Automatic Inversion Generates Divide-and-Conquer Parallel Programs

Kazutaka Morita Akimasa Morihata Kiminori Matsuzaki Zhenjiang Hu Masato Takeichi

Graduate School of Information Science and Technology, University of Tokyo

{kazutaka,morihata,kmatsu}@ipl.t.u-tokyo.ac.jp {hu,takeichi}@mist.i.u-tokyo.ac.jp

Abstract

Divide-and-conquer algorithms are suitable for modern parallel machines, tending to have large amounts of inherent parallelism and working well with caches and deep memory hierarchies. Among others, list homomorphisms are a class of recursive functions on lists, which match very well with the divide-and-conquer paradigm. However, direct programming with list homomorphisms is a challenge for many programmers. In this paper, we propose and implement a novel system that can automatically derive cost-optimal list homomorphisms from a pair of sequential programs, based on the third homomorphism theorem. Our idea is to reduce extraction of list homomorphisms to derivation of *weak right inverses*. We show that a weak right inverse always exists and can be automatically generated from a wide class of sequential programs. We demonstrate our system with several nontrivial examples, including the maximum prefix sum problem, the prefix sum computation, the maximum segment sum problem, and the line-of-sight problem. The experimental results show practical efficiency of our automatic parallelization algorithm and good speedups of the generated parallel programs.

Categories and Subject Descriptors D.1.2 [*Programming Techniques*]: Automatic Programming; D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel programming

General Terms Algorithms, Design, Languages

Keywords Divide-and-conquer parallelism, Inversion, Program transformation, Third homomorphism theorem

1. Introduction

Divide-and-conquer algorithms solve problems by breaking them up into smaller subproblems, recursively solving the subproblems, and then combining the results to generate a solution to the original problem. They match very well for modern parallel machines, tending to have large amounts of inherent parallelism and working well with caches and deep memory hierarchies [28]. Among others, list homomorphisms are a class of recursive functions on lists, which match very well with the divide-and-conquer paradigm [9, 11, 24, 27, 30]. Function h is said to be a list homomorphism, if there is an associated operator $\odot$ such that for any list x and list y

$$h\ (x +\!\!+\ y) = h(x) \odot h(y)$$

where $+\!\!+$ is the list concatenation. When function h is defined as the equation above, the computation of h on a longer list, which is a concatenation of two shorter ones, can be carried out by computing h on each piece in parallel and then combining the results. For instance, the function that sums up the elements in a list can be described as a list homomorphism

$$sum\ (x +\!\!+\ y) = sum(x) + sum(y).$$

List homomorphisms are attractive in parallel programming for several reasons. First, being a class of natural recursive functions on lists, they enjoy many nice algebraic properties, among which, the three well-known homomorphism lemmas form the basis of the formal development of parallel programs [9, 19, 21, 22]. Secondly, and very importantly, they are useful to solve really practical problems. For example, many algorithms executed on Google's clusters are on MapReduce [24], and most of them, such as distributed grep, count of URL access frequency, and inverting index, are certainly nothing but list homomorphisms. Moreover, homomorphisms (catamorphisms) are suitable for developing robust parallel programs, and are considered to be a primitive parallel loop structure in the design of the new parallel programming language Fortress in Sun Microsystems [30].

Despite these appealing advantages of list homomorphisms in parallel programming, a challenge remains for a programmer to use them to solve their problems, particularly when the problems are a bit complicated. Consider the maximum prefix sum problem [6], which is to compute the maximum sum of all the prefix sublists. For instance, supposing mps is the function that solves the problem, we have

$$\begin{aligned} mps\ [1,-1,2] &= 0 \uparrow 1 \uparrow (1+(-1)) \uparrow (1+(-1)+2) \\ &= 2 \end{aligned}$$

where $\uparrow$ is an infix operator returning the bigger of two numbers. It is not straightforward to obtain a parallel program by finding an operator $\odot$ such that

$$mps\ (x +\!\!+\ y) = mps(x) \odot mps(y).$$

It is, however, easy to obtain two sequential programs. We may compute the maximum prefix sum either by scanning the list from left to right as

$$mps\ (x +\!\!+\ [b]) = mps(x) \uparrow (sum(x) + b)$$

or by scanning the list from right to left as

$$mps\ ([a] +\!\!+\ y) = 0 \uparrow (a + mps(y)).$$

These two sequential programs are specialized ones of list homomorphisms: In the former program y is specialized to a list with

PLDI'07 June 11–13, 2007, San Diego, California, USA.

a single element b, and in the latter program x is specialized to a list with a single element a, This ease of sequential programming suggests us to look into possibilities of obtaining list homomorphisms from sequential programs. Noticing that not every sequential program can be parallelized, that is, not all functions can be described as list homomorphisms (in fact mps cannot be a list homomorphism), it is important to clarify under what condition list homomorphisms exist and can be automatically derived.

Interestingly, in the context of list homomorphisms, there is a famous theorem, called *the third homomorphism theorem*, which says that

> *if* two *sequential programs in some specific form exist in solving a problem, then there must exist a list homomorphism that solves the problem too.*

This theorem suggests a new parallel programming paradigm, that is, developing a parallel program with a list homomorphism from a pair of sequential ones. Although this theorem gives a necessary and sufficient condition for the existence of list homomorphisms, it mentions nothing of how to construct them. In fact, it remains open whether there is a general and automatic way to extract an efficient list homomorphism from two sequential programs [15].

In this paper, we propose a novel approach to automatic derivation of cost-optimal list homomorphisms from a wide class of sequential programs. Our idea is to reduce automatic extraction of list homomorphisms to automatic derivation of *a weak right inverse* of a function. We show that a weak right inverse always exists and can be automatically generated for a wide class of sequential functions. As will be seen later, this new approach is applicable to many nontrivial examples, including the maximum prefix sum problem [6, 14, 15], the prefix sum computation [6], the maximum segment sum problem [3], and the line-of-sight problem [6].

Our main contribution can be summarized as follows.

- We design a new automatic parallelization algorithm based on the third homomorphism theorem, by reformalizing the third homomorphism theorem with *the weak right inverse* and generating parallel programs by deriving weak right inverses. The optimization procedure in the algorithm plays an important role in making parallelized programs be efficient.
- We define a language in which users can describe sequential programs for solving various kinds of problems on lists. It is guaranteed that under a reasonable condition an efficient parallel program can be automatically derived from a pair of sequential programs in the language.
- We have implemented the new automatic parallelization algorithm, and tested the generated parallel programs using the SkeTo parallel programming environment, which directly supports parallel programming with map and reduce (two special cases of list homomorphisms.) The experimental results show practical efficiency of our automatic parallelization algorithm and good speedups of the generated parallel programs. This indicates the promise of our new approach.

The rest of this paper is organized as follows. In Section 2, we briefly explain the base theory used in our parallelization framework, our parallel computation patterns, and the third homomorphism theorem. In Section 3, we define the new concept of the weak right inverse. In Section 4, we describe our automatic parallelization algorithm: we explain the source language we deal with, the algorithm to derive a weak right inverse, which is the core part of our parallelization algorithm, and the optimization algorithm for the derived programs. We discuss the implementation issues and show experimental results, together with several application examples in Section 5, and discuss the extensions of our technique and related work in Section 6. We conclude the paper in Section 7.

2. Basic Theory of List Homomorphisms

In this section, we briefly explain the base theory of our parallelization framework, our parallel computation pattern, and the third homomorphism theorem that gives a necessary and sufficient condition for the existence of list homomorphisms.

