Formal Derivation of Parallel Program for 2-Dimensional Maximum Segment Sum Problem

Zhenjiang Hu[1], Hideya Iwasaki[2], Masato Takeichi[1]

[1] Department of Information Engineering, University of Tokyo
7–3–1 Hongo, Bunkyo-ku, Tokyo 113, Japan
Email: hu@ipl.t.u-tokyo.ac.jp and takeichi@u-tokyo.ac.jp
[2] Department of Computer Science, Faculty of Technology
Tokyo University of Agriculture and Technology
2–24–16 Naka-cho, Koganei, Tokyo 184, Japan
Email: hiwasaki@cc.tuat.ac.jp

Abstract. It has been attracting much attention to make use of list homomorphisms in parallel programming because they ideally suit the divide-and-conquer parallel paradigm. However, they have been usually treated rather informally and ad-hoc in the development of efficient parallel programs. This paper reports a case study on systematic and formal development of a new parallel program for the 2-dimensional maximum segment problem. We show how a straightforward, and "obviously" correct, but quite inefficient solution to the problem can be successfully turned into a semantically equivalent "almost list homomorphism" based on two transformations, namely tupling and fusion, which are defined according to the specific recursive structures of list homomorphisms.

1 Introduction

It has been attracting wide attention to make use of list homomorphisms in parallel programming. *List homomorphisms* are those functions on finite lists that *promote* through list concatenation — that is, function h for which there exists an associative binary operator $\oplus$ such that, for all finite lists xs and ys, we have $h\,(xs +\!\!+ ys) = h\,xs \oplus h\,ys$, where $+\!\!+$ denotes list concatenation. 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 $h\,xs$ and $h\,ys$ are independent 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 of parallel programming.

Therefore, the implications for parallel program development become clear; *if the problem is a list homomorphism*, then it only remains to define a cheap $\oplus$ in order to produce a highly parallel solution. However, there are a lot of useful and interesting list functions that are not list homomorphisms and thus have no corresponding $\oplus$. One example is the function *mss* known as (*1-dimensional*) *maximum segment sum problem*, which finds the maximum of the sums of contiguous segments within a list of integers. For example, $mss\ [3, -4, 2, -1, 6, -3] = 7$,

where the result is contributed by the segment $[2, -1, 6]$. The *mss* is not a list homomorphism, since knowing *mss xs* and *mss ys* is not enough to allow computation of *mss* $(xs +\!\!+ ys)$.

This paper reports a case study of a formal and systematic derivation of a new efficient and correct $O(\log^2 n)$ (n denotes the size of a matrix) parallel program for the *2-dimensional maximum segment sum problem* via construction of (almost) list homomorphisms based on the idea in [HIT96]. This problem is of interest because there are efficient but non-obvious algorithms to compute it in parallel. In [Smi87], the tuple consisting of eleven functions is used for the definition of a $O(\log^2 n)$ parallel algorithm, but the detailed derivation, which would be rather cumbersome with Smith's approach, was not given at all.

This paper is organized as follows. In Sect. 2, we review the notational conventions and some basic concepts used in this paper. After giving a specification for the 2-dimensional maximum segment sum problem in Sect. 3, we focus ourselves on deriving an efficient (almost) list homomorphism from the specification by using our two important theorems, namely the Tupling and the Almost Fusion Theorems, in Sect. 4. Concluding remarks are given in Sect. 5.

2 Preliminary

In this section, we briefly review the notational conventions known as Bird-Meertens Formalisms [Bir87] and some basic concepts which will be used in the rest of this paper.

Functions

Functional application is denoted by a space and the argument which may be written without brackets. Thus $f\,a$ means $f\,(a)$. Functions are curried and application associates to the left. Thus $f\,a\,b$ means $(f\,a)\,b$. Functional application is regarded as more binding than any other operator, so $f\,a \oplus b$ means $(f\,a) \oplus b$ but not $f\,(a \oplus b)$. Functional composition is denoted by a centralized circle $\circ$. By definition, $(f \circ g)\,a = f\,(g\,a)$. Functional composition is an associative operator, and the identity function is denoted by *id*. Infix binary operators will often be denoted by $\oplus, \otimes$ and can be *sectioned*; an infix binary operators like $\oplus$ can be turned into unary functions by: $(a\oplus)\,b = a \oplus b = (\oplus\, b)\,a$.

