A Library of Constructive Skeletons for Sequential Style of Parallel Programming

(Invited Paper)

Kiminori Matsuzaki*, Kento Emoto*, Hideya Iwasaki† and Zhenjiang Hu*

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

†Department of Computer Science
The University of Electro-Communications
iwasaki@cs.uec.ac.jp

***Abstract*— With the increasing popularity of parallel programming environments such as PC clusters, more and more *sequential* programmers, with little knowledge about parallel architectures and parallel programming, are hoping to write *parallel* programs. Numerous attempts have been made to develop high-level parallel programming libraries that use abstraction to hide low-level concerns and reduce difficulties in parallel programming. Among them, libraries of parallel skeletons have emerged as a promising way towards this direction. Unfortunately, these libraries are not well accepted by sequential programmers, because of incomplete elimination of lower-level details, ad-hoc selection of library functions, unsatisfactory performance, or lack of convincing application examples. This paper addresses principle of designing skeleton libraries of parallel programming and reports implementation details and practical applications of a skeleton library *SkeTo*. The SkeTo library is unique in its feature that it has a solid theoretical foundation based on the theory of Constructive Algorithmics, and is practical to be used to describe various parallel computations in a sequential manner. The promise of the new approach is illustrated by many interesting applications.**

I. Introduction

Parallel programming offers the potential for substantial performance improvements, but exploration of this potential has been regarded as sort of privilege of expert programmers. This situation is changing dramatically with the availability of low-cost hardware and fast computer networks which makes it easy to build up parallel computer systems such as PC clusters. More and more *sequential* programmers, with little knowledge about parallel architecture and parallel programming, are hoping to write *parallel* programs.

For non-experienced parallel programmers who are used to sequential programming, the rapid development and deployment of a correct parallel application are as important as its execution time; they would like to build parallel applications with a minimum of effort. This eagerly calls for a new parallel programming paradigm such that a sequential programmer can easily and quickly develop parallel programs running reasonably fast on some nearby parallel machines.

There have been numerous attempts to develop high-level parallel programming tools that use abstraction to hide low-level concerns and reduce complexity so that users can quickly build correct parallel programs. These tools may appear as a new high-level parallel programming language [1] or a complicated parallel programming library such as MPI and PVM, and therefore they usually require non-experienced parallel programmers to spend much time on learning due to the gap between the styles of sequential and parallel programming. It would be ideal if one could *write parallel programs as if he wrote sequential programs*, except that some specific functions have both sequential and parallel interpretations. When programs run on parallel machines, the parallel interpretation could be automatically adopted.

Skeletal parallel programming[1] (or structured parallel programming) [2]–[6] has emerged as a promising way towards this direction. In skeletal parallel programming, programmers are encouraged to build a parallel program from ready-made components (i.e., skeletons or patterns) for which efficient parallel implementations are known to exist, making the parallelization process simpler. There are many definitions of skeletons, but the point is that useful patterns of parallel computation and interaction can be packaged up as constructs, such as a framework [6], a second order structure [7], or a template [8]. Skeletal parallel programs are easy to understand even for non-experienced parallel programmers, because they look very much like sequential programs with just a *single execution stream*.

Unfortunately, few widely-used applications are actually developed with parallel skeletons. There are mainly four reasons for this. First, like design patterns in software engineering, parallel skeletons have been introduced in a rather ad-hoc manner mostly based on application domains. There is no clear discipline on skeleton design for adding new skeletons or combining different sets of skeletons developed for different purposes. Second, because parallel programming relies on a fixed set of parallel primitive skeletons for specifying parallelism, programmers often find it hard to choose proper

[1]See `http://homepages.inf.ed.ac.uk/mic/Skeletons/`.

ones and then to integrate them well to develop *efficient* parallel programs. In many cases, the reason programmers feel reluctant to use skeletons is simply because they are not sure whether the provided skeletons are powerful enough to solve their problems and they give up too early. Third, since parallelism should be specified with the skeletons, big overheads could be introduced due to, for example, unnecessary intermediate data structures passed between skeletons. Finally, since most skeleton systems are defined upon some new languages or require extension of existing languages with new syntax, and they do not completely hide all lower-level details of parallel programming from programmers, sequential programmers (non-experienced parallel programmers) often feel uncomfortable to use them.

To solve these problems, we borrowed the theory of Constructive Algorithmics to define and structure skeletons and developed a new skeleton library named *SkeTo*.[2] The theory of *Constructive Algorithmics* [9]–[14], also known as Bird-Meertens Formalisms (BMF for short), was initially proposed for systematical construction of efficient sequential programs by means of *program calculation*. In Constructive Algorithmics, programs are structured so that calculation rules and theories are constructively established for deriving efficient programs from initial specifications. The key observation is that *each control structure (of BMF) should be derivable from the data structure it manipulates.* The fact that skeletons are basically control structures encourages us to apply the theory of Constructive Algorithmics to define and structure skeletons according to the data structures.

This paper addresses principle of designing skeleton libraries of parallel programming, and reports implementation details and practical applications of a skeleton library SkeTo that supports *complete sequential style of parallel programming*. The library is unique in that it has a solid theoretical foundation based on the theory of Constructive Algorithmics, and is practical to be used to describe various parallel computations in a sequential manner. Our main contributions are two folds.

- In principle, we propose a unified approach to defining and structuring skeletons on various parallel data structures. Though many works have been devoted to applying Constructive Algorithmics to formal development of parallel programs on regular data structures like lists [7], [15]–[17], as far as we are aware, we give the first constructive definition for irregular data structures like trees. This enables us to define and structure skeletons on irregular data structures as we do on regular data structures.
- In practice, we have successfully implemented a practical skeleton library SkeTo, with which many interesting and practical applications have been developed (Section VI). Our skeleton library system has the following features.
  - Our system gives the first efficient parallel implementation of *constructive* skeletons for manipulating trees and matrices on distributed memory architectures.
  - Our system hides all lower-level details of parallel programming from programmers, and the parallel programs with the skeletons look like sequential ones. Such parallel programs are easy to understand and debug. They have neither explicit data communication and nor danger of deadlock.
  - A meta mechanism is introduced to support extension of new skeletons and for optimizing skeletal parallel programs. In contrast, most high-level parallel programming tools use a programming model that suffers from a lack of *openness*, making it difficult to tune performance.

The rest of this paper is organized as follows. We start by briefly explaining the related work in Section II. Then, we show how to apply the theory of Constructive Algorithmics to design parallel skeletons for manipulating parallel data structures, regular or irregular, in Section III. We illustrate our SkeTo library with a simple example in Section IV, and explain in detail the implementation issues in Section V. We list applications and give experimental results in Section VI, and conclude the paper and highlight future works in Section VII.

II. Related Work

Skeletal parallel programming was first proposed by Cole [2], and much research [5] has been devoted to it.

Darlington [3], [4] is one of the pioneers in this research area. In his framework, each application program has two-layers structure: higher skeleton level and lower base language level. In the higher level, the user writes a program with skeletons, abstracting its parallel behavior using the SCL (structured coordination language) whose syntax has some kind of functional notation. In the lower level, the user describes sequential aspect of the program in the base language. Darlington selected Fortran as the base language in his implementation.

The idea that separates skeletal parallel part from the base, sequential part in a parallel program is also adopted in the P3L [18], [19] system. In a P3L program, skeletal part is written in a functional notation with the clear specification of its input and output, while the base part is described in the C language. From these descriptions, the P3L compiler generates a C code that calls MPI library functions.