2.1 Notations on Functions and Lists

Our notations are basically based on the functional programming language Haskell [5]. Functional application is denoted by a space and an argument may be written without brackets. Thus $f\ a$ means $f(a)$. Functions are curried, i.e. functions take one argument and return a function or a value, and the function application associates to the left. Thus $f\ a\ b$ means $(f\ a)\ b$. Infix binary operators will often be denoted by $\oplus, \otimes, \odot$. Functional application binds stronger than any other operators, so $f\ a \oplus b$ means $(f\ a) \oplus b$, but not $f(a \oplus b)$. A functional composition is denoted by a centered circle $\circ$. By definition, $(f \circ g)\ x = f\ (g\ x)$. A functional composition is an associative operator. The identity function is denoted by id. The operator $\triangle$ is for tupling two functions, defined by

$$(f \triangle g)\ a = (f\ a,\ g\ a).$$

The operator $\uparrow$ expresses the operation that computes the maximum, and is defined as

$$x \uparrow y = \mathbf{if}\ (x \geq y)\ \mathbf{then}\ x\ \mathbf{else}\ y.$$

Lists are (nonempty) finite sequences of values of the same type. A list is either a singleton or a concatenation of two lists. We denote $[a]$ for a singleton list with element a, and $x +\!\!+ y$ for a concatenation of two lists x and y. The concatenation operator is associative. Lists are destructed by pattern matching.

2.2 Leftwards, Rightwards, and List Homomorphisms

One point of our parallelization method lies in good capture of the structure of recursive computation. We classify most of the list manipulation computation into three classes, namely rightwards functions, leftwards functions, and list homomorphisms [4].

Definition 2.1 (Leftwards function). Function h is *leftwards* if it is defined in the following form with function f and operator $\oplus$.

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

That is, a leftwards function iterates the computation by sequentially scanning the list from right to left. □

Definition 2.2 (Rightwards function). Function h is *rightwards* if it is defined in the following form with function f and operator $\otimes$.

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

That is, a rightwards function iterates the computation by sequentially scanning the list from left to right. □

Definition 2.3 (List homomorphism [4]). Function h is a *list homomorphism* $(\!|f,\ \odot|\!)$, where $\odot$ is an associative operator, if it is defined in the following divide-and-conquer form.

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

Leftwards and rightwards functions have good correspondence to usual sequential programs, while list homomorphisms can be seen as a general form for efficient divide-and-conquer parallel computation on list: If a function is a list homomorphism then its result does not depend on the place where the input list is split, because the operator $\odot$ is associative.

2.3 Parallel Skeletons

List homomorphisms play a central role in skeletal parallel programming [8, 27], in which users are provided with a fixed set of parallel computation patterns (skeletons) to write parallel programs. The following four parallel skeletons, namely map, reduce, and two scans, are considered to be the most fundamental to describe parallel computation on lists.

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

$$map\ f\ [x_1, x_2, \ldots, x_n] = [f\ x_1, f\ x_2, \ldots, f\ x_n].$$

If f uses $\mathrm{O}(1)$ computation time, then $map\ f$ can be implemented using $\mathrm{O}(1)$ parallel time.

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

$$reduce\ (\odot)\ [x_1, x_2, \ldots, x_n] = x_1 \odot x_2 \odot \cdots \odot x_n.$$

If $\odot$ uses $\mathrm{O}(1)$ computation time, then $reduce\ (\odot)$ can be implemented using $\mathrm{O}(\log n)$ parallel time.

In fact, we can express every instance of list homomorphisms by map with $reduce$.

$$(\!|f,\ \odot|\!) = reduce\ (\odot) \circ map\ f$$

Therefore, list homomorphisms can be efficiently implemented by map and reduce.

Scanl is the operator that accumulates all intermediate results for computation of reduce from left to right. Informally, for associative binary operator $\odot$, we have

$$scanl\ (\odot)\ [x_1, x_2, \ldots, x_n] = [x_1, x_1 \odot x_2, \ldots, x_1 \odot x_2 \odot \cdots \odot x_n].$$

The dual operator $scanr$ is to scan a list from right to left. If $\odot$ uses $\mathrm{O}(1)$ computation time, both $scanl\ (\odot)$ and $scanr\ (\odot)$ can be implemented using $\mathrm{O}(\log n)$ parallel time.

Scans have a strong relationship with list homomorphisms. Consider the functions $inits$ and $tails$, for computing all the prefix sublists and the postfix sublists of a list, respectively.

$$\begin{aligned} inits\ [x_1, x_2, \ldots, x_n] &= [[x_1], [x_1, x_2], \ldots, [x_1, x_2, \ldots, x_n]] \\ tails\ [x_1, x_2, \ldots, x_n] &= [[x_1, x_2, \ldots, x_n], \ldots, [x_{n-1}, x_n], [x_n]] \end{aligned}$$

Then, scans can be defined in terms of map, $reduce$, $inits$, and $tails$.

$$\begin{aligned} scanl\ (\odot) &= map\ (reduce\ (\odot)) \circ inits \\ scanr\ (\odot) &= map\ (reduce\ (\odot)) \circ tails \end{aligned}$$

This implies that computation patterns of $map\ (reduce\ (\odot)) \circ inits$ and $map\ (reduce\ (\odot)) \circ tails$ can be efficiently computed in parallel. This fact will be useful to parallelize sequential programs that use accumulation parameters.

Our objective is to derive programs written by parallel skeletons from usual sequential programs.

2.4 Third Homomorphism Theorem

When writing a program in terms of list homomorphisms or parallel skeletons, the most difficult step is to find the associative binary operator. *The third homomorphism theorem* [16] is the theorem that gives a necessary and sufficient condition for the existence of the associative binary operator.

Theorem 2.1 (Third homomorphism theorem [16]). Function h can be described as a list homomorphism, if and only if it can be defined by both leftwards and rightwards functions. That is, there exists associative operator $\odot$ and function f such that

$$h = (\!|f, \odot|\!)$$

if and only if there exist f, $\oplus$, and $\otimes$ such that

$$\begin{aligned} h\ [a] &= f\ a \\ h\ ([a] +\!\!+ x) &= a \oplus h\ x \\ h\ (x +\!\!+ [a]) &= h\ x \otimes a. \end{aligned}$$

□

Corollary 2.2 ([16]). If function h can be defined as both leftwards and rightwards functions, and if there is function g satisfying $h \circ g \circ h = h$, then there exists an associative binary operator $\odot$ such that the following equation holds.

$$\begin{aligned} &h\ (x +\!\!+ y) = h\ x \odot h\ y \\ &\quad \mathbf{where}\ a \odot b = h\ (g\ a +\!\!+ g\ b) \end{aligned}$$

□

It is worth noting that the third homomorphism theorem only shows the existence of a list homomorphism. We need an automatic method to derive such list homomorphisms, which is the main topic of this paper.

3. Weak Right Inverse

In this section, we introduce the basic concept of *weak right inverses*, which play a very important role in construction of our parallelization algorithm in Section 4.