The following are some important operators (functions) used in the paper.

- The *projection* function π_i will be used to select the i-th component of tuples, e.g., $\pi_1\,(a, b) = a$. The $\triangle$ and $\times$ are two important operators related to tuples, defined by

$$(f \triangle g)\,a = (f\,a,\, g\,a), \quad (f \times g)\,(a, b) = (f\,a,\, g\,b).$$

  The $\triangle$ can be naturally extended to functions with two arguments. So, we have $a\,(\oplus \triangle \otimes)\,b = (a \oplus b,\, a \otimes b)$.

– The *cross* operator $\mathcal{X}_\oplus$, which crosswisely combines elements in two lists with operator $\oplus$, is defined informally by:

$$[x_1, \cdots, x_n]\ \mathcal{X}_\oplus\ [y_1, \cdots, y_m] = [x_1 \oplus y_1, \cdots, x_1 \oplus y_m,\ \cdots,\ x_n \oplus y_1, \cdots, x_n \oplus y_m].$$

The cross operator enjoys many algebraic identities, e.g., $(f*) \circ \mathcal{X}_\oplus = \mathcal{X}_{f \circ \oplus}$.

– The *concat*, a function to flatten a list, is defined by:

$$concat\ [xs_1, \cdots, xs_n] = xs_1 +\!\!+ \cdots +\!\!+ xs_n.$$

– The *zip-with* operator $\curlyvee_\oplus$, a function to apply $\oplus$ pairwisely to two lists, is informally defined by

$$[x_1, \cdots, x_n]\ \curlyvee_\oplus\ [y_1, \cdots, y_n] = [x_1 \oplus y_1, \cdots, x_n \oplus y_n].$$

Lists

Lists are finite sequences of values of the same type. A list is either empty, a singleton, or the concatenation of two other lists. We write $[\,]$ for the empty list, $[a]$ for the singleton list with element a (and $[\cdot]$ for the function taking a to $[a]$), and $xs +\!\!+ ys$ for the concatenation of xs and ys. Concatenation is associative, and $[\,]$ is its unit. For example, the term $[1] +\!\!+ [2] +\!\!+ [3]$ denotes a list with three elements, often abbreviated to $[1, 2, 3]$.

List Homomorphisms

A function h satisfying the following three equations will be called a *list homomorphism*.

$$\begin{aligned} h\ [\,] &= \iota_\oplus \\ h\ [x] &= f\ x \\ h\ (xs +\!\!+ ys) &= h\ xs \oplus h\ ys \end{aligned}$$

It soon follows from this definition that $\oplus$ must be an associative binary operator with unit $\iota_\oplus$. For notational convenience, we write $([f, \oplus])$ for the unique function h[3], e.g., $sum = ([id, +])$ and $max = ([id, \uparrow])$, where $\uparrow$ denotes the binary maximum function whose unit is $-\infty$. Note when it is clear from the context, we usually abbreviate "list homomorphisms" to "homomorphism."

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

$$f * [x_1, x_2, \cdots, x_n] = [f\ x_1, f\ x_2, \cdots, f\ x_n].$$

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

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

[3] Strictly speaking, we should write $([\iota_\oplus, f, \oplus])$ to denote the unique function h. We can omit the $\iota_\oplus$ because it is the unit of $\oplus$.

It is not difficult to see that $*$ and / have simple massively parallel implementations on many architectures. For example, $\oplus/$ can be computed in parallel on a tree-like structure with the combining operator $\oplus$ applied in the nodes, whereas $f*$ is totally parallel. The relevance of list homomorphisms to parallel programming can be seen clearly from the Homomorphism Lemma [Bir87]: $(\![f, \oplus]\!) = (\oplus/) \circ (f*)$. Every list homomorphism can be written as the composition of a reduction and a map.

As stated in the introduction, quite a lot of interesting functions are not list homomorphisms. Fortunately, Cole [Col93a] argued informally that some of them can be converted into so-called *almost* (*list*) *homomorphisms* by tupling with some extra functions. To make this conversion be more formal and systematic, we proposed the idea of construction of such almost homomorphisms via tupling and fusion transformations [HIT96]. As a matter of fact, an almost list homomorphism is a composition of a projection function and a list homomorphism. Since projection functions are simple, almost homomorphisms are also suitable for parallel implementation as list homomorphisms do.

