Generators-of-generators Library with Optimization Capabilities in Fortress

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

[1]University of Tokyo
{emoto@mist.i / k_kakehi@ducr / takeichi@mist.i}.u-tokyo.ac.jp
[2]National Institute of Informatics
hu@nii.ac.jp
[3]Kochi University of Technology
matsuzaki.kiminori@kochi-tech.ac.jp

Abstract. To resolve difficulties in parallel programming, a large number of studies are conducted on parallel skeletons and optimization theorems over skeleton programs. However, two nontrivial tasks still remain unsettled when we need nested data structures: One is composing skeletons to generate and consume them; the other is applying optimization theorems to obtain efficient parallel programs. In this paper, we propose a novel library named *GoG (Generators of Generators) library*. It provides a set of primitives, GoGs, for production of nested data structures. A program developed with these GoGs is automatically optimized, even in asymptotic complexity, by the optimization mechanism in the library. We show its implementation on the Fortress language and report some experimental results.

1 Introduction

Consider the following variant of the maximum segment sum problem: given a sequence of numbers, find the maximum sum of 4-flat segments. Here, '4-flat' means that each difference of successive elements in the segment is less than 4. For example, the answer of a sequence below is 13 contributed by bold numbers.

$$[2, 1, -5, 3, 6, \mathbf{2}, \mathbf{4}, \mathbf{3}, \mathbf{4}, -5, 3, 1, -2, 8]$$

This is a simplified example of combinatorial optimization [1] that is one of the most important classes of computational problems.

Developing an efficient parallel program to solve the problem, especially in a cost linear to the length, is difficult. Even if one can use parallel skeletons [2, 3] such as map, reduce, and scan, it is still difficult to generate all the segments by composing them. In addition, we often require to optimize skeleton programs, but deriving efficient programs from naive programs is still a difficult task even though we have various theorems for shortcut derivation [4–7].

If we have a generation function *segs* that returns all the segments, then we can solve the problem rather easily. Such a program is written as follows with

comprehension notation [8–12]. Here, x is the given sequence, s is bound to each segment of x, flat_4 is a predicate to check 4-flatness, and $\sum$ and $\mathtt{MAX}$ means reductions to take summation and the maximum, respectively.

$$\mathtt{MAX}\left\langle \sum s \mid s \leftarrow \mathit{segs}\, x, \mathit{flat}_4\, s \right\rangle$$

Normal execution of this naive program clearly has a cubic cost w.r.t. the length of x. Therefore, we need to optimize the program to obtain an efficient one.

In this paper, we propose a novel library with which we can run the above naive program efficiently with a linear cost parallel reduction (i.e., it runs in $\mathrm{O}(n/p + \log p)$ parallel time for an input x of length n on p processors). Main features of the library are as follows. (1) It provides a set of primitives, *GoGs (Generators of Generators)*, for production of nested data structures. (2) It is equipped with an automatic optimization mechanism that exploits knowledge of optimization theorems developed in the field of skeletal parallel programming so far. (3) Its optimization is lightweight and applying optimization theorems requires no deep analysis of program code. The main contributions of this paper are the novel design of the library as well as its implementation in Fortress [12]. Note that the implementation has been merged into the Fortress interpreter/compiler.

The rest of this paper is organized as follows. Section 2 clarifies the problems we tackle with the GoG library. Section 3 describes our GoG library. Section 4 shows programming examples and experimental results of the library. Finally, Section 5 reviews related work, and Section 6 concludes this paper.

2 Motivating Example and Problems We Tackle

Let's consider again the maximum 4-flat segment sum problem (MFSS for short) shown in the introduction. We point out the two problems we tackle in this paper, through making an efficient parallel program for MFSS via parallel skeletons and optimization theorems. Notation in this section follows that of Haskell [13].