Definition 3.1 (Weak right inverse). Function g is a weak right inverse of function f, if g satisfies

$$\forall b \in \mathrm{range}(f),\ g\ b = a \Rightarrow f\ a = b$$

where $\mathrm{range}(f)$ denotes the range of the function f. □

Compared to the standard right inverse, the above g is weak in the sense that the domain of g can be larger than the range of f. In other words, g is a right inverse of f only if the domain of g is within the range of the original function f. Unlike inverses, any function has one or many weak right inverses. Consider the weak right inverse of sum. Each function that returns a list whose sum is equal to the input value is a weak right inverse of sum. For instance, each of the the following functions is a weak right inverse of sum.

$$\begin{aligned} g_1\ a &= [a] \\ g_2\ a &= [a-1,\ 1] \\ g_3\ a &= [1,\ 2,\ a-3] \\ g_4\ a &= [a/2,\ a/2]. \end{aligned}$$

While a function may have many weak right inverses, there exists at least one weak right inverse for any function.

Lemma 3.1. At least one weak right inverse exists for any function.

Proof. Let f be a function. We define function g by returning one of those x that satisfies $f\ x = y$ for an input y. This g is obviously a weak right inverse from the definition. Therefore, a weak right inverse exists for any functions. □

For notational convenience, we write f° to denote a weak right inverse of function f. Below we give more examples, where $sort$ is a function for sorting the elements of a list.

$$\begin{aligned} &sum^\circ\ a = [a] \\ &sort^\circ\ x = x \\ &(map\ f)^\circ\ x = map\ f^\circ\ x \end{aligned}$$

Weak right inverses is useful for parallelization, because of the following lemma.

Lemma 3.2. Let f be a function. The following property holds for a weak right inverse f°.

$$f \circ f^\circ \circ f = f$$

Proof. By the definition of the weak right inverse, we know that for any b in the range of f, if $f^\circ\ b = a$ then $f\ a = b$. So $(f \circ f^\circ)\ b = f\ (f^\circ\ b) = f\ a = b$, and thus $f \circ f^\circ \circ f = f$ holds. □

Recall Corollary 2.2, which says that an associative operator for parallel computation can be derived if function g satisfying $f \circ g \circ f = f$ exists. Now, by Corollary 2.2 and Lemma 3.2, we have the following parallelization theorem.

Theorem 3.3 (Parallelization with weak right inverse). If function h is both leftwards and rightwards, then

$$\begin{array}{l} h = (\!|f, \odot|\!) \\ \quad \textbf{where } f\ a = h\ [a] \\ \qquad\quad a \odot b = h\ (h^\circ\ a +\!\!+ h^\circ\ b) \end{array}$$

holds. □

Theorem 3.3 shows that, when the function h is leftwards and rightwards, we can obtain the definition of the list homomorphism of h, provided that we have a weak right inverse of h.

Example 3.1. Let us see how Theorem 3.3 works. Function sum is leftwards and rightwards:

$$\begin{array}{lcl} sum\ [a] & = & a \\ sum\ ([a] +\!\!+ x) & = & a + sum\ x \\ sum\ (x +\!\!+ [a]) & = & sum\ x + a \end{array}$$

so sum can be defined in the form of the list homomorphism according to the third homomorphism theorem. Unlike the third homomorphism theorem that helps little in deriving the list homomorphism for sum, Theorem 3.3 shows us a way to go. As seen before, we have obtained $sum^\circ\ a = [a]$, so we can derive the following list homomorphism:

$$\begin{array}{l} sum = (\!|f, \odot|\!) \\ \quad \textbf{where } f\ a = a \\ \qquad\quad a \odot b = a + b \end{array}$$

by two calculations, $sum\ [a] = a$ and $sum\ (sum^\circ\ a +\!\!+ sum^\circ\ b) = sum\ ([a] +\!\!+ [b]) = a + b$. □

Example 3.2. As a more involved example, consider the maximum prefix sum problem, which we introduced in the introduction.

We start with a quick but incorrect derivation of a list homomorphism for mps. Noticing that $mps^\circ\ a = [a]$, one may calculate as follows:

$$\begin{array}{lcl} a \odot b & = & mps\ (mps^\circ\ a +\!\!+ mps^\circ\ b) \\ & = & mps\ ([a] +\!\!+ [b]) \\ & = & 0 \uparrow a \uparrow (a + b) \end{array}$$

and conclude (see Theorem 3.3) that $mps\ (x +\!\!+ y) = mps\ x \odot mps\ y$. This derived homomorphism is actually incorrect, because $mps\ [1, -2, 2, 1]$ should be 2, but $mps\ [1, -2] \odot mps\ [2, 1]$ gives 3. The problem in this derivation is in its wrong application of Theorem 3.3; it did not check whether the function can be written by both leftwards and rightwards functions, which is required by the theorem. In fact, mps is leftwards:

$$mps\ ([a] +\!\!+ x) = 0 \uparrow (a + mps\ x)$$

but it is not rightwards in the sense that there does not exist $\otimes$ such that $mps\ (x +\!\!+ [a]) = mps\ x \otimes a$. However, it can be defined rightwards if the auxiliary function sum is allowed to be used:

$$mps\ (x +\!\!+ [a]) = mps\ x \uparrow (sum\ x + a)$$

This suggest us to consider tupling of the two functions. In fact, the tupled function $(mps \vartriangle sum)$ is both leftwards and rightwards:

$$\begin{array}{l} \begin{array}{lcl} (mps \vartriangle sum)\ [a] & = & (a \uparrow 0, a) \\ (mps \vartriangle sum)\ ([a] +\!\!+ x) & = & a \oplus (mps \vartriangle sum)\ x \\ (mps \vartriangle sum)\ (x +\!\!+ [a]) & = & (mps \vartriangle sum)\ x \otimes a \end{array} \\ \quad \textbf{where } a \oplus (p, s) = (0 \uparrow (a + p), a + s) \\ \qquad\quad (p, s) \otimes a = (p \uparrow (s + a), s + a) \end{array}$$

To derive the list homomorphic form of $(mps \vartriangle sum)$ by Theorem 3.3, we need to find a weak right inverse of $(mps \vartriangle sum)$, namely $(mps \vartriangle sum)^\circ$, which should take a pair of two values (p, s), and output a list whose maximum prefix sum must be p and whose sum must be s. That is,

$$\begin{array}{l} (mps \vartriangle sum)^\circ\ (p, s) = x \\ \quad \textbf{such that } mps\ x = p \wedge sum\ x = s\ . \end{array}$$

Derivation of $(mps \vartriangle sum)^\circ$ is not obvious at all, which will be studied in the next section. □

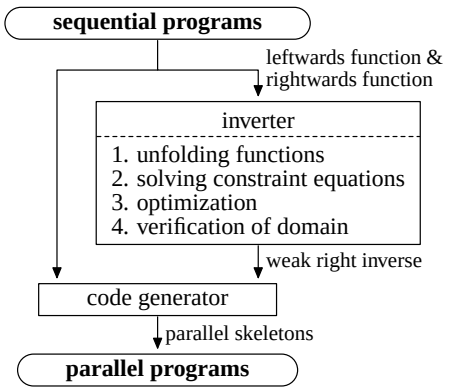

Figure 1. Parallelization framework.

4. Automatic Parallelization Algorithm

Theorem 3.3 indicates that in order to obtain a list homomorphism, it is sufficient to derive its weak right inverse. In this section, we propose a novel parallelization algorithm based on this observation. With our parallelization algorithm, users can obtain efficient parallel programs for free, by focusing only on the development of a pair of sequential programs.

4.1 Overview

Figure 1 shows the framework of our parallelization algorithm. It accepts a sequential program as input, which may contain a list of functions that are both leftwards and rightwards, derives weak right inverses of the functions in the input, and generates a parallel program composed by parallel skeletons.