3 Specification

Before giving a specification for the 2-dimensional maximum segment sum problem, let's start with the simpler 1-dimensional maximum segment sum problem *mss* (refer [HIT96, Col93b, CS92] for more detailed discussions), an example given in the introduction. An obviously correct solution to the problem is $mss : [Int] \rightarrow Int$ defined by:

$$mss = max \circ (sum*) \circ segs$$

which is implemented by three passes; (1) computing all contiguous segments of a sequence by *segs*, (2) summing up each contiguous segment by *sum*, (3) selecting the largest value by *max*.

The only unknown function in the specification is $segs : [Int] \rightarrow [[Int]]$, computing all segments of a list. It would be likely to define it simply as

$$segs\ (xs +\!\!+ ys) = segs\ xs +\!\!+ segs\ ys +\!\!+ (tails\ xs\ \mathcal{X}_{+\!\!+}\ inits\ ys).$$

The equation reads that all segments in the sequence $xs +\!\!+ ys$ are made up of three parts: all segments in xs, all segments in ys, and all segments produced by crosswisely concatenating every *tail segment* of xs (i.e., the segment in xs ending with xs's last element) with every *initial segment* of ys (i.e., the segment in ys starting with ys's first element). Here, *inits* and *tails* are standard functions in [Bir87], though our definitions are slightly different as will be seen later. Being simple, it is a *wrong definition* for *segs*, as you may have noticed that, for example, $segs\ ([1,2] +\!\!+ [3]) \neq segs\ ([1] +\!\!+ [2,3])$ while they are expected to be equal (to $segs\ [1,2,3]$). A closer look reveals that the two resulting lists indeed consist of all segments of $[1,2,3]$ but in different order. One way to remedy this

$$
\begin{array}{l}
mss \ : \ [(Index, Int)] \rightarrow Int \\
mss \ = \ max \circ (sum'*) \circ segs \\
\\
\text{where} \\
\\
\begin{array}{ll}
max & = ([id, \uparrow]) \\
sum' & = ([id, \lambda((i,x),(j,y)).x + y]) \\
segs \ [\,] & = [\,] \\
segs \ [x] & = [[x]] \\
segs \ (xs +\!\!+ ys) & = segs \ xs +\!\!+_{\prec} segs \ ys +\!\!+_{\prec} (tails \ xs \ \mathcal{X}_{+\!\!+} \ inits \ ys) \\
inits \ [\,] & = [\,] \\
inits \ [x] & = [[x]] \\
inits \ (xs +\!\!+ ys) & = inits \ xs +\!\!+ (xs +\!\!+) * (inits \ ys) \\
tails \ [\,] & = [\,] \\
tails \ [x] & = [[x]] \\
tails \ (xs +\!\!+ ys) & = (+\!\!+ ys) * (tails \ xs) +\!\!+ tails \ ys
\end{array}
\end{array}
$$

Fig. 1. Specification for *mss* Problem

situation is to force *segs* to give result of a sorted list under a total order, say $\prec$, and thus we can define *segs* correctly as

$$segs \ (xs +\!\!+ ys) = segs \ xs +\!\!+_{\prec} segs \ ys +\!\!+_{\prec} (tails \ xs \ \mathcal{X}_{+\!\!+} \ inits \ ys)$$

where $+\!\!+_{\prec}$ merges two sorted lists into one with respect to the order of $\prec$.

Let's see how we can define such $\prec$ in a simple way. Let $[x_{i_1}, x_{i_1+1}, \cdots, x_{j_1}]$ and $[x_{i_2}, x_{i_2+1}, \cdots, x_{j_2}]$ be two segments of the presumed list $[x_1, \cdots, x_n]$. Then, $\prec$ is defined by $[x_{i_1}, x_{i_1+1}, \cdots, x_{j_1}] \prec [x_{i_2}, x_{i_2+1}, \cdots, x_{j_2}] =_{def} (i_1, \cdots, j_1) <_D (i_2, \cdots, j_2)$, where $<_D$ stands for the lexicographic order. To capture the index information in our specification, we extend the input type of *mss* and *segs* from lists of integers, $[Int]$, to lists of pairs of index and integer, $[(Index, Int)]$.