From the user's point of view, both Darlington's system and P3L have higher level layer which has to be written in the specific language in functional-like notation. Thus the user, especially outside the computer science field, may feel difficulties in using these systems in the development of parallel programs, because the user has to acquire the ability of programming in the new language.

Some systems have only a single layer but syntactically extend the base language for the description of skeleton-related part of the program. An instance of such systems is Skil [20],

[2]See `http://www.ipl.t.u-tokyo.ac.jp/SkeTo/` for details including information about how to download the SkeTo system.

[21], which is an imperative, C-based language with some enhancements of functional features. The enhancements include introduction of type variables for polymorphism, special data structure for distributed objects, and higher-order functions. Although Skil is an epoch-making system in the research of skeletal parallel programming, it is now somewhat obsolete because most of the enhanced features can be easily achieved by the C++ language.

HPC++ [22] is another system that introduces extensions (compiler directives) into the base language. HPC++ is a C++ library, developed from the viewpoint of parallelization of the standard template library. Although it is not explicitly addressed that the library implements parallel skeletons, the library includes parallel algorithms that correspond to such data parallel skeletons as map, reduce and scan.

In contrast to the above systems with enhanced syntax into the base language, our SkeTo library introduces no special extension to the base C++ language. Thanks to this design principle, users who can develop a *standard* C++ program can use the SkeTo library without being annoyed with the acquisition of new syntax or new language.

Among recent skeleton libraries that have no syntactic enhancements, Muskel and eSkel are well known. Muskel, developed by Kuchen [23], is a C++ library that works using MPI. It supports data parallel skeletons for distributed array and matrix, and control parallel skeletons. eSkel [24]–[26] is a library of C functions, also on top of MPI. The latest version of eSkel supports control parallel skeletons, putting emphasis on addressing the issues of nesting of skeletons and interaction between parallel activities. Compared with these libraries, SkeTo stands on the data parallel programming model with a solid foundation based on Constructive Algorithmics, and we can offer data parallel skeletons on wide variety of distributed data structures; SkeTo supports binary tree besides array and matrix in a constructive way. In addition, SkeTo has a new skeleton accumulate [27], [28] that abstracts a general and nice combination of data parallel skeletons.

Our work was inspired by the development of Muskel. We are developing the SkeTo library as the practical product of our researches on Constructive Algorithmics. One of its important results is systematic program optimization by fusion transformation. This transformation merges two successive function calls into a single one and eliminates the overhead of both function calls and generation of intermediate data structures passed between the functions. SkeTo is equipped with automatic fusion transformation based on the idea of shortcut deforestation [29] with some modifications that makes it adapt to parallel data structures [27]. By the shortcut deforestation, we can reduce the number of transformation rules and make the implementation of SkeTo simple. This is in sharp contrast to other transformation approaches [30] with large number of transformation rules. This simple optimization mechanism by fusion transformation is SkeTo's distinguishing feature that has not been implemented in other systems.

III. Constructive Skeletons

In this section, we explain the principle in the design of our skeleton library, which is based on the theory of Constructive Algorithmics. We show what kind of basic skeletons should be defined, and how to add and structure new skeletons. Since our skeletons are designed based on Constructive Algorithmics, they are considered to be *constructive*.

A. Constructive Algorithmics

Constructive Algorithmics [9]–[14] is a theory proposed for systematic development of algebraic rules/laws based on which efficient programs are derived from specification via program calculation (manipulation). It has been proved to be very useful for development of efficient sequential programs [13], [14].

The key point of Constructive Algorithmics is that each computation structure used in a function (program) should be derivable from the data structure on which the computation is defined. We will not detail the theory; rather we explain the facts which actually motivated us to use it in the design of our parallel skeletons.

1) Algebraic Data Structures: In Constructive Algorithmics, data structures are defined constructively (algebraically). For instance, integer lists can be defined by[3]

$$\begin{array}{rcl} \mathsf{IntList} & = & \mathsf{Nil} \\ & | & \mathsf{Cons\ Int\ IntList} \end{array}$$

saying that a list may be empty denoted by Nil, or a list denoted by $\mathsf{Cons}\ a\ x$, which is built up from an integer element a and a (shorter) list x by the constructor Cons. So $\mathsf{Cons}\ 1\ (\mathsf{Cons}\ 2\ (\mathsf{Cons}\ 3\ \mathsf{Nil}))$ constructs a list with three elements 1, 2 and 3.

2) Homomorphic Computation Structures: Each algebraic data structure is equipped with a basic computation pattern called homomorphism. For instance, a homomorphism, say h, on the integer lists is the following computation pattern:

$$\begin{array}{lcl} h\ (\mathsf{Nil}) & = & e \\ h\ (\mathsf{Cons}\ a\ x) & = & a\ \oplus\ h\ (x) \end{array}$$

where e is a constant and $\oplus$ is an infixed binary operation. Choosing different pairs of e and $\oplus$ denotes different computation by $h(x)$, which will be represented by $\mathsf{hom}_{\mathsf{IntList}}(e, \oplus, x)$. When it is clear from the context, we may omit the subscript. For instance, computation for summing up elements of a list x is described as $\mathsf{hom}(0, +, x)$.

3) Properties of Homomorphisms: Homomorphism plays a central role in program development [14]. We briefly review those features that are much related to the design of our skeleton library, which has not been well recognized so far in the community of skeletal parallel programming.

- Homomorphism, though being simple, is *powerful* in theory to describe any computation on the data structure it is

[3]Although the skeleton library is in C++, we explain the principle in this section using Bird Meertens Formalism (BMF for short) [9], [31], a functional notation designed for program development. BMF is very similar to the functional language Haskell [32].

defined upon, appearing either as a single homomorphism or a *compositional* use of homomorphisms. We believe that the descriptive power of compositionally is worth emphasizing in skeletal parallel programming.
- Homomorphism enjoys many nice algebraic rules which are useful to eliminate overheads caused by unnecessary data passed between skeletons, or to construct new rules to optimize programs that are defined in terms of homomorphisms. So if we consider homomorphisms as skeletons, skeletal programs can be *efficient* enough.
- Homomorphism can serve as the basis for building practical computation such as the dynamic programming [33], [34], incremental computation [35], and the inverse computation [36]. So with homomorphism, we are able to *add* other useful algorithmic computation patterns as skeletons.

B. Design Issues

1) Challenges in Design of Parallel Skeletons: The features of homomorphism indicate that homomorphism can serve as the basis of our skeleton design. A big challenge is how to formalize *parallel data structures* in an algebraic way such that homomorphisms manipulating these data in parallel can be derived.

It has been shown that useful data structures like lists, nested lists, and trees can be described algebraically but sequentially. Recall the above definition of IntList. Though it is constructive, it is not in parallel in the sense that there is an order on the construction. As a matter of fact, it remains open how to constructively define parallel data structures, in particular for the irregular data structures like imbalanced trees and for the nested data structures like dense or sparse matrices.

In the following, we will first explain in detail the basic idea in the design of data parallel skeletons through the case study on parallel lists. Then, we briefly explain the design of the data parallel skeletons on parallel matrices and parallel trees with a focus on showing how we can constructively define these two parallel data structures.