Input: Sequential Programs

The input to our parallelization algorithm is a sequential program that describes computation on lists. Different from the other parallelizing tools, our algorithm requires each function in the input to be implemented by a pair of sequential functions that perform computation by scanning lists leftwards and rightwards respectively. This requirement is to guarantee the existence of parallel programs.

To see concretely what the input sequential programs look like, consider again the maximum prefix sum problem. If the input is a list with only a single element, easily we obtain

$$mps\ [a] = 0 \uparrow a.$$

Otherwise, we hope to define mps by scanning the lists leftwards and rightwards, that is, to find t_1 and t_2 such that

$$\begin{array}{lcl} mps\ ([a] +\!\!+ x) & = & t_1 \\ mps\ (x +\!\!+ [a]) & = & t_2. \end{array}$$

The basic restriction on the terms t_1 and t_2 is that x must appear in the form of $f\ x$ where f is a function that is defined both leftwards and rightwards. This restriction ensures the order of visiting the elements of the list. As seen in Example 3.2, such t_1 and t_2 exist

with an auxiliary function sum that can be defined both leftwards and rightwards.

To parallelize more general programs, we consider another example. It is known that accumulation parameters are important in designing efficient programs. Efficient programs often appear in the following extended leftwards and rightwards forms:

$$\begin{array}{l} f\ ([a] +\!\!+\ x)\ c\ =\ t_1 \\ f\ (x +\!\!+\ [a])\ c\ =\ t_2 \end{array}$$

where an accumulation parameter c is used. They seem out of the scope discussed above, but it has been shown [1] that this kind of programs can be decomposed into a combination of skeletons

$$\begin{array}{l} g_1' \circ map\ f_1' \circ inits \\ g_2' \circ map\ f_2' \circ tails \end{array}$$

where f_1', f_2', g_1', and g_2' are sequential functions defined without accumulation parameters. Our algorithm can deal with the latter style of sequential programs, whenever f_1', f_2', g_1', and g_2' are defined leftwards and rightwards. As an example, consider the prefix sum problem [7, 14, 15]: given a list, compute the sums of all the prefix sublists. It can be sequentially described by

$$\begin{array}{lcl} psum\ x & = & psum'\ x\ 0 \\ psum'\ [a]\ c & = & [a+c] \\ psum'\ ([a] +\!\!+\ x)\ c & = & [a+c] +\!\!+\ psum'\ x\ (a+c) \end{array}$$

which can be equivalently described by

$$psum = map\ sum \circ inits$$

where sum can be sequentially implemented by both leftwards and rightwards functions, and our algorithm can generate a parallel program for $psum$.

The syntax of the source language is summarized in Figure 2. The sequential program, namely the input to our algorithm, is of the following three forms:

- $lrseqs$: leftwards/rightwards programs;
- $map\ lrseqs \circ inits$: leftwards accumulative programs;
- $map\ lrseqs \circ tails$: rightwards accumulative programs.

The main functions are specified by the special identifier `main`. The language to define leftwards/rightwards programs will be discussed in Section 4.2.

Parallelization Engine: Inverter and Code Generator

Our parallelization engine consists of two layers: The first is the inverter and the second is the code generator.

The first layer of our algorithm, namely the inverter, derives a weak right inverse from $lrseqs$. Let $f_1,\ f_2, \ldots,\ f_n$ be all the functions defined in the $lrseqs$. Then the inverter derives a weak right inverse of the tupled function, that is $(f_1 \vartriangle f_2 \vartriangle \cdots \vartriangle f_n)^\circ$. This layer is the main part of our parallelization engine, because weak right inverses derive associative operators that are necessary to construct list homomorphisms. We will show the automatic weak right inversion algorithm in Section 4.3.

The second layer, namely the code generator, does three things. Firstly, it derives list homomorphism for $lrseqs$ based on Theorem 3.3, from the weak right inverse generated by the inverter. Secondly, it transforms the sequential program defined in one of the three basic forms into a skeletal parallel program by the following rules.

$$\begin{array}{lcl} (\!|f, \odot|\!) & = & reduce\ (\odot) \circ map\ f \\ map\ (\!|f, \odot|\!) \circ inits & = & scanl\ (\odot) \circ map\ f \\ map\ (\!|f, \odot|\!) \circ tails & = & scanr\ (\odot) \circ map\ f \end{array}$$

Finally, it generates executable parallel code for the skeletal parallel program. Our parallelization never fails whenever the derivation of weak right inverses succeeds.

Output: Skeletal Parallel Programs

Our algorithm automatically generates parallel programs that are defined with parallel skeletons, which can be executed efficiently in parallel [27]. For instance, our algorithm generates the following skeletal parallel program for mps:

$$\begin{array}{l} mps_{para} = fst \circ reduce\ (\odot) \circ map\ f \\ \quad \textbf{where} \\ \quad\quad fst\ (a, b) = a \\ \quad\quad f\ a = (a \uparrow 0,\ a) \\ \quad\quad (p_x,\ s_x) \odot (p_y,\ s_y) \\ \quad\quad\quad = (mps \vartriangle sum)\ ([p_x,\ s_x - p_x] +\!\!+\ [p_y,\ s_y - p_y]) \end{array}$$

This program is an $O(\log n)$ parallel program for mps where n is the length of the input list.

4.2 Language to Specify Leftwards/Rightwards Programs

We provide users with a language to write leftwards and rightwards sequential programs, which are the input of the inverter. The language captures a wide class of functions that accepts a list as input and computes a numeric value. Leftwards and rightwards sequential programs are specified by the nonterminal $lrseqs$ in Figure 2. The nonterminal $lrseqs$ consists of one or more function definitions (def), which provide leftwards definitions and rightwards definitions. The body of each definition is a linear term $lterm$, which is constructed by additions of two linear terms, multiplications by constants, applications of a function to the rest of the list `x`, values of the element of the list `a`, constants, or conditional expressions. Any function used in $lterm$ must be defined in the program, and the list element `a` should appear in the definition body at least once.

As an example, recall the maximum prefix sum problem. We can use the language to describe a sequential program as follows. In the following program, the $\uparrow$ operator is unfolded with the conditional expression.

```
main = mps;
  mps [a]      = if (a <= 0) then 0 else a;
  mps ([a]++x) = if (0 <= a + mps(x))
                   then a + mps(x)
                   else 0;
  mps (x++[a]) = if (mps(x) <= sum(x) + a)
                   then sum(x) + a
                   else mps(x);
  sum [a]      = a;
  sum ([a]++x) = a + sum(x);
  sum (x++[a]) = sum(x) + a;
```

It is worth noting that we need to write the leftwards and rightwards program of sum, because sum is necessary to write the rightwards program of mps. From this program, our inverter derives a weak right inverse of $(mps \vartriangle sum)$.

For successful and automatic derivation of a weak right inverse, we impose some restriction on our language. This restriction excludes some functions that are necessary for parallel computation. We can relax this restriction and deal with wider classes of functions in our framework. We will discuss these extensions in Section 6.

