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. A large number of studies have been conducted on parallel skeletons and optimization theorems over skeleton programs to resolve difficulties with parallel programming. However, two nontrivial tasks still remain unresolved when we need nested data structures: The first is composing skeletons to generate and consume them; and the second is applying optimization theorems to obtain efficient parallel programs. In this paper, we propose a novel library called *Generators of Generators (GoG) library*. It provides a set of primitives, GoGs, to produce nested data structures. A program developed with these GoGs is automatically optimized by the optimization mechanism in the library, so that its asymptotic complexity can be improved. We demonstrate 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 in successive elements in the segment is less than four. For example, the answer to the sequence below is 13 contributed to by the 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], which is one of the most important classes of computational problems.

It is difficult to develop an efficient parallel program to solve problems, especially in a cost linear to the length. Even if one can use parallel skeletons [2, 3] such as a map, reduce, and scan, it is still difficult to generate all the segments by composing them. In addition, we often need to optimize skeleton programs, but deriving efficient programs from naive programs is still a difficult task even though we have various theorems for shortcut derivations [4, 5, 6, 7].

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

P. D'Ambra, M. Guarracino, and D. Talia (Eds.): Euro-Par 2010, Part II, LNCS 6272, pp. 26–37, 2010.

comprehension notation [8,9,10,11,12]. 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 MAX mean reductions to find the summation and maximum.

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

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 make it efficient.

This paper proposes a novel library with which we can run the above naive program efficiently with a linear-cost parallel reduction (i.e., it runs in $O(n/p + \log p)$ parallel time for an input x of length n on p processors). The library has three features. (1) It provides a set of primitives, *Generators of Generators (GoGs)*, to produce nested data structures. (2) It is equipped with an automatic optimization mechanism that exploits knowledge on optimization theorems that have been developed in the field of skeletal parallel programming thus far. (3) Its optimization is lightweight and no deep analysis of program code is required to apply optimization theorems. 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 have tackled with the GoG library. Section 3 describes our GoG library. Section 4 presents programming examples and experimental results for the library. Finally, Section 5 reviews related work, and Section 6 concludes the paper.

2 Motivating Example and Problems We Tackle

Let us again consider the maximum 4-flat segment sum problem (MFSS for short) discussed in the introduction. We will identify two problems we tackle in this paper, through creating an efficient parallel program for MFSS via parallel skeletons and optimization theorems. The notation follows that of Haskell [13].

2.1 Composing Parallel Skeletons to Create Naive Program

We will introduce the following parallel skeletons [14, 2] on lists to describe a naive parallel program. Here, a 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.

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

Here, $+\!\!+$ means list concatenation, and an application of 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, 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 create segs is difficult for usual programmers.

Such compositions of skeletons to generate nested data structures is generally a difficult task. For example, the generation of all subsequences (subsets) of a list is far more difficult and complicated.

2.2 Applying Theorem to Derive Efficient Parallel Program

Let us introduce a theorem to derive an efficient program from the naive program. Of the various optimization theorems that have been studied thus far [4,5,6,7,15], the following theorem [15] can be applied to the naive program, *mfss*.

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

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

□

Applying this theorem to *mfss*, we can obtain an efficient parallel program that is shown in the right hand side of the equation. The resulting 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 manually the derived operators correctly without bugs.

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

2.3 Problems We Tackled

The two main problems we tackled were 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 propose a library to conceal these difficult tasks from users.

3 GoG Library in Fortress

To overcome the two problems, we propose a novel library called the GoG library. The library provides a set of GoGs equipped with an optimization mechanism. The whole structure of the library is outlined 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 computation (nested reductions, specifically) on the nested data structure. The combination of GoGs and comprehension notation provides a concise way of describing 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 result 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 feature of the library from others is that a GoG optimizes computation when it is executed. A GoG automatically checks whether given parameters (such as functions, predicates, and operators) satisfy the 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.

