A Uniform Approach toward Nested Parallelism

Kazuhiko Kakehi\textsuperscript{1}, Kiminori Matsuzaki\textsuperscript{1}, Zhenjiang Hu\textsuperscript{1,2} and Masato Takeichi\textsuperscript{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.

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 \(\ast\), is to apply a function to every element in a list.

\[ k \ast [a_1, a_2, \ldots, a_n] = [k a_1, k a_2, \ldots, k a_n] \]

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

\[ [a_1, a_2, \ldots, a_n] \oplus [b_1, b_2, \ldots, b_n] = [a_1 \oplus b_1, a_2 \oplus b_2, \ldots, a_n \oplus b_n]. \]

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

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

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 \(k\ast\) and zipw
map \( f \) (Node \( a \ t \))
\[=\] Node \( (f \ a) (\text{map} \ f \ast t)\)

dAcc \( (\otimes) \) c (Node \( a \ t \))
\[=\] Node \( (c \oplus a) (\text{dAcc} (\otimes) (c \oplus a)) \ast t\)

reduce \( (\oplus) (\otimes) \) c (Node \( a \ t \))
\[=\] \( a \oplus (\otimes/c) (\text{reduce} (\oplus) (\otimes) c \ast t)\)

\(\text{uAcc} (\oplus) (\otimes) \) c (Node \( a \ t \))
\[=\] let \( t' = (\text{uAcc} (\oplus) (\otimes) c \ast t)\)
\[a' =\] reduce \( (\oplus) (\otimes) \) c (Node \( a \ t\))
in Node \( a' t'\)

\(\otimes\) is an associative operator

\[\prod\] can be implemented using \(O(1)\) parallel time, and both reduce \(\otimes/\) and scan \(\otimes\#\) can be implemented using \(O(\log n)\) parallel time over an input list of length \(n\). For example, \(\otimes/\) 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 [1]) and an associative operator \(\otimes\) (for list joins \(\oplus\)) as

\[h \ [a] = f \ a\]
\[h \ (x \oplus y) = (h \ x) \otimes (h \ y)\, .\]

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.

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

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 \(RTree \ \alpha\) 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 \([\text{Int}, \ \alpha]\). 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; \(\otimes\) performs computation where adequate information appeared. This \(\otimes\) 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
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 $O(d \log n)$.

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
dominated by the constant term. Thus, the entire expression is dominated by the constant term.


g_{mcs} = \begin{cases} 0 & \text{if } a \text{ is not marked} \\ 1 & \text{if } a \text{ is marked} \end{cases} \in \mathbb{N}

This predicate is used to determine whether a node is marked. If a node is marked, it contributes 1 to the result, otherwise 0.


g_{reduce} = \begin{cases} f_{c} & \text{if } a \text{ is marked} \\ \emptyset & \text{otherwise} \end{cases} \in \mathbb{N}

This predicate combines the marked nodes with the function \(f\) applied to the marked nodes.


g_{filter} = \begin{cases} \emptyset & \text{if } a \text{ is not marked} \\ \{a\} & \text{if } a \text{ is marked} \end{cases} \in \mathbb{N}

This predicate filters out the marked nodes from the table.


g_{map} = \begin{cases} \emptyset & \text{if } a \text{ is not marked} \\ \{f_{c}(a)\} & \text{if } a \text{ is marked} \end{cases} \in \mathbb{N}

This predicate maps the marked nodes with the function \(f\).

These predicates are used to manipulate the RTree structure efficiently. The \(g_{mcs}\) predicate checks if a node is marked, \(g_{reduce}\) combines the marked nodes with \(f\), \(g_{filter}\) removes unmarked nodes, and \(g_{map}\) maps marked nodes. These operations are designed to be parallelizable, making the solution scalable for large datasets.
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 $O(d \log n)$ 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 $\circ$ and $\odot$ 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 $O(\log d \cdot \log n)$.

References