2) Skeletons on Parallel Lists: Following Constructive Algorithmics, we start with the following definition of a *constructive* and *parallel* view of lists (i.e., parallel lists or distributed lists).

$$\begin{array}{lcl} \mathsf{DList}\ \alpha & = & \mathsf{Empty} \\ & | & \mathsf{Singleton}\ \alpha \\ & | & \mathsf{DList}\ \alpha +\!\!+ \mathsf{DList}\ \alpha \end{array}$$

A list (of type $\mathsf{DList}\ \alpha$) is either the empty list, a singleton list containing a single element (of type α), or the concatenation of two lists (of type $\mathsf{DList}\ \alpha$) by $+\!\!+$. Concatenation $+\!\!+$ is associative, and Empty is its unit.

$$(x +\!\!+ y) +\!\!+ z = x +\!\!+ (y +\!\!+ z)$$

Parallelism in this constructive list lies in the associativity of $+\!\!+$, giving many ways of constructing a list, i.e., no specific (sequential) order is imposed on the list construction. For simplicity, we write $[\,]$ for Empty, $[a]$ for $\mathsf{Singleton}\ a$, and the term $[1] +\!\!+ [2] +\!\!+ [3]$ denotes a list with three elements, often abbreviated to $[1, 2, 3]$.

Homomorphism on Parallel Lists: List homomorphisms (or *homomorphisms* when it is clear from context) [9] are those functions on parallel lists that *promote* through list concatenation. More precisely, a function h satisfying the following three equations is called a *list homomorphism*:

$$\begin{array}{lcl} h([\,]) & = & \iota_\oplus \\ h([a]) & = & f(a) \\ h(x +\!\!+ y) & = & h(x) \oplus h(y) \end{array}$$

where f and $\oplus$ are given functions, and $\oplus$ is *associative* with the identity unit $\iota_\oplus$. We shall write $\mathsf{hom}_L(f, \oplus, x)$ for $h(x)$. For example, the function $sum(x)$, summing up all elements in a list x, can be defined as a homomorphism $\mathsf{hom}_L(id, +, x)$.

List homomorphism captures a natural recursive computation pattern and can serve as the most essential parallel skeleton on parallel lists, which has been recognized in [7], [37], [38]. Intuitively, the above h means that the value of h on the larger list depends in a particular way (using binary operation $\oplus$) on the values of h applied to the two pieces of the list. The computations of $h(x)$ and $h(y)$ are independent of each other and can thus be carried out in parallel. This simple equation can be viewed as expressing the well-known divide-and-conquer paradigm of parallel programming.

Basic Skeletons: While list homomorphism is a good parallel skeleton in general, specialized list homomorphisms can be implemented more efficiently. The most important three such skeletons are *map*, *reduce* and *scan*.

Map is the operator which applies a function to every element in a list. Informally, we have

$$\mathsf{map}_L(f, [x_1, x_2, \ldots, x_n]) = [f(x_1), f(x_2), \ldots, f(x_n)].$$

Reduce is the operator which collapses a list into a single value by repeated application of some associative binary operator. Informally, for an associative binary operator $\oplus$, we have

$$\mathsf{reduce}_L(\oplus, [x_1, x_2, \ldots, x_n]) = x_1 \oplus x_2 \oplus \cdots \oplus x_n.$$

Scan is the operator that accumulates all intermediate results for computation of reduce. Informally, for an associative binary operator $\oplus$, we have

$$\begin{array}{l} \mathsf{scan}_L(\oplus, [x_1, x_2, \ldots, x_n]) \\ \quad = [x_1, x_1 \oplus x_2, \ldots, x_1 \oplus x_2 \oplus \cdots \oplus x_n]. \end{array}$$

Map, reduce and scan have nice massively parallel implementations on many parallel architectures [7], [39]. If f and $\oplus$ use $O(1)$ computation time, then $\mathsf{map}_L(f, x)$ can be implemented using $O(1)$ parallel time, and both $\mathsf{reduce}_L(\oplus, x)$ and $\mathsf{scan}_L(\oplus, x)$ can be implemented using $O(\log n)$.

It is worth noting not only that the three skeletons are specialized version of homomorphisms, but also that a homomorphism is a composition of them:

$$\mathsf{hom}_L(f, \oplus, x) = \mathsf{reduce}_L(\oplus, \mathsf{map}_L(f, x)).$$

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

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

$$\mathsf{scan}_M(\oplus, \otimes, \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & \cdots & x_{mn} \end{pmatrix}) = \begin{pmatrix} y_{11} & y_{12} & \cdots & y_{1n} \\ y_{21} & y_{22} & \cdots & y_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ y_{m1} & y_{m2} & \cdots & y_{mn} \end{pmatrix} \textbf{ where } y_{ij} = \begin{array}{c} (x_{11} \otimes x_{12} \otimes \cdots \otimes x_{1j}) \oplus \\ (x_{21} \otimes x_{22} \otimes \cdots \otimes x_{2j}) \oplus \\ \vdots \\ (x_{i1} \otimes x_{i2} \otimes \cdots \otimes x_{ij}) \end{array}$$

Fig. 1. Three basic skeletons on parallel matrices ($\oplus$ and $\otimes$ are associative and satisfy the abide property.)

Adding New Skeletons: Quite often, we need to introduce new parallel computation skeletons for some specific application domains, but extension of skeletons has been treated in a ad-hoc way.

In our framework, new skeletons are defined around homomorphisms, that is, new parallel computation patterns are defined in terms of the basic skeletons or the skeletons that are defined upon the basic ones. Theoretically, this is always possible if one does not care about efficiency. Any function f on lists can be expressed as a composition of a projection function with a homomorphism, often called *almost homomorphism* [37]:

$$f(x) \equiv \mathit{fst}(\mathsf{hom}_L(g, \oplus, x))$$

where *fst* is a projection function to return the first element of a pair, and the function g and the associative operator $\oplus$ are defined by

$$\begin{array}{lcl} g(x) & = & (f([x]), [x]) \\ (r_1, x_1) \oplus (r_2, x_2) & = & (f\,(x_1 +\!\!+\, x_2),\ x_1 +\!\!+\, x_2). \end{array}$$

In practice, we care about efficiency when introducing new domain-specific skeletons. Let us see how we define new skeletons. Suppose we want to introduce a new skeleton $\mathsf{poly}(\oplus, \otimes, x)$ for capturing the pattern of general polynomial computation: computing on list $x = [x_1, x_2, \ldots, x_n]$ with two associative operators $\oplus$ and $\otimes$ and producing the result of

$$x_1 \ \oplus\ (x_1 \otimes x_2)\ \oplus \cdots \oplus\ (x_1 \otimes x_2 \otimes \cdots \otimes x_n).$$

We may define this new skeleton by

$$\mathsf{poly}(\oplus, \otimes, x) = \mathsf{reduce}_L(\oplus, \mathsf{scan}_L(\otimes, x))$$

and this new skeleton inherits nice properties of homomorphism. As a simple use of this this skeleton, if we choose $\otimes$ as $+$ and $\oplus$ as the function *bigger* to return the bigger of two numbers, then $\mathsf{poly}(\mathit{bigger}, +, x)$ will compute the maximum of all the prefix sums of a list x.

As a matter of fact, a new skeleton can be defined in various ways even with homomorphisms. For the above poly skeleton, we can do better by defining it by

$$\begin{array}{l} \mathsf{poly}(\oplus, \otimes, x) = \mathit{fst}(\mathsf{hom}_L(\mathit{dup}, \odot, x)) \\ \quad \textbf{where } \mathit{dup}(a) = (a, a) \\ \qquad\qquad (a_1, b_1) \odot (a_2, b_2) = (a_1 \oplus (b_1 \otimes a_2), b_1 \otimes b_2) \end{array}$$

provided that $\otimes$ is distributive over $\oplus$. This new definition is more efficient since it is a one-pass algorithm.

3) Skeletons on Regularly Nested Data Structures: Following the same thought of parallel lists, we are defining skeletons to manipulate matrices (a list of lists of the same length), a kind of regularly nested data structures. As explained in Section III-B.1, we focus ourselves on the key step of giving a constructive and parallel view (representation) of matrices.

