Background Paper Report: Theoretical Background on Balanced Trees and Parallelism

Samuel Teporten Zhili Pan

November 7, 2024

Contents

1 Introduction

To better utilize modern processors with multiple cores or CPUs, developers write parallel programs with techniques like divide-and-conquer. However, traditional algebraic data structures in standard functional programs do not support efficient parallel programming. For example, when calculating the sum of list elements, one usually has:

$$\begin{aligned} \mathrm{sum}_{\mathrm{List}}[] &= 0 \\ \mathrm{sum}_{\mathrm{List}}(a : x) &= a + \mathrm{sum}_{\mathrm{List}} x \end{aligned}$$

No expressions in the above code can be evaluated in parallel. Similar problems exist in list-like tree data structures. There is no efficient way to divide usual data structures and programming idioms because they are inherently sequential.
To achieve efficient parallel computation, Morihata and Matsuzaki (2011) propose to use balanced trees as the abstract data structure and develop recursive functions on the encapsulated divide-and-conquer parallelism (p.117). In this paper, the authors first systematically relate parallel algorithms to balanced trees. They then perform systematic development of parallel programs that uniformly work on both lists and binary trees.

2 Preliminaries

This section outlines the core data structures and notation essential for understanding the parallel processing strategies discussed in the paper.

Greek Letters:

- α, β, γ, etc., are used as *type variables*. They represent arbitrary types in type signatures, enabling polymorphism in functions and data structures.

English Letters:

- Capital letters like A, B, C, etc., are *meta-variables* ranging over types. They are placeholders for specific types in theoretical discussions.

Identity Function:

$$\texttt{id} :: \forall \alpha.\ \alpha \to \alpha$$

This denotes the identity function, which takes a value of any type α and returns the same value without modification.

Haskell Constructs:

- `Single` and `Join` are data constructors in the definition of `JList`.
  - `Single` represents a join list containing a single element.
  - `Join` combines two `JList` structures into one, facilitating the creation of binary tree structures for parallel processing.
- $\otimes, \oplus$: 2-nary functions with associativity property.
- $|$: Used in Haskell to separate different constructors in a data type definition. It indicates that a type can have multiple forms.

3 Balance Trees for Parallel Operations

3.1 Parallel List Operations

3.1.1 Aggregation Computation Structure

As Morihata and Matsuzaki (2011) initiate the exploration of parallel evaluation for list operations, they emphasize the need to extract a computation structure that supports divide-and-conquer summations. Functions that aggregate information in the input list, like list summation, usually compute on each singleton list and then merge the results of independent sublists. Bird(1987) categorized these functions in a set called list homomorphism (pp.12-13). Formally, Morihata and Matsuzaki (2011) state that a function

$$h :: \forall\beta.(\beta \to \beta \to \beta) \to (A \to \beta) \to [A] \to \beta$$

is of list-homomorphism scheme if, for any list x, it satisfies

$$h(++)(\lambda a \to [a])\, x = x$$

Given this skeleton of computation, one can achieve efficient divide-and-conquer parallelism aggregation if there is a proper list-dividing mechanism. List homomorphism forms a basis for handling previously sequential-only functions (like *foldr* and *foldl*) in parallel.

3.1.2 Join List

The list homomorphism doesn't specify how to divide the input list, while the standard data structure doesn't support split by construction. Hence Morihata and Matsuzaki consider using Join Lists to represent the list. The Join List data structure represents lists as binary trees and is defined as follows:

$$\textbf{data JList}\,\alpha = \textbf{Single}\,\alpha \mid \textbf{Join}\,(\textbf{JList}\,\alpha)\,(\textbf{JList}\,\alpha)$$

Using this scheme, any list x, can be transformed into its equivalent join list format using the list-homomorphism scheme h Join Single x. The transformation is reversible. The function *j2l* converts a join list back into its original list representation:

$$\begin{aligned} &j2l :: \forall\alpha.\,\textbf{JList}\,\alpha \to [\alpha] \\ &j2l\,(\textbf{Single}\,a) = [a] \\ &j2l\,(\textbf{Join}\,l\,r) = j2l\,(l) +\!+ j2l\,(r) \end{aligned}$$

Moreover, *j2l* satisfies the lambda function A → [A], effectively mapping each element in a singleton join list back to its original list form, preserving the elements' sequential order in a parallel-compatible structure. This process supports divide-and-conquer parallelism by enabling list recombination through associativity.