2.1 Composing Parallel Skeletons to Make Naive Program

We introduce the following parallel skeletons [2, 14] on lists to describe a naive parallel program. Here, map applies a function to each element of a list, reduce takes a summation of a list with an associative operator, scan and scanr produce forward and backward accumulations with associative operators, respectively, and filter removes elements that do not satisfy a predicate.

$$\begin{aligned}
&\mathsf{map}\ f\ [a_1, a_2, \ldots, a_n] &&= [f\ a_1, f\ a_2, \ldots, f\ a_n]\\
&\mathsf{reduce}\ (\oplus)\ [a_1, a_2, \ldots, a_n] &&= a_1 \oplus a_2 \oplus \cdots \oplus a_n\\
&\mathsf{scan}\ (\oplus)\ [a_1, a_2, \ldots, a_n] &&= [b_1, b_2, \ldots, b_n]\ \mathbf{where}\ b_i = a_1 \oplus \cdots \oplus a_i\\
&\mathsf{scanr}\ (\oplus)\ [a_1, a_2, \ldots, a_n] &&= [c_1, c_2, \ldots, c_n]\ \mathbf{where}\ c_i = a_i \oplus \cdots \oplus a_n\\
&\mathsf{filter}\ p &&= \mathsf{reduce}\ (+\!\!+) \circ \mathsf{map}\ (\lambda a.\mathbf{if}\ p\ a\ \mathbf{then}\ [a]\ \mathbf{else}\ [\,])
\end{aligned}$$

Here, $+\!\!+$ means list concatenation, and an application of $\mathsf{reduce}\ (\oplus)$ to an empty list results in the identity of $\oplus$.

Now, we can compose a naive parallel program *mfss* for MFSS as follows. Here, $\uparrow$ is the max operator, segs generates all segments of a list, inits and tails generate all initial and tail segments, respectively, and $flat_4$ checks the 4-flatness.

$$
\begin{aligned}
mfss &= \mathsf{reduce}\ (\uparrow) \circ \mathsf{map}\ (\mathsf{reduce}\ (+)) \circ \mathsf{filter}\ flat_4 \circ \mathsf{segs} \\
\mathsf{segs} &= \mathsf{reduce}\ (+\!\!+) \circ \mathsf{map}\ \mathsf{inits} \circ \mathsf{tails} \\
\mathsf{inits} &= \mathsf{scan}\ (+\!\!+) \circ \mathsf{map}\ (\lambda a.[a]) \\
\mathsf{tails} &= \mathsf{scanr}\ (+\!\!+) \circ \mathsf{map}\ (\lambda a.[a]) \\
flat_4 &= \mathsf{rpred}\ (\lambda(u,v).|u-v|<4) \\
\mathsf{rpred}\ r\ [a_1,a_2,\ldots,a_n] &= \mathsf{reduce}\ (\wedge)\ (\mathsf{map}\ r\ [(a_1,a_2),(a_2,a_3),\ldots,(a_{n-1},a_n)]
\end{aligned}
$$

Since *mfss* is described with parallel skeletons, it is a parallel program.

The program *mfss* is clear, once we know segs generates all segments. However, composing skeletons to make segs is difficult for usual programmers.

In general, such composition of skeletons to generate nested data structures is a difficult task. For example, generation of all subsequences (subsets) of a list is far more difficult and complicated.

2.2 Applying Theorem to Derive Efficient Parallel Program

We introduce a theorem to derive efficient program from the naive program. Among various optimization theorems studied so far [4–7, 15], the following theorem [15] is applicable to the naive program *mfss*.

Theorem 1. *Provided that $\oplus$ with the identity $\imath_\oplus$ is associative and commutative, and $\otimes$ is associative and distributive over $\oplus$, the following equation holds.*

$$
\begin{aligned}
&\mathsf{reduce}\,(\oplus) \circ \mathsf{map}\,(\mathsf{reduce}\,(\otimes)) \circ \mathsf{filter}\,(\mathsf{rpred}\,r) \circ \mathsf{segs} = \pi_1 \circ \mathsf{reduce}\,(\odot) \circ \mathsf{map}\,hex \\
&\textbf{where} \\
&(m_1,t_1,i_1,s_1,h_1,l_1) \odot (m_2,t_2,i_2,s_2,h_2,l_2) \\
&= (m_1 \oplus m_2 \oplus (t_1 \otimes i_2)_{l_1,h_2},\ (t_1 \otimes s_2)_{l_1,h_2} \oplus t_2,\ i_1 \oplus (s_1 \otimes i_2)_{l_1,h_2},\ (s_1 \otimes s_2)_{l_1,h_2},\ h_1,\ l_2) \\
&hex\ a = (a,a,a,a,a,a)\ \ ;\ \ (a)_{l,h} = \textbf{if}\ r\ l\ h\ \textbf{then}\ a\ \textbf{else}\ \imath_\oplus
\end{aligned}
$$

□

Applying this theorem to *mfss*, we can get an efficient parallel program shown in the right hand side of the equation. The resultant parallel program is a simple reduction with a linear cost, and thus runs in $O(n/p + \log p)$ parallel time for an input of size n on p processors. However, a difficult task here is to select the theorem from a sea of optimization theorems. Moreover, it is also difficult to implement the derived new operators/functions correctly without bugs by hand.