4.3 Automatic Weak Right Inversion for Parallelization

Now we consider how to derive a weak right inverse. If $lrseqs$ consists of only one function, it is easy to derive its weak right inverse, as seen in Example 3.1. But as seen in the Example 3.2, it is hard to derive a weak right inverse of a tupled function. Since many

```
prog   ::= main; lrseqs                          (program)
main   ::= main = fun                            (main function)
         | main = map(fun) . (inits | tails)
lrseqs ::= def+
def    ::= fun [a] = lterm;                      (function definition)
           fun ([a]++x) = lterm;
           fun (x++[a]) = lterm;
lterm  ::= lterm (+ | -) lterm                   (addition)
         | lterm (* | /) Integer                 (multiplication)
         | fun(x)                                (function application)
         | a                                     (element of the list)
         | Integer                               (constant number)
         | if(cond) then lterm else lterm
                                                 (condition)
cond   ::= lterm (== | != | <= | >= | < | >) lterm
         | cond (|| | &&) cond
```

Figure 2. The source language of our parallelization algorithm

functions are defined in terms of both leftwards and rightwards by using other auxiliary functions, we will focus on the derivation of a weak right inverse of such a function.

As shown in Figure 3, our derivation of a weak right inverse is carried out by the following procedure. Firstly, we get the constraint equations by unfolding the sequential functions for an input list of fixed length. Secondly, we solve the constraint equations and get a weak right inverse that is correct but may be inefficient. Next, we optimize the weak right inverse by eliminating redundant conditional branches. Finally, we verify whether the domain of the weak right inverse is correct with respect to our preconditions. Figure 3 summarizes the four steps of the procedure. We explain these four steps from Section 4.3.1 to Section 4.3.4, and after that, we demonstrate how they work with a concrete example in Section 4.3.5.

The inverter takes only one program, either leftwards or rightwards, to derive a weak right inverse. Note that the inverter with a single program does not make up the correct parallelization algorithm without the other one, since both sequential programs of the pair are required to guarantee the existence of a parallel program as stated in Theorem 3.3.

4.3.1 Unfolding Functions and Getting Constraint Equations

As the first step to obtain a weak right inverse, we unfold the original functions to get relational expressions (constraints) that explicitly describe the relation between their input list and their output values. Since the input list of the original functions corresponds to the output of the objective weak right inverse, our objective is to derive the computation that computes the input list from the output values.

Taking the maximum prefix sum problem (Example 3.2) as an example, we may assume that the weak right inverse $(mps \triangle sum)^\circ$ takes $(p,\ s) \in \mathrm{range}(mps \triangle sum)$ and returns a singleton list $[a]$ as

$$(mps \triangle sum)^\circ\ (p,\ s) = [a].$$

Then we get the following equation according to the definition of the weak right inverse as

$$(p,\ s) = (mps \triangle sum)\ [a]$$

that is,

$$p = mps\ [a],\ \ s = sum\ [a].$$

By unfolding the functions mps and sum, we get

$$p = \mathbf{if}\ (a \leq 0)\ \mathbf{then}\ 0\ \mathbf{else}\ a,\ \ s = a$$

```
// function
//     Inversion
// input
//     funcSet : all functions included in the program
// output
//     a weak right inverse of the tupled function

function Inversion (funcSet)
    return Optimize(Solve(Unfold(funcSet)))

function Unfold (funcSet)
    constraintEquation = funcSet.Unfold(funcSet)
    return constraintEquation.EnumCond()

function Solve (constraintEquationSet)
    foreach ((cond, eq) in constraintEquationSet)
        exprs = eq.solve()
        cond.Subst(exprs)
        weakRightInverse.Add(cond, exprs)
    return weakRightInverse

function Optimize (weakRightInverse)
    foreach ((cond, exprs) in weakRightInverse)
        if (!cond || weakRightInverse.Or())
            then weakRightInverse.Remove(cond,exprs)
    return weakRightInverse

// function
//     Verification
// input
//     preCond : the precondition that the functions satisfy
//     weakRightInverse : a weak right inverse
// output
//     whether the weak right inverse is valid

function Verification (preCond, weakRightInverse)
    return (!preCond || weakRightInverse.Or())
```

Figure 3. Algorithm to derive a weak right inverse

which can be expressed by the constraint equations with conditions in the form of $C \Rightarrow E_1, E_2, \ldots, E_n$, where C is a constraint and each E_i is an equation:

$$\begin{array}{l} \{a \leq 0\} \ \Rightarrow\ p = 0,\ s = a \\ \{a > 0\} \ \Rightarrow\ p = a,\ s = a. \end{array}$$

These two constraint equations imply that the weak right inverse $(mps \triangle sum)^\circ\ (p, s)$ returns the singleton list $[s]$ when (1) p is 0 and s is 0 or less, or (2) s is positive and equal to p.

The result is not satisfactory: The result is a partial definition of a weak right inverse and we know nothing about the case where the weak right inverse should return a longer list. It is obviously impossible to run the same algorithm for all the possible lengths of the output lists, though it would derive a full definition of a weak right inverse.

To resolve this problem, we assume that a weak right inverse returns a list whose length is the same as the number of defined functions. For example, we assume that $(mps \triangle sum)^\circ$ returns a list of two elements. This assumption is problematic because the assumption may narrow the domain of the derived weak right inverse. We will discuss the correctness of the derived weak right inverse in Section 4.3.4.

4.3.2 Solving Constraint Equations

After unfolding the functions, we get the simultaneous equations that express the relation between the input variables and the output list of the objective weak right inverse. We then solve these equations using the constraint solver. If there are some variables we cannot decide their values uniquely, we substitute an arbitrary value for one of them. After that, we substitute the result of the simultaneous equations for the variables of the conditional expressions, and derive a weak right inverse.

4.3.3 Optimization

In general, the weak right inverse derived by the above-mentioned process is inefficient: If there are n functions with m conditional branches, the number of constraints can be $2^{m(n+1)}$ in the worst case. Since there usually exist many unnecessary branches, we can improve the efficiency of the weak right inverse by eliminating unnecessary branches. Let C_i be the ith conditional branch. If

$$C_i \Rightarrow \bigvee_{i \neq k} C_k \tag{1}$$

holds, then the domain of the weak right inverse does not change even if C_i is removed. Moreover, all the branches return a correct list as a weak right inverse if the input value is in the domain. Therefore, the expression corresponding to C_i is redundant and can be removed. The expression (1) is in the form of Presburger arithmetic [25], and we can compute it by quantifier elimination [10].

4.3.4 Verification

As mentioned in Section 4.3.1, the derived weak right inverse may be a partial function. That is, $\bigvee_i C_i = \mathit{true}$ may not hold, where C_i denotes the ith condition. However, it is sufficient for a weak right inverse to return the value in the range of the original function. In other words, if

$$P \Rightarrow \bigvee_i C_i \tag{2}$$

holds, where P corresponds to the range of the original function, then the weak right inverse meets the requirement of the definition. Our algorithm works as follows. At first, the algorithm checks whether $\bigvee_i C_i = \mathit{true}$ holds. If not, the algorithm requires the precondition that corresponds to the range of the original function, and tries checking the condition (2). These checks can be also done by quantifier elimination. If the verification fails, we fail to derive a weak right inverse.

4.3.5 Example: Inversion of Maximum Prefix Sum

Now let us demonstrate the whole process of inversion by deriving a weak right inverse of $(\mathit{mps} \vartriangle \mathit{sum})$.