So much for the specification of the *mss* problem, which is summarized in Fig. 1. It is a naive solution of the problem without concerning efficiency and parallelism at all, but its correctness is obvious.

Let's turn to the specification for the 2-dimensional maximum segment sum problem, *mss2*, a generalization of *mss*, which finds the maximum over the sum of all rectangular subregions of a matrix. The matrix can be naturally represented by a list of lists with the same length as shown in Fig. 2 (a), and so does its rectangular subregion as in Fig. 2 (b). Following the same thought we did for *mss*, we define *mss2* straightforwardly as in Fig. 3. Here, *segs2* computes all rectangular subregions of a matrix, then *sum2* is applied to every rectangular subregion and sums up all elements, and finally *max* returns the largest.

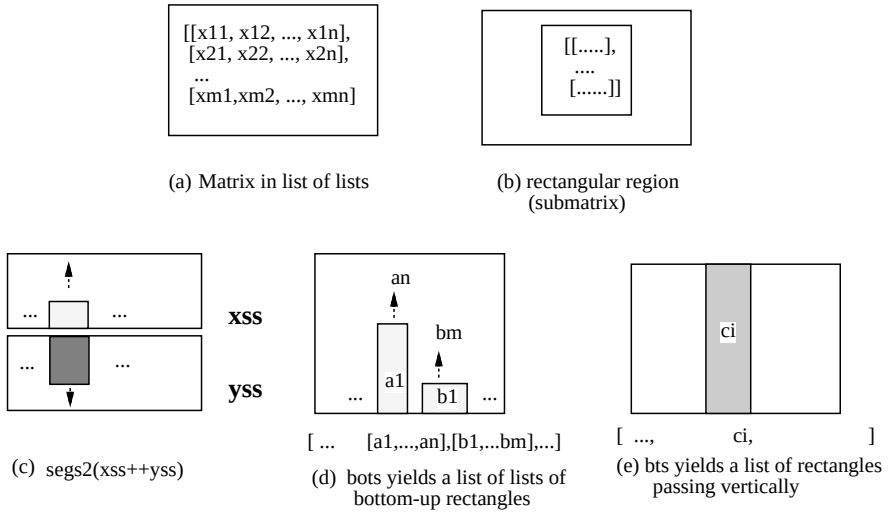

Fig. 2. The *mss2* Problem

$$
\begin{array}{l}
mss2 \;:\; [[(Index', Int)]] \rightarrow Int \\
mss2 \;=\; max \circ (sum2*) \circ segs2
\end{array}
$$

where

$$
\begin{array}{ll}
sum2 & = sum \circ sum'* \\
segs2\ [\,] & = [\,] \\
segs2\ [xs] & = [\cdot] * (segs'\ xs) \\
segs2\ (xss +\!\!+ yss) & = segs2\ xss +\!\!+_{\prec'} segs2\ yss +\!\!+_{\prec'} \\
 & \quad concat\ ((bots\ xss) \curlyvee_{\#} (tops\ yss)) \\
bots\ [\,] & = [\,] \\
bots\ [xs] & = [\cdot] * ([\cdot] * (segs'\ xs)) \\
bots\ (xss +\!\!+ yss) & = ((bots\ xss)\ \curlyvee_{\lambda(x,y).(\#\,y)*x}\ (bts\ yss))\ \curlyvee_{\#}\ (bots\ yss) \\
tops\ [\,] & = [\,] \\
tops\ [xs] & = [\cdot] * ([\cdot] * (segs'\ xs)) \\
tops\ (xss +\!\!+ yss) & = (tops\ xss)\ \curlyvee_{\#}\ ((bts\ xss)\ \curlyvee_{\lambda(x,y).((x\,\#\,)*y)}\ (tops\ yss)) \\
bts\ [\,] & = [\,] \\
bts\ [xs] & = [\cdot] * (segs'\ xs) \\
bts\ (xss +\!\!+ yss) & = (bts\ xss)\ \curlyvee_{\#}\ (bts\ yss) \\
segs'\ [\,] & = [\,] \\
segs'\ [x] & = [[x]] \\
segs'\ (xs +\!\!+ ys) & = segs'\ xs +\!\!+_{\prec'} segs'\ ys +\!\!+_{\prec'} (tails\ xs\ \mathcal{X}_{\#}\ inits\ ys)
\end{array}
$$

Fig. 3. Specification for *mss2* Problem

Function *segs2* is defined in a quite similar way to *segs*. The last equation reads that all rectangular subregions of $xss +\!\!+ yss$, a matrix connecting *xss* and *yss* vertically (Fig. 2 (c)), are made up from those in both *xss* and *yss*, and those produced by combining every *bottom-up rectangular subregion* in *xss* (depicted by shallow-grey rectangle) with every *top-down rectangular subregion* in *yss* (depicted by dark-grey rectangle) sharing the same edge.