In general, there are two difficult tasks in applying optimization theorems: finding a suitable theorem from a sea of optimization theorems; and implementing the given efficient program correctly.

2.3 Problems We Tackle

The problems we tackle are the following two difficult tasks in the development of efficient parallel programs: (1) composing skeletons for producing nested data structures used to describe naive programs; and (2) selecting and applying suitable optimization theorems to derive efficient programs. To tackle these problems, we will propose a library to hide these difficult tasks from users.

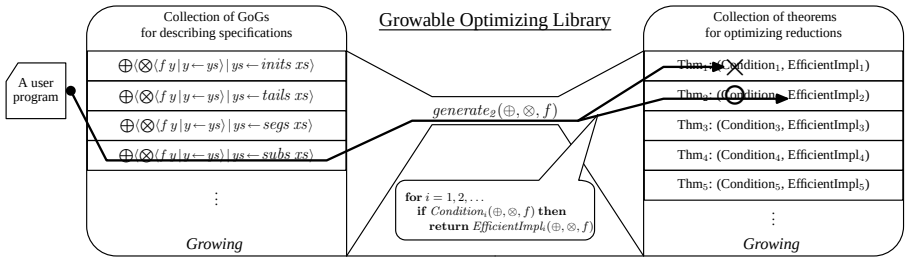

Fig. 1. Two collections form an optimizing GoG library. It optimizes naively-described computation using knowledge of optimization theorems.

3 GoG Library in Fortress

To overcome the problems, we propose a novel library named GoG library. The library provides a set of GoGs equipped with an optimization mechanism. The whole structure of the library is illustrated in Figure 1.

A GoG is, basically, an object representing a nested data structure, such as a list of all segments. It also has the ability to carry out a computation (nested reductions, specifically) on the nested data structure. The combination of GoGs and comprehension notation gives a concise way to describe naive parallel programs.

For example, a naive program for MFSS can be written with GoGs and comprehension notation as follows. Here, *segs* is a function to create a GoG object that represents a list of all segments of the given list x. Note that the resultant is not a simple list of all segments.

$$\texttt{MAX}\left\langle \sum s \mid s \leftarrow segs\, x, flat_4\, s \right\rangle$$

Since the generation of all segments is implemented in the GoG, users are freed from the difficult task of composing skeletons to produce segments. They only need to learn what kind of GoGs are given.

The outstanding point of the library among others is that a GoG does optimization at the execution of the computation. A GoG automatically checks whether given parameters (such as functions, predicates, and operators) satisfy application conditions of optimization theorems. Once it finds an applicable theorem, it executes the computation using an efficient implementation given by the theorem. For example, the library applies Theorem 1 to the above naive program, so that it runs with a linear cost. This mechanism clearly frees users from the difficult task of applying optimization theorems.

In the rest of this section, we explain GoGs and the optimization mechanism as well as their implementation in Fortress. Also, we mention how the library can be extended. Details are found in thesis [15]. We have selected Fortress as an implementation language, because it has both comprehension notation and generators, which share the same concept as GoGs.

3.1 GoG: Generation and Consumption of Nested Data Structures

First of all, we introduce generators in Fortress. A generator is basically an object holding a set of elements. For example, a list is a generator. Its difference from a simple data set is that it also carries out parallel computation on the elements. The computation is implemented in a method *generate*, and has the following semantics. Here, generator g is a list, and its method *generate* takes the pair of an associative operator (enclosed in an object) and a function.

$$g.generate(\oplus, f) \equiv \mathsf{reduce}\ (\oplus)\ (\mathsf{map}\ f\ g)$$

Important is that a generator (a data structure) itself carries out the computation, which enables optimization of the whole computation at its execution. For example, a generator may fuse the above reduce and map, and may use specific efficient implementation exploiting the zero of $\oplus$ when it exists.