Because the number of defined function is two, we assume that a weak right inverse of $(\mathit{mps} \vartriangle \mathit{sum})$ returns a list of two elements:

$$(\mathit{mps} \vartriangle \mathit{sum})^\circ\ (p,\ s) = [a,\ b]$$

where $(p,\ s) \in \mathrm{range}(\mathit{mps} \vartriangle \mathit{sum})$. This equation is equivalent to the following equation:

$$(p,\ s) = (\mathit{mps} \vartriangle \mathit{sum})\ [a,\ b]$$

By unfolding the definitions of $\vartriangle$, sum, and mps, we get the following equations:

$$\begin{aligned} p\ &=\ \mathbf{if}\ (b \leq 0) \\ &\quad\ \mathbf{then\ if}\ (0 \leq a)\ \mathbf{then}\ a\ \mathbf{else}\ 0 \\ &\quad\ \mathbf{else\ if}\ (0 \leq a+b)\ \mathbf{then}\ a+b\ \mathbf{else}\ 0 \\ s\ &=\ a+b \end{aligned}$$

Let us solve these simultaneous equations for a and b to get a weak right inverse. To solve these simultaneous equations, we enumerate all conditional branches and get a set of constraint equations with conditions as follows:

$$\begin{array}{ll} \{b \leq 0 \wedge 0 \leq a\} & \Rightarrow\ p = a,\ \ s = a+b \\ \{b \leq 0 \wedge 0 > a\} & \Rightarrow\ p = 0,\ \ s = a+b \\ \{b > 0 \wedge 0 \leq a+b\} & \Rightarrow\ p = a+b,\ \ s = a+b \\ \{b > 0 \wedge 0 > a+b\} & \Rightarrow\ p = 0,\ \ s = a+b \end{array}$$

Solving the equations gives the following result. Two things are worth noting. Firstly, we cannot uniquely decide the values of a and b in the second, third and fourth equations, so let a be 0 here. Secondly, there is a dependency between p and s in the first and second equations, and we add it to the conditional part:

$$\begin{array}{ll} \{b \leq 0 \wedge 0 \leq a\} & \Rightarrow\ a = p,\ \ b = s-p \\ \{b \leq 0 \wedge 0 > 0 \wedge p = 0\} & \Rightarrow\ a = 0,\ \ b = s \\ \{b > 0 \wedge 0 \leq b \wedge p = s\} & \Rightarrow\ a = 0,\ \ b = s \\ \{b > 0 \wedge 0 > b \wedge p = 0\} & \Rightarrow\ a = 0,\ \ b = s \end{array}$$

Removing clearly unreachable branches and replacing a and b in the conditions by their solution yield the following equations:

$$\begin{array}{ll} \{s \leq p \wedge 0 \leq p\} & \Rightarrow\ a = p,\ \ b = s-p \\ \{s > 0 \wedge p = s\} & \Rightarrow\ a = 0,\ \ b = s \end{array}$$

Therefore we get the following program of $(\mathit{mps} \vartriangle \mathit{sum})^\circ$, because $(\mathit{mps} \vartriangle \mathit{sum})^\circ\ (p,\ s) = [a,\ b]$:

$$\begin{aligned} &(\mathit{mps} \vartriangle \mathit{sum})^\circ (p,\ s) \\ &\quad =\ \mathbf{if}\ (s \leq p \wedge 0 \leq p)\ \mathbf{then}\ [p,\ s-p] \\ &\qquad\ \mathbf{else\ if}\ (s > 0 \wedge p = s)\ \mathbf{then}\ [0,\ s] \end{aligned}$$

Next, we optimize this weak right inverse. Since the condition $\{s \leq p \wedge 0 \leq p\}$ includes $\{s > 0 \wedge p = s\}$, our optimizer removes the branch and yields the following result:

$$\begin{aligned} &(\mathit{mps} \vartriangle \mathit{sum})^\circ (p,\ s) \\ &\quad =\ \mathbf{if}\ (s \leq p \wedge 0 \leq p)\ \mathbf{then}\ [p,\ s-p] \end{aligned}$$

Finally, we verify the weak right inverse. While the weak right inverse is not a total function, it is correct, because the condition (2) holds; $s \leq p \wedge 0 \leq p$ holds for all list x, where $(\mathit{mps} \vartriangle \mathit{sum})\ x = (p,\ s)$, because the return value of the function mps is larger than or equal to that of sum. To verify the correctness, users specify $s \leq p \wedge 0 \leq p$ when our algorithm requires the precondition of $(\mathit{mps} \vartriangle \mathit{sum})$.

Now that we have got a weak right inverse of $(\mathit{mps} \vartriangle \mathit{sum})$, Theorem 3.3 soon gives a parallel program seen in Section 4.1.

4.4 Properties

Two remarks are worth making on the properties of our parallelization algorithm.

First, our inversion procedure always terminates, and moreover, the derived weak right inverse is always correct provided that the verification step succeeds.

Second, our derived parallel programs are guaranteed to be efficient in the sense that they use $\mathrm{O}(\log n)$ parallel time, where n is the length of the input list. This is because the weak right inverse returns a list of constant length, and thus the computation of $\odot$ in Theorem 3.3 uses constant time.

5. Implementation and Experiments

We have implemented an automatic parallel code generator on the parallelization algorithm in C++. Two major issues in our implementation are the speedup of the optimization step and the generation of architecture-independent parallel programs.

In the optimization step (Section 4.3.3), we need computation on a large number of constraint equations, which occupies most of the time in our parallelization algorithm. To resolve this problem, we note that all the expressions in the optimization step are in the

form of Presburger arithmetic, and we used the Omega library that solves truth judgments in the Presburger arithmetic fast based on the *omega test* method [26]. The use of the Omega library enables our parallel code generator to work in practical time.

The generation of architecture-independent executable parallel programs is another implementation issue. We need to take into account the architecture of parallel computers to generate efficient parallel programs, but there are so many parallel architectures from shared-memory ones to distributed-memory ones that architecture-specific implementation is almost impossible. Our approach is to generate parallel programs that can be combined with the SkeTo library [24], which provides not only map and $reduce$ but also $scanl$ and $scanr$ as parallel primitives based on the MPI environments. Our parallelization system generates C++ code of the main routine and the function objects used with the parallel primitives.

In the rest of this section, we demonstrate the ability of our parallelization tools with two more examples, and give experimental results on the efficiency of our system.

5.1 Maximum Segment Sum Problem

To show the power of our system, we consider the maximum segment sum problem, which computes the maximum of the sums for all the segments (contiguous sublists) of a list. It is an instance of the maximum weight-sum problems [29] that capture many dynamic-programming problems. Our system can automatically parallelize all the problems on lists in [29]. As an example, we show automatic parallelization of the maximum segment sum problem.

The input sequential programs for the maximum segment sum is given as follows:

```
main = mss

mss [a]      = max(a, 0);
mss ([a]++x) = max(a + mps(x), mss(x));
mss (x++[a]) = max(mss(x), mts(x) + a);

mps [a]      = max(a, 0);
mps ([a]++x) = max(0, a + mps(x));
mps (x++[a]) = max(mps(x), sum(x) + a);

mts [a]      = max(a, 0);
mts ([a]++x) = max(mts(x), a + sum(x));
mts (x++[a]) = max(mts(x) + a, 0);

sum [a]      = a;
sum ([a]++x) = a + sum(x);
sum (x++[a]) = sum(x) + a;
```

where `max` is a macro and defined as follows:

$$\texttt{max(x, y)} \Rightarrow \texttt{if (x>=y) then x else y}$$

and `mts` is a function to compute the maximum sum of all the postfix sublists. From this input source code, our parallelization system generates the following weak right inverse for the function $(\texttt{mss} \triangle \texttt{mps} \triangle \texttt{mts} \triangle \texttt{sum})$.

```
weakinv(mss & mps & mts & sum) (m, p, t, s)
  = if(0 <= p <= m && 0 <= t <= m && s+m <= t+p)
      then [p, -p-t+s, m, -m+t]
```

And based on the Theorem 3.3 with this weak right inverse, our code generator generates the parallel program for the SkeTo library as shown in Figure 4.

