MmpP: Maximum Marking Problems in Parallel

Kazuhiko Kakehi[1], Zhenjiang Hu[1,2] and Masato Takeichi[1]

1. Graduate School of Information Science and Technology, University of Tokyo
2. PRESTO, Japan Science and Technology Corporation

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

Maximum marking problems (Mmp for short) are to put a mark on the entries of some given data structure in a way such that a given constraint is satisfied and the sum of the weights associated with marked entries is as large as possible. It was shown that the linear time algorithm can be obtained provided that the characterizing function of the constraint is a finite homomorphism. This paper demonstrates that we can also have the parallelizable form for solving Mmp through incorporating accumulation into list homomorphisms and focusing on the property of finiteness in the accumulation.

1 Introduction

Maximum marking problems (Mmp for short) are to put a mark on the entries of some given data structure in a way such that a given constraint is satisfied and the sum of the weights associated with marked entries is as large as possible. For example, k-maximum segment sum problems (k-mss for short) find the maximum weight of the marked adjacent segment(s) separated at most in k parts. When $[3, -4, 2, -1, 6, -3]$ is given, the following mark (denoted by underline) returns the maximum weight for each k.

$$\begin{array}{ll} [3, -4, \underline{2, -1, 6}, -3] & (k = 1) \\ [\underline{3}, -4, \underline{2, -1, 6}, -3] & (k = 2) \\ [\underline{3}, -4, \underline{2}, -1, \underline{6}, -3] & (k \geq 3) \end{array}$$

This paper demonstrates the parallelization of Mmp. It was shown that the linear time algorithm can be obtained provided that the characterizing function of the constraint is a finite homomorphism [8, 2]. Homomorphism displays good characteristics for parallelization [3, 4, 5, 6], but currently it does not co-operate with accumulation which often enables us to describe functions smoothly. We first define $\mathcal{H}$-homomorphism which incorporates the notion of accumulation into homomorphisms and show equivalence between sequential specifications called $\mathcal{H}$-form and $\mathcal{H}$-homomorphisms. We then focus on the cases where the accumulation domain is finite, which are formalized in $\mathcal{H}'$-form, and see they are translated into homomorphisms.

We finally show that Mmp can be reformatted in an almost homomorphism via specifying their constraints in $\mathcal{H}'$-form. As a notable example, we derive a solution for k-mss which runs in parallel efficiently.

Preliminaries

Throughout this paper, we shall use the notation of BMF (Bird-Meertens Formalism) [1, 9], which enables us to describe both of programs and their transformation concisely. For readability we also borrow the notation of Haskell [7].

2 Accumulative homomorphisms

In order to extend homomorphisms with accumulation, we define *$\mathcal{H}$-homomorphism* that utilizes an accumulation for inheriting computation from the previous call.

Definition 1 ($\mathcal{H}$-Homomorphism) Function h is said to be an *$\mathcal{H}$-homomorphism*, if there exist associative operators $\oplus$ and $\otimes$, a binary operator $\ominus$, and a homomorphism g, such that

$$\begin{array}{lcl} h\ [a]\ c & = & c \ominus a \\ h\ (x +\!\!+ y)\ c & = & h\ x\ c \oplus h\ y\ (c \otimes g\ x)\ . \end{array}$$

Here, g is a homomorphism satisfying

$$\begin{array}{lcl} g\ [a] & = & k\ a \\ g\ (x +\!\!+ y) & = & g\ x \otimes g\ y\ . \end{array}$$

□

$\mathcal{H}$-homomorphism is a natural extension of homomorphism. The additional accumulating parameter c serves for information propagation. When a list is divided into x and y, computation on x receives the original c while h on y uses accumulative information also related to x.

Cyclic dependency is avoided from information propagation as the definition indicates. Therefore we do not treat the case where update of the accumulation parameter for computation on x uses information related to y. If the accumulation parameter is not used at all (*i.e.*, *dead*), $\mathcal{H}$-homomorphism degenerates to homomorphism.

The following lemma shows that we can evaluate $\mathcal{H}$-homomorphisms in parallel. It is implemented using the four parallel skeletons *map* (*fun* $*$ *list*), *reduce* (*fun* / *list*), *scan* (*fun* $/\!\!/_{unit}$ *list*) and *zipwith* (*list* Υ_{fun} *list*), written as infix operators.

Lemma 1 ($\mathcal{H}$-Homomorphism in Skeletons) An $\mathcal{H}$-homomorphism h defined above can be decomposed into the form using parallel skeletons.

$$h\ x\ c\ =\ \oplus\ /\ ((\otimes\ /\!\!/_c\ (k * x))\ \Upsilon_{\ominus}\ x)$$

□

3 $\mathcal{H}$-form

It is often the case that a function is defined *sequentially* by induction on the cons list, rather than on the join list. In order to enable smooth transformation, we define the following *$\mathcal{H}$-form*. The following lemma shows the function in $\mathcal{H}$-form can be transformed in $\mathcal{H}$-homomorphism.

Definition 2 ($\mathcal{H}$-Form) Let p, q, r be given functions, $\oplus$ and $\otimes$ be associative operators. The function f is said to be in *$\mathcal{H}$-form*, if it is defined in the following (sequential) recursive form, which has a parallel equivalent of $\mathcal{H}$-homomorphism.

$$\begin{array}{lcl} f\ [\,]\ c & = & r\ c \\ f\ (a : x)\ c & = & p\ a\ c \oplus f\ x\ (c \otimes q\ a) \end{array}$$

We write $f = \mathcal{H}[\![r, (p, \oplus), (q, \otimes)]\!]$. □

Lemma 2 ($\mathcal{H}$-Form into $\mathcal{H}$-Homomorphism) A $\mathcal{H}$-form function $f = \mathcal{H}[\![r, (p, \oplus), (q, \otimes)]\!]$ can be redefined in terms of a $\mathcal{H}$-homomorphism h as follows.

$$\begin{array}{lcl} f\ x\ c_0 & = & \mathit{fst}\ (h\ x\ c_0) \\ h\ [a]\ c & = & (p\ a\ c \oplus r\ (c \otimes q\ a),\ p\ a\ c) \\ h\ (x +\!\!+ y)\ c & = & h\ x\ c \oplus' h\ y\ (c \otimes g\ x) \\ (a_1, b_1) \oplus' (a_2, b_2) & = & (b_1 \oplus a_2, b_1 \oplus b_2) \\ g\ [a] & = & q\ a \\ g\ (x +\!\!+ y) & = & g\ x \otimes g\ y \end{array}$$

fst takes out the first in a pair. □

4 $\mathcal{H}'$-Form: $\mathcal{H}$-Form with Finite Accumulation

Consider a example to count the number of 'mountains' from a list of three tags, *Up*, *Dn*, *Fl*.

$$\begin{array}{l} \mathit{cmnt}\ x\ \ = \mathit{cmnt}'\ x\ (\mathit{Up}, \mathit{False}) \\ \quad \mathit{cmnt}'\ [\,]\qquad (c_1, c_2) \\ \qquad\qquad = \mathtt{if}\ (\mathit{isUp}\ c_1 \wedge c_2)\ \mathtt{then}\ 1\ \mathtt{else}\ 0 \\ \quad \mathit{cmnt}'\ (a : x)\ (c_1, c_2) \\ \qquad\qquad = (\mathtt{if}\ (\mathit{isUp}\ c_1 \wedge \mathit{isDn}\ a)\ \mathtt{then}\ 1\ \mathtt{else}\ 0) \\ \qquad\qquad\quad + \mathit{cmnt}'\ x\ (\ \mathtt{if}\ \mathit{isFl}\ a\ \mathtt{then}\ (c_1, c_2) \\ \qquad\qquad\qquad\qquad\qquad\qquad \mathtt{else}\ (a, \mathit{True})\ ) \end{array}$$

These tags indicate the position is a place of climbing up, sloping down, or flat plane, respectively. With the help of accumulation we can write a program *cmnt*: The accumulation passes the current mark (*Up* or *Dn*) to the successive computation. When the current mark is *Fl*, the new accumulation refers to the incoming accumulation not to lose which direction the preceding list has. The boolean value in the accumulating pair works to return the correct value 0 when the input list is either empty or has *Fl*s only. Due to the dependency, we cannot easily find an associative operator $\otimes$ when we try to fit *cmnt*′ in the $\mathcal{H}$-form.

4.1 Closures in a Finite Domain

One idea for this is to use closures, where we can enjoy associativity of composition. Representing them in a naive way, however, just results in holding a big and enlarging closure.

Closures on a finite domain and range have good properties. Given a function $f :: A \to B \to B$ with

finite $B = \{b_1, \ldots, b_n\}$, a closure of this function can be represented using case branching once the first parameter a_1 is specified:

$$\begin{array}{rcl} f\ a_1 & = & \lambda\, b\ .\ \texttt{case}\ x\ \texttt{of} \quad b_1 \to b'_1 \\ & & \qquad\qquad \cdots \\ & & \qquad\qquad b_n \to b'_n\ , \end{array}$$

where $b'_i = f\ a_1\ b_i$. Closures commonly keep their function body as it is, except for the parameters already filled. When the domain of the unfilled parameter is finite, we can perform preemptive computation by specifying the unfilled part exhaustively. Naturally $\{b'_1, \ldots, b'_n\} \subseteq B$. Different a_2 may construct different case branching, yet they are composible since they share B as their domain and range. Therefore