3.1.3 Balancing Join Lists

Intuitively, a split in the middle of the list would be an ideal list dividing, while join lists have multiple equivalences when representing the same list. To achieve better parallelism, one would need balanced joint lists. Morihata and Matsuzaki (2011) propose implementing join lists with self-balancing binary search trees like biased search tree and rope.
For instance, biased search trees are suggested due to their efficient handling of modifications while maintaining balance. This balance is crucial for ensuring that the join list height remains within *O(log n)*, allowing the list operations to be performed in a time-efficient manner. Similarly, ropes are employed for lists where concatenation operations are frequent, as they provide an optimized balance for joining and splitting segments.

3.1.4 Implementation Examples

Based on balanced joint lists, Morihata and Matsuzaki (2011) demonstrate that one can easily implement functions of list-homomorphism scheme. For example, a join-list version of sum_{List}, named sum_J,

$$sum_J\ (\textbf{Single}\ a) = a$$
$$sum_J\ (\textbf{Join}\ l\ r) = sum_J(l) + sum_J(r)$$

Comparing to the sum_{List}, sum_J can easily benefit from parallel execution on multiple cores. Each subtree can be computed independently, meaning that the computation can be split across processors or threads. This divide-and-conquer approach is effective in reducing computation time, as each processor handles a portion of the tree simultaneously. For example, consider a join list represented by:

$$Join\,\big(Join\,(\textbf{Single}\,3)\,(\textbf{Single}\,5)\big)\,\big(Join\,(\textbf{Single}\,2)\,(\textbf{Single}\,7)\big)$$

Using sum_J, we can evaluate each pair of Single elements in parallel:

- Calculate sum_J(Join (Single 3) (Single 5)) $= 3 + 5 = 8$ on one processor.
- Simultaneously, compute sum_J(Join (Single 2) (Single 7)) $= 2 + 7 = 9$ on another processor.
- Finally, combine the results of the two subtrees: $8 + 9 = 17$.

This approach highlights the strength of join lists in parallel environments. Since the computation for each subtree is independent, it minimizes the need for sequential processing and leverages concurrent resources, making sum_J highly efficient for large data sets when implemented on balanced join lists.
Another function of interests is $foldr$. The Joint-list version, named $foldr_J$, would be:

$$\begin{aligned} &foldr_J\,f\,e\,(\textbf{Single}\,a) = f\,a\,e \\ &foldr_J\,f\,e\,(\textbf{Join}\,l\,r) = (foldr'_J\,f\,l \circ foldr'_J\,f\,r)\,e \\ &\quad \text{where}\ foldr'_J\,f\,x\,e = foldr_J\,f\,e\,x \end{aligned}$$

Note that $foldr_J$ is still sequential due to the composition of $foldr'_J$, unless there is an efficient implementation of composition. In the meantime, the function signature of $foldr'_J$ shows more potential in parallel evaluation. In this scenario, we can calculate $foldr'_J\,f$ for each subtree in a bottom up manner if the following theorem given by Morihata and Matsuzaki (2010) is respected, namely it states that: *$foldr_J\,f\,e$ can be computed by a bottom-up divide-and-conquer parallel algorithm provided that there exists a set of constant-time functions F such that $f_1, f_2 \in F$ implies $f_1 \circ f_2 \in F$.*

3.2 Parallel Tree Operations

Based on the method described above, Morihata and Matsuzaki (2011) explain how it scales up to binary trees defined as:

$$\textbf{data}\ \textbf{Tree}\,\alpha\,\beta = \textbf{Tip}\,\alpha \mid \textbf{Bin}\,(\textbf{Tree}\,\alpha\,\beta)\,\beta\,(\textbf{Tree}\,\alpha\,\beta)$$

By this definition, each node in a tree is either a leaf node (i.e. with no child) or an internal node (with two children).

3.2.1 Aggregation Computation Structure

Functions that aggregate information on trees have a similar behaviour as those on the list. They would compute on nodes and merge the result from a node with the results from both its subtrees. If one considers aggregation as replacing a subtree with a single node that "holds" the aggregation result of the subtree, then such aggregations are essentially reducing a tree to a leaf, while doing computations during the process. This matches the definition of the parallel tree contraction algorithm stated by Reif (1993). Hence, one can implement aggregation functions via a parallel tree contraction algorithm. An example of a tree contraction algorithm is Shunt, which operates on leaves (Reif, 1993). Since Shunt operations aggregate information from three nodes, Morihata and Matsuzaki (2011) proposed a record type to express aggregating operations in three cases.

$$\begin{aligned}\textbf{data Shunt}\,\alpha\,\beta = \textbf{Shunt}\,\{&\text{left} :: \beta \to \beta \to \alpha \to \beta,\\ &\text{right} :: \alpha \to \beta \to \beta \to \beta,\\ &\text{none} :: \alpha \to \beta \to \alpha \to \alpha\}\end{aligned}$$

For simplicity, the shorthand of above definition is ([left, right, none]). With this record type, the aggregation functions can be expressed as the following polymorphic type:

$$scs :: \forall\alpha, \beta.(A \to \alpha) \to (B \to \beta) \to \textbf{Shunt}\,\alpha\,\beta \to \textbf{Tree}\,A\,B \to \alpha \tag{1}$$

However, one disadvantage of tree contraction algorithms is that they completely destroy the input tree. To reconstruct the input tree after the aggregating, Morihata and Matsuzaki (2011) proposed the following:

$$\begin{aligned}&scs\ \textbf{Tip}\ hole\,connect\,t = t\\ &\textbf{type Context}\ \alpha\,\beta = (\textbf{Tree}\ \alpha\,\beta, \textbf{Tree}\ \alpha\,\beta) \to \textbf{Tree}\ \alpha\,\beta\\ &hole\ :: \forall\alpha, \beta.\beta \to \textbf{Context}\ \alpha\,\beta\\ &hole\ = \lambda(l, r) \to \textbf{Bin}\ l\,a\,r\\ &connect\ :: \forall\alpha, \beta.\ \textbf{Shunt}\ (\textbf{Tree}\ \alpha\,\beta)\,(\textbf{Context}\ \alpha\,\beta)\\ &connect\ = ([connect_L, connect_R, connect_N])\\ &\textbf{where}\ connect_L\ b\ a\ c = \lambda x \to a\,(b\,x, c)\\ &\qquad connect_R\ b\ a\ c = \lambda x \to a\,(b, c\,x)\\ &\qquad connect_N\ b\ a\ c = a\,(b, c)\\ &\qquad x \text{ is of type } (\textbf{Tree}\ \alpha\,\beta, \textbf{Tree}\ \alpha\,\beta)\end{aligned}$$

Moreover, Morihata and Matsuzaki (2011) formally defines that a function *scs* of the polymorphic type (1) is a Shunt-contraction scheme if

$$scs\ \textbf{Tip}\ hole\,connect\,t = t$$

holds for any tree t.

3.2.2 Shunt Tree

From the Shunt-contraction scheme, Morihata and Matsuzaki (2011) proposed Shunt Tree as a binary tree structure.

$$\begin{aligned}&\textbf{data ST}\ \alpha\,\beta = \textbf{T}\,\alpha\\ &\qquad\qquad\mid \textbf{N}\,(\textbf{ST}\ \alpha\,\beta)\ (\textbf{ST}_\bullet\,\alpha\,\beta)\ (\textbf{ST}\ \alpha\,\beta)\\ &\textbf{data ST}_\bullet\,\alpha\,\beta = \textbf{H}_\bullet\,\beta\\ &\qquad\qquad\mid \textbf{L}_\bullet\,(\textbf{ST}_\bullet\,\alpha\,\beta)\ (\textbf{ST}_\bullet\,\alpha\,\beta)\ (\textbf{ST}\ \alpha\,\beta)\\ &\qquad\qquad\mid \textbf{R}_\bullet(\textbf{ST}\ \alpha\,\beta)\ (\textbf{ST}_\bullet\,\alpha\,\beta)\ (\textbf{ST}_\bullet\,\alpha\,\beta)\end{aligned}$$

ST corresponds to **Tree**, while $\mathbf{ST}_\bullet$ corresponds to **Context**. One can transform a tree t to its Shunt-Tree representation by $scs\ \mathbf{T}\ \mathbf{H}_\bullet\ ([\mathbf{L}_\bullet\ \mathbf{R}_\bullet\ \mathbf{N}])\ t$, while reconstructing the tree structure via $s2t$ and $s2c$.