5.2 Line-of-Sight Problem

Given an observation point and a list of buildings along a line (see Figure 5), the line-of-sight problem is to find which buildings are visible originating at the observation point [7]. Let each building be represented by the vertical angle from the observation point to the top of the building. Then, a building on the line is visible if and only if no other building between it and the observation point has a greater vertical angle.

We start by solving this problem sequentially. Since we have to compute on each point, it is natural to use the form of $map\ f \circ inits$ as discussed in Section 4.1.

```
main = map(visible) . inits;

visible [a]      = 1;
visible ([a]++x) = if (a <= amax(x) && visible(x) == 1)
                     then 1 else 0;
visible (x++[a]) = if (amax(x) <= a) then 1 else 0;

amax [a]      = a;
amax ([a]++x) = max(a, amax(x));
amax (x++[a]) = max(amax(x), a);
```

From this source code the system automatically generates a weak right inverse for the function $(\texttt{visible} \triangle \texttt{amax})$.

```
weakinv(visible & amax) (v,m)
  = if(v == 1 && 0 <= m) then [0, m]
    else if(v == 0 && 0 <= m) then [m, 0]
```

Finally, based on this weak right inverse, the system generates parallel code using skeletal primitives, map and $scanl$. Note that

```
struct mss_tuple_t {
  int m, p, t, s;
  mss_tuple_t(int m_, int p_, int t_, int s_) {
    m = m_; p = p_; t = t_; s = s_; }
  mss_tuple_t(){}
};

struct func_t : public skeleton::unary_function
                        <int, mss_tuple_t> {
  mss_tuple_t operator()(int a) const {
    return mss_tuple_t(max(a, 0), max(a, 0),
                       max(a, 0), a); }
} func;

inline mss_tuple_t fwd(int *ar, int n) {
  if (n == 1) {
    return mss_tuple_t(max(*ar, 0), max(*ar, 0),
                       max(*ar, 0), *ar);
  } else {
    mss_tuple_t x = fwd(ar + 1, n - 1);
    return mss_tuple_t(max(*ar + x.p, x.m),
      max(0, *ar + x.p),max(x.t, *ar + x.s), *ar + x.s);
  }
}

inline void bwd(const mss_tuple_t &x, int* ar) {
  ar[ 0 ] = x.p; ar[ 1 ] = -x.p - x.t + x.s;
  ar[ 2 ] = x.m; ar[ 3 ] = -x.m + x.t;
}

struct odot_t : public skeleton::binary_function
            <mss_tuple_t, mss_tuple_t, mss_tuple_t> {
  mss_tuple_t operator()(const mss_tuple_t &x,
                         const mss_tuple_t &y) const {
    int ar[4*2]; bwd(x, ar); bwd(y, ar+4);
    return fwd(ar, 4*2);
  }
} odot;

 ...
  /* main routine */
  dist_list<mss_tuple_t> *list1
    = list_skeletons::map(func, data);
  mss_tuple_t result
    = list_skeletons::reduce(odot, list1);
  return result.m;
```

Figure 4. The generated parallel program for the maximum segment sum problem. The `bwd` function is the implementation of the weak right inverse and `func` and `odot` are function objects that are used in calling skeletal primitives.

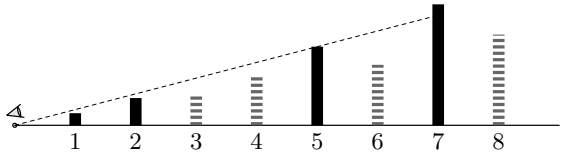

Figure 5. Line-of-sight problem. This figure shows the sequence of buildings. There is no obstacle to see the bldg-7, so the bldg-7 is visible. In this case, the answer is $[1, 1, 0, 0, 1, 0, 1, 0]$.

Table 1. Execution time (in seconds) for inversion.

	Branches	Execution time
Max. prefix sum	4	0.021
Max. segment sum	70	0.209
Line of sight	2	0.021

Table 2. Execution time (in seconds) for generated parallel programs.

Processors	1	4	16	32
Max. prefix sum	3.30	0.845	0.212	0.126
Max. segment sum	6.25	1.58	0.401	0.236
Line of sight	6.53	1.70	0.445	0.250

the following equation holds for the main program:

$$\begin{aligned} los &= map\ visible \circ inits \\ &= map\ (fst \circ (visible\ \triangle\ amax)) \circ inits \\ &= map\ fst \circ map\ (visible\ \triangle\ amax) \circ inits \\ &= map\ fst \circ scanl\ (\odot) \end{aligned}$$

where fst is a function that returns the first value of the pair and $\odot$ is the associative operator derived from the weak right inverse of $(\texttt{visible} \triangle \texttt{amax})$.

5.3 Experimental Results

To verify the efficiency of our parallelization algorithm and the generated parallel programs, we have made the following experiments.

First, we give the experimental result of the parallelization algorithm. It is worth remarking that for mss, nine if-statements implied in the function `max` cause 70 branches in total before optimization. But due to the use of the Omega library, it takes only 0.209 s in deriving the weak right inverse. Table 1 shows the execution times of computing weak right inverses for three examples.

Next we show the experimental result of the generated parallel programs. We used our PC-clusters that consists of uniform PCs with Pentium 4 Xeon 2.4 GHz CPU and 2 GB of memory connected Gigabit Ethernet. The OS is Linux 2.4 and the C++ compiler and the base MPI library used in SkeTo library are gcc 4.1.1 and mpich 1.2.7 respectively. We executed three parallel programs for the maximum prefix sum problem, the maximum segment sum problem, and the line-of-sight problem using an array of 64M (almost 67 million) elements using up to 32 processors. The execution times are given in Table 2 and the relative speedups against each programs executed on one processor are given in Figure 6. With these results, we can confirm the efficiency of the parallel programs generated by our system.

6. Discussion and Related Work

In this section, we discuss most related work in addition to that in the introduction, and highlight limitations and extensions of our parallelization system.

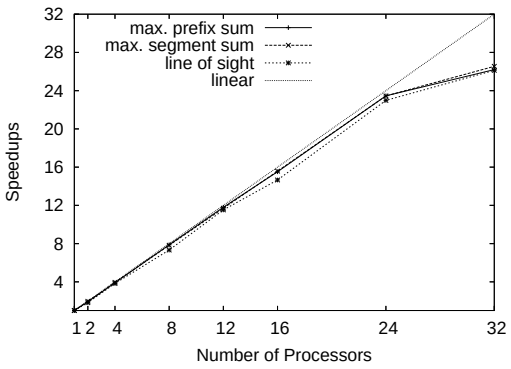

Figure 6. Speedups for the three programs

6.1 Derivation of List Homomorphism

The research on parallelization via derivation of list homomorphisms has gained great interest, and there have been many approaches, such as the third homomorphism theorem based method [15, 19], function composition based method [14], matrix multiplication based method [31] and recurrence equation based method [2]. Our approach is unique in its use of weak right inverse in derivation of parallel programs. One of the advantages of our approach is that our parallelization framework works well for any function, if we have a way to obtain a weak right inverse. We used the Omega library to solve truth judgments in the Presburger arithmetic. We would derive the more parallel programs from sequential programs if we use the more powerful constraint solvers. In addition, It might be possible to extend our approach to other data structures such as trees, but this could be difficult with the other approaches.

