D4-2

Implementing List Homomorphisms in Parallel

胡 振江 Zhenjiang HU　岩崎 英哉 Hideya IWASAKI　武市 正人 Masato TAKEICHI

東京大学 大学院工学系研究科
Department of Information Engineering
University of Tokyo
({hu,iwasaki,takeichi}@ipl.t.u-tokyo.ac.jp)

概 要

List homomorphisms are a useful parallel template (skeleton) in parallel programming, not only because they enjoy many algebraic rules suitable for program manipulation, but also because they can be "obviously" implemented in parallel using the divide-and-conquer paradigm. However, such obvious implementation may be potentially sequential because of some hidden dependence. This paper is intended to show that homomorphisms can be efficiently implemented in parallel, but the implementation is far from being that trivial. We shall propose a new algorithm for transforming list homomorphisms into NESL programs in which the so-called *scan* operator effectively encapsulates a parallel structure common to many parallel architectures. We illustrate our method by deriving a novel parallel algorithm for bracket matching.

1 Introduction

So many studies have been devoted to making use of homomorphisms, a parallel skeleton, in parallel programming [Col95] [GDH96] [Gor96] [HIT97] [HTC98].

List homomorphisms (or homomorphisms for short) [Bir87] are those functions on (nonempty) finite lists that *promote* through list concatenation:

$$h\ (x +\!\!+ y) \quad = \quad h\,x \oplus h\,y$$

where $\oplus$ is an associative operator. Intuitively, the definition of list homomorphisms means that the value of h on the larger list depends in a particular way (using binary operation $\oplus$) on the values of h applied to the two pieces of the list. The computations of $h\,x$ and $h\,y$ are independent of each other and can thus be carried out in parallel. This simple equation can be viewed as expressing the well-known divide-and-conquer paradigm in parallel programming.

So we have taken it for granted that the parallelism in homomorphisms can be easily developed because of its divide-and-conquer form. However, the thing is not so simple as it appears. To appreciate the problem, consider the homomorphism in Figure 1, as being derived in [HTC98], determining whether the brackets of '(' and ')' is matched or not in a a given string.

Note that *tup* function is a homomorphism returning a *function* (rather than a basic value) as its result. This kind of homomorphisms usually hide dependence between two recursive calls which are expected to be computed independently. In the definition of *tup* $(x +\!\!+ y)$, the computation of *tup* y in fact requires (depends on) the

$$
\begin{aligned}
&\mathit{sbp}\ x\ c = s \\
&\quad \text{where } (s, g_1, g_2) = \mathit{tup}\ x\ c \\
&\mathit{tup}\ [a] = \lambda c. \\
&\quad \text{if } a == \text{'('} \text{ then } (c+1 == 0, \mathit{True}, 1) \\
&\quad \text{else if } a == \text{')'} \text{ then } (c-1 == 0, c > 0, -1) \\
&\quad \text{else } (c == 0, \mathit{True}, 0) \\
&\mathit{tup}\ (x +\!\!+ y) = \lambda c. \\
&\quad \text{let } (s_x, g_{1x}, g_{2x}) = \mathit{tup}\ x\ c \\
&\qquad\ (s_y, g_{1y}, g_{2y}) = \mathit{tup}\ y\ (g_{2x} + c) \\
&\quad \text{in } (g_{1x} \wedge s_y,\ g_{1x} \wedge g_{1y},\ g_{2x} + g_{2y})
\end{aligned}
$$

図 1 A homomorphism for bracket matching

result of g_{2x} from the computation of $\mathit{tup}\ x\ c$. As a result, if we simply use the traditional way of computing $\mathit{tup}\ x$ and $\mathit{tup}'\ y$ independently to get result for $\mathit{tup}\ (x +\!\!+ y)$, many computations are remained and will be performed *sequentially* until the second parameter c for tup is applied.

