CSE305 Project: Parallel Expression Evaluation

Leal Köksal
leal.koksal@polytechnique.edu

Sixtine Lyonnet
sixtine.lyonnet@polytechnique.edu

GitHub: https://github.com/Leal-Koeksal/CSE305Project/

Contents

1 Introduction

Evaluating expression trees is a foundational problem in computer science, with a wide range of applications, notably in compilers, where expressions written in high-level programming languages must be parsed and evaluated correctly. Traditionally, expression tree evaluations are preformed recursively in a "top-down" manner: the algorithm visits each internal node and recursively evaluates it left and right subtrees, and combines the results. While conceptually simple and effective for balances trees, this divide-and-conquer approach can lead to inefficient parallel execution when the expression tree is highly unbalanced.

Parallel tree contraction is an alternative "bottom-up" strategy proposed by Miller and Reif [MR85]. Their approach introduces two key operations, *rake* and *compress*, which systematically reduce the tree in logarithmic time steps, making it highly suitable for parallel computation.

The goal of this project is to implement and evaluate *parallel tree contraction* for expression trees. Our implementation involves construction tress from individual `Node` objects. We explore several approaches to tree evaluation: a sequential recursive algorithm, a manually parallelised "top-down" approach with controlled depth, as well as a deterministic "bottom-up" algorithm based on rake and compress operations, and a randomised bottom-up algorithm.

In this project, Sixtine worked on the sequential tree evaluation algorithm, the fixed thread parallel tree evaluation as well as both randomised evaluation algorithms. Sixtine also wrote the tree constructor functions. Leal worked on the implementation of sequential and parallel `rake` and `compress` as proposed by Miller and Reif.

2 Nodes and Trees

The expression tree is a binary tree composed of `Node` objects. Each node stores either a numeric operand or an operator (e.g., +, −), and links to its left and right children, as well as its parent. Nodes also carry metadata to support evaluation strategies, including deletion status, computed value, and markers for randomized passes.

Leaf nodes represent constants and have no children; internal nodes represent operators and connect subexpressions. The tree structure allows evaluation and transformation through recursive or parallel traversal.

The `Tree` class wraps the root node and provides utility methods for accessing, evaluating, and deallocating the tree. It serves as the main interface for building and manipulating expression trees throughout the contraction process.

3 Sequential Tree Evaluation

The sequential `Tree::evaluate()` method recursively traverses the tree to compute the result of the expression. At each node:

- If the node is `nullptr`, evaluation returns 0.
- If the node is a leaf, its string token is parsed into a `double` and returned.
- If the node is an operator, recursively evaluate the left and right subtrees. Then apply the corresponding operation (+, −, $\star$, or /) to the results. The final value is taken modulo `LARGE_PRIME` (6101) to avoid overflow.

4 Parallel Tree Evaluation

Our parallel evaluator recursively traverses the tree and selectively parallelizes the evaluation of the left subtree, up to a global thread limit `MAX_THREADS`, enforced by a global atomic counter `active_threads`.

At each operator node:

- If the node is null, an exception is thrown. If it is a leaf, its token is parsed and the result is returned via a `promise`.
- If `active_threads < MAX_THREADS`, a new thread is launched to evaluate the left subtree asynchronously using a `promise`/`future` pair. The right subtree is evaluated sequentially.
- Otherwise, both subtrees are evaluated sequentially.
- Results are combined according to the operator (+, −, ⋆, /), with all computations modulo `LARGE_PRIME = 6101` to prevent overflow.
- Any exceptions during evaluation are captured and propagated via the `promise`.

Evaluation begins from the root via:

```
double evaluate_parallel(Node* root) {
    auto p = std::make_shared<std::promise<double>>();
    auto f = p->get_future();
    evaluate_parallel(root, p);
    return f.get();
}
```

This approach guarantees that at most `MAX_THREADS` are active simultaneously.

Performance Results

We benchmarked the algorithm on three tree types: full, left skewed and random binary tree.