Let's see the definition of the total order $\prec'$ among rectangular subregions. Note that the index type $Index'$ in this case should be a pair denoting the row and column of elements. So we define $\prec'$ by $[[((r_1, c_1), x_1), \cdots], \cdots, [\cdots, ((r_2, c_2), x_2)]] \prec' [[((r_1', c_1'), y_1), \cdots], \cdots, [\cdots, ((r_2', c_2'), y_2)]] =_{def} ((r_1, c_1), (r_2, c_2)) <_D ((r_1', c_1'), (r_2', c_2'))$.

For other functions in Fig. 3, *bots* is used to calculate a list of lists each of which comprises all rectangles with the same bottom edge. Symmetrically, *tops* calculates a list of lists each of which comprises all rectangles with the same top edge. They are defined by using another function *bts* which yields a list of rectangles passing through the matrix vertically (Fig. 2 (e)). The function $segs'$ is almost the same as *segs* except that it is defined under the order of $\prec'$ rather than $\prec$. Although segs and segs' could be unified into a single function by means of overloading, they are defined independently for simplicity.

4 Derivation

Our derivation of an almost homomorphism from the specification in Fig. 3 is carried out in the following procedure.

1. Derive an almost homomorphism from the recursive definition of *segs2* (Sect. 4.1);
2. Fuse $(sum2*)$ with the derived almost homomorphism to obtain another almost homomorphism and repeat this fusion for max (Sect. 4.2);
3. Let $\pi_1 \circ ([f, \oplus])$ be the result obtained in (2). If f or $\oplus$ are much complicated, repeat (1) and (2) to find an efficient parallel implementation for f and $\oplus$ (Sect. 4.3).

4.1 Deriving almost homomorphisms

Our approach is based on the following theorem. For notational convenience, we define $\Delta_1^n f_i = f_1 \mathbin{\triangle} f_2 \mathbin{\triangle} \cdots \mathbin{\triangle} f_n$.

Theorem 1 (Tupling [HIT96]). Let $h_1, \cdots, h_n$ be mutually defined by:

$$\begin{aligned} h_i\ [\,] &= \iota_{\oplus_i} \\ h_i\ [x] &= f_i\ x \\ h_i\ (xs +\!\!+ ys) &= ((\Delta_1^n h_i)\ xs) \oplus_i ((\Delta_1^n h_i)\ ys) \end{aligned} \tag{1}$$

Then $\Delta_1^n h_i = ([\Delta_1^n f_i,\ \Delta_1^n \oplus_i])$, and $(\iota_{\oplus_1}, \cdots, \iota_{\oplus_n})$ is the unit of $\Delta_1^n \oplus_i$. □

Theorem 1 says that if h_1 is mutually defined with other functions (i.e., $h_2, \cdots h_n$) *traversing over the same lists* in the *specific form* of (1), then tupling $h_1, \cdots, h_n$ will give a list homomorphism. Let's see how the tupling theorem is used in deriving an almost homomorphism from the definition of *segs2* given in Sect. 3.

First, we determine what functions are to be tupled, i.e., $h_1, \cdots, h_n$. As the tupling theorem suggests, the functions to be tupled are those traversing over the same lists in the mutual definitions. So, from the definition of *segs2*:

$$segs2\ (xss +\!\!+ yss) = \underline{segs2\ xss} +\!\!+_{\prec'} \underline{segs2\ yss} +\!\!+_{\prec'} concat((\underline{bots\ xss})\Upsilon_{\mathcal{X}_{+\!\!+}}\ (\underline{tops\ yss}))$$