$$
\begin{aligned}
&s2t :: \forall \alpha, \beta.\, \mathbf{ST}\ \alpha\, \beta \rightarrow \mathbf{Tree}\ \alpha\, \beta \\
&s2t\, (\mathbf{T}\ a) = \mathbf{Tip}\ a \\
&s2t\, (\mathbf{N}\ b\, a\, c) = connect_N\, (s2t\ b)\, (s2c\ a)\, (s2t\ c)
\end{aligned}
$$

$$
\begin{aligned}
&s2c :: \forall \alpha, \beta.\, \mathbf{ST}_\bullet\, \alpha\, \beta \rightarrow \mathbf{Context}\ \alpha\, \beta \\
&s2c\, (\mathbf{H}_\bullet\ a) = hole\ a \\
&s2c\, (\mathbf{L}_\bullet\ b\ a\ c) = connect_L\, (s2c\ b)\, (s2c\ a)\, (s2t\ c) \\
&s2c(\mathbf{R}_\bullet\ b\ a\ c) = connect_R\, (s2t\ b)\, (s2c\ a)\, (s2c\ c)
\end{aligned}
$$

3.2.3 Balancing Shunt Trees

Similar to the join list, Shunt Trees have multiple equivalences when representing the same tree. To achieve the best divide-and-conquer parallelism, Morihata and Matsuzaki (2011) proposed using a balanced binary tree implementation that supports self-balancing, like directed topology trees, when implementing Shunt Trees.

3.2.4 Implementation Examples

On normal tree, the sum operations would look like:

$$
\begin{aligned}
&sum_{Tree}\, (\mathbf{Tip}\ a) = a \\
&sum_{Tree}\, (\mathbf{Bin}\ l\ a\ r) = sum_{Tree}\, l\ +\ a\ +\ sum_{Tree}\, r
\end{aligned}
$$

Because the normal tree may have list-like shape (i.e. all left substrees are leaves), the above operation does not guarantee to have optimal parallel executions. With a balanced Shunt Tree, one can guarantee have efficient parallel executions. Based on the Shunt-contraction scheme, a Shunt Tree version of sum operation should have its polymorphic type $= scs\ id\ id\ ([add3,\ add3,\ add3])$. In the Shunt Tree data type, the implementation looks like:

$$
\begin{aligned}
&sum_t\, (\mathbf{T}\ a) = a \\
&sum_t\, (\mathbf{N}\ b\, a\, c)\ = sum_t\, b + sum_c\, a +\ sum_t\, c
\end{aligned}
$$

$$
\begin{aligned}
&sum_c\, (\mathbf{H}_\bullet\ a) = a \\
&sum_c\, (\mathbf{L}_\bullet\ b\, a\, c)\ = sum_c\, b + sum_c\, a +\ sum_t\, c \\
&sum_c\, (\mathbf{R}_\bullet\ b\, a\, c)\ = sum_t\, b + sum_{sc}\, a +\ sum_c\, c
\end{aligned}
$$

4 Conclusion and Future Work

In this paper, the authors proposed a way to achieve parallel computations in purely functional programming. The use of the balanced tree allows the implementation of various complicated parallel algorithms and enables concurrent computations in traditionally sequential calculations. However, the authors didn't provide concrete implementations of the underlying data structures. For instance, it is unclear how to implement the basic tree operations like insert and delete on a balanced Shunt Tree with a directed topology tree. Besides, the authors sometimes assume results based on theorems that do not always apply. This makes some properties hard to prove in the scope of our project. As a result, we will focus more on implementing a balanced Shunt Tree rather than digging into those theorems about parallelism. Having that said, we would like to implement different operations, such as zip and map, on a balanced Shunt Tree and then formally verify their correctness. If time permits, we will explore more applications of Shunt Tree and verify those operations.

5 References

Bird, R. S. (1987). An introduction to the theory of lists. *Logic of Programming and Calculi of Discrete Design*, 5–42. https://doi.org/10.1007/978-3-642-87374-4_1

Morihata, A., & Matsuzaki, K. (2010). Automatic parallelization of recursive functions using quantifier elimination. *Proceedings of the 10th International Conference on Functional and Logic Programming (FLOPS'10)*, 321–336. https://doi.org/10.1007/978-3-642-12251-4_23

Morihata, A., & Matsuzaki, K. (2011). Balanced trees inhabiting functional parallel programming. *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, 117–128. https://doi.org/10.1145/2034773.2034791

Reif, J. H. (1993). Synthesis of parallel algorithms. Morgan Kaufmann Publishers Inc.