Surrounding Theorem: Developing Parallel Programs for Matrix-Convolutions

Kento Emoto, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi

Department of Mathematical Informatics,
University of Tokyo
{emoto,kmatsu,hu,takeichi}@ipl.t.u-tokyo.ac.jp

Abstract. Computations on two-dimensional arrays such as matrices and images are one of the most fundamental and ubiquitous things in computational science and its vast application areas, but development of efficient parallel programs on two-dimensional arrays is known to be hard. To solve this problem, we have proposed the skeletal framework on two-dimensional arrays based on the theory of constructive algorithmics. It supports users, even with little knowledge about parallel machines, to develop systematically both correct and efficient parallel programs on two-dimensional arrays. In this paper, we apply our framework to the matrix-convolutions often used in image filters and difference methods. We show the efficacy of the framework by giving a general parallel program described with the skeletons for the matrix-convolutions, and a theorem that optimizes the general program into an application-specific one.

1 Introduction

Computations on two-dimensional arrays, such as matrix computations, image processing, and difference methods, are both fundamental and ubiquitous in scientific computations and other application areas [6, 10, 11]. However, development of efficient parallel programs on two-dimensional arrays is known to be a hard task due to the necessity of considering data allocation, synchronization and communication between processors. *Skeletal parallel programming* is one promising solution to the situation [4, 12]. In this model, users build parallel programs by composing ready-made components (called *skeletons*) implemented efficiently in parallel for various parallel architectures. Since low-level parallelism is concealed in the skeletons, users can obtain a comparatively efficient parallel program without needing technical details of parallel computers or being conscious of parallelism explicitly.

We have proposed a skeletal framework on two-dimensional arrays [8], based on the theory of constructive algorithmics (also known as *Bird-Meertens Formalism*) [1,3,13]. Our framework provides users, even with little knowledge about parallel machines, a concise way to describe safe and efficient parallel computation over two-dimensional arrays, and theories for deriving and optimizing programs. The main features of our framework are: (1) *a novel use of the abide-tree representation* [1] in developing parallel programs for manipulating two-dimensional arrays; (2) *a strong support* for systematic development of both efficient and correct parallel programs on two-dimensional arrays in a highly abstract way; (3) *an efficient implementation* of basic skeletons in

C++ and MPI on PC clusters, guaranteeing that programs composed with these parallel skeletons can run efficiently in parallel. In this framework, users write an easy and general program that covers a class of problems, derive its efficient version using general techniques such as fusion, tupling and generalization, summarize it as a theorem (tool), and then instantiate it to solve concrete problems. These general techniques have already been developed in the framework, however, domain-specific tools are not so much presented.

In this paper, we give an domain-specific tool and show the efficacy of the framework. We focus on a set of computations on two-dimensional arrays known as matrix-convolutions [**?**], in which each element in the resulting array depends on its surrounding elements. This set of computations includes important and fundamental problems such as image filters, difference methods and the N-body problem (although this last problem seems more difficult than the others, it merely sees not only the nearest neighbours but all the surrounding elements.) The most general form $mconv$ is described with three components in our framework:

$$mconv\ f\ shrink = \mathsf{map}\ \ f \circ \mathsf{map}\ \ shrink \circ surrounds\ .$$

Here, $surrounds$ gathers all the surrounding elements for each element, $shrink$ picks the necessary parts up from those gathered elements, and f calculates the resulting element from them. This general form is parameterized by the two functions $shrink$ and f, and users can solve many problems by specifying suitable ones. For example, users can develop a sharpen-filter by choosing the function $shrink$ that reduces the surroundings into 3×3 matrix, and the function f that calculates the weighted sum of them. We can further optimize instances of the general program to application-specific ones with the *surrounding theorem*.

The main contribution of this paper is as follows.

- We show the general parallel program for the matrix-convolution described with parallel skeletons. Users can solve their problems as an instance.
- We give the *surrounding theorem* which enables users to get an efficient program easily. The experimental results show that the derived program can be executed efficiently in parallel.

Technical details of this paper is available in the master's thesis [7].

2 Notations

Notation in this paper follows that of Haskell [2], a pure functional language that can describe both algorithms and algorithmic transformation concisely.

Function application is denoted by a space and the argument may be written without brackets. Thus, $f\ a$ means $f(a)$ in ordinary notation. Functions are curried, i.e. functions take one argument and return a function or a value, and the function application associates to the left. Thus, $f\ a\ b$ means $(f\ a)\ b$. The function application binds more strongly than any other operator, so $f\ a \otimes b$ means $(f\ a) \otimes b$, but not $f\ (a \otimes b)$. Function composition is denoted by $\circ$, so $(f \circ g)\ x = f\ (g\ x)$ from its definition. Binary operators can be used as functions by sectioning as follows: $a \oplus b = (a\oplus)\,b = (\oplus b)\,a = (\oplus)\,a\,b$. Two binary operators $\ll$ and $\gg$ are defined by $a \ll b = a$, $a \gg b = b$. Pairs are Cartesian products of plural data, written like (x, y). A function that applies functions f and g respectively to the elements of a pair (x, y) is denoted by $(f \times g)$. Thus, $(f \times g)\,(x, y) = (f\,x, g\,y)$.

3 Skeletal Framework on Two-Dimensional Arrays

In this section, we introduce our parallel skeletal framework on two-dimensional arrays [8] based on the theory of constructive algorithmics [1, 3, 13].

3.1 Abide-trees for Two-Dimensional Arrays

To represent two-dimensional arrays, we define the following abide-trees, which are built up by three constructors $|\cdot|$ (singleton), $\ominus$ (above) and ϕ (beside) following the idea in [1].

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

Here, $|\cdot|\ a$, or abbreviated as $|a|$, means a singleton array of a, i.e. a two-dimensional array with a single element a. For two-dimensional arrays x and y of the same width, $x \ominus y$ means that x is located above y. Similarly, for two-dimensional arrays x and y of the same height, $x \phi y$ means that x is located on the left of y. Moreover, $\ominus$ and ϕ are associative binary operators and satisfy the following *abide* (a coined term from $\underline{\text{ab}}$ove and bes$\underline{\text{ide}}$) property.

Definition 1 (Abide Property). *Two binary operators $\oplus$ and $\otimes$ are said to satisfy the abide property or to be abiding, if the following equation is satisfied:*

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

In the rest of the paper, we will assume that x has the same width of y when $x \ominus y$ appears, and that u has the same height of v for $u \phi v$.

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

$$\begin{pmatrix}1\ 2\\3\ 4\end{pmatrix} \Rightarrow \begin{cases}(|1| \phi |2|) \ominus (|3| \phi |4|)\\ (|1| \ominus |3|) \phi (|2| \ominus |4|)\end{cases}$$

This is in sharp contrast to the quadtree representation of matrices [9], which does not allow such freedom.

From the theory of constructive algorithmics [3], it follows that each constructively built-up data structure (i.e., algebraic data structure) is equipped with a powerful computation pattern called homomorphism.

$$\mathsf{map}\ f \begin{pmatrix} x_{11} & \cdots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{m1} & \cdots & x_{mn} \end{pmatrix} = \begin{pmatrix} f\ x_{11} & \cdots & f\ x_{1n} \\ \vdots & \ddots & \vdots \\ f\ x_{m1} & \cdots & f\ x_{mn} \end{pmatrix}$$

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

$$\mathsf{zipwith}\ f \begin{pmatrix} x_{11} & \cdots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{m1} & \cdots & x_{mn} \end{pmatrix} \begin{pmatrix} y_{11} & \cdots & y_{1n} \\ \vdots & \ddots & \vdots \\ y_{m1} & \cdots & y_{mn} \end{pmatrix} = \begin{pmatrix} f\ x_{11}\ y_{11} & \cdots & f\ x_{1n}\ y_{1n} \\ \vdots & \ddots & \vdots \\ f\ x_{m1}\ y_{m1} & \cdots & f\ x_{mn}\ y_{mn} \end{pmatrix}$$

$$\mathsf{scan}(\oplus,\otimes) \begin{pmatrix} x_{11} & \cdots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{m1} & \cdots & x_{mn} \end{pmatrix} = \begin{pmatrix} y_{11} & \cdots & y_{1n} \\ \vdots & \ddots & \vdots \\ y_{m1} & \cdots & y_{mn} \end{pmatrix} \ \textbf{where}\ y_{ij} = \begin{array}{c} (x_{11} \otimes \cdots \otimes x_{1j}) \oplus \\ \ddots \\ (x_{i1} \otimes \cdots \otimes x_{ij}) \end{array}$$

$$\mathsf{scanr}(\oplus,\otimes) \begin{pmatrix} x_{11} & \cdots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{m1} & \cdots & x_{mn} \end{pmatrix} = \begin{pmatrix} z_{11} & \cdots & z_{1n} \\ \vdots & \ddots & \vdots \\ z_{m1} & \cdots & z_{mn} \end{pmatrix} \ \textbf{where}\ z_{ij} = \begin{array}{c} (x_{ij} \otimes \cdots \otimes x_{in}) \oplus \\ \ddots \\ (x_{mj} \otimes \cdots \otimes x_{mn}) \end{array}$$

Fig. 1. Intuitive Definition of Parallel Skeletons on Two-Dimensional Arrays

Definition 2 ((Abide-tree) Homomorphism). *A function h is said to be an abide-tree homomorphism, if it is defined as follows for a function f and binary operators $\oplus, \otimes$.*

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

For notational convenience, we write $(\!| f, \oplus, \otimes |\!)$ to denote h. When it is clear from the context, we just call $(\!| f, \oplus, \otimes |\!)$ homomorphism. Note that $\oplus$ and $\otimes$ in $(\!| f, \oplus, \otimes |\!)$ should be associative and satisfy the abide property, inheriting the properties of $\ominus$ and Φ.

Intuitively, a homomorphism $(\!| f, \oplus, \otimes |\!)$ is a function to replace the constructors $|\cdot|$, $\ominus$ and Φ in an input abide-tree by f, $\oplus$ and $\otimes$ respectively.

3.2 Parallel Skeletons on Two-Dimensional Arrays

We introduce the parallel skeletons map, reduce, zipwith, scan and scanr for manipulating two-dimensional arrays. In the theory of Constructive Algorithmics [1, 3, 13], 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. Intuitive definitions of the skeletons are shown in Fig. 1. All the skeletons are implemented efficiently in parallel and their costs are shown in Table 1.

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

Table 1. Parallel Complexity of the Skeletons for a Two-Dimensional Array of $n \times n$

	P processors	n^2 processors
map, zipwith	$\mathrm{O}(n^2/P)$	$\mathrm{O}(1)$
reduce	$\mathrm{O}(n^2/P + \log P)$	$\mathrm{O}(\log n)$
scan, scanr	$\mathrm{O}(n^2/P + \sqrt{n^2/P}\log P)$	$\mathrm{O}(\log n)$

the shape of the structure. The skeleton reduce collapses a two-dimensional array to a value using two abiding binary operators $\oplus$ and $\otimes$. They are defined formally as map $f = (\!|\, |\cdot| \circ f, \ominus, \phi \,|\!)$, and reduce$(\oplus, \otimes) = (\!|\, id, \oplus, \otimes \,|\!)$.

The skeleton zipwith, an extension of map, takes two arrays of the same shape, applies a function f to corresponding elements of the arrays and returns a new array of the same shape. The skeletons scan and scanr, extensions of reduce, hold all values generated in reducing an array by reduce. The scan generates the result of reducing the upper-left subarray, while the scanr generates that of the lower-right subarray. We omit the formal definition of zipwith, scan and scanr for the space limitation.

4 Developing Parallel Programs for Matrix-Convolutions

In this section, focusing on the matrix-convolutions such as image filters and difference methods, we give the general form described with parallel skeletons, and then give the theorem to get optimized program from the general form.

The matrix-convolution is computation in which each element of the resulting array depends on the surrounding elements. For example, the sharpen-filter that sharpens the input image is one instance of the matrix-convolution. A pixel of the resulting image is the weighted sum of the surrounding pixels of the input image. Similarly, the difference method is another instance of matrix-convolution since it calculates the new value of each point from the old values of the surrounding points. We show a code in C++ for the sharpen-filter in Fig. 2, to give a concrete image of the problems dealt with here.

The idea of our general from is illustrated in Fig. 3 that shows an image of execution of the sharpen-filter: (1) *surrounds* gathers all the surrounding elements for each element, (2) *shrink* picks he necessary parts up from those gathered elements, and (3) f calculates the resulting element from them. This general form has clear correspondences to the code in Fig. 2. The function f corresponds to `f` of the code, *shrink* corresponds to which elements are the arguments passed to f, and *surrounds* corresponds to `for`-loops. Thus, users can easily write their programs using the general form.

4.1 A General Form Described with Parallel Skeletons

As argued in the introduction, the most general form of this kind of computation is thought to consist of three components: gathering all the surrounding elements of each element to it, shrinking those to the necessary amount, and applying a function to get a new element from them. Thus, the program is described as follows:

$$mconv\ f\ shrink = \mathsf{map}\ f \circ \mathsf{map}\ shrink \circ surrounds\ .$$

```
int sharpen_filter(int **b, int **a, int n, int m){
  for(int i = 0; i < m; i++)
    for(int j = 0; j < n; j++)
      b[i][j] = f(a[i][j], a[i-1][j], a[i+1][j], a[i][j+1], a[i][j-1],
                  a[i-1][j+1], a[i-1][j-1], a[i+1][j+1], a[i+1][j-1]);
}
int f(int c, int n, int s, int e, int w, int ne, int nw, int se, int sw){
  return 5*c + (-1)*n + (-1)*s + (-1)*e + (-1)*w + 0*ne + 0*nw + 0*se + 0*sw;}
```

Fig. 2. C++ Code of the Sharpen Filter (Sequential Program)

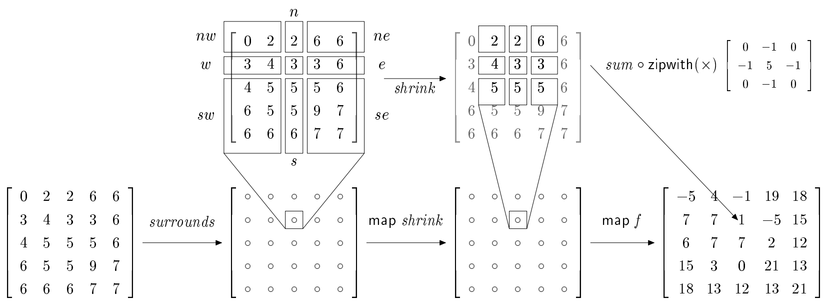

Fig. 3. An Image of the Sharpen Filter in the General Program

This general form is parameterized by the two functions $shrink$ and f, and users can solve many problems by specifying application-specific ones, as shown below. The function $surrounds$, which is commonly used in those problems, has two-phase calculation as follows: (1) calculation of the parts of the north-west (i.e. c, n, w and nw) by scan, and (2) that of the other parts by scanr. Its definition is as follows.

$$surrounds = \mathsf{scanr}(\oplus_r, \otimes_r) \circ \mathsf{map}\ f_r \circ \mathsf{scan}(\oplus_f, \otimes_f) \circ \mathsf{map}\ f_f$$

where

$$f_f\ a = (a, Nil, Nil, Nil)$$

$$(c_a, n_a, w_a, nw_a) \oplus_f (c_b, n_b, w_b, nw_b) = (\underbrace{c_b}_{c}, \underbrace{n_a \ominus |c_a| \ominus n_b}_{n}, \underbrace{w_b}_{w}, \underbrace{nw_a \ominus w_a \ominus nw_b}_{nw})$$

$$(c_a, n_a, w_a, nw_a) \otimes_f (c_b, n_b, w_b, nw_b) = (\underbrace{c_b}_{c}, \underbrace{n_b}_{n}, \underbrace{w_a \phi |c_a| \phi w_b}_{w}, \underbrace{nw_a \phi n_a \phi nw_b}_{nw})$$

$$f_r\ (c, n, w, nw) = (c, n, Nil, Nil, w, Nil, nw, Nil, Nil)$$

$$(c_a, n_a, s_a, e_a, w_a, ne_a, nw_a, se_a, sw_a) \oplus_r (c_b, n_b, s_b, e_b, w_b, ne_b, nw_b, se_b, sw_b)$$
$$= (\underbrace{c_a}_{c}, \underbrace{n_a}_{n}, \underbrace{s_a \ominus |c_b| \ominus s_b}_{s}, \underbrace{e_a}_{e}, \underbrace{w_a}_{w}, \underbrace{ne_a}_{ne}, \underbrace{nw_a}_{nw}, \underbrace{se_a \ominus e_b \ominus se_b}_{se}, \underbrace{sw_a \ominus w_b \ominus sw_b}_{sw})$$

$$(c_a, n_a, s_a, e_a, w_a, ne_a, nw_a, se_a, sw_a) \otimes_r (c_b, n_b, s_b, e_b, w_b, ne_b, nw_b, se_b, sw_b)$$
$$= (\underbrace{c_a}_{c}, \underbrace{n_a}_{n}, \underbrace{s_a}_{s}, \underbrace{e_a \phi |c_b| \phi e_b}_{e}, \underbrace{w_a}_{w}, \underbrace{ne_a \phi n_b \phi ne_b}_{ne}, \underbrace{nw_a}_{nw}, \underbrace{se_a \phi s_b \phi se_b}_{se}, \underbrace{sw_a}_{sw})$$

Here, Nil is a special value to indicate that there is no value, and we treat it as an identity of $\ominus$ and ϕ for simplification of the notation. Thus, $Nil \ominus x = x$, $x \ominus Nil = x$, $Nil \phi x = x$, and $x \phi Nil = x$. Each element of the resulting array is a tuple of nine elements. The meaning of each element of the tuple is as follows: c is the center element; s is an array of the elements on the south of the element; similarly n, e and w are arrays of the elements on the north, east and west respectively; ne, nw, se and sw are arrays of the elements on the north-east, north-west, south-east and south-west. Note that this $surrounds$ needs $\mathrm{O}(n^4)$ memory space for a matrix of $n \times n$.

We show some examples written with the general form.

$$\begin{aligned}
&\mathit{imagefilter}\ \mathit{ker} = mconv\ (\mathit{conv}\ \mathit{ker})\ \mathit{shrink}_1\\
&\mathit{FDM}\ n\ \mathit{ker} \quad = \mathit{iter}\ n\ (mconv\ (\mathit{conv}\ \mathit{ker})\ \mathit{shrink}_1)\\
&\quad \textbf{where}\\
&\quad \mathit{shrink}_1 = \mathit{id} \times B \times T \times L \times R \times \mathit{BL} \times \mathit{BR} \times \mathit{TL} \times \mathit{TR}\\
&\quad B\ = (\!|\ |\cdot|, \gg, \phi\ |\!),\ \ T\ = (\!|\ |\cdot|, \ll, \phi\ |\!),\ \ L\ = (\!|\ |\cdot|, \ominus, \ll\ |\!),\ \ R\ = (\!|\ |\cdot|, \ominus, \gg\ |\!),\\
&\quad \mathit{BL} = (\!|\ |\cdot|, \gg, \ll\ |\!), \mathit{BR} = (\!|\ |\cdot|, \gg, \gg\ |\!), \mathit{TL} = (\!|\ |\cdot|, \ll, \ll\ |\!), \mathit{TR} = (\!|\ |\cdot|, \ll, \gg\ |\!)
\end{aligned}$$

The function $imagefilter\ ker$ is an image filter with the coefficient matrix ker, which is used to weighted sum of the surrounding pixels. The $shrink_1$ reduces each part of the gathered surrounding elements to the element closest to the center, and the function $conv\ ker$ calculates the weighted sum of them. The functions B and T take the bottom row and the top row of the input array respectively. Similarly, each of L, R, BL, BR, TL and TR takes corresponding part of the input array. Figure 3 shows an image of execution of the sharpen-filter by the above general program. The function $FDM\ n\ ker$ performs the finite difference method, where $iter$ is an iteration function and each iteration step is the same as image filters with specific coefficients.

The following example calculates the array of which element at (i, j) is the maximum in the i-th row and the j-th column, i.e. the maximum in the cross. The $shrink_{max}$ reduces each part of the gathered surrounding elements to the biggest element in the part, where the binary operator $\uparrow$ takes the bigger element. The function max_5 takes the maximum of the column and the row including the center element.

$$\begin{aligned}
&crossmax = mconv\ max_5\ shrink_{max}\\
&\quad \textbf{where}\ shrink_{max} = max \times \cdots \times max\\
&\qquad\qquad max = (\!|id, \uparrow, \uparrow|\!)\\
&\qquad\qquad max_5\ (c, n, s, e, w, \_, \_, \_, \_) = c \uparrow n \uparrow s \uparrow e \uparrow w
\end{aligned}$$

As shown in this example, $shrink$ is allowed not only to shrink the shape of the surroundings but to perform some calculation.

4.2 Surrounding Theorem

In this section, we give the theorem to optimize the general form by fusing $shrink$ to $surrounds$.

Image filters and difference methods usually have the $shrink$ of the fixed size window that takes the fixed-size rectangle region (window) of the surrounding elements. The function that takes a fixed number of columns (rows) can be written as a homomorphism. For example, the function $right = (\!|\ |\cdot|, \ominus, \gg\ |\!)$ takes the right-most column, which is used in examples in the previous section. Thus, we here consider the general *shrink* that consists of homomorphisms. It is defined as follows.

$$\begin{aligned} &\mathit{shrink} = g_c \times h_n \times h_s \times h_e \times h_w \times h_{ne} \times h_{nw} \times h_{se} \times h_{sw} \\ &\quad \textbf{where} \\ &\quad h_n = (\!|g_n, \oplus_n, \otimes_n|\!)\ , \quad h_s = (\!|g_s, \oplus_s, \otimes_s|\!)\ , \quad h_e = (\!|g_e, \oplus_e, \otimes_e|\!) \\ &\quad h_w = (\!|g_w, \oplus_w, \otimes_w|\!)\ , \quad h_{ne} = (\!|g_{ne}, \oplus_{ne}, \otimes_{ne}|\!)\ , h_{nw} = (\!|g_{nw}, \oplus_{nw}, \otimes_{nw}|\!) \\ &\quad h_{se} = (\!|g_{se}, \oplus_{se}, \otimes_{se}|\!)\ , h_{sw} = (\!|g_{sw}, \oplus_{sw}, \otimes_{sw}|\!) \end{aligned}$$

Here, $\oplus_X$ and $\otimes_X$ are extended to satisfy the following equations: $\mathit{Nil} \oplus_X x = x$, $x \oplus_X \mathit{Nil} = x$, $\mathit{Nil} \otimes_X x = x$, and $x \otimes_X \mathit{Nil} = x$. The general form using this *shrink* uses $\mathrm{O}(n^4)$ operations for a two-dimensional array of $n \times n$.

Then, we give the result of the optimization by fusing *shrink* to *surrounds*.