Generators are equipped with comprehension notation; we can use the concise notation instead of direct invocations of the method. An expression described in the comprehension notation is desugared into invocations of *generate* as follows.

$$\bigoplus \langle f\ a \mid a \leftarrow g \rangle \Rightarrow g.generate(\oplus, f)$$

It is worth noting that a generator has a method *filter* to return another generator holding filtered elements. Also, expression $\langle e \mid a \leftarrow g, p\ x \rangle$, which involves filtering by predicate p, is interpreted as $\langle e \mid x \leftarrow g.filter(p) \rangle$. The actual filtering is delayed until the resultant generator of $g.filter(p)$ carries the computation on its elements. This may enable optimization exploiting properties of the predicate. It is also worth noting that the body of a comprehension expression can contain another comprehension expression to describe complex computation.

Now, we introduce our GoGs extending the concept of generators. A GoG is an object representing a nested data structure, such as a list of all segments, but it also carries out computation on the nested data structure. The computation is implemented in a method $generate_2$, and has the following semantics. Here, GoG gg is a list of lists, such as segs x for list x.

$$gg.generate_2(\oplus, \otimes, f) \equiv \mathsf{reduce}\ (\oplus)\ (\mathsf{reduce}\ (\otimes)\ (\mathsf{map}\ f\ gg))$$

Again, the encapsulation of the computation into a GoG enables optimization of the whole computation. Its details are shown in the next section.

The combination of GoGs and comprehension notation gives us a concise way to describe naive nested computations, which may be optimized by GoGs. A nested comprehension expression is desugared into an invocation of $generate_2$ as follows.

$$\bigoplus \left\langle \bigotimes \langle f\ a \mid a \leftarrow g \rangle \,\middle|\, g \leftarrow gg \right\rangle \Rightarrow gg.generate_2(\oplus, \otimes, f)$$

It is worth noting that we have extended the desugaring process of the Fortress interpreter to deal with our GoGs. The extension of desugaring will be implemented completely within our library, when the syntax extension feature of Fortress becomes available in the future.

The library gives a set of functions to make GoG objects, such as *segs*. Using such functions, we can write a naive parallel program for MFSS with comprehension notation as shown in the beginning of this section. Note that the generation of all segments is delayed until the GoG carries out the computation, and also that the generation may be canceled when an efficient implementation is used there.

3.2 Optimization Mechanism in GoGs

The outstanding point of the library is GoG's optimization of the computation. In the previous section, we have designed GoGs to carry out the computation by themselves so that they can do the optimization.

We need the following functionalities to implement the optimization exploiting knowledge of theorems: (1) knowing mathematical properties of parameters such as predicates and operators; (2) judging application conditions of theorems; and (3) dispatching efficient implementations given by applicable theorems. Once these are given, the optimization is straightforward: if an applicable theorem is found, a GoG executes the computation with the dispatched implementation.

Now we see the implementation of required functionalities below.

Knowing Mathematical Properties of Parameters For example, to use Theorem 1, we have to know whether the operators have mathematical properties such as distributivity. In general, it is very difficult to find such properties from definitions of operators and functions. Therefore, we take another way: parameters are annotated about such properties beforehand by implementors.

The annotation about properties is put on types of parameters. We use types as place of the annotation, because the annotation is not a value necessary for computation, and the type hierarchy is useful for reuse.

For example, Figure 2 shows annotation about the distributivity of $+$ (enclosed in an object SumReduction) over $\uparrow$ (enclosed in an object MaxReduction). To indicate the distributivity, the object SumReduction extends the trait DistributesOver⟦MaxReduction⟧. In Fortress, type arguments are enclosed in $[\![\cdot]\!]$.

It is worth noting that we can annotate predicates in another way. Since a predicate is just a function, we cannot add annotation on its type directly like objects of reduction operators. However, we can add annotation on the type of its return value, because Fortress allows extension of Boolean.

Judging Application Conditions To use knowledge of optimization theorems correctly, we have to judge their application conditions about parameters. Since properties of parameters are annotated on their types, we can implement such judgment by expressions branching based on types.

For example, Fortress has `typecase` expression that branches on types of the given arguments. Figure 2 shows implementation of judgment about distributivity. The judgment checks whether the second reduction object (r) extends the trait DistributesOver⟦Q⟧, in which Q is the type of the first reduction object

```
trait DistributesOver⟦E⟧ end (* used for annotation: distributive over E *)
object SumReduction extends { DistributesOver⟦MaxReduction⟧, ... }
    empty(): Number = 0; join(a: Number, b: Number): Number = a + b
end
distributes⟦Q, R⟧(q : Q, r : R) : Boolean = typecase (q, r) of (Q, DistributesOver⟦Q⟧) ⇒ true
                                                              else ⇒ false end
```

Fig. 2. Annotation and judgment about distributivity.

```
generate₂⟦R⟧(q: Reduction⟦R⟧, r: Reduction⟦R⟧, f : E → R) : R =
    if distributes(q, r) ∧ commutative(q) ∧ relational(p) then efficientImpl(q, r, f)
    else naiveImpl(q, r, f) end
```

Fig. 3. Simplified dispatching of efficient implementation about Theorem 1. Here, *commutative* is judgment of commutativity, predicate p is stored in a field variable, and *relational* is judgment to check if p is defined by `rpred`.