Table 1: Parallel Timing and Speedup for Different Tree Shapes

(a) Full Tree (N=100 000)

Threads	Time (s)	Speedup
1	0.433424	1.00×
2	0.000102333	4 238×
3	0.000099375	4 361×
4	0.000175084	2 475×
5	0.000167542	2 589×
6	0.000190958	2 271×
7	0.000228334	1 890×
8	0.000195500	2 214×
9	0.000262584	1 651×

(b) Unbalanced Tree (height=1000)

Threads	Time (s)	Speedup
1	0.000345375	1.00×
2	0.000079000	4.37×
3	0.000105000	3.29×
4	0.000104375	3.31×
5	0.000124459	2.78×
6	0.000146375	2.36×
7	0.000239250	1.44×
8	0.000292667	1.18×
9	0.000288959	1.20×

(c) Random Tree (height=100)

Threads	Time (s)	Speedup
1	0.0505036	1.00×
2	0.000075709	667×
3	0.000107250	471×
4	0.000132542	381×
5	0.000175459	288×
6	0.000167083	302×
7	0.000171792	294×
8	0.000211083	239×
9	0.000231083	218×

Conclusion: For all three tree shapes, the fastest evaluation occurred at `MAX_THREADS = 3` for the balanced tree (speedup $\sim 4361\times$), and at `MAX_THREADS = 2` for both the unbalanced and random trees. Beyond those points, the overhead of spawning and synchronizing additional threads outweighs any further parallel benefit, causing speedups to diminish.

5 Deterministic Tree Evaluation

This section details the implementation and analysis of *Deterministic Tree Evaluation* based on the tree contraction proposed by Miller and Reif. To accommodate this method, modifications were made to the `Node` class. In particular, the division operation has been excluded, and nodes can now also represent linear functions, where applicable. Otherwise, we cannot assure the linearity of node functions or their compositions.

We focus on the core operations defined in the Miller-Reif algorithm: `rake` and `compress`. These two operations are the building blocks of the tree contraction process. We define the sequential application of `rake` followed by `compress` as the `contract` operation. The overall contraction algorithm repeatedly applies `contract` until the tree is reduced to a single node.

The following subsections will describe the `rake`, `compress`, and `contract` operations in detail, along with the implementation-specific considerations and their implications on performance.

5.1 Algorithm

5.1.1 Rake

The `rake` operation is responsible for evaluating and simplifying subtrees in a binary expression tree. It targets nodes where evaluation is already possible (i.e. both children are leaves) or where a function transformation can occur. More specifically, it does this in three cases:

1. **Case 1: Evaluation**

   If a node is an operator and both its children are numeric leaves, we can evaluate the result of this operation. The node is updated to store the result of the operation applied to its children, and both child nodes are marked as deleted.

2. **Case 2: Functional Transformation**

   Given a node, if one child is a numeric leaf and the other an unevaluated expression, the operation is rewritten as a *linear function*, represented in the form `a,b` to mean $f(x) = ax + b$. The constant leaf child is deleted, and the operator node is reclassified as a function node.

3. **Case 3: Functional Evaluation**

   If a node already stores a linear function and has a single numerical leaf child, the function is evaluated with that child as input. The result replaces the node and its child is marked as deleted.

This is achieved through a recursive function `collect_rakeable_nodes()` that traverses the tree and gathers candidate nodes belonging to the three categories illustrated above.

Therefore, each pass of rake simplifies parts of the tree and removes leaf nodes that are no longer necessary. This prepares the tree for further evaluation (or compression).

5.1.2 Compress

The `compress` function is used to contract *chain of function nodes* into a single evaluation function. A chain is a sequence of function nodes, each having exactly one child, a none being a leaf. A chain is compressed by pairwise composing the function nodes, halving the length of the chain. It should be notes that in the sequential solution, chains are fully collapsed, whereas in the parallelised the intended design of Miller and Reif is followed.

For instance, if you have a node `3,2` representing $f(x) = 3x + 2$ and its only child `2,-5` representing $g(x) = 2x - 5$, the composition becomes $f(g(x)) = 6x - 13$. This composition is stored in a new function string `6,-13` in the parent node, and the child is deleted.

This is done recursively in post-order traversal. At each step, the function checks if a node has exactly one child and both the node and child are function nodes. If so, it composes them and replaces the node's function.

5.1.3 Parallelisation

To parallelise the rake and compress operations in the contraction-based expression evaluation algorithm, the sequential contraction procedure was modified to allow multiple nodes to be processed concurrently. The key idea was to identify independent nodes in each contraction round that could be safely contracted in parallel without conflict.

In the sequential version, `rake` and `compress` were applied to one node at a time during the traversal. To parallelise a batch of eligible nodes is collected to be raked, as in the sequential implementation. These nodes were then processed in parallel using threads, ensuring no two threads operated on overlapping subtrees. Further, the parallelised version of `compress` finds chains of unary nodes, them compresses them pairwise in parallel. In order to do that, we introduce a new class `ThreadPool` [Sta14].

5.1.4 A New Class: **`ThreadPool`**

A thread pool is a programming patters used to manage a group of reusable threads to execute tasks efficiently and concurrently. Instead of creating and destroying threads every time you need one, a thread pool maintains a fixed number of threads that wait for a task. This is used instead of creating and destroying threads, as that is costly.