Theorem 1 (Surrounding). *Let the function shrink be defined by homomorphisms as above. Then, there exist a projection function proj and operators* $\oplus'_f$, $\otimes'_f$, $\oplus'_r$ *and* $\otimes'_r$, *where complexity is the same of* $\oplus_X$ *and* $\otimes_X$, *and the program*

$$\mathit{mconv}\ f\ \mathit{shrink}$$

is optimized to the following program.

$$\mathsf{map}\ \ (f \circ \mathit{proj}) \circ \mathsf{scanr}(\oplus'_r, \otimes'_r) \circ \mathsf{map}\ f_r{}' \circ \mathsf{scan}(\oplus'_f, \otimes'_f) \circ \mathsf{map}\ f_f{}'$$

Proof. The theorem is proved by the promotion of map *shrink* with extending the tuples. See the master's thesis [7] for its details .

The resulting program uses $\mathrm{O}(n^2)$ operations for a two-dimensional array of $n \times n$, while the original general form uses $\mathrm{O}(n^4)$ operations. The parallel complexity of the resulting program is $\mathrm{O}((n^2/P + \sqrt{n^2/P}\log P)T_{(\oplus_X, \otimes_X)})$ for P processors, provided that the calculational complexity of $\oplus_X$ and $\otimes_X$ in the homomorphisms are $T_{(\oplus_X, \otimes_X)}$.

All the examples shown in the previous section have the *shrink* functions described with homomorphisms. Thus, we can apply this theorem to all of them, and they are executed in $\mathrm{O}(n^2/P + \sqrt{n^2/P}\log P)$ complexity using the skeletons.

As mentioned above, the function that takes a fixed number of columns (rows) can be written as a homomorphism. Thus, this theorem holds for the *shrink* of the fixed size window that shrinks the surrounding elements to a fixed size, which is often seen in image filters and difference methods.

Corollary 1 (Fixed Size Window). *Let the function shrink be the fixed size window. Then, the program* $\mathit{mconv}\ f\ \mathit{shrink}$ *is optimized to that of* $\mathrm{O}(n^2)$ *operations.*

Note that the homomorphism taking $h \times w$ subarray of a two-dimensional array has the operators of $\mathrm{O}(wh)$ complexity. Thus, the total complexity of the program of fixed size window is $\mathrm{O}(n^2 wh)$.

Finally, we note that we may perform more optimization by using the shifting of the edges instead of butterfly computation for the global computation of scan and scanr, provided that the operators influences only a fixed number of elements. This leads to the parallel complexity of $\mathrm{O}((n^2/P + \sqrt{n^2/P})T_{(\oplus_X, \otimes_X)})$ for P processors.

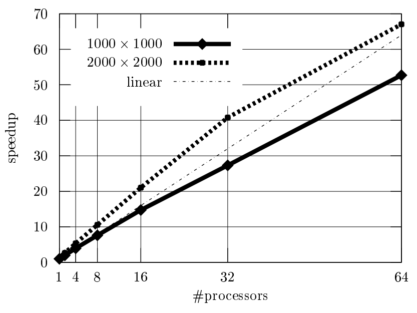

Fig. 4. Speedup of Image Filter

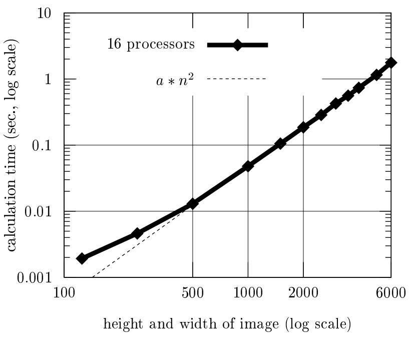

Fig. 5. Calculation Time vs. Size of Image

5 Experimental Results

We implemented the program[1] using our parallel skeleton library [**?**] and did our experiment on a cluster (distributed memory). Each of the nodes connected with Gigabit Ethernet has a CPU of Intel® Xeon®2.80GHz and 2GB memory, with Linux 2.4.21 for the OS, gcc 2.96 for the compiler, and mpich 1.2.7 for the MPI.

Figures 4 and 5 show the speedups and the calculation times of the sharpen-filter. The program is an optimized one from the general form (an equivalent of the program in Fig. 2). The inputs are images of 1000×1000 and 2000×2000. The computation times of the program on one processor are 0.70s and 3.85s respectively.

