A Uniform Approach toward Nested Parallelism

Kazuhiko Kakehi[1], Kiminori Matsuzaki[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 Agency

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

Parallel skeletons and homomorphisms over lists provide successful methods for parallelization. The structural regularity of lists helps simple and efficient implementation of their systems, which casts a contrast to more irregular, complicated data structures like nested lists or trees.

This paper develops a uniform approach toward such nested structures through converting these data structures into lists of tuples with depth information. Generic parallel computation schemes like reduce and accumulation are analyzed using the parallelization techniques over lists. We demonstrate its expressiveness and efficacy using a class of optimization problems called maximum marking problems.

1 Introduction

Skeletal parallelism which consists of generic computation structures called *skeletons* or homomorphisms provide a convenient model for parallel computation [14]. Structural regularity of lists promotes and facilitates not only active researches but also several simple and efficient implementations on PC clusters. On the contrary, similar efficient parallel treatments on other data structures like sparse matrices or trees are not straightforward due to their structural irregularity.

One of the basic algorithms is to convert information in one data structure into another, traversals over trees into lists for instance; association of additional information makes translation invertible. This fact naturally raises a question: Are skeletons for such nested structures implementable using list skeletons?

This paper shows a positive answer to this question. We clarify the behavior of their skeletons as list operations uniformly, and demonstrate their expressiveness and parallel efficiency when the depth of nesting is fixed (written as d in this paper) using a class of optimization problems called maximum marking problems.

Notation We use the notation of BMF (Bird-Meertens Formalism) [1, 16] throughout this paper. We also borrow the Haskell notation [12] for readability.

2 List Parallelization Techniques

Four higher order functions are often called *list skeletons*, that is map, zipw, reduce and scan. Map, denoted as an infix $*$, is to apply a function to every element in a list.

$$k * [a_1, a_2, \ldots, a_n] = [k\, a_1, k\, a_2, \ldots, k\, a_n]$$

Zipw is the skeleton that takes two lists and returns a new list through applying a specified operation $\oplus$ to every pair of corresponding elements from the two lists of the same length; therefore

$$\begin{aligned}&[a_1, a_2, \ldots, a_n]\, \Upsilon_{\oplus}\, [b_1, b_2, \ldots, b_n]\\ &\quad = \ [a_1 \oplus b_1, a_2 \oplus b_2, \ldots, a_n \oplus b_n]\ .\end{aligned}$$

Reduce, written as an infix /, is the skeleton which collapses a list into a single value by repeated application of some associative binary operator $\otimes$.

$$\otimes\ /_c\ [a_1, a_2, \ldots, a_n] = c \otimes a_1 \otimes a_2 \otimes \cdots \otimes a_n$$

Scan accumulates all intermediate results for associative computation $\otimes$. Informally we have

$$\begin{aligned}&\otimes /\!\!/_c\ [a_1, a_2, \ldots, a_n]\\ &= \ [c, c \otimes a_1, c \otimes a_1 \otimes a_2, \ldots, c \otimes a_1 \otimes a_2 \otimes \cdots \otimes a_n]\ .\end{aligned}$$

This left-to-right scanning is called scanl while the symmetric right-to-left is called scanr.

These four skeletons have nice massively parallel implementations on many architectures [3, 7, 16]. If k, $\oplus$ and $\otimes$ need O(1), then both map $k*$ and zipw

$\mathsf{map}\ f\ (Node\ a\ t)$
$= Node\ (f\ a)\ (\ (\mathsf{map}\ f)\ *\ t\ )$

$\mathsf{zipw}\ f\ (Node\ a\ t_a)\ (Node\ b\ t_b)$
$= Node\ (f\ a\ b)\ (\ t_a\ \Upsilon_{(\mathsf{zipw}\ f)}\ t_b\ )$

$\mathsf{dAcc}\ (\otimes)\ c\ (Node\ a\ t)$
$= Node\ (c \oplus a)\ (\ (\mathsf{dAcc}\ (\otimes)\ (c \oplus a))\ *\ t\ )$