(q). If the second has the type, then it means that the second distributes over the first. In this case, the judgment returns true. It is worth noting that we can implement such judgments also by overloading functions.

Judgment of an application condition is implemented straightforwardly by composing judgment functions about required properties.

Dispatching Efficient Implementations The dispatching process is straightforward, once we have judgments about application conditions.

Figures 3 shows a simplified dispatching process of the GoG for all segments. The process is implemented in the *generate₂* method, and checks whether the parameters satisfy the application condition of Theorem 1. If the condition is satisfied, then it computes the result by the efficient implementation (i.e., RHS of the equation in Theorem 1). Otherwise it computes the result by its naive semantics. It is worth noting that each of the new operators in the efficient implementation uses the original operators for a fixed number of times.

In general, each GoG has a list of theorems (pairs of conditions and efficient implementations). It checks their application conditions one by one. If an applicable theorem is found, then it computes the result of computation by the efficient implementation. If no applicable theorem is found in the list, then it computes the result based on its naive semantics.

3.3 Growing GoG Library

The library is extended easily. We can add GoGs and accompanying functions to extend the application area of the library. Also, we can add new pairs of application conditions and efficient implementations to strengthen its optimization.

We have grown the library to have the following GoGs (and accompanying functions): all segments of a list (*segs*), all initial segments of a list (*inits*), all tail segments of a list (*tails*), and all subsequences of a list (*subs*). Naive semantics of the former three is shown in Section 2. The last one is trivial.

Also, we have added various optimization theorems to the library as well as Theorem 1. The following optimizations were used during the experiment in Section 4. Here, x is an input of the computation, and each of the LHS programs can be replaced with the corresponding efficient program in the RHS under some conditions. The common application condition is that the associative operator $\otimes$ distributes over the other associative operator $\oplus$. In addition, the theorem for *segs* requires commutativity of $\oplus$, and theorems involving predicates require predicate p to be defined by a certain relation r as $p = \mathsf{rpred}\ r$. We omit definitions of new constant-cost reduction operators $\bigodot_x$. See thesis [15] for details.

$$\begin{aligned}
\bigoplus\langle\bigotimes\langle f\,a \mid a \leftarrow i\rangle \mid i \leftarrow \mathit{inits}\ x\rangle &= \pi_1\ (\bigodot_i\langle(f\,a, f\,a) \mid a \leftarrow x\rangle)\\
\bigoplus\langle\bigotimes\langle f\,a \mid a \leftarrow t\rangle \mid t \leftarrow \mathit{tails}\ xs\rangle &= \pi_1\ (\bigodot_s\langle(f\,a, f\,a) \mid a \leftarrow x\rangle)\\
\bigoplus\langle\bigotimes\langle f\,a \mid a \leftarrow s\rangle \mid s \leftarrow \mathit{segs}\ x\rangle &= \pi_1\ (\bigodot_t\langle(f\,a, f\,a, f\,a, f\,a) \mid a \leftarrow x\rangle)\\
\bigoplus\langle\bigotimes\langle f\,a \mid a \leftarrow i\rangle \mid i \leftarrow \mathit{inits}\ x, p\,i\rangle &= \pi_1\ (\bigodot_{i'}\langle(f\,a, f\,a, a, a) \mid a \leftarrow x\rangle)\\
\bigoplus\langle\bigotimes\langle f\,a \mid a \leftarrow t\rangle \mid t \leftarrow \mathit{tails}\ x, p\,t\rangle &= \pi_1\ (\bigodot_{t'}\langle(f\,a, f\,a, a, a) \mid a \leftarrow x\rangle)
\end{aligned}$$

It is worth noting that these optimization are applicable to not only the usual plus and maximum operator but also any operators that satisfy the required conditions. It is also worth noting that the RHSs run in $O(n/p + \log p)$ parallel time for an input of size n on p processors, when $\oplus$ and $\otimes$ have constant costs.