The result shows programs described with skeletons can be executed efficiently in parallel, and proves the success of our framework. The program achieves almost linear speedups by the parallel implementation of the skeletons, and the total computational complexity of the optimized program is $\mathrm{O}(n^2)$ (thus, its parallel complexity is $\mathrm{O}(n^2/P)$ for small P). However, the serial performance is rather poor due to the overhead of using general skeletons (i.e. scan and scanr). We think this problem can be solved by replacing the general skeletons with those specialized for this domain, and it can be automatically done by compilers (future work).

6 Related Work

SKiPPER [**?**] is a skeleton-based parallel programming environment for real-time image processing. They use skeletons specialized for image processing (not for images or two-dimensional arrays), while we use general skeletons on two-dimensional arrays. Thus, a program developed with SKiPPER may be faster than that written with our skeletons, however, the latter program can be easily composed with other programs and be optimized by fusion due to generality and solid foundation of our skeletons.

There are several other skeletal parallel approaches (libraries), such as eSkel [**?**], Muesli [**?**] and P3L [**?**]. However, their formalization of skeletons on two-dimensional

[1] The source code of the test program as well as the skeleton library are available at the web page `http://www.ipl.t.u-tokyo.ac.jp/sketo/`.

arrays are not enough, so that matrix-convolutions cannot be suitably dealt with. Our skeletons have solid foundation, so that we can easily deal with matrix-convolutions and perform optimizations.

7 Conclusion

In this paper, we focused on the set of matrix-convolution problems, and applied our framework for that domain. This domain is important and fundamental including wide-area of applications such as image filters, difference methods and N-body problems. Applying our skeletal framework, we gave the general program for those computations described with parallel skeletons. Users can easily develop instances of the general form for their problems. We then proposed the *surrounding theorem* to optimize the programs under the assumption that the shrinking function consists of abide-tree homomorphisms. Users can easily get an application-specific program from the program in general form with the theorem. The experimental results support that the optimized program can be executed efficiently in parallel.

8 Acknowledgement

This work is partially supported by the Grant-in-Aid for Scientific Research (B), No. 17300005, Japan Society for the Promotion of Science. We are also grateful to the referees for detailed and helpful comments.

References

1. R. S. Bird. Lectures on Constructive Functional Programming. Technical Report Technical Monograph PRG-69, Oxford University Computing Laboratory, 1988.
2. R. S. Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, 1998.
3. R. S. Bird and O. de Moor. *Algebras of Programming*. Prentice Hall, 1996.
4. M. Cole. *Algorithmic Skeletons : A Structured Approach to the Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing, Pitman, London, 1989.
5. M. Cole. eSkel Home Page. `http://homepages.inf.ed.ac.uk/mic/eSkel/`, 2002.
6. 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.
7. K. Emoto. A Compositional Framework for Parallel Programming on Two-Dimensional Arrays. Master's thesis, Graduate School of Information Science and Technology, the University of Tokyo, 2006. Available at http://www.ipl.t.u-tokyo.ac.jp/˜emoto/master_thesis.pdf.
8. K. Emoto, Z. Hu, K. Kakehi, and M. Takeichi. A Compositional Framework for Developing Parallel Programs on Two Dimensional Arrays. Technical Report METR2005-09, Department of Mathematical Informatics, University of Tokyo, 2005.
9. J. D. Frens and D. S. Wise. QR Factorization with Morton-Ordered Quadtree Matrices for Memory Re-use and Parallelism. In *Proceedings of 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming(PPoPP'03)*, pages 144–154, 2003.

10. 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*.
11. L. Mullin, editor. *Arrays, Functional Languages, and Parallel Systems*. Kluwer Academic Publishers, 1991.
12. F. A. Rabhi and S. Gorlatch, editors. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer-Verlag, 2002.
13. D. B. Skillicorn. *Foundations of Parallel Programming*. Cambridge University Press, 1994.