$$\begin{array}{l} (f\ a_2) \circ (f\ a_1) \\ = \left(\begin{array}{r} \lambda\, b\ .\ \texttt{case}\ b\ \texttt{of} \\ b_1 \to b''_1 \\ \cdots \\ b_n \to b''_n \end{array} \right) \circ \left(\begin{array}{r} \lambda\, b\ .\ \texttt{case}\ b\ \texttt{of} \\ b_1 \to b'_1 \\ \cdots \\ b_n \to b'_n \end{array} \right) \\ = \lambda\, b\ .\ \texttt{case}\ b\ \texttt{of} \quad b_1 \to b''_{j_1} \\ \qquad\qquad\qquad\qquad \cdots \\ \qquad\qquad\qquad\qquad b_n \to b''_{j_n} \end{array}$$

where $b'_{j_i} = b_i$. Composition is obtained by matching the results of the right closure with the case branching in the left. Now we see the resulting closure keeps the shape, meaning composition of such closures stays in some fixed size. This amount depends on the size of B. This closure can be suitably implemented by an array whose size depends on the B's size.

4.2 $\mathcal{H}'$-Form

Finiteness of the domain and the range settles the problems of associativity through reducing to exhaustive case branching. We then give a variant of $\mathcal{H}$-form, namely *$\mathcal{H}'$-form*, which has its parallel equivalent in homomorphism.

Definition 3 ($\mathcal{H}'$-Form) The function f' is said to have *$\mathcal{H}'$-form* $\mathcal{H}'[\![r, (p, \oplus), q']\!]$, if it is defined in the following (sequential) recursive form with an associative operator $\oplus$ and a function q' which has the finite range C.

$$\begin{array}{lcl} f'\ [\,]\ c & = & r\ c \\ f'\ (a : x)\ c & = & p\ a\ c \oplus (f'\ x\ (q'\ a\ c)) \end{array}$$

□

Lemma 3 ($\mathcal{H}'$-Form into Homomorphism) A $\mathcal{H}'$-form function $f' = \mathcal{H}'[\![r, (p, \oplus), q']\!]$ can be redefined with a homomorphism. The finite domain C is assumed to take the form of a list.

$$\begin{array}{ll} f'\ x\ c_0 & = \mathit{accept}\ (h'\ x)\ c_0 \\ h'\ [a] & = [\ \mathit{tup}\ a\ c \mid c \leftarrow C\ ] \\ h'\ (x +\!\!+ y) & = h\ x \oplus' h\ y \\ \mathit{accept}\ x\ c_0 & = [\cdot]^{-1} \\ & \quad [\ b \oplus r\ c' \mid (b, (c, c')) \leftarrow x,\ c == c_0\ ] \\ \mathit{tup}\ a\ c & = (\ p\ a\ c\ ,\ (c,\ q'\ a\ c)\ ) \\ x \oplus' y & = [\ (b_x \oplus b_y, (c_x, c'_y)) \\ & \quad \mid (b_x, (c_x, c'_x)) \leftarrow x, \\ & \quad\ (b_y, (c_y, c'_y)) \leftarrow y,\ c'_x == c_y\ ] \end{array}$$

$[\cdot]^{-1}$ takes out the element from a singleton list. □

5 Mmp in Parallel

Mmp can be formally specified as follows.

$$\begin{array}{ll} \mathit{mmp} & :: [\mathit{Int}] \to \mathit{Int} \\ \mathit{mmp}\ P\ x & = (\mathit{maximum} \circ \mathit{map}\ \mathit{sumM} \circ \mathit{filter}\ P \\ & \qquad \circ \mathit{marking})\ x \end{array}$$

Given a list of values x, we use *marking* to enumerate all the ways of marking; through filtering out lists which do not satisfy P, namely the constraints on how lists are marked, we finally choose the maximum weight among the sums of the marked elements of those marked lists.

Mmp is parameterized by the predicate P for the marking constraint. On the same list, different predicates define different problems, and calculate different maximum weights. One of them is *adj*, which describes whether the marked elements are adjacent or not.