we know that *segs2* should be tupled with *bots* and *tops*, because *segs2* and *bots* traverse over the same list *xss* whereas *segs2* and *tops* traverse over the same list *yss* as underlined. Similarly, the definitions of *bots* and *tops* requires that *bts* be tupled with *bots* and *tops*. In summary, the functions to be tupled are *segs2*, *bots*, *tops* and *bts*, i.e., our tuple function will be:

$$segs2 \vartriangle bots \vartriangle tops \vartriangle bts.$$

Next, we rewrite the definition of each function in the above tuple to be in the form of (1), i.e., deriving $f_1, \oplus_1$ for *segs2*, $f_2, \oplus_2$ for *bots*, $f_3, \oplus_3$ for *tops*, and $f_4, \oplus_4$ for *bts*. This is straightforward. The results are as follows. For example, from the definition of *segs2*, we can easily derive that

$$\begin{array}{l}
f_1\ xs = [\cdot] * (segs'\ xs) \\
(s_1, b_1, t_1, d_1) \oplus_1 (s_2, b_2, t_2, d_2) = s_1 +\!\!+_{\prec'} s_2 +\!\!+_{\prec'} concat\ (b_1 \Upsilon_{\mathcal{X}_{+\!\!+}}\ t_2) \\
f_2\ xs = [\cdot] * ([\cdot] * (segs'\ xs)) \\
(s_1, b_1, t_1, d_1) \oplus_2 (s_2, b_2, t_2, d_2) = (b_1\ \Upsilon_{\lambda(x,y).(+\!\!+ y)*x}\ d_2)\ \Upsilon_{+\!\!+}\ b_2 \\
f_3\ xs = [\cdot] * ([\cdot] * (segs'\ xs)) \\
(s_1, b_1, t_1, d_1) \oplus_3 (s_2, b_2, t_2, d_2) = t_1\ \Upsilon_{+\!\!+}\ (d_1\ \Upsilon_{\lambda(x,y).((x +\!\!+)*y)}\ t_2) \\
f_4\ xs = [\cdot] * (segs'\ xs) \\
(s_1, b_1, t_1, d_1) \oplus_4 (s_2, b_2, t_2, d_2) = d_1\ \Upsilon_{+\!\!+}\ d_2
\end{array}$$

Finally, we apply Theorem 1 and get the following list homomorphism.

$$segs2 \vartriangle bots \vartriangle tops \vartriangle bts = ([\Delta_1^4 f_i, \Delta_1^4 \oplus_i])$$

And our almost homomorphism for *segs2* is thus obtained:

$$segs2 = \pi_1 \circ ([\Delta_1^4 f_i, \Delta_1^4 \oplus_i]). \tag{2}$$

4.2 Fusion with Almost Homomorphisms

In this section, we show how to fuse a function with an almost homomorphism. Our fusion theorem for this purpose is given below.

Theorem 2 (Almost Fusion [HIT96]). Let $([\Delta_1^n f_i,\ \Delta_1^n \oplus_i])$ and h be given. If there exist $\otimes_i$ $(i = 1, \cdots, n)$ and a map $\Lambda h = h_1 \times \cdots \times h_n$ where $h_1 = h$ such that for all i, $\forall x, y.\ h_i\ (x \oplus_i y) = (\Lambda h)\ x \otimes_i (\Lambda h)\ y$, then $h \circ (\pi_1 \circ ([\Delta_1^n f_i, \Delta_1^n \oplus_i])) = \pi_1 \circ ([\Delta_1^n (h_i \circ f_i), \Delta_1^n \otimes_i])$. □

Returning to our example, recall that we have reached the point:

$$mss2 = max \circ (sum2*) \circ (\pi_1 \circ ([\Delta_1^4 f_i, \Delta_1^4 \oplus_i])).$$

We can fuse $sum2*$ with $\pi_1 \circ ([\Delta_1^4 f_i, \Delta_1^4 \oplus_i])$ by Theorem 2, and then repeat this fusion for max, giving the following result.