$\mathsf{reduce}\ (\oplus)\ (\otimes)\ c\ (Node\ a\ t)$
$= a \oplus (\ \otimes/_c\ (\ (\mathsf{reduce}\ (\oplus)\ (\otimes)\ c)\ *\ t\ )\ )$

$\mathsf{uAcc}\ (\oplus)\ (\otimes)\ c\ (Node\ a\ t)$
$=$ `let` $t' = (\mathsf{uAcc}\ (\oplus)\ (\otimes)\ c)\ *\ t$
$\quad a' = \mathsf{reduce}\ (\oplus)\ (\otimes)\ c\ (Node\ a\ t)$
`in` $Node\ a'\ t'$

- $\otimes$ is an associative operator

Figure 1: Skeleton definitions for the data type *RTree*

$\Upsilon_\oplus$ can be implemented using O(1) parallel time, and both reduce $\otimes\ /_c$ and scan $\otimes /\!\!/_c$ can be implemented using $O(\log n)$ parallel time over an input list of length n. For example, $\otimes\ /_c$ is implemented over a tree-like structure with the combining operator $\oplus$ applied in the nodes.

List homomorphism is a divide-and-conquer formalization over lists [6]. A list homomorphism h consists of a function f (applied for list singletons $[\cdot]$) and an associative operator $\odot$ (for list joins $+\!\!+$) as

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

It is factored into map and reduce (*First Homomorphism Theorem*).

3 Skeletons for Nested Structures

In contrast to lists, data structures in general can be and are indeed irregular, as is seen in lists of lists, sparse matrices, or trees. As a general form of such nested structures, we choose a data structure *RTree* which is known as *rose trees*.

$$\text{data}\ RTree\ \alpha = Node\ \alpha\ [RTree\ \alpha]$$

We implicitly assume maximum depth is bounded to d. Consider $[\,[1],\ [2,3,4]\,]$, a list of lists which is also in this scope by assigning values only in terminal nodes. Since two list elements have different length and naive divide-and-conquer approaches do not realize satisfactory parallelism.

There are researches on such nested structures, called nested parallelism [4]. Here we rather choose the approach taken in the previous section, and deal with higher-order function over *RTree*, called *tree skeletons* (Figure 1). We see how these skeletons are implemented uniformly and efficiently using list parallelization techniques.

3.1 List representation and data structure

First we need to represent *RTree* α by a list in order to apply list parallelization techniques. We choose preorder traversal to obtain its list representation. Since preorder traversal alone loses the original structural information, we assign information about the depth of nodes. Therefore the obtained list is of type $[(Int, \alpha)]$. Figure 2 shows an example of *RTree* (left) and how it is represented by a list (upper right). We can easily see how the original tree is obtained with the help of additional lines and node arrangement (lower right).

3.2 Implementing skeletons under the list representation

Due to the limit of space we give the illustration of algorithms instead of detailed codes. List homomorphism serves as the main tool here.

In our case, f in the homomorphism does almost nothing; $\odot$ performs computation where adequate information appeared. This $\odot$ is shown to be associative (proofs omitted). It is helpful to observe that any part of a list representation is partitioned into a *hill* (Figure 3). Each part keeps its intermediate value during computation.

The skeleton reduce computes an single value

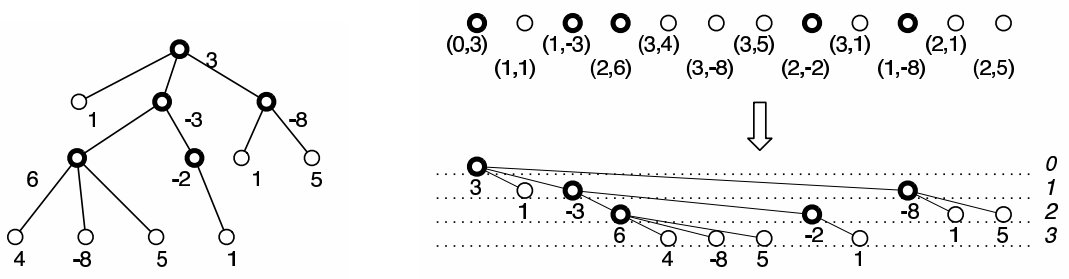

Figure 2: An example of *RTree* and its list representation in a flat and height-adjusted manner