The traditional representations of matrices by nested lists (row-majored or column-majored representations) [7], [13] impose much restriction on the access order of elements. Wise et al. [40], [41] represented a two-dimensional array by a quadtree, which does not fully expose freedom in construction of matrices.

We borrow the idea in [42] where a more flexible construction of matrices is given for derivation of sequential programs. We define that a matrix is built up by three constructors $|\cdot|$ (singleton), $\circ$ (above) and $\diamond$ (beside) [43].

$$\begin{array}{lcl} \mathsf{DMatrix}\,\alpha & = & |\alpha| \\ & | & (\mathsf{DMatrix}\,\alpha) \circ (\mathsf{DMatrix}\,\alpha) \\ & | & (\mathsf{DMatrix}\,\alpha) \diamond (\mathsf{DMatrix}\,\alpha) \end{array}$$

Here, $|a|$ constructs a matrix with a single element a. Given two matrices x and y of the same width, $x \circ y$ constructs a new matrix where x is located above y. Similarly, given matrices x and y of the same height, $x \diamond y$ constructs a matrix where x is located on the left of y.

The parallelism in this definition of matrices is revealed by the following properties of $\circ$ and $\diamond$.

1) $\circ$ and $\diamond$ are associative.

$$\mathsf{map}_T\ (f,\ \begin{array}{c} x_1 \\ /\backslash \\ x_2\ \ x_3 \\ /\backslash \\ x_4\ \ x_5 \end{array}\ ) \quad = \quad \begin{array}{c} f(x_1) \\ /\backslash \\ f(x_2)\ \ f(x_3) \\ /\backslash \\ f(x_4)\ \ f(x_5) \end{array}$$

$$\mathsf{reduce}_T\ (\oplus, \otimes, f,\ \begin{array}{c} x_1 \\ /\backslash \\ x_2\ \ x_3 \\ /\backslash \\ x_4\ \ x_5 \end{array}\ ) \quad = \quad f(x_1) \oplus ((f(x_2) \oplus (f(x_4) \otimes f(x_5))) \otimes f(x_3))$$

$$\mathsf{uAcc}_T\ (\oplus, \otimes, k,\ \begin{array}{c} x_1 \\ /\backslash \\ x_2\ \ x_3 \\ /\backslash \\ x_4\ \ x_5 \end{array}\ ) \quad = \quad \begin{array}{c} y_1 \\ /\backslash \\ y_2\ \ y_3 \\ /\backslash \\ y_4\ \ y_5 \end{array} \quad \textbf{where} \begin{cases} y_1 = f(x_1) \oplus ((f(x_2) \oplus (f(x_4) \otimes f(x_5))) \otimes f(x_3)) \\ y_2 = f(x_2) \oplus (f(x_4) \otimes f(x_5)) \\ y_3 = f(x_3) \\ y_4 = f(x_4) \\ y_5 = f(x_5) \end{cases}$$

$$\mathsf{dAcc}_T\ (\odot, g_l, g_r, c,\ \begin{array}{c} x_1 \\ /\backslash \\ x_2\ \ x_3 \\ /\backslash \\ x_4\ \ x_5 \end{array}\ ) \quad = \quad \begin{array}{c} z_1 \\ /\backslash \\ z_2\ \ z_3 \\ /\backslash \\ z_4\ \ z_5 \end{array} \quad \textbf{where} \begin{cases} z_1 = c \\ z_2 = c \odot g_l(x_1) \\ z_3 = c \odot g_r(x_1) \\ z_4 = c \odot g_l(x_1) \odot g_l(x_2) \\ z_5 = c \odot g_l(x_1) \odot g_r(x_2) \end{cases}$$

Fig. 2. Four basic skeletons on parallel trees ($\odot$, $\oplus$ and $\otimes$ are associative, and $\otimes$ is distributive over $\oplus$.)

2) $\ominus$ and $\oplus$ satisfy the following *abide* (a coined term from above and beside) property.

$$(x \oplus u) \ominus (y \oplus v) \quad = \quad (x \ominus y) \oplus (u \ominus v)$$

Thanks to these properties, a matrix can be represented in many ways. For example, the following 3×3 matrix

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

can be represented by many ways, two of which are given below.

$$(|1| \oplus |2| \oplus |3|) \ominus (|4| \oplus |5| \oplus |6|) \ominus (|7| \oplus |8| \oplus |9|)$$
$$(|1| \ominus |4| \ominus |7|) \oplus (|2| \ominus |5| \ominus |8|) \oplus (|3| \ominus |6| \ominus |9|)$$

This is in sharp contrast to the quadtree representation of matrices [41], which does not allow such freedom.

With this parallel matrices, it is direct to define matrix homomorphisms and basic skeletons of map_M, reduce_M and scan_M on parallel matrices. Fig. 1 gives informal definitions of three important skeletons on matrices.

4) Skeletons on Irregular Data Structures: Trees are important and widely used in representing hierarchical structures such as XML, but trees are typically irregular; a tree may be ill-balanced and some tree nodes may have too many children. It is a big challenge to give a constructive definition of parallel trees, which, as far as we are aware, is an open problem.

For simplicity, we consider binary trees:

$$\begin{array}{rcl} \mathsf{Tree}\ \alpha & = & \mathsf{Leaf}\ \alpha \\ & | & \mathsf{Fork}\ \alpha\ (\mathsf{Tree}\ \alpha)\ (\mathsf{Tree}\ \alpha) \end{array}$$

reading that a tree is either a tree with just a leaf node, or a tree with a root node and left and right trees. This algebraic representation of trees, as often seen very often in literature, cannot describe "good" parallelism in ill-balanced trees. To resolve this problem, we propose the following novel definition for parallel trees:

$$\begin{array}{rcl} \mathsf{DTree}\ \alpha & = & \mathsf{Leaf}\ \alpha \\ & | & \mathsf{ForkL}\ (\mathsf{DTree}\ \alpha)\ (\mathsf{DTree}\ \alpha)\ (\mathsf{DTree}\ \alpha) \\ & | & \mathsf{ForkR}\ (\mathsf{DTree}\ \alpha)\ (\mathsf{DTree}\ \alpha)\ (\mathsf{DTree}\ \alpha) \end{array}$$

DTree can be regarded as a generalization of Tree where internal nodes are trees instead of node elements.

Parallelism of this DTree is expressed by the requirement that ForkL and ForkR satisfy the following *tree-shift* property.

$$\begin{array}{l} \mathsf{ForkL}\ nt\ (\mathsf{ForkL}\ nt'\ lt'\ rt')\ rt \\ \qquad = \mathsf{ForkL}\ (\mathsf{ForkL}\ nt\ nt'\ rt)\ lt'\ rt' \\ \mathsf{ForkL}\ nt\ (\mathsf{ForkR}\ nt'\ lt'\ rt')\ rt \\ \qquad = \mathsf{ForkR}\ (\mathsf{ForkL}\ nt\ nt'\ rt)\ lt'\ rt' \\ \\ \mathsf{ForkR}\ nt\ lt\ (\mathsf{ForkL}\ nt'\ lt'\ rt') \\ \qquad = \mathsf{ForkL}\ (\mathsf{ForkR}\ nt\ lt\ nt')\ lt'\ rt' \\ \mathsf{ForkR}\ nt\ (\mathsf{ForkR}\ nt'\ lt'\ rt')\ rt \\ \qquad = \mathsf{ForkR}\ (\mathsf{ForkR}\ nt\ lt\ nt')\ lt'\ rt' \end{array}$$

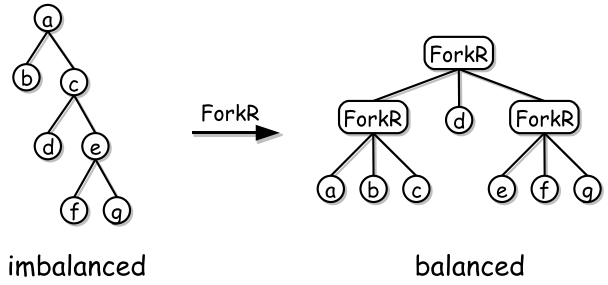

Fig. 3. Balancing an imbalanced tree.