$$mss2 = \pi_1 \circ ([\Delta_1^4 f_i', \Delta_1^4 \oplus_i']) \tag{3}$$

where

$$\begin{aligned}
(s_1, b_1, t_1, d_1) \oplus_1' (s_2, b_2, t_2, d_2) &= s1 \uparrow s_2 \uparrow (\uparrow / (b_1 \ \Upsilon_{\mathcal{X}_+} \ t_2)) \\
(s_1, b_1, t_1, d_1) \oplus_2' (s_2, b_2, t_2, d_2) &= (b_1 \ \Upsilon_+ \ d_2) \ \Upsilon_\uparrow \ b_2 \\
(s_1, b_1, t_1, d_1) \oplus_3' (s_2, b_2, t_2, d_2) &= t_1 \ \Upsilon_\uparrow \ (d_1 \ \Upsilon_+ \ t_2) \\
(s_1, b_1, t_1, d_1) \oplus_4' (s_2, b_2, t_2, d_2) &= d_1 \ \Upsilon_+ \ d_2
\end{aligned}$$

and

$$\begin{aligned}
f_1' &= max \circ (sum'*) \circ segs' \\
f_2' &= (sum'*) \circ segs' \\
f_3' &= (sum'*) \circ segs' \\
f_4' &= (sum'*) \circ segs'
\end{aligned}$$

4.3 Improving Operators in List Homomorphisms

Equation (3) has given a homomorphic solution to the 2-dimensional maximum segment sum problem. Let n be the size of the input matrix. By a simple divide-and-conquer implementation of list homomorphisms, the derived program can expect an $max(O(\Delta_1^4 f_i'),\ (O(\log n) * O(\Delta_1^4 \oplus_i')))$ parallel algorithm. With assumptions that $\Upsilon_\otimes$ and $\mathcal{X}_\otimes$ can be implemented fully in parallel, i.e., $O(\Upsilon_\otimes) = O(\otimes)$ and $O(\mathcal{X}_\otimes) = O(\otimes)$, we can see that $O(\Delta_1^4 \oplus_i') = O(\log\ n)$ due to the inherited parallelism in the reduction $(\uparrow /)$. It follows that $mss2$ is a

$$max(O(\Delta_1^4 f_i'),\ O(\log^2 n))$$

parallel algorithm. It is, however, not so obvious about efficient parallel implementation of, e.g., f_1' (similar to 1-dimensional mss problem except for different index order). We can derive (almost) list homomorphisms for it using the above derivation strategy again, giving a $O(\log n)$ parallel algorithm. (see [HIT96] for a detailed derivation for mss).

5 Concluding Remarks

In this paper, we demonstrate our derivation of an efficient parallel algorithm for the 2-dimensional maximum segment sum problem. It is based on the *manipulation* of (almost) homomorphisms, namely the construction of almost homomorphisms from recursive definitions (Theorem 1) and the fusion of a function with almost homomorphisms (Theorem 2). After the initial solution, all the derivation are proceeded in a formal setting based on our theorems and algebraic identities of list functions. Therefore, the resulting parallel algorithm is guaranteed to be

semantically equivalent to the initial naive but inefficient solution. This is in sharp contract to Cole's informal approach [Col93b, Col93a].

Smith [Smi87] applied a strategy of divide-and-conquer approach to the same problem as an application. He constructs the composing operator (analog to our associative operator $\oplus$) by employing the suitable mathematical properties of the problem. Although our initial specification is less abstract than his, our derivation is more systematic and less prone to errors (Sect. 4) In [Smi87], the tuple consisting of eleven functions is given for the 2-dimensional *mss* problem, and the corresponding manipulation with Smith's approach must be cumbersome (Smith does not present them).

References

[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.

[Col93a] M. Cole. List homomorphic parallel algorithms for bracket matching. Technical report CSR-29-93, Department of Computing Science, The University of Edinburgh, August 1993.

[Col93b] M. Cole. Parallel programming, list homomorphisms and the maximum segment sum problems. Report CSR-25-93, Department of Computing Science, The University of Edinburgh, May 1993.

[CS92] W. Cai and D.B. Skillicorn. Calculating recurrences using the Bird-Meertens Formalism. Technical report, Department of Computing and Information Science, Queen's University, 1992.

[HIT96] Z. Hu, H. Iwasaki, and M. Takeichi. Construction of list homomorphisms via tupling and fusion. In *21st International Symposium on Mathematical Foundation of Computer Science* (MFCS '96), Cracow, September 1996. Springer-Verlag Lecture Notes in Computer Science.

[Smi87] D.R. Smith. Applications of a strategy for designing divide-and-conquer algorithms. *Science of Computer Programming*, (9):213–229, 1987.

This article was processed using the LaTeX macro package with LLNCS style