This paper is intended to show that homomorphisms can be efficiently implemented in parallel, but this implementation is far from being that trivial. To be more concrete, we shall propose a novel transformation effectively turning list homomorphisms into NESL programs [Ble92] in which the so-called *apply-to-each* and *scan* operators encapsulate efficient parallel computational patterns. We choose the NESL as our target, because practically efficient code can be generated from NESL programs for a variety of architectures, from vector multiprocessors (CRAY C90 and J90) to distributed memory machines (IBM SP2, Intel Paragon, CM-5). Therefore, efficiently transforming homomorphisms to NESL programs can lead to efficient codes for these parallel architectures.

2 List Homomorphisms

List homomorphisms (or *homomorphisms* for short) are an important concept in Bird-Meertens Formalisms (BMF) [Bir87], and are central to this paper. They are functions on finite lists that *promote* through list concatenation, as precisely defined later. Before giving the definition, we introduce some notational conventions in BMF which will be used in this paper.

In BMF, *Function application* is denoted by a space and the argument which may be written without brackets. Thus $f\ a$ means $f\ (a)$. Function application binds stronger than any other operator, so $f\ a \oplus b$ means $(f\ a) \oplus b$, but not $f\ (a \oplus b)$. *Function composition* is denoted by a centralized circle $\circ$. By definition, we have $(f \circ g)\ a = f\ (g\ a)$. Function composition is an associative operator, and the identity function is denoted by *id*.

Definition 1 ((List) Homomorphism) A function h satisfying the following equations is called a *list homomorphism*:

$$
\begin{aligned}
h\ [a] &= k\ a \\
h\ (x +\!\!+ y) &= h\ x \oplus h\ y
\end{aligned}
$$

where $\oplus$ is an *associative* binary operator. We write $(\!|k, \oplus|\!)$ for the unique function h. □

Two important homomorphisms are *map* and *reduction*. Map is the operator which applies a function to every element in a list. It is written as an infix $*$. Informally, we have

$$k * [x_1, x_2, \ldots, x_n] = [k\ x_1, k\ x_2, \ldots, k\ x_n].$$

Reduction is the operator which collapses a list into a single value by repeated application of some binary operator. It is written as an infix /. Informally, for an associative binary operator $\oplus$, we have

$$\oplus/\ [x_1, x_2, \ldots, x_n] = x_1 \oplus x_2 \oplus \cdots \oplus x_n.$$

It has been argued that $*$ and / have simple massively parallel implementations on many architectures [Ski90]. For example, $\oplus/$ can be computed in parallel on a tree-like structure with the combining operator $\oplus$ applied in the nodes, while $k*$ is computed in parallel with k applied to each of the leaves.

The relevance of homomorphisms to parallel programming is basically from the *homomorphism lemma* [Bir87]:

$$(\!|k, \oplus|\!) = (\oplus/) \circ (k*)$$

saying that every list homomorphism can be written as the composition of a reduction and a map.

However, as argued in the introduction, efficient implementing homomorphisms in parallel turns out to be difficult when the homomorphisms de-

note functions that accept a list and return a *function* rather than a basic value. To deal with it, in this paper, we assume that our input is a first order homomorphism, but it may have additional accumulation parameters and its result may contain many components. To be precise, we give the following definition of our input list homomorphisms.

Definition 2 (Input Homomorphism) Let h be a *first order* homomorphism, which accepts p accumulation parameters and returns a result with q components. It is defined by

$$
\begin{array}{l}
h\ ([a], (w_1, \ldots, w_p)) = (E_{k_1}[a, W], \ldots, E_{k_q}[a, W]) \\
h\ (x +\!\!+ y, (w_1, \ldots, w_p)) = \\
\quad \text{let} \\
\quad\quad (h_{x_1}, \ldots, h_{x_q}) = h\ (x,\ (E_{x_1}[W, H], \ldots, E_{x_p}[W, H])) \\
\quad\quad (h_{y_1}, \ldots, h_{y_q}) = h\ (y,\ (E_{y_1}[W, H], \ldots, E_{y_p}[W, H])) \\
\quad \text{in} \\
\quad\quad (E_1[H, W], \ldots, E_q[H, W])
\end{array}
$$