This property can be considered as a tree-version associatively, compared to the associativity of $+\!\!+$ in the definition of parallel lists. With this property, any tree, balanced or imbalanced, can be expressed as a balanced tree, as seen in the example of Fig. 3.

For lack of space, we just give informal definitions of the four important skeletons on parallel trees in Fig. 2. All of these skeletons can be efficiently implemented by the tree contraction algorithm [44], as will be seen later.

IV. The SkeTo Library

A. An Overview

Fig. 4 depicts the framework of the SkeTo library[4] with which programmers are allowed to write skeletal parallel programs in C++ in a sequential style. The SkeTo library itself is implemented in standard C++ and MPI, and the optimization mechanism is in OpenC++ [45], a meta-language for C++.

The SkeTo library provides parallel skeletons for data structures of lists, matrices, and trees. For each data structure, the library consists of two classes (in C++); one provides the definition of parallel data structure, and the other provides the parallel skeletons.

The parallel data structures are implemented based on the theory discussed in Section III, and the implementation conceals the detail of data distribution from the user. The user thus can use the parallel data structures as he uses the sequential ones.

For each parallel data structure, homomorphism and basic skeletons such as *map*, *reduce*, and *scan* are provided. These skeletons are implemented carefully using the MPI library, and thus skeletal programs with these skeletons run efficiently. The domain-specific skeletons such as poly are not provided but can be implemented easily with our basic skeletons as shown in Section V-B.

The skeleton library also provides several wrapper functions of MPI to make the skeletal programs look just like sequential ones. We can see some of them in the sample program in the following section (Fig. 5).

The parallel programs composed with the skeletons may suffer from poor performance. To resolve this problem, the SkeTo library provides meta optimization mechanism based on the fusion transformation techniques. This mechanism is implemented in OpenC++ and automatically optimizes the skeletal programs. See [46] for the details of this optimization mechanism.

B. A Programming Example

To give a flavor of how to write parallel programs in a sequential style using the SkeTo library, let us consider the

[4]SkeTo is an abbreviation for "**Ske**leton Library in **To**kyo", and means "supporter" in Japanese.

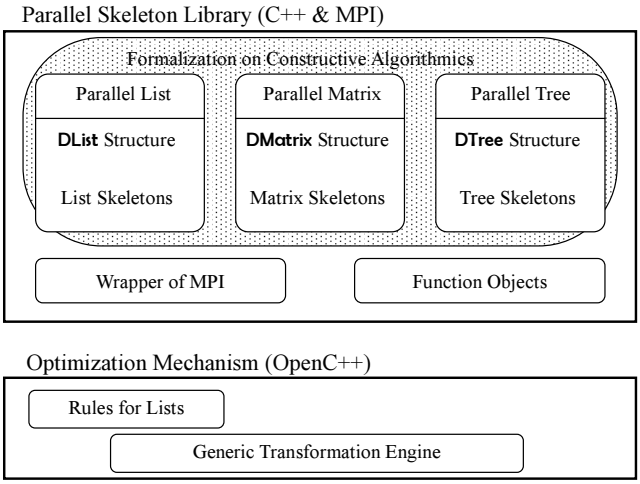

Fig. 4. The framework of the SkeTo Library.

computation of the variance of a list $[x_1, x_2, \ldots, x_n]$.

$$var = \sum_{i=1}^{n} (x_i - ave)^2 / n$$
$$ave = \sum_{i=1}^{n} x_i / n$$

We can specify the algorithm in the following clear parallel program with our skeletons:

$$\begin{array}{l} \mathit{variance}(x) = \mathit{var} \\ \textbf{where} \\ \quad n = \mathit{length}(x) \\ \quad ave = \mathsf{reduce}_L(+, x) \ / \ n \\ \quad var = \mathsf{reduce}_L(+, \mathsf{map}_L(\mathit{sqr}, \mathsf{map}_L(\mathit{sub}(ave), x))) \ / \ n \end{array}$$

where *sqr* is a function to square a number, and $\mathit{sub}(ave)$ is a function to subtract *ave* from a number. The executable C++ code for this problem is given in Fig. 5.

The program starts from the `SketoMain` function (line 9). MPI functions such as `MPI_Init` and `MPI_Finalize` are concealed under the implementation of this function. There are several other wrapper functions of the MPI, for example, the class `skeleton::cout` (line 19) enables us to output from the master processor, which often has rank 0, with the same usage as the `std::cout`. These wrapper functions are helpful for programmers who are not familiar with the MPI.

The sample program first generates an instance of parallel list structure, `data_list` (line 11), by specifying a generator function and the size of list. Note that no details about data distribution are revealed on the program, since the constructor automatically distributes the elements in a proper way.

The sample program then computes the average and the variance with the reduce and the map skeletons (lines 13–17). The functional arguments passed to the skeletons are function objects, and thus the functions are inline-expanded by the C++ compiler and the overhead of function calls are removed. The `map_ow` (lines 15, 16) is a specialized version

```
1  /* definitions of header files and other function objects */
2
3  struct Sub : public skeleton::unary_function<double, double> {
4    double val;
5    Sub(double val_) : val(val_){ }
6    double operator()(double x) const { return x - val; }
7  };
8
9  int SketoMain(int argc, char **argv)
10 {
11   dist_list<double> *as = new dist_list<double>(gen, SIZE);
12
13   double ave = list_skeletons::reduce(add, add_unit, as) / SIZE;
14
15   list_skeletons::map_ow(Sub(ave), as);
16   list_skeletons::map_ow(sqr, as);
17   double var = list_skeletons::reduce(add, add_unit, as) / SIZE;
18
19   skeleton::cout << "average:" << ave << "\n" << "variance:" << var << "\n";
20
21   delete as;
22   return 0;
23 }
```

Fig. 5. A sample program that computes the variance.

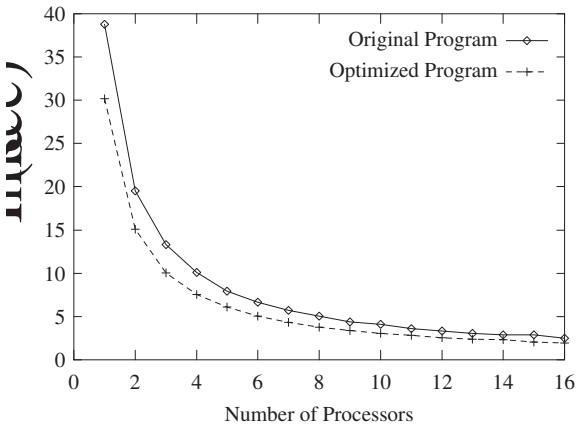

Fig. 6. The computation time before/after the optimization.

of the map skeleton which overwrites the output on the input list. These specialized skeletons enable users to write more efficient skeletal parallel programs.

Thanks to the efficient implementation of the skeletons, the skeletal programs show good scalability. Furthermore, by using meta optimization mechanism we can obtain more efficient programs, in which intermediate structures passed between the skeletons are removed. In the sample program, the meta optimizer automatically fuses the successive calls of `map_ow` and `reduce` into one, and the optimized program runs 1.3 times faster (Fig. 6), which is almost the same performance as the program hand-coded with the MPI library.

As seen so far, users can compose parallel programs easily in a sequential manner, and the obtained programs are reasonably efficient.

V. Implementation Issues

A. Functional Arguments of the Skeletons

The skeletons take function objects for their functional arguments, and this is beneficial in the point of efficiency and extensibility of the skeletons.

In the classic C programs we implement skeletons using pointers to the functions. This implementation is much worse than the hand-coded programs because of sorrowful overhead of calling functions. The function objects on the other hand can be inline-expanded by the C++ compiler, and thus the skeletal programs implemented with function objects are much more efficient.

Furthermore, the function objects are manipulatable like functional programming in conjunction with the template mechanism of C++, for example, composition of functions and partial-binding of arguments. This advantage is essential in importing the theoretical results based on Constructive Algorithmics to extend the skeletons (the following section) and to implement the meta optimization mechanism [46].

B. Extensibility of the Skeletons

Homomorphism and basic skeletons are provided as ready-made skeletons in the SkeTo library. Other skeletons can be easily implemented with these skeletons. Here, we briefly show how we can extend the SkeTo library for domain-specific purposes, with the example of the poly skeleton.

A new skeleton can be implemented in the following two steps; first we define the function objects passed to the ready-made skeletons, and then we define the new skeleton by calling the ready-made skeletons with the newly defined function objects. The sample code of implementing poly is given in Fig 7. In lines 1–11, we define a new function object for $\odot$ with function objects for $\oplus$ and $\otimes$. The function object for

```
1  template<typename A, typename OP, typename OT>
2  struct poly_odot : public skeleton::binary_function<std::pair<A,A>, std::pair<A,A>,
3                                                      std::pair<A,A> >
4  {
5    OP oplus; OT otimes;
6    poly_odot(OP oplus_, OT otimes_) : oplus(oplus_), otimes(otimes_) {}
7    std::pair<A,A> operator()(const std::pair<A,A>& lhs, const std::pair<A,A>& rhs) const {
8      return std::pair<A,A>(oplus(lhs.first,otimes(lhs.second,rhs.first)),
9                            otimes(lhs.second, rhs.second));
10   }
11 };
12
13 template<typename A, typename OP, typename OT>
14 A static poly(const OP& oplus, const A& e_oplus,
15               const OT& otimes, const A& e_otimes, const dist_list<A>* as) {
16   dist_list<std::pair<A,A> > *bs = map(poly_dup( ), as);
17   A result = reduce(poly_odot<A,OP,OT>(oplus, otimes),
18                     std::pair<A,A>(e_oplus, e_otimes), bs).first;
19   if ( bs ) delete bs;
20   return result;
21 }
```

Fig. 7. A sample code for imlementing the poly skeleton.

dup can also be defined in the same way. In lines 13–21, we implement the new skeleton by calling basic skeletons map and reduce.

Defining the new skeletons with basic skeletons is much easier than defining them from scratch. Furthermore, the inline-expansion of function objects and the fusion transformation by the optimization mechanism provide reasonably efficient implementations of the newly defined skeletons.

C. Implementation Issues on Matrices

The parallel representation of matrices in Section III enables us to split any matrix and distribute it in arbitrary sizes and shapes. This is in sharp contrast to traditional representations such as nested lists or quad-trees that restrict the shape of a distributed submatrix to be a list (vector) or a square matrix of the power of two. This freedom of distribution enables the library to use an arbitrary number of processors, and supports the library to be potentially scalable even in heterogeneous environments.

Using the freedom, the library automatically distributes a matrix among processors so that the division of the matrix becomes as square as possible. This block distribution is suitable for the memory hierarchy that recent computers have. Some algorithms require particular distribution manners, such as row-major, column-major or square blocking, for its efficient execution. For example, an efficient algorithm of matrix multiplication needs the shape of the distribution to be square. In those cases, users can control the distribution of matrices by specifying a distribution policy to the library so that the algorithm can be executed efficiently. We provide policies for row-major, column-major, strictly square blocking and roughly square blocking. The skeletons also allow manual distribution for expert users.

We implemented three specialized homomorphisms in Section III and some other useful functions as skeletons. The implementation of the basic matrix skeletons is straightforward extension of that of list skeletons: the execution of the skeleton consists of local computation on each processor and global tree-style communication among the processors, where the difference is that the computation proceeds in two directions in an arbitrary order.

The skeletons take two operators satisfying the abide property, while other existing libraries only take one associative and commutative operator. In fact, an associative and commutative operator satisfies the abide property with itself, and thus our matrix skeletons capture wider range of parallel programs.

D. Implementation Issues on Trees

The parallel structure of binary trees introduced in Section III provides the basis of efficient parallel programs manipulating trees. For a given binary tree, we can construct its parallel structure in more than one way, and thus finding a well-distributable parallel structure with reasonably small cost is important. By borrowing the idea of m-*bridges* [47], which are segments in a tree satisfying certain properties on their size, we implemented a two-pass algorithm for constructing parallel structures of trees. This algorithm guarantees the scalability of the parallel programs for binary trees of arbitrary shapes.

The parallel skeletons for binary trees are implemented based on the tree contraction algorithms. The tree contraction algorithms, first proposed on shared memory architectures [44], [48], and then extended on hypercube networks [49], are distinguished parallel algorithms for computing on trees. We reformalized the algorithms in order to apply them on our parallel tree structures.

We implemented seven parallel skeletons including the four basic skeletons shown in Section III carefully using the MPI. The other skeletons are implementable by composing these skeletons. For example, consider the following more specific reduction skeleton defined on (sequential) tree with

TABLE I

SPEEDUPS FOR PARALLEL PROGRAMS WITH THE SKELETONS

Problem	$P=1$		$P=2$		$P=4$		$P=8$		$P=16$	
	time	ratio	time	ratio	time	ratio	time	ratio	time	ratio
Variance	38.8	1	19.5	1.95	10.1	3.85	5.07	7.64	2.48	15.6
Bracket Matching	115.1	1	57.7	1.99	29.3	3.92	14.7	7.80	7.4	15.47
Heat Equation	86.7	1	43.0	2.02	20.4	4.25	7.38	11.8	5.36	16.2
Matrix Multiplication	14.1	1	5.21	2.71	2.52	5.60	1.05	13.40	0.530	26.62
Maximum Subarray Sum	6.21	1	3.51	1.77	2.30	2.70	1.72	3.61	1.50	4.15
F-Norm	0.21	1	0.104	1.97	0.053	3.88	0.026	7.72	0.013	15.01
QR Factorization	297.1	1	-	-	96.60	3.09	64.76*	4.64*	70.47	4.22
Height of Tree	0.547	1	0.214	2.55	0.128	4.28	0.104	5.25	0.080	6.83
XPath Query	1.92	1	0.762	2.52	0.694	2.77	0.476	4.04	0.360	5.35
Party Planning	0.143	1	0.143	1.00	0.094	1.51	0.069	2.07	0.040	3.66

(* $P = 9$)

an associative and commutative operator.

$$\begin{aligned}&\mathsf{reduce}'_T(\oplus, \mathsf{Fork}\ n\ l\ r)\\ &\quad = n \oplus \mathsf{reduce}'_T(\oplus, l) \oplus \mathsf{reduce}'_T(\oplus, r)\end{aligned}$$

We can implement it with the ready-made basic skeletons in the same way as the implementation of the poly skeleton. The simplicity of extending our skeleton library is more worth noting for tree skeletons since writing parallel tree algorithms from scratch is not straightforward.