Equipped with things above, GoG library enables us to describe various parallel programs naively, and carry out efficient computation exploiting the theorems implicitly.

4 Programming Examples and Experimental Results

We show how we can write naive parallel programs with GoG library, and give experimental results that show the naive programs actually run efficiently.

4.1 Example Programming with GoG Library

Figure 4 shows complete code with GoG library for MFSS. Here, $\sum[\![\mathrm{Number}]\!]\,s$ is an abbreviation of $\sum[\![\mathrm{Number}]\!]\,\langle a \mid a \leftarrow s\rangle$, *relationalPredicate* corresponds to rpred, and type information (i.e., $[\![\mathrm{Number}]\!]$) is explicitly written as workaround of the current type system. The program clearly looks a cubic-cost naive program. But it runs with a linear cost owing to the optimization mechanism implemented in the GoG (in this program, the GoG is the object returned by the expression *segs* x). This will be demonstrated in the next section.

It is worth noting that the expressiveness is at least equal to the set of our list skeletons. This is because we can make scan and scanr by inits and tails as follows: scan $(\oplus)\ x = \langle\bigoplus i \mid i \leftarrow \mathit{inits}\ x\rangle$, and scanr $(\oplus)\ x = \langle\bigoplus t \mid t \leftarrow \mathit{tails}\ x\rangle$. Here, a comprehension expression without reduction operations results in a list of the elements in the usual sense. Then, their computation can be executed with a linear cost exploiting well-known scan lemmas.

```
component ExampleProgram
import List.{...}; import Generator2.{...}; export Executable
run() : () = do x = array⟦Number⟧( 400 ).fill(fn a ⇒ ⌊random(10) − 5⌋)
              flat_4 = relationalPredicate⟦Number⟧(fn (a, b) ⇒ |a − b| < 4)
              mfss = MAX⟦Number⟧ ⟨ ⟦Number⟧ ∑⟦Number⟧ s | s ← segs x, flat_4 s ⟩
              println("the maximum 4-flat segment sum of x is " mfss)
end end
```

Fig. 4. Complete code of an example program with GoG library for MFSS.

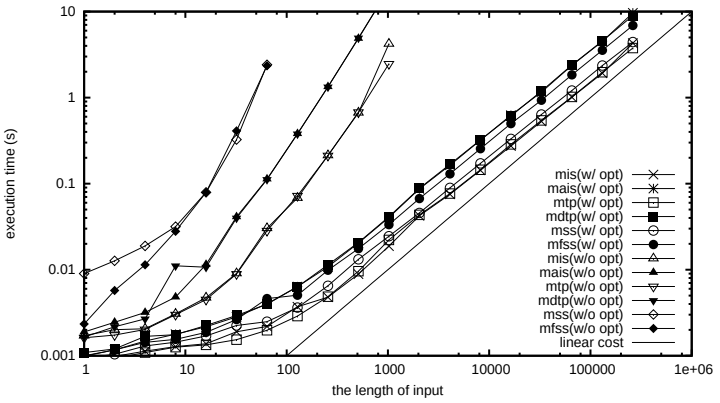

Fig. 5. Execution time of micro programs. They achieve linear costs by optimization.

4.2 Experimental Results

We show experimental results to demonstrate effect of the optimization, parallel performance, and the overhead of the dispatching process. The measurement was taken on the current Fortress interpreter (release 4444 from the subversion repository) running on a PC with two quadcore CPUs (two Intel®Xeon®X5550, 8 cores in total, without hyper-threading), 12GB memory, and Linux 2.6.31.

Figure 5 shows measured execution time of the following micro programs with and without optimization in a logarithmic scale.

$$\begin{array}{l} mis = \mathtt{MAX}\langle \sum s \mid s \leftarrow inits\ x \rangle \ ;\ mais = \mathtt{MAX}\langle \sum s \mid s \leftarrow inits\ x, ascending\ s \rangle \\ mtp = \mathtt{MIN}\langle \prod s \mid s \leftarrow tails\ x \rangle \ ;\ mdtp = \mathtt{MIN}\langle \prod s \mid s \leftarrow tails\ x, descending\ s \rangle \\ mss = \mathtt{MAX}\langle \sum s \mid s \leftarrow segs\ x \rangle \ ;\ mfss = \mathtt{MAX}\langle \sum s \mid s \leftarrow segs\ x, flat_4\ s \rangle \end{array}$$

Here, *inits* and *tails* make GoGs of all initial and tail segments, respectively, and *ascending* and *descending* are predicates to check whether given arguments are sorted ascendingly and descendingly, respectively. Note that the input for *mtp* and *mdtp* is a list of positive real numbers.