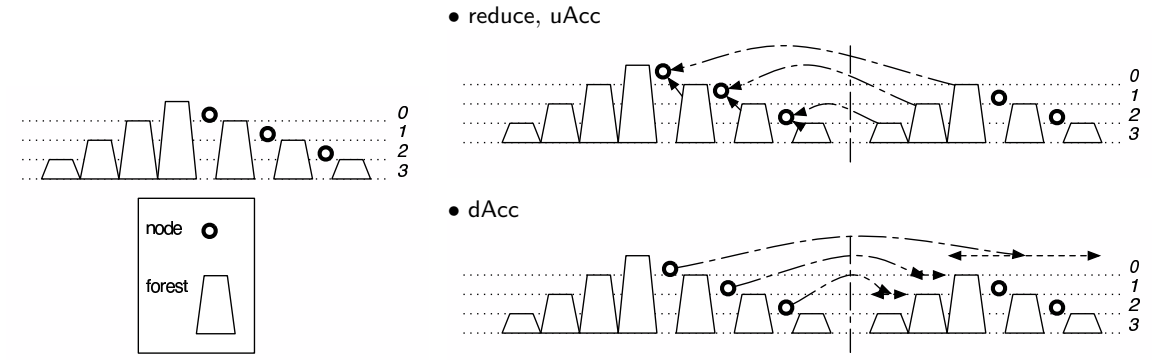

Figure 3: *Hill*-like partitioning and data passage in skeletal computation

from a tree, and uAcc recursively produces a tree whose root nodes have the value computed by reduce over the tree. Computation of a node in a tree cannot proceed until all of its subtrees appear. In terms of the list representation, joining two adjacent hills creates in between a *valley* where computation takes place. It is illustrated as the upper right in Figure 3. Even if a node does not have its whole subtrees, results of adjacent subtrees can be computed in advance using the associative operator $\otimes$. Each step passes O(d) information and requires O(d) computation time. Therefore parallel computation as a whole takes $O(\mathsf{d} \log n)$.

The skeleton dAcc recursively propagates the value in a node toward its subtrees. While reduce and uAcc pass information from right to left, the lower right part in Figure 3 indicates there is a flow of information from left to right. The values of nodes in the left list are mapped onto their corresponding segments in the right. While this information passing takes O(d) for each step, the computation is essentially a map operation in O(1) parallel time. Total computation takes $O(\mathsf{d} \log n)$.

We do not mention map and zipw since list skeletons map and zipw directly work, respectively, and they require O(1) parallel time.

4 Maximum Marking Problems over Nested Parallelism

In this section we demonstrate the expressiveness of these skeletons using an optimization problem called *maximum marking problems*, MMP for short. This is 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 is shown that a linear time algorithm based on dynamic programming exists provided that the characterizing function of the constraint is a finite

$$
\begin{array}{l}
P_{\text{k-MCS}}\ x \\
\quad = (f_{\text{MCS}}\ \mathit{False}\ x) \le \mathsf{k} \\
f_{\text{MCS}}\ c\ (\mathit{Node}\ a\ t) \\
\quad = (\mathtt{if}\ \neg c \wedge \mathit{Marked}?\ a\ \mathtt{then}\ 1\ \mathtt{else}\ 0) \\
\qquad +\ (+/_0\ ((f_{\text{MCS}}\ (\mathit{Marked}?\ a)) * t))
\end{array}
$$

$$
\begin{array}{l}
P_{\text{MIS}}\ x \\
\quad = f_{\text{MIS}}\ \mathit{False}\ x \\
f_{\text{MIS}}\ c\ (\mathit{Node}\ a\ t) \\
\quad = (\mathtt{if}\ c \wedge \mathit{Marked}?\ a\ \mathtt{then}\ \mathit{False}\ \mathtt{else}\ \mathit{True}) \\
\qquad \wedge\ (\wedge/_{\mathit{True}}\ ((f_{\text{MIS}}\ x\ (\mathit{Marked}?\ a)) * t))
\end{array}
$$

Figure 4: k-MCS and MIS over *RTree*

homomorphism [2, 15], and that it can be parallelized over lists automatically when the characterizing function is written in some format [8].