6.2 Inversion and Parallelization

We have reduced parallelization process to derivation of a weak right inverse, and thus our approach is related to researches on automatic inversion [12, 13, 17, 18, 20, 23]. Different from the inverse that does not always exist for a given function, our weak right inverse always exists for any function. To guarantee the efficiency of the weak right inverse, as in Section 4, we impose some restriction on our objective functions and obtain a weak right inverse by solving linear equations. Though it may look restrictive, our framework is powerful enough to solve many practical examples including most recurrence equations, and to generate efficient parallel programs within the framework of Presburger arithmetics. It will be interesting to see if we can make better use of techniques on inversion to parallelize more functions.

6.3 Generalization

We have an assumption (as in Section 4.3.1) that the output list of a weak right inverse is of constant length. This assumption excludes many functions that are necessary for parallel computation. One example is the function $length$, which computes the length of a list. Our algorithm cannot derive a weak right inverse of $length$ though $length$ is both leftwards and rightwards.

$$\begin{aligned} length\ [a] &= 1 \\ length\ ([a] +\!\!+ x) &= 1 + length\ x \\ length\ (x +\!\!+ [a]) &= length\ x + 1 \end{aligned}$$

This is because variable a does not appear in the definition body, which is required by our system. This problem may be solved by generalization: generalizing constant 1 to variable a. This generalization would split $length$ into two functions, $length'$ and map, as follows:

$$
\begin{aligned}
length &= length' \circ map\ one \\
length'\ [a] &= a \\
length'\ ([a] +\!\!+\ x) &= a + length'\ x \\
length'\ (x +\!\!+\ [a]) &= length'\ x + a
\end{aligned}
$$

where one is the function which always returns 1 for any input. Now we can derive a parallel program of $length$ because our inversion algorithm can deal with $length'$. (In fact, $length'$ is sum.)

7. Conclusion

In this paper, we have introduced a new concept called *weak right inverse*, proposed a novel parallelization framework based on the third homomorphism theorem and derivation of a weak right inverse, and implemented a parallelization system that can automatically generate efficient parallel programs suitable for the divide-and-conquer paradigm. The experimental results show promise of the approach.

We are now looking into how to formalize and automate the generalization step discussed in Section 6. In addition, we are considering using more powerful constraint solvers in our system so that more involved constraint equations can be solved and more sequential programs can be parallelized. Extending our approach to other data structures such as trees is interesting future work.

Acknowledgments

The authors would like to thank Isao Sasano and Shin-Cheng Mu for valuable discussions with them, and the anonymous referees for their variable advice. This work was partially supported by Japan Society for the Promotion of Science, Grant-in-Aid for Scientific Research (B) 17300005, and the Ministry of Education, Culture, Sports, Science and Technology, Grant-in-Aid for Young Scientists (B) 18700021.

References

[1] J. Ahn and T. Han. An analytical method for parallelization of recursive functions. *Parallel Processing Letters*, 10(1):87–98, 2000.

[2] Y. Ben-Asher and G. Haber. Parallel solutions of simple indexed recurrence equations. *IEEE Transactions on Parallel and Distributed Systems*, 12(1):22–40, 2001.

[3] J. Bentley. Algorithm design techniques. In *Programming Pearls,* Column 7, pages 69–80. Addison-Wesley, 1986.

[4] R. S. Bird. An introduction to the theory of lists. In *Logic of Programming and Calculi of Discrete Design*, *NATO ASI Series F* 36, pages 5–42. 1987.

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

[6] G. E. Blelloch. Scans as primitive operations. *IEEE Transactions on Computers*, 38(11):1526–1538, 1989.

[7] G. E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, 1990.

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

[9] M. Cole. Parallel programming with list homomorphisms. *Parallel Processing Letters*, 5(2):191–203, 1995.

[10] D. C. Cooper. Theorem proving in arithmetic without multiplication. *Machine Intelligence*, 7:91–99, 1972.

[11] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *6th Symposium on Operating System Design and Implementation (OSDI 2004)*, pages 137–150, 2004.

[12] E. W. Dijkstra. Program inversion. In *Program Construction*, *LNCS* 69, pages 54–57. 1978.

[13] D. Eppstein. A heuristic approach to program inversion. In *Proceedings of the 9th International Joint Conferences on Artificial Intelligence*, pages 219–221, 1985.

[14] A. L. Fisher and A. M. Ghuloum. Parallelizing complex scans and reductions. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI '94)*, pages 135–146, 1994.

[15] A. Geser and S. Gorlatch. Parallelizing functional programs by generalization. In *Algebraic and Logic Programming (ALP'97)*, *LNCS* 1298, pages 46–60. 1997.

[16] J. Gibbons. The third homomorphism theorem. *Journal of Functional Programming*, 6(4):657–665, 1996.

[17] R. Glück and M. Kawabe. A program inverter for a functional language with equality and constructors. In *Programming Languages and Systems. Proceedings*, *LNCS* 2895, pages 246–264. 2003.

[18] R. Glück and M. Kawabe. Derivation of deterministic inverse programs based on LR parsing. In *Functional and Logic Programming, 7th International Symposium (FLOPS 2004), Proceedings*, *LNCS* 2998, pages 291–306. 2004.

[19] S. Gorlatch. Systematic extraction and implementation of divide-and-conquer parallelism. In *Programming languages: Implementation, Logics and Programs. PLILP'96*, *LNCS* 1140, pages 274–288. 1996.

[20] D. Gries. Inverting programs. In *The Science of Programming*, chapter 21, pages 265–274. 1981.

[21] 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*, 19(3):444–461, 1997.

[22] Z. Hu, M. Takeichi, and W. N. Chin. Parallelization in calculational forms. In *25th ACM Symposium on Principles of Programming Languages (POPL '98)*, pages 316–328, 1998.

[23] R. E. Korf. Inversion of applicative programs. In *Proceedings of the 7th International Conferences on Artificial Intelligence (IC-AI '81)*, pages 1007–1009, 1981.

[24] K. Matsuzaki, K. Emoto, H. Iwasaki, and Z. Hu. A library of constructive skeletons for sequential style of parallel programming (invited paper). In *1st International Conference on Scalable Information Systems (InfoScale 2006)*, 2006.

[25] M. Presburger. Uber die vollstandigkeit eines gewissen systems der arithmetik ganzer zahlen, in welchem die addition als einzige operation hervorstritt. *Sprawozdanie z I Kongresu Matematikow Krajow Slowcanskich Warszawa*, pages 92–101, 1929.

[26] W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 4–13, 1991.

[27] F. Rabhi and S. Gorlatch. *Patterns and Skeletons for Parallel and Distributed Computing*. 2002.

[28] R. Rugina and M. C. Rinard. Automatic parallelization of divide and conquer algorithms. In *Proceedings of the 7th ACM Symposium on Principles Practice of Parallel Programming (PPoPP '99)*, pages 72–83, 1999.

[29] I. Sasano, Z. Hu, M. Takeichi, and M. Ogawa. Make it practical: A generic linear time algorithm for solving maximum-weightsum problems. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*, pages 137–149. 2000.

[30] G. Steele. Parallel programming and parallel abstractions in fortress. In *Functional and Logic Programming, 8th International Symposium (FLOPS 2006), Proceedings*, *LNCS* 3945, page 1. 2006.

[31] D. N. Xu, S. C. Khoo, and Z. Hu. PType system: A featherweight parallelizability detector. In *Proceedings of 2nd Asian Symposium on Programming Languages and Systems (APLAS 2004)*, *LNCS* 3302, pages 197–212. 2004.