The graph shows that the optimization works well so that the naively described micro programs run with linear costs, while the naive execution of these programs suffer from quadratic and cubic cost.

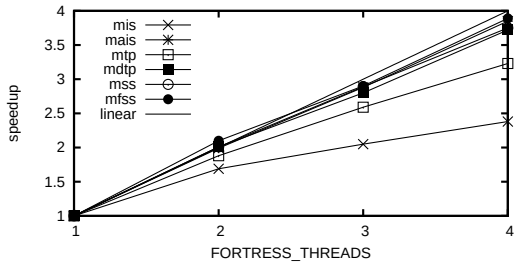

Fig. 6. Speedup of the micro programs for a large input with optimization.

The number of theorems	1	4	16	64	256	1024	4096
Time of dispatching (ms)	3.55	3.18	4.3	9.6	28.7	103	395

Table 1. Overheads of the dispatching process for various numbers of theorems.

It is worth noting that the program code used in the measurement of the case without optimization is the same as the case with optimization, except that annotations about mathematical properties were removed from reduction objects. This means that the library applies its known theorems correctly. The big task of applying suitable theorems is now of the GoG library, and we only have a small task to tell mathematical properties of objects to the library. It is also worth noting that well-used objects are annotated by library implementors.

Next, we see parallel performance of the programs. Figure 6 shows measured speedup of the micro programs for a large input (2^{18} elements) with optimization. The graph shows good speedup of the programs, although it is a little less than the ideal because of lightness of the computation. Unfortunately, the current Fortress interpreter is not mature and has a problem of limitations on parallelism; no Fortress program including our library can achieve more than 4 times speedup. Therefore, the graph only shows the results of at most 4 native threads. This limitation will be removed in the future Fortress interpreter or compiler, and thus the programs will be able to achieve better speedup for a larger number of threads. It is worth noting that a program achieves good speedup even if it was not optimized and thus executed by the naive semantics. This is because the naive semantics uses the existing generators of Fortress in the computation.

Finally, we mention about the overhead of dispatching process in *generate2*. We measured execution time of dispatching process of dummy GoGs with l dummy non-applicable theorems. For the dummy GoGs, the dispatching process checks application conditions l times, then fails in finding an applicable theorem, and finally executes their dummy naive implementations that do nothing. Table 1 shows measured execution time of the dispatching process. The time is very small and ignorable against that of the main computation, unless too many (more than hundreds) theorems are given. If so many theorems are given, we would need to organize them for efficient dispatching, which is a part of future work.

The results show naive programs with GoGs run efficiently in parallel.

5 Related Work

SkeTo library [4, 14] is a parallel skeleton library equipped with optimization mechanisms. Its optimization is designed for fusions of successive flat calls of skeletons, but not for optimization of nested use of skeletons. The work in this paper deals with optimization of nestedly composed skeletons. It can be seen as a complement of the previous work.

The FAN skeleton framework [3] is an skeletal parallel programming framework with an interactive transformation (optimization) mechanism. It has the same goal as ours. It helps programmers interactively to refine naive skeleton compositions into efficient ones, by a graphical tool that locates applicable transformations and provides performance estimates. Our GoG library is designed for automatic optimization, and thereby is equipped with transformations (optimization theorems) that always improve performance for specific cases. Also, optimization mechanism of GoG library is lightweight in the sense that it does not need extra tools such as preprocessors.

Programming using comprehension notation has been considered a promising approach for concise parallel programming, with a history of decades-long research [8–11]. The previous work [16] studied optimization through flattening of nested comprehension expressions to exploit flat parallelism effectively. Their optimizations are focusing on balancing computation tasks. The work in this paper mainly focuses on improving the complexity of computation.

6 Conclusion

We have proposed GoG library to tackle two difficult problems in the development of efficient parallel programs. The library frees users from difficult tasks: composing skeletons to generate nested data structures; and applying optimization theorems (transformations) correctly by hand. In the paper, with the MFSS problem, we demonstrated that a naively-composed program appearing to have a cubic cost actually runs in parallel with a linear cost. The drastic improvement is due to the automatic optimization based on theories of parallel skeletons. It is worth noting that we can implement the library also in other modern languages. For example, we can implement it in C++ using OpenMP [17] or MPI [18] for parallel execution and template techniques for providing a new notation.