We now briefly remark on the parallel skeletons for general trees. There have been only a few studies on the parallel programming on general trees and they were rather ad hoc. We have formalized the parallel tree skeletons for general trees based on binary tree skeletons [50], and under this formalization we have implemented general tree skeletons as wrapper functions of the binary tree skeletons with the same techniques as in adding new skeletons.

VI. APPLICATIONS AND EXPERIMENTS

SkeTo allows users to write parallel programs for various problems. We have so far written a set of skeletal parallel programs listed below[5], and measured their speedups. TABLE I shows the execution time of the programs without the initial data distribution and final data gathering. The first three programs are written with list skeletons, the following four with matrix skeletons, and the last three with tree skeletons.

The parallel computer environment for the experiments is a PC cluster of sixteen uniform PCs connected with Gigabit Ethernet. Each PC has a CPU of Pentium4 3.0GHz (Hyper Threading ON) and 1GB memory, with Linux 2.6.8 for the OS, gcc 2.95 for the compiler, and mpich 1.2.6 for the MPI.

- **Variance**: The program that computes the variance, listed in Fig. 5 in Section IV. The number of elements of the input is 10000000. The measurement is of repetition of 100 times.
- **Bracket Matching**: The program that solves the bracket matching problem of 4 kinds of brackets [28]. The length of the input string is 10000000. The measurement is of repetition of 100 times.
- **Heat Equation**: The program that solves the one-dimensional heat equation. The number of elements of the input is 100000. The measurement is of ten unit time.
- **Matrix Multiplication**: The program that executes matrix multiplication which is fundamental matrix manipulation. The size of the input matrices is 1000×1000.
- **Maximum Subarray Sum**: The program that calculates the maximum of all the subarray (submatrix) sums. The size of the input matrices is 400×400.
- **F-Norm**: The program that calculates the Frobenius norm of matrices (the square root of the sum of squares of the elements). The size of the input matrices is 4000×4000.
- **QR Factorization**: The program that calculates QR factorization of a given matrix. It uses Frens and Wise's algorithm [41], which is originally developed for quadtrees, and thus this program restricts the number of processors involved in the calculation to be a square number.
- **Height of Tree**: The program that computes the height of given binary tree. The number of nodes of the input tree is 2000001.
- **XPath Query**: The program that executes XPath query in parallel. The number of nodes of the input tree is 2000001, and the executed query is of five axes.
- **Party Planning**: The program that solves a dynamic programming problem on general trees called party planning problem [51], which is to find a set of nodes that maximize the sum under certain condition. The number of nodes of the input tree is 1000000.

The experimental results show good speedups in general. In particular, the results of Variance, Bracket Matching, and F-Norm shows almost linear scalability. The results of Heat Equation and Matrix Multiplication shows super-linear speedups. This can happen in memory-intensive applications where a large memory space is needed with respect to the cache size. These results prove the efficiency of the programs written with the SkeTo library.

The programs with tree skeletons are a bit less scalable than those with list or matrix skeletons. This is because even parallel trees cannot be as uniformly distributed as parallel lists or matrices.

Unfortunately the results of Maximum Subarray Sum and

[5]The sources of all the parallel programs are available at the SkeTo web page, so we shall omit detailed explanation due to space limit.

QR Factorization shows poor speedups. The former is due to the current implementation of the SkeTo which only allows the parallel execution of the outer skeleton when nested calls of skeletons are used. Future version of the SkeTo will allow fully parallelization of nested calls of skeletons. The latter is due to the program that is written to parallelize a part of the recursive execution.

VII. Conclusion

This paper applies the theory of Constructive Algorithms to the design and implementation of the SkeTo library supporting parallel programming in a sequential way. Compared with the existing skeleton libraries [23], the SkeTo library attains several new features, which make the system practically useful.

- *It has high description power*. Skeletons and their combination are powerful to describe various parallel computations. This is not only theoretically correct, but also convinced by many practical applications being developed.
- *The skeletons can be extended in a compatible way.* A new skeleton can be introduced systematically, either as a special case of homomorphism or as a generalization being a combination of homomorphisms, and it inherits the nice properties that homomorphisms have. Therefore, all the skeletons, old or new, coexist in a compatible way.
- *Efficiency can be achieved through systematic optimization*. Since all skeletons are basically homomorphisms, many useful calculation rules such as fusion and tupling are easily to be transported here for optimization of skeletal programs.
- *The skeleton library supports a sequential style of parallel programming.* It completely hides lower-level details about parallel programming, so that programmers can stand on the algorithmic level and write parallel programs sequentially without problems such as deadlock.

We are now working to make the SkeTo library more practical to be used widely for parallel programming in sequential style. One of the current issues is to implement the optimization rules for matrix and tree skeletons. In addition, we are interested in how to formalize the control parallel skeletons in theory of Constructive Alogrithmics, which is a challenge but worth consideration.

References

[1] J.-P. Banâtre and D. L. Métayer, Eds., *Research Directions in High-Level Parallel Programming Languages, Mont Saint-Michel, France, June 17-19, 1991, Procedings*, ser. Lecture Notes in Computer Science, vol. 574. Springer, 1992.

[2] M. Cole, *Algorithmic skeletons : a structured approach to the management of parallel computation*. Pitman, London: Research Monographs in Parallel and Distributed Computing, 1989.

[3] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, Q. Wu, and R. L. While, "Parallel programming using skeleton functions," in *Proc. of the Conference on Parallel Architectures and Reduction Languages Europe (PARLE'93)*, ser. Lecture Notes in Computer Science, vol. 694. Springer-Verlag, 1993, pp. 146–160.

[4] J. Darlington, Y. Guo, H. To, and J. Yang, "Parallel skeletons for structured composition," in *Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'95*, Santa Barbara, California, 1995, pp. 19–28. [Online]. Available: citeseer.nj.nec.com/darlington95parallel.html

[5] F. Rabhi and S. G. (eds), *Patterns and Skeletons for Parallel and Distributed Computing*. Springer Verlag, 2002.

[6] K. Tan, D. Szafron, J. Schaeffer, J. Anvik, and S. MacDonald, "Using generative design patterns to generate parallel code for a distributed memory environment," in *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM Press, 2003, pp. 203–215.

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

[8] H. Bischof, S. Gorlatch, and R. Leshchinskiy, "Dattel: A data-parallel C++ template library." *Parallel Processing Letters*, vol. 13, no. 3, pp. 461–472, 2003.

[9] R. Bird, "An introduction to the theory of lists," in *Logic of Programming and Calculi of Discrete Design*, M. Broy, Ed. Springer-Verlag, 1987, pp. 5–42.

[10] R. Backhouse, "An exploration of the Bird-Meertens formalism," in *STOP Summer School on Constructive Algorithmics, Ameland*, Sept. 1989.

[11] E. Meijer, M. Fokkinga, and R. Paterson, "Functional programming with bananas, lenses, envelopes and barbed wire," in *Proc. Conference on Functional Programming Languages and Computer Architecture* (LNCS 523), Cambridge, Massachuetts, Aug. 1991, pp. 124–144.

[12] M. Fokkinga, "A gentle introduction to category theory — the calculational approach —," Dept. INF, University of Twente, The Netherlands, Tech. Rep. Lecture Notes, Sept. 1992.

[13] J. Jeuring, "Theories for algorithm calculation," Ph.D Thesis, Faculty of Science, Utrecht University, 1993.

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

[15] S. Gorlatch, "Systematic efficient parallelization of scan and other list homomorphisms," in *Euro-Par'96. Parallel Processing*, ser. Lecture Notes in Computer Science 1124, L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, Eds. Springer-Verlag, 1996, pp. 401–408.