5.1 Describing Predicates P in $\mathcal{H}'$-Form

We first see whether predicates P can be written in $\mathcal{H}'$-form. In the previous section we have obtained homomorphism from $\mathcal{H}'$-form, wrapped with

a single function. Similarly, consider defining predicates in the following form:

$$P\ x = judge\ (f\ x\ c_0)$$

After computing a $\mathcal{H}'$-form function f as defined in Definition 2 with a suitable initial value c_0 in its accumulating parameter, the function *judge* maps the result of f to a boolean value. This relaxation enables us to find a definition of the generalized k-adjacentness adj_k, which is required for k-mss to examine whether the number of marked segments in a list is at most k, as follows.

$$\begin{array}{ll}
adj_k\ x & = judge\ (adj'_k\ x\ False) \\
adj'_k\ [\,]\ m & = 0 \\
adj'_k\ (a:x)\ m & = (\texttt{if}\ \neg m \wedge isM\ a\ \texttt{then}\ 1\ \texttt{else}\ 0) \\
 & \quad +'_k adj'_k\ x\ (isM\ a) \\
judge\ v & = v \leq k \\
a +'_k b & = \texttt{if}\ a+b>k\ \ \texttt{then}\ k+1 \\
 & \qquad\qquad\ \ \texttt{else}\ a+b
\end{array}$$

The intuition is to count the beginning of segments where the current element is marked and the element before it is unmarked. The function *isM* returns *True* if the element is marked, *False* otherwise. Summation $+'_k$ treats integers more than k are all equal to $k+1$, since they are judged *False* eventually. This abstraction later works to keep the table size in Mmp independent from the length of given lists.

5.2 Mmp in Homomorphism

Now that predicates are known to be transformed into homomorphism, we tackle the main problem to reduce Mmp on list structures in some homomorphisms. The same idea in Lemma 3 also applies here, namely to specify the possible cases exhaustively to make a table, which gives us the following theorem.

Theorem 1 (Mmp in Almost Homomorphism) Assume the constraint is $P\ x = judge\ (f\ x\ c_0)$, where $f = \mathcal{H}'[\![r, (\oplus, p), q']\!]$ and C is the list which holds all elements in the domain of the f's accumulation.

$$\begin{array}{lcl}
mmp\ P\ x & = & accept\ (mmpp\ x)\ c_0 \\
mmpp\ [a] & = & \mathcal{G}\ [\ tup_{mmpp}\ m\ a\ c \mid c \leftarrow C, \\
 & & \quad m \leftarrow [True, False]\ ] \\
mmpp\ (x +\!\!+ y) & = & \mathcal{G}\ (\ mmpp\ x \oplus'\ mmpp\ y)
\end{array}$$

Given a list of pairs $[(x_1, y_1), \ldots, (x_n, y_n)]$, the function $\mathcal{G}$ is to group the pairs of the same first component x in a pair (x, y) such that y is the maximum of the second components. For instance,

$$\mathcal{G}\ [(2,1),(1,5),(2,10),(2,2),(1,1)] = [(1,5),(2,10)]\ .$$

The definitions of other auxiliary functions are as follows.

$$\begin{array}{l}
accept\ x\ c_0 \\
= maximum\ [\ w \mid ((b,(c,c')),w) \leftarrow x\ , \\
\qquad\qquad c == c_0\ ,\ judge\ (b \oplus r\ c')\ ] \\
tup_{mmpp}\ m\ a\ c \\
=\quad (\quad (p\ (m,a)\ c\ , (c, q\ (m,a)\ c)), \\
\qquad \texttt{if}\ m\ \texttt{then}\ a\ \texttt{else}\ 0\ ) \\
x \oplus' y \\
=\quad [\ ((b_x \oplus b_y, (c_x, c'_y)), w_x + w_y) \\
\qquad\qquad \mid ((b_x,(c_x,c'_x)),w_x) \leftarrow x, \\
\qquad\qquad\ ((b_y,(c_y,c'_y)),w_y) \leftarrow y, \\
\qquad\qquad\ c'_x == c_y\ ]
\end{array}$$

□

Finiteness of the range of $+'_k$ and the accumulation guarantees the number of pairs $(b,(c,c'))$ in the resulting list elements is within some constant. The associative operation $\oplus'$ can produce the same pair $(b,(c,c'))$ to have different weights w; $\mathcal{G}$ finds out their maximum which is of our interest and reduces the number again within the constant. Due to this constant each of $\oplus'$ and $\mathcal{G}$ is executed in some constant time for any cases, and the obtained function can be executed efficiently in parallel.

6 Conclusion

In this paper we demonstrated how accumulation is incorporated in the framework of homomorphisms. We have seen that finiteness of accumulations' domain played an interesting role for parallelization. Our formalization derived a parallelizable form of Mmp, k-mss as its notable example.

References

[1] 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.

[2] R. Bird. Maximum marking problems. *Journal of Functional Programming*, 11(4):411–424, 2001.

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

[4] S. Gorlatch. Constructing list homomorphisms. Technical Report MIP-9512, Fakultät für Mathematik und Informatik, Universität Passau, August 1995.

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

[6] 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.

[7] S. Peyton Jones and J. Hughes, editors. *Haskell 98: A Non-strict, Purely Functional Language*. Available online: `http://www.haskell.org`, February 1999.

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

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