We extend the approach [8] toward *RTree*. Two problems are considered: k-MCS and MIS. Maximum connected sum k-MCS finds a marking whose resulting connected parts are at most k. Maximum independent sum MIS is to find a marking where marked nodes do not have marked parents.

4.1 Problem description under lists

First, we consider the case in lists. Naive specification of MMP in general is given as follows.

$$mmp\ P\ x = (max \circ map\ sumM \circ filter\ P \circ marking)\ x$$

This generate-and-test specification is of course general but inefficient.

We automatically have a $O(\log n)$ parallel solution provided that the constraint P is defined as

$$
\begin{array}{lcl}
P\ x & = & \mathit{judge}\ (f\ c_0\ x) \\
f\ c\ (a\ :\ x) & = & p\ a\ c \oslash f\ (q'\ a\ c)\ x \\
f\ c\ [\,] & = & r\ c\ ,
\end{array}
$$

with an associative operator $\oslash$, finite domains F of the function f and C of the accumulation c. Finiteness of the accumulation derives associativity using a table working as closures, and finiteness of the function itself limits the size of the memoization table during computation.

4.2 Specification on nested structures

The description under lists are naturally extended toward *RTree*. The predicate P is defined in the format of reduce with an accumulating parameter in addition.

$$
\begin{array}{l}
P\ x = \mathit{judge}\ (f\ c_0\ x) \\
f\ c\ (\mathit{Node}\ a\ t) \\
\qquad = p\ a\ c \oslash (\odot/_{\iota_\odot}\ ((f\ (q'\ a\ c))\ *\ t))
\end{array}
$$

Note that associativity is required for the operator $\odot$, but not for $\oslash$; $\iota_\oslash$ is a unit of $\oslash$.

The predicates for k-MCS and MIS are defined in Figure 4. Since their operators with a finite accumulation fulfill conditions for reduce, these predicates are computed in $O(\mathsf{d} \log n)$ parallel time.

4.3 Parallel Mmp on nested structures

Once the predicate are shown to be parallelizable, the parallel solution for the main MMP is almost straightforward. Figure 5 shows the general paralellizable form for solving MMP with predicates specified in Section 4.2. Initially a table is created for each node by map, and we apply reduce to merge tables into a table. The table holds the maximum weight for each index specified by a triple $(f, (c_{in}, c_{out}))$, which are value computed by f, incoming accumulation and outgoing accumulation, respectively. The function $\mathcal{G}$ is to return the pairs (x, y) where y is the maximum for each x; for instance, $\mathcal{G}\ [(2,1),(1,5),(2,10),(2,2),(1,1)] = [(1,5),(2,10)]$. The domain F of f_{MCS} in Figure 4 is not bounded; we can regard values more than k equals to $\mathsf{k}+1$ since addition is monotone.

Each table has size at most $|F| \cdot |C|^2$, and each computation by $\oplus$ or $\otimes$ may take at most $|F|^2 \cdot |C|^4$. This implies that the program runs $O(|F|^2 \cdot |C|^4 \cdot \mathsf{d} \log n)$. Since $|F|$ and $|C|$ is decided along with the predicate P, like $|F| = 4$, $|C| = 2$ for k-MCS, and $|F| = 2$, $|C| = 2$ for MIS, the factor of $O(\mathsf{d} \log n)$ becomes more essential.