One direction of our future work is to widen the application area of the library. We will extend the set of GoGs as well as optimizations over them, so that we can describe more applications such as combinatorial optimization problems. Also, we will extend the optimization to higher-level nesting, though the current implementation deals only with 2-level nesting. Moreover, we will apply the idea of GoGs to programming on matrices and trees, based on our previous research about parallel skeletons on them.

Another direction of our future work is to study automatic discovery of mathematical properties of operators and functions from their definitions. It will reduce users' tasks more. We believe that the rapid growth of recent computers will enable such automatic discovery in the near future.

References

1. Schrijver, A.: Combinatorial Optimization—Polyhedra and Efficiency. Springer-Verlag (2003)
2. Rabhi, F.A., Gorlatch, S., eds.: Patterns and Skeletons for Parallel and Distributed Computing. Springer-Verlag (2002)
3. Aldinucci, M., Gorlatch, S., Lengauer, C., Pelagatti, S.: Towards parallel programming by transformation: the FAN skeleton framework. Parallel Algorithms Applications **16**(2-3) (2001)
4. Emoto, K., Hu, Z., Kakehi, K., Takeichi, M.: A compositional framework for developing parallel programs on two-dimensional arrays. International Journal of Parallel Programming **35**(6) (2007)
5. Iwasaki, H., Hu, Z.: A new parallel skeleton for general accumulative computations. International Journal of Parallel Programming **32**(5) (2004)
6. Hu, Z., Takeichi, M., Iwasaki, H.: Diffusion: Calculating efficient parallel programs. In: Proceedings of the 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation. (1999)
7. Gorlatch, S.: Systematic efficient parallelization of scan and other list homomorphisms. In: Euro-Par '96 Parallel Processing, Second International Euro-Par Conference. Volume 1124 of Lecture Notes in Computer Science, Springer (1996)
8. Blelloch, G.E., Sabot, G.W.: Compiling collection-oriented languages onto massively parallel computers. Journal of Parallel and Distributed Computing **8**(2) (1990)
9. Chakravarty, M.M.T., Keller, G., Lechtchinsky, R., Pfannenstiel, W.: Nepal - nested data parallelism in haskell. In: Euro-Par '01: Proceedings of the 7th International Euro-Par Conference on Parallel Processing, Springer-Verlag (2001)
10. Chakravarty, M.M.T., Leshchinskiy, R., Jones, S.P., Keller, G., Marlow, S.: Data parallel haskell: a status report. In: DAMP '07: Proceedings of the 2007 workshop on Declarative aspects of multicore programming. (2007)
11. Fluet, M., Rainey, M., Reppy, J., Shaw, A., Xiao, Y.: Manticore: a heterogeneous parallel language. In: DAMP '07: Proceedings of the 2007 workshop on Declarative aspects of multicore programming. (2007)
12. Allen, E., Chase, D., Hallett, J., Luchangco, V., Maessen, J.W., Ryu, S., Jr., G.L.S., Tobin-Hochstadt, S.: The Fortress language specification version 1.0. http://research.sun.com/projects/plrg/fortress.pdf (2008)
13. Peyton Jones, S., ed.: Haskell 98 Language and Libraries: The Revised Report. Cambridge University Press, Cambridge, UK (2003)
14. Matsuzaki, K., Emoto, K.: Implementing fusion-equipped parallel skeletons by expression templates. In: Draft Proceedings of the 21st International Symposium on Implementation and Application of Functional Languages (IFL 2009), Technical Report: SHU-TR-CS-2009-09-1, Seton Hall University (2009)
15. Emoto, K.: Homomorphism-based Structured Parallel Programming. PhD thesis, University of Tokyo (2009)
16. Leshchinskiy, R., Chakravarty, M.M.T., Keller, G.: Higher order flattening. In: Computational Science - ICCS 2006, 6th International Conference. Volume 3992 of Lecture Notes in Computer Science, Springer (2006)
17. Chapman, B., Jost, G., van der Pas, R.: Using OpenMP: Portable Shared Memory Parallel Programming. The MIT Press (2007)
18. Gropp, W., Lusk, E., Skjellum, A.: Using MPI (2nd ed.): portable parallel programming with the message-passing interface. MIT Press (1999)