s elements} = 1
\end{aligned}
$$

Proof.

$$
\begin{aligned}
& \mathsf{reduce}(\_, \phi)\ (\mathsf{map}\ \ f\ |x|) \\
= & \quad \{ \text{ def. of } \mathsf{reduce}, \mathsf{map} \} \\
& f\ x \\
= & \quad \{ \text{ def. of } \mathsf{map}_c,\ \mathit{height}\ x = 1 \} \\
& \mathsf{map}_c\ \ f\ x \\
= & \quad \{ \text{ def. of } \mathsf{reduce} \} \\
& \mathsf{map}_c\ \ f\ (\mathsf{reduce}(\_, \phi)\ |x|)
\end{aligned}
$$

$$
\begin{aligned}
& \mathsf{reduce}(\_, \phi)\ (\mathsf{map}\ \ f\ (x_1 \phi x_2)) \\
= & \quad \{ \text{ def. of } \mathsf{reduce}, \mathsf{map} \} \\
& \mathsf{map}\ \ f\ x_1 \phi \mathsf{map}\ \ f\ x_2 \\
= & \quad \{ \text{ hypo. of induction } \} \\
& \mathsf{map}_c\ \ f\ (\mathsf{reduce}(\_, \phi)\ x_1) \phi \mathsf{map}_c\ \ f\ (\mathsf{reduce}(\_, \phi)\ x_2) \\
= & \quad \{ \text{ def. of } \mathsf{map}_c \} \\
& \mathsf{map}_c\ \ f\ (\mathsf{reduce}(\_, \phi)\ (x_1 \phi x_2))
\end{aligned}
$$

To complete the proof, we prove the next equation by the induction on the structure of abide trees.

$$
\begin{aligned}
& \mathit{right}'\ x = \mathit{right}\ (\mathsf{reduce}(\_, \phi)\ x) \\
& \Leftarrow \mathit{height}\ x = 1, \mathit{width} \text{ of } x\text{'s elements} = 1
\end{aligned}
$$

Proof.

$$
\begin{aligned}
& \mathit{right}'\ |x| \\
= & \quad \{ \text{ def. of } \mathit{right}' \} \\
& x \\
= & \quad \{ \text{ def. of } \mathit{right},\ \mathit{width}\ x = 1 \} \\
& \mathit{right}\ x \\
= & \quad \{ \text{ def. of } \mathsf{reduce} \} \\
& \mathit{right}\ (\mathsf{reduce}(\_, \phi)\ |x|)
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{right}'\ (x_1 \phi x_2) \\
= & \quad \{ \text{ def. of } \mathit{right}' \} \\
& \mathit{right}'\ x_2 \\
= & \quad \{ \text{ hypo. of induction } \} \\
& \mathit{right}\ (\mathsf{reduce}(\_, \phi)\ x_2) \\
= & \quad \{ \text{ def. of } \mathit{right} \} \\
& \mathit{right}\ (\mathsf{reduce}(\_, \phi)\ x_1 \phi \mathsf{reduce}(\_, \phi)\ x_2) \\
= & \quad \{ \text{ def. of } \mathsf{reduce} \} \\
& \mathit{right}\ (\mathsf{reduce}(\_, \phi)\ (x_1 \phi x_2))
\end{aligned}
$$

B.19 Rule XIX

$$\begin{aligned}&\mathsf{reduce}(\phi,\_)\ (\mathsf{map}\ (\mathsf{zipwith}(+)\ (top'\ x))\ y)\\ &= \mathsf{map}_c\ (\mathsf{zipwith}(+)\ (top\ (\mathsf{reduce}(\phi,\_)\ x)))\ (\mathsf{reduce}(\phi,\_)\ y)\\ &\Leftarrow width\ x = 1, width \text{ of } x\text{'s elements} = 1\end{aligned}$$

Proof. Similar to Rule XVIII.

B.20 Rule XX

$$\begin{aligned}&\mathsf{reduce}(\_,\phi)\ (\mathsf{zipwith}\ \ f\ x\ y)\\ &= \mathsf{map}_r\ (\mathsf{zipwith}(+)\ (top\ (\mathsf{reduce}(\_,\phi)\ y)))\ (\mathsf{reduce}(\phi,\_)\ x) \ominus (\mathsf{reduce}(\phi,\_)\ y)\\ &\Leftarrow height\ x = 1, height\ y = 1, width \text{ of } x \text{ and } y\text{'s elements} = 1\\ &\quad f\ x\ y = \mathsf{map}\ (+(top'\ y))\ x \ominus y\end{aligned}$$

Proof. Rule XIV with $f'\ a\ b = \mathsf{map}_r\ (\mathsf{zipwith}(+)\ (top\ b))\ a \ominus\ b$ and $\otimes = \phi$, $\otimes_1 = \phi$, $\otimes_2 = \phi$.

B.21 Rule XXI

$$\begin{aligned}&\mathsf{reduce}(\_,\phi)\ (\mathsf{zipwith}\ \ f\ x\ y)\\ &= \mathsf{map}_r\ (\mathsf{zipwith}(+)\ (top\ (\mathsf{reduce}(\_,\phi)\ y)))\ (\mathsf{reduce}(\phi,\_)\ x) \ominus (\mathsf{reduce}(\phi,\_)\ y)\\ &\Leftarrow width\ x = 1, width\ y = 1, width \text{ of } x \text{ and } y\text{'s elements} = 1\\ &\quad f\ x\ y = \mathsf{map}\ (+(top'\ y))\ x \ominus y\end{aligned}$$

Proof. Rule XIV with $f'\ a\ b = \mathsf{map}_r\ (\mathsf{zipwith}(+)\ (top\ b))\ a \ominus\ b$ and $\oplus = \phi$, $\oplus_1 = \phi$, $\oplus_2 = \phi$.