$$
\begin{array}{lcl}
mmp\ P\ x & = & accept\ (\mathsf{reduce}\ \oplus\ \otimes\ \iota_{\otimes}\ (\mathsf{map}\ g\ x)) \\
\\
g\ a & = & \mathcal{G}\ [\ tab\ m\ a\ c \mid c \leftarrow C,\ m \leftarrow [True, False]\ ] \\
a \oplus x & = & \mathcal{G}\ [\ ((f_a \oslash f_x, (c_a, \_)), w_a + w_x) \mid ((f_a, (c_a, c'_a)), w_a) \leftarrow a,\ ((f_x, (c_x, c'_x)), w_x) \leftarrow x, \\
 & & \qquad\qquad c'_a\ \texttt{==}\ c_y\ ] \\
x \otimes y & = & \mathcal{G}\ [\ ((f_x \odot f_y, (c_x, \_)), w_x + w_y) \mid ((f_x, (c_x, c'_x)), w_x) \leftarrow x,\ ((f_y, (c_y, c'_y)), w_y) \leftarrow y, \\
 & & \qquad\qquad c_x\ \texttt{==}\ c_y\ ] \\
\iota_{\otimes} & = & [\ ((\iota_{\odot}, (c, \_)), 0) \mid c \leftarrow C\ ] \\
\\
accept\ x\ c_0 & = & max\ [\ w \mid ((b, (c, c')), w) \leftarrow x\ , c\ \texttt{==}\ c_0\ ,\ judge\ (b \oplus r\ c')\ ] \\
tab\ m\ a\ c & = & ((p\ (m, a)\ c\ , (c, q\ (m, a)\ c)),\ \texttt{if}\ m\ \texttt{then}\ a\ \texttt{else}\ 0\ )
\end{array}
$$

Figure 5: Parallelizable form for MMP toward nested parallelism

We conducted an experiment of MIS over a tree with 1 million nodes, and depth at most 50 using our library [11]. Good scalability is observed as 4 cpus takes 11 secs. while a single cpu takes 43 secs.

5 Conclusion

We uniformly realized skeletons for nested structures using list skeletons which run in $O(\mathsf{d} \log n)$ parallel time. Maximum marking problems demonstrated their expressiveness.

Related works would be *flattening* [5, 13] on nested parallelism or its extension to trees [9]. Our formalization is uniform enough and suits better for computation on trees.

The data structure *RTree* is nothing but rose trees if we lift the limit of depth d. Our current approach may run at $O(n \log n)$ parallel time if imbalanced trees are concerned. We already know a strong property called *extended distributivity* [10] on $\oplus$ and $\otimes$ for reduce and uAcc, and this implements parallel computation in time logarithmic to the depth. Our future work is how to extend our present approach to realize efficient parallel tree skeletons which run in the time closer to $O(\log \mathsf{d} \cdot \log n)$.

References

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

[2] R. Bird. Maximum marking problems. *J. Funct. Prog.*, 11(4): 411–424, 2001.

[3] G.E. Blelloch. Scans as primitive operations. *IEEE Trans. Comput.*, 38(11): 1526–1538, Nov. 1989.

[4] G.E. Blelloch. *Vector models for data-parallel computing.* MIT Press, 1990.

[5] G.E. Blelloch and G.W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *J. Par. Distr. Comput.* 8(2): 119–134, 1990.

[6] M. Cole. Parallel programming with list homomorphisms. *Par. Proc. Let.*, 5(2): 191–203, 1995.

[7] M. Cole. Skeletal Parallelism home page. `http://homepages.inf.ed.ac.uk/mic/Skeletons/` .

[8] K. Kakehi, et al. List homomorphism with accumulation. In *Proc. SNPD*, 250–259, Oct. 2003.

[9] G. Keller and M.M.T. Chakravarty. Flattening trees. In *Proc. Euro-Par*, 709–719, Sep. 1998.

[10] K. Matsuzaki, et al. Systematic derivation of tree contraction algorithms. In *Proc. CMPP*, Jul. 2004.

[11] K. Matsuzaki, et al. A fusion-embedded skeleton library. In *Proc. Euro-Par*, Aug.–Sep. 2004.

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

[13] D.W. Palmer, et al. Piecewise execution of nested data-parallel programs. In *Proc. LCPC*, 346–361, Aug. 1995.

[14] F.A. Rabhi and S. Gorlatch, eds. *Patterns and skeletons for parallel and distributed computing.* Springer-Verlag, 2002.

[15] I. Sasano, et al. Make it practical: A generic linear time algorithm for solving maximum weightsum problems. In *Proc. ICFP*, 137–149, Sep. 2000.

[16] D.B. Skillicorn. Architecture-independent parallel computation. *IEEE Comput.*, 23(12):38–50, Dec. 1990.