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

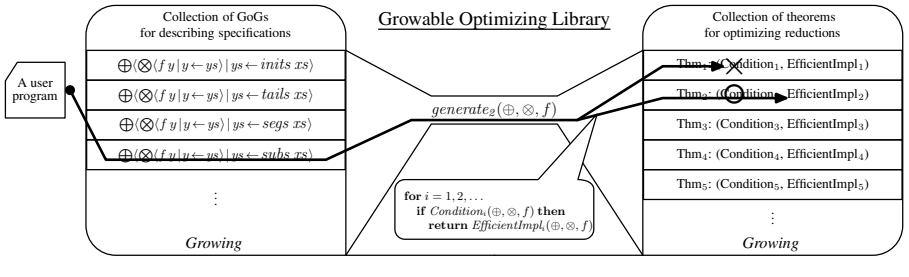

Fig. 1. Two collections form an optimizing GoG library. It optimizes naively-described computation using knowledge of optimization theorems.

3.1 GoG: Generation and Consumption of Nested Data Structures

First, let us we introduce generators in Fortress. A generator is basically an object holding a set of elements. For example, a list is a generator. Its differs from a simple data set in that it also carries out parallel computation on the elements. The computation is implemented in the method, *generate*, and it 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.

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

What is important is that the generator (data structure) itself carries out the computation, which enables the whole computation to be optimized when it is executed. For example, a generator may fuse the reduce and map above, and may use specific efficient implementation exploiting the zero of $\oplus$ when it exists.

As generators are equipped with comprehension notation, we can use a concise notation instead of the direct method invocations. An expression described in the comprehension notation is desugared into invocations of *generate* as

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

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 resulting generator of $g.filter(p)$ carries the computation on its elements. This may enable optimization by 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 will 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 the whole computation to be optimized. The details are presented in the next section.

The combination of GoGs and comprehension notation gives us a concise way of describing naive nested computations, which may be optimized with 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 create GoG objects, such as *segs*. Using such functions, we can write a naive parallel program for MFSS with comprehension notation as explained at the beginning of this section. Note that the generation of all segments is delayed until the GoG carries out computation, and also that generation may be canceled when an efficient implementation is used there.

3.2 Optimization Mechanism in GoGs

The outstanding feature of the library is GoG's optimization of computation. In the previous section, we explained how we designed GoGs to carry out computation by themselves so that they could optimize computation.

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

Know Mathematical Properties of Parameters. We have to know whether the operators have mathematical properties such as distributivity, for example, to use Theorem 1. It is generally very difficult to find such properties from definitions of operators and functions. Therefore, we took another route: parameters were annotated about such properties beforehand by implementors.

The annotations about properties were embedded on the types of parameters. We used types as the location of annotations, because annotations were not values necessary for computation, and the type hierarchy was useful for reuse.

For example, Figure 2 has an annotation about the distributivity of $+$ (enclosed in an object, SumReduction) over $\uparrow$ (enclosed in an object, MaxReduction). To indicate 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 an annotation to its type directly like objects of reduction operators. However, we can add an annotation to the type of its return value, because Fortress allows Boolean extension.

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

For example, Fortress has a `typecase` expression that branches on the types of given arguments. Figure 2 shows an implementation of a judgment on 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 (q). If r, then it means that the second reduction object distributes over the first. In this case, the judgment returns true. It is worth noting that we can also implement such judgments by overloading functions.

```
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.

The judgment of an application condition is implemented simply by composing judgment functions about required properties.

Dispatch Efficient Implementations. The dispatch process is straightforward, once we have judgments about application conditions.

Figures 3 shows a simplified dispatch process of the GoGs for all segments. The process is implemented in the $generate_2$ method, and it checks whether the parameters satisfy the application condition of Theorem 1. If the condition is satisfied, then it computes the result by efficient implementation (i.e., the RHS of the equation in Theorem 1). Otherwise, it computes the result with 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.

Each GoG generally 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 done by the efficient implementation. If no applicable theorem is found in the list, then it computes the result based on its naive semantics. The current library uses the first-match strategy in this process.

3.3 Growing GoG Library

The library can easily be extended. We can add GoGs and accompanying functions to extend its application area. We can also add new pairs of application conditions and efficient implementations to strengthen its optimization.

We have extended 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*). The naive semantics of the former three were discussed in Section 2. The last one is trivial.

We have also 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 a corresponding efficient program in the RHS under various 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 have omitted definitions of new constant-cost reduction operators $\bigodot_x$. See Emoto [15] for details.

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

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 the GoGs and theorems above, the GoG library enables us to describe various naive parallel programs, and carry out efficient computations by implicitly exploiting the theorems.

4 Programming Examples and Experimental Results

Here, we explain how we can write naive parallel programs with the GoG library, and present experimental results that demonstrate the naive programs actually run efficiently.

4.1 Example Programming with GoG Library

Figure 4 has the complete code written with the GoG library for MFSS. Here, $\sum[\![\text{Number}]\!]\, s$ is an abbreviation of $\sum[\![\text{Number}]\!]\,\langle a \mid a \leftarrow s\rangle$, *relationalPredicate* corresponds to rpred, and type information (i.e., $[\![\text{Number}]\!]$) is explicitly written as a workaround of the current type system. The program clearly looks like a cubic-cost naive program. However, it runs with a linear cost due to the optimization mechanism implemented in the GoG (in this program, the GoG is the

```
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

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 create scan and scanr with 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.

4.2 Experimental Results

We present the experimental results to demonstrate the effect of optimization and parallel execution performance. The measurements were done with 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), 12-GB of memory, and Linux 2.6.31.

Figure 5 plots measured execution time for the following micro-programs with and without optimization on a logarithmic scale. Here, *inits* and *tails* creates the GoGs of all initial and tail segments, and *ascending* and *descending* are predicates to check whether given arguments are sorted ascendingly and descendingly. Note that the input for *mtp* and *mdtp* is a list of positive real numbers.

$$\begin{array}{l} \mathit{mis} = \mathtt{MAX}\langle \sum s \mid s \leftarrow \mathit{inits}\ x \rangle \ ;\ \mathit{mais} = \mathtt{MAX}\langle \sum s \mid s \leftarrow \mathit{inits}\ x, \mathit{ascending}\ s \rangle \\ \mathit{mtp} = \mathtt{MIN}\langle \prod s \mid s \leftarrow \mathit{tails}\ x \rangle \ ;\ \mathit{mdtp} = \mathtt{MIN}\langle \prod s \mid s \leftarrow \mathit{tails}\ x, \mathit{descending}\ s \rangle \\ \mathit{mss} = \mathtt{MAX}\langle \sum s \mid s \leftarrow \mathit{segs}\ x \rangle \ ;\ \mathit{mfss} = \mathtt{MAX}\langle \sum s \mid s \leftarrow \mathit{segs}\ x, \mathit{flat}_4\ s \rangle \end{array}$$

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

It is worth noting that the program code used in measuring the case without optimization is the same as that with optimization, except that the annotations

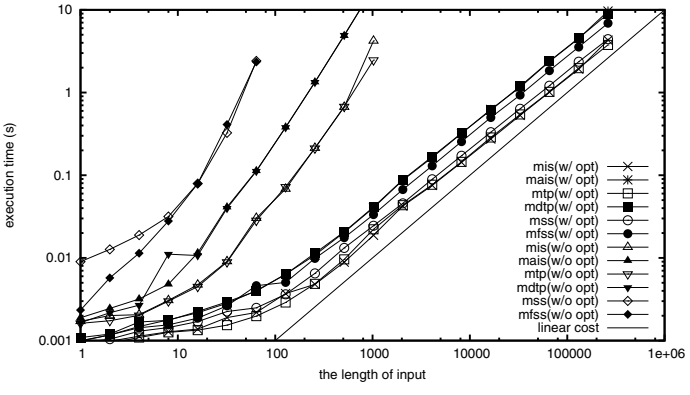

Fig. 5. Execution time of micro programs. They achieve linear costs by optimization.

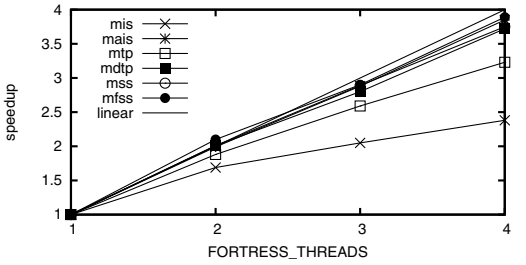

Fig. 6. Speedup of the micro programs for a large input with optimization

about mathematical properties were removed from the reduction objects. This means that the library correctly applies its known theorems. The GoG library now has the huge task of applying suitable theorems, and we only have the small task of notifying the library of mathematical properties of objects. It is also worth noting that well-used objects are annotated by library implementors.

Next, let us look at parallel execution performance of the programs. Figure 6 plots the measured speedup of the micro-programs for a large input (2^{18} elements) with optimization. The graph indicates good speedup of programs, although this is slightly less than optimal because the computation is light. Unfortunately, the current Fortress interpreter is not mature and has a problem with limitations on parallelism; no Fortress program including our library can achieve more than four-fold speedup. Therefore, the graph only shows the results for at most four 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 can achieve good speedup even if it is not optimized and thus executed by the naive semantics. This is because the naive semantics uses the existing generators of Fortress in the computation.

It is worth noting that the overhead for the dispatch process in *generate2* is negligibly small against the execution time of the main computation, unless too many (more than hundreds) theorems are given. If that many theorems are given, we would need to organize them to dispatch them efficiently, which we intend to do in future work.

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

5 Related Work

The SkeTo library [14,4] is a parallel skeleton library equipped with optimization mechanisms. Its optimization is designed to fuse successive flat calls of skeletons, but not to optimize nested use of skeletons. The work in this paper deals with the optimization of nestedly composed skeletons. It can be seen as a complement to 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 to refine naive skeleton compositions interactively so that they are efficient, with a graphical tool that locates applicable transformations and provides performance estimates. Our GoG library was designed for automatic optimization, and thereby is equipped with transformations (optimization theorems) that always improve performance for specific cases. Also, the optimization mechanism for the 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 to concise parallel programming, with a history of decades-long research [8, 9, 10, 11]. Previous work [16] has studied optimization through flattening of nested comprehension expressions to exploit flat parallelism effectively. Their optimizations have focused on balancing computation tasks. The work discussed in this paper mainly focuses on improving the complexity of computation.

6 Conclusion

We proposed the GoG library to tackle two difficult problems in the development of efficient parallel programs. The library frees users from the difficult tasks of composing skeletons to generate nested data structures, and manually applying optimization theorems (transformations) correctly. In the paper, we demonstrated that a naively-composed program appearing to have a cubic cost actually runs in parallel with a linear cost, with the MFSS problem. The drastic improvement was due to automatic optimization based on theories of parallel skeletons. It is worth noting that we can also implement the library 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 new notations.

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

Another direction in future work is to study automatic discovery of the mathematical properties of operators and functions from their definitions. This will greatly reduce the number of user tasks. We believe that the rapid progress with recent computers will enable such automatic discovery in the near future.

Acknowledgments

This paper reports the first result of the joint research project “Development of a library based on skeletal parallel programming in Fortress” with Sun Microsystems. The authors would like to thank Guy L. Steele Jr. and Jan-Willem Maessen for fruitful discussions. This study was partially supported by the Global COE program “The Research and Training Center for New Development in Mathematics”.

References

1. Schrijver, A.: Combinatorial Optimization—Polyhedra and Efficiency. Springer, Heidelberg (2003)
2. Rabhi, F.A., Gorlatch, S. (eds.): Patterns and Skeletons for Parallel and Distributed Computing. Springer, Heidelberg (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: Fraigniaud, P., Mignotte, A., Robert, Y., Bougé, L. (eds.) Euro-Par 1996. LNCS, vol. 1124. Springer, Heidelberg (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: Sakellariou, R., Keane, J.A., Gurd, J.R., Freeman, L. (eds.) Euro-Par 2001. LNCS, vol. 2150, p. 524. Springer, Heidelberg (2001)
10. Chakravarty, M.M.T., Leshchinskiy, R., Jones, S.P., Keller, G., Marlow, S.: Data parallel haskell: a status report. In: DAMP 2007: 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 2007: 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., Steele Jr., G.L., Tobin-Hochstadt, S.: The Fortress language specification version 1.0 (2008), `http://research.sun.com/projects/plrg/fortress.pdf`
13. Peyton Jones, S. (ed.): Haskell 98 Language and Libraries: The Revised Report. Cambridge University Press, Cambridge (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: Alexandrov, V.N., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds.) ICCS 2006. LNCS, vol. 3992, pp. 920–928. Springer, Heidelberg (2006)
17. Chapman, B., Jost, G., van der Pas, R.: Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation). MIT Press, Cambridge (2007)
18. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: portable parallel programming with the message-passing interface, 2nd edn. MIT Press, Cambridge (1999)