where W is an abbreviation for $w_1, \ldots, w_p$, and H for $h_{x_1}, \ldots, h_{x_q}, h_{y_1}, \ldots, h_{y_q}$. In addition, we use $E[x_1, \ldots, x_n]$ to represent an expression containing free variables *partly* from $x_1, \ldots, x_n$.

□

For instance our *sbp* can be reexpressed as follows.

$$
\begin{array}{l}
sbp\ (x, c) = s \\
\quad \text{where } (s, g_1, g_2) = tup\ (x, c) \\
tup\ ([a], c) = (k_s\ (a, c), k_{g_1}\ (a, c), k_{g_2}\ (a)) \\
tup\ (x +\!\!+ y, c) = \\
\quad \text{let } (s_x, g_{1x}, g_{2x}) = tup\ x\ c \\
\quad\quad\ (s_y, g_{1y}, g_{2y}) = tup\ y\ (c + g_{2x}) \\
\quad \text{in } (g_{1x} \wedge s_y,\ g_{1x} \wedge g_{1y},\ g_{2x} + g_{2y})
\end{array}
$$

where

$$
\begin{array}{lcl}
k_s\ (a, c) & = & \text{if } a == \text{'('} \text{ then } c + 1 == 0 \\
 & & \text{else if } a == \text{')'} \text{ then } c - 1 == 0 \\
 & & \text{else } c == 0 \\
k_{g_1}\ (a, c) & = & \text{if } a == \text{'('} \text{ then } \mathit{True} \\
 & & \text{else if } a == \text{')'} \text{ then } c > 0 \\
 & & \text{else } \mathit{True} \\
k_{g_2}\ (a) & = & \text{if } a == \text{'('} \text{ then } 1 \\
 & & \text{else if } a == \text{')'} \text{ then } -1 \\
 & & \text{else } 0
\end{array}
$$

3 NESL

This paper intends to implement a first order homomorphism in NESL, a practical parallel language which can run on variety of parallel architectures (see Introduction). NESL [Ble92] is a strongly-typed strict *first-order* functional language. It runs with an interactive environment, and is loosely based on the ML language. The language uses sequences like

$$[x_1, x_2, \ldots, x_n]$$

as a primitive parallel data type. Parallelism is achieved essentially by using two parallel constructs. One is the so-called *apply-to-each* construct:

$$\{f(a)\ :\ a \text{ in } seq\}$$

which is read as "in parallel for each a in the sequence *seq*, apply f to a". It is similar to *list comprehension* in many functional languages.

The other is the *scan* construct [Ble89]:

$$\oplus\text{-}scan\ [x_1, x_2, \ldots, x_n] = [\iota_\oplus, x_1 \oplus x_2, x_1 \oplus \cdots \oplus x_{n-1}]$$

where $\oplus$ is an associative operator with $\iota_\oplus$ as its identity element. Note that *reduct* is a special case of *scan* defined by

$$\oplus\text{-}reduct\ [x_1, x_2, \ldots, x_n] = x_1 \oplus \cdots \oplus x_n.$$

4 Transformation Algorithm

The major difficulty for efficiently implementing homomorphisms in parallel, as explained in the introduction, is the dependence relationship inside homomorphisms themselves. So, we should be clear about such dependence.

We shall consider our input homomorphism h in Definition 2, which has p (accumulation) parameters $w_1, \ldots, w_p$, and returns q components $h_1, \cdots, h_q$ as its result. We have four kinds of dependence relationship, namely

1. *Parameter-Parameter Dependence*: one parameter depends on another parameter;
2. *Parameter-Result Dependence*: a parameter depends on a component of the result;
3. *Result-Parameter Dependence*: a component

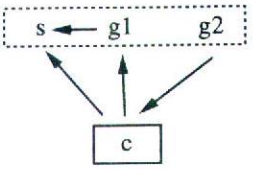

図 2 Dependence graph for *tup*.

of the result depends on an accumulating parameter;

4. *Result-Result Dependence*: a component of the result depends on another component.

Take as an example the homomorphism *tup* for defining *sbp*. It has a single accumulating parameter c and three components s, g_1 and g_2 in the result. Fig. 2 summarizes the dependences in *tup*. It has the parameter-result dependence between c and g_2, the result-parameter dependences between c and g_1 and between c and s, and the result-result dependence between s and g_1.

The idea underlying our transformation algorithm is to reduce the number of both the parameters and components by memoization in vectors (sequences) with scans. We present our transformation algorithm by *induction over the number of parameters*. To simplify our presentation, we assume that the dependence graph has no cyclic. Our algorithm is summarized as follows.

1. If h has no parameter, we use the homomorphism lemma to implement h by apply-to-each and reduct.
2. Otherwise, we repeatedly compute and memoize the components that do not depend on the accumulating parameters, and compute and memoize parameters not depending on components that have not been computed, until we finish computing all parameters. Goto 1.

Recall our example of *tup*. It has a single accumulating parameter depending on the g_2 component. We can apply the transformation algorithm. The following gives our parallel program for *sbp* in a pseudo NESL code, where we use several variables to show memoized intermediate result in each computation step.

$$
\begin{array}{l}
sbp\ (x, c) = s \\
\quad \text{where } (s, g_1) = tup\ (x, c) \\
tup\ (x, c) = \\
\quad \text{let } g_2 s = +\_scan\ [k_{g_2}\ (a)\ :\ a \text{ in } x] \\
\qquad cs = [c + a\ :\ a \text{ in } g_2 s] \\
\qquad (x_1, y_1) \oplus (x_2, y_2) = (x_2 \wedge y_1, x_2 \wedge y_2) \\
\quad \text{in } \oplus\_reduce \\
\qquad [(k_s\ (a, c'), k_{g_1}\ (a, c'))\ :\ c' \text{ in } cs,\ a \text{ in } x]
\end{array}
$$

As shown above, we compute and memoize g_2 with scan first, and then compute and memoize the parameter c with scan again, and finally compute g_1 and s_1 in parallel.

According to the cost model for NESL program, we actually come to an efficient $O(\log N)$ parallel program, where N denotes the size of the input string.

参考文献

[Bir87] R. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 5–42. Springer-Verlag, 1987.

[Ble89] Guy E. Blelloch. Scans as primitive operations. *IEEE Trans. on Computers*, 38(11):1526–1538, November 1989.

[Ble92] G.E. Blelloch. NESL: a nested data parallel language. Technical Report CMU-CS-92-103, School of Computer Science, Carnegie-Mellon University, January 1992.

[Col95] M. Cole. Parallel programming with list homomorphisms. *Parallel Processing Letters*, 5(2), 1995.

[GDH96] Z.N. Grant-Duff and P. Harrison. Parallelism via homomorphism. *Parallel Processing Letters*, 6(2):279–295, 1996.

[Gor96] S. Gorlatch. Systematic extraction and implementation of divide-and-conquer parallelism. In *Proc. Conference on Programming Languages: Implementation, Logics and Programs, LNCS 1140*, pages 274–288. Springer-Verlag, 1996.

[HIT97] 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.

[HTC98] Z. Hu, M. Takeichi, and W.N. Chin. Parallelization in calculational forms. In *25th ACM Symposium on Principles of Programming Languages*, pages 316–328, San Diego, California, USA, January 1998.

[Ski90] D.B. Skillicorn. Architecture-independent parallel computation. *IEEE Computer*, 23(12):38–51, December 1990.