[16] Z. Hu, H. Iwasaki, and M. Takeichi, "Formal derivation of efficient parallel programs by construction of list homomorphisms," *ACM Transactions on Programming Languages and Systems*, vol. 19, no. 3, pp. 444–461, 1997.

[17] Z. Hu, M. Takeichi, and W. Chin, "Parallelization in calculational forms," in *25th ACM Symposium on Principles of Programming Languages*, San Diego, California, USA, Jan. 1998, pp. 316–328.

[18] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi, "P3L: A structured high level programming language and its structured support," *Concurrency Practice and Experience*, vol. 7, no. 3, pp. 225–255, 1995.

[19] M. Danelutto, F. Pasqualetti, and S. Pelagatti, "Skeletons for data parallelism in P3L," in *Proc. 3rd International Euro-Par Conference (Euro-Par'97)*, ser. Lecture Notes in Computer Science, vol. 1300. Springer-Verlag, 1997, pp. 619–628.

[20] G. H. Botorog and H. Kuchen, "Skil: An imperative language with algorithmic skeletons," in *Proc. 5th International Symposium on High Performance Distributed Computing (HDPC'96)*, 1996, pp. 243–252.

[21] ——, "Efficient high-level parallel programming," *Theoretical Computer Science*, vol. 196, no. 1–2, pp. 71–107, 1998.

[22] E. Johnson and D. Gannon, "HPC++: Experiments with the parallel standard template library." in *Proc. 1997 International Conference on Supercomputing (ICS'97)*, 1997, pp. 124–131.

[23] H. Kuchen, "A skeleton library," in *Proc. 8th International Euro-Par Conference (Euro-Par2002)*, ser. Lecture Notes in Computer Science, vol. 2400. Springer-Verlag, 2002, pp. 620–629.

[24] M. Cole, "Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming." *Parallel Computing*, vol. 30, no. 3, pp. 389–406, 2004.

[25] A. Benoit and M. Cole, "Two fundamental concepts in skeletal parallel programming." in *International Conference on Computational Science (2)*, ser. Lecture Notes in Computer Science, vol. 3515. Springer-Verlag, 2005, pp. 764–771.

[26] A. Benoit, M. Cole, J. Hillston, and S. Gilmore, "Flexible skeletal programming with eskel," in *Proc. 11th International Euro-Par Conference*

(Euro-Par2005), ser. Lecture Notes in Computer Science, vol. 3648. Springer-Verlag, 2005, pp. 761–770.

[27] Z. Hu, H. Iwasaki, and M. Takeichi, "An accumulative parallel skeleton for all," in *11st European Symposium on Programming (ESOP 2002)*. Grenoble, France: Springer, LNCS 2305, Apr. 2002, pp. 83–97.

[28] H. Iwasaki and Z. Hu, "A new parallel skeleton for general accumulative computations," *International Journal of Parallel Programming*, vol. 32, no. 5, pp. 389–414, Oct. 2004.

[29] A. Gill, J. Launchbury, and S. P. Jones, "A short cut to deforestation," in *Proc. Conference on Functional Programming Languages and Computer Architecture*, Copenhagen, June 1993, pp. 223–232.

[30] S. Gorlatch, C. Wedler, and C. Lengauer, "Optimization rules for programming with collective operations," in *IPPS/SPDP'99. 13th Int. Parallel Processing Symp. & 10th Symp. on Parallel and Distributed Processing*, M. Atallah, Ed., 1999, pp. 492–499.

[31] R. Bird, "Constructive functional programming," in *STOP Summer School on Constructive Algorithmics, Abeland*, 9 1989.

[32] ——, *Introduction to Functional Programming using Haskell*. Prentice Hall, 1998.

[33] O. de Moor, "Categories, relations and dynamic programming," Ph.D Thesis, Programming research group, Oxford Univ., 1992, technical Monograph PRG-98.

[34] I. Sasano, Z. Hu, M. Takeichi, and M. Ogawa, "Make it practical: A generic linear time algorithm for solving maximum weightsum problems," in *The 2000 ACM SIGPLAN International Conference on Functional Programming (ICFP 2000)*. Montreal, Canada: ACM Press, Sept. 2000, pp. 137–149.

[35] J. Jeuring, "Incremental Data Compression - extended abstract," in *Proceedings of the Navy Environmental Systems Workshop*, P. Fisher, Ed., 1992. [Online]. Available: citeseer.ist.psu.edu/83060.html

[36] S.-C. Mu and R. Bird, "Theory and applications of inverting functions as folds," *Sci. Comput. Program.*, vol. 51, no. 1-2, pp. 87–116, 2004.

[37] M. Cole, "Parallel programming, list homomorphisms and the maximum segment sum problems," Department of Computing Science, The University of Edinburgh," Report CSR-25-93, May 1993.

[38] S. Gorlatch, "Constructing list homomorphisms," Fakultät für Mathematik und Informatik, Universität Passau, Tech. Rep. MIP-9512, Aug. 1995.

[39] G. E. Blelloch, "Scans as primitive operations," *IEEE Trans. on Computers*, vol. 38, no. 11, pp. 1526–1538, Nov. 1989.

[40] D. S. Wise, "Representing Matrices as Quadtrees for Parallel Proce ssors," *Information Processing Letters*, vol. 20, no. 4, pp. 195–199, 1984.

[41] 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*, 2003, pp. 144–154.

[42] R. S. Bird, "Lectures on Constructive Functional Programming," Oxford University Computing Laboratory, Tech. Rep. Technical Monograph PRG-69, 1988.

[43] K. Emoto, Z. Hu, K. Kakehi, and M. Takeichi, "A Compositional Framework for Developing Parallel Programs on Two Dimensional Arrays," Department of Mathematical Informatics, University of Tokyo, Tech. Rep. METR2005-09, 2005.

[44] G. Miller and J. Reif, "Parallel tree contraction and its application," in *6th Annual IEEE Syrup. on the Foundations of Comp. Science*, 1985, pp. 478–489.

[45] S. Chiba, "A metaobject protocol for C++," in *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*, ser. SIGPLAN Notices 30(10), Austin, Texas, USA, Oct. 1995, pp. 285–299. [Online]. Available: citeseer.nj.nec.com/chiba95metaobject.html

[46] K. Matsuzaki, K. Kakehi, H. Iwasaki, Z. Hu, and Y. Akashi, "A fusion-embedded skeleton library," in *International Conference on Parallel and Distributed Computing (EuroPar 2004)*. Springer, LNCS 3149, Aug. 2004, pp. 644–653.

[47] M. Reid-Miller, G. L. Miller, and F. Modugno, "List ranking and parallel tree contraction," in *Synthesis of Parallel Algorithms*, J. H. Reif, Ed. Morgan Kaufmann Publishers, 1996, ch. 3, pp. 115–194.

[48] K. Abrahamson, N. Dadoun, D. G. Kirkpatrik, and T. Przytycka, "A simple parallel tree contraction algorithm," *Journal of Algorithms*, vol. 10, no. 2, pp. 287–302, June 1989.

[49] E. W. Mayr and R. Werchner, "Optimal tree constraction and term matching on the hypercube and related networks," *Algorithmica*, vol. 18, no. 3, pp. 445–460, 1997.

[50] K. Matsuzaki, Z. Hu, and M. Takeichi, "Design and implementation of general tree skeletons," Department of Mathematical Engineering and Information Physics, University of Tokyo, Tech. Rep. METR2005-30, 2005.

[51] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. MIT Press, 2001.