The thread pool is composed of worker threads, a task queue, and synchronisation components to protect shared data and notify waiting threads of new work.

In `rake` for instance, each group of nodes is processed in batches using thread pool tasks:

```
pool.enqueue([=]() { /* process a batch of nodes */ });
```

The variable `BATCH` controls how many nodes are processed together by a single thread pool task. Instead of submitting one task per node, which could create massive synchronisation load, we group nodes into chunks. Each chunk is processed by one task running on one thread. The ideal batch size depends on the cost of processing a single node, the total number of nodes, and the number of available threads.

5.2 Results

5.2.1 Sequential Contraction

Below is a table summarising several runs of `contract` as well as the slowdown in comparison to the serial recursion.

Tree Depth	Contraction Result	Serial Recursion Time (s)	Contraction Time (s)	Slowdown
100000	3008	2×10^{-7}	1.83388	9.17×10^6
100000	2694	3×10^{-7}	1.94713	6.49×10^6
10000	3095	3×10^{-7}	0.211164	7.04×10^5
10000	4537	2×10^{-7}	0.21114	1.06×10^6
10000	5450	3×10^{-7}	0.184749	6.16×10^5
100000	3182	3×10^{-7}	2.7883	9.29×10^6
1000	240	3×10^{-7}	0.0211247	7.04×10^4
1000	4073	6×10^{-7}	0.0282999	4.72×10^4
1000	1828	3×10^{-7}	0.0300297	1.00×10^5

Table 2: Sequential Contraction vs. Serial Recursion: Performance and Slowdown

The result in Table 2 clearly shows that `contract` is significantly slower than the serial recursion. This is expected and justified for several reasons:

- The contraction involved identifying reducible patterns, marking nodes for deletion, updating parent-child links, and composing function representations. These operations add considerable computational memory overhead compared to the straightforward recursive descent used in serial evaluation.
- Contraction is primarily designed as a parallelisable algorithm. When executed with multiple threads. When contraction steps can be applied concurrently, it should yield large performance gains.

5.2.2 Parallel Contraction

We were unable to complete this code fully. The code compiles and run, but the parallel contraction outputs the wrong result. See the table below as an example:

Threads	Batch Size	Serial Result	Parallel Result	Time (s)	Correct?	Speed-down (×)
1	200	1114	2987	0.0079	False	1.27
1	50	5815	2865	0.0074	False	1.18
1	500	5479	3862	0.0062	False	1.00
2	25	3875	4168	0.0075	False	1.20
2	50	3585	4869	0.0096	False	1.54
2	100	3557	2819	0.0088	False	1.41
3	20	637	5556	0.0141	False	2.26
3	40	3580	4281	0.0127	False	2.03
3	80	2899	1694	0.0119	False	1.90
4	15	2882	4175	0.0126	False	2.01
4	30	881	1953	0.0149	False	2.38
4	60	1682	2452	0.0115	False	1.84
4	200	3832	2019	0.0130	False	2.07

Table 3: Parallel contraction results at tree depth 1000 with various thread counts and batch sizes

All parallel versions are slower than the best one (batch size 500, 1 thread). No configuration achieves a meaningful speed-up. Increasing threads does not improve correctness or speed, suggesting race conditions or structural bugs persist.

Here are likely issues that should be checked and fixed if this project was to be continued: The class `ThreadPool` is thread-safe, but the tree operations are not. Multiple threads may access or modify the same node or its children simultaneously. Further, there is no locking mechanism around shared `Node` access. For instance, one thread might read from anode while another deletes or updates it. This leads to undetermined behaviour.

6 Randomised Tree Evaluation

The randomised evaluator simplifies the expression tree in parallel rounds, operating on a vector `nodes` of all active (non-deleted) pointers. An atomic counter tracks the number of remaining nodes.

1. **Initialization.** Construct `nodes` by traversing the tree. Set

   $$\texttt{active\_node\_count} = \texttt{count\_active\_nodes(nodes)},$$

   and determine $T = $ `hardware_concurrency()` threads [Cpp].

2. **Contraction rounds.** In each round, spawn T threads to process disjoint chunks [Gee18]. Each thread performs:

   - *Leaf removal:* Leaf nodes mark their parent and remove themselves, decrementing the counter.
   - *Unary shortcutting:* Degree-1 nodes bypass their parent, linking directly to the grandparent.

   In **randomised rounds**, each degree-1 node randomly selects whether to shortcut (via a “sex” bit), avoiding worst-case patterns.

3. **Main loop.** Run $O(\log\log n)$ dynamic rounds, then alternate randomised contraction until only one active node remains, which holds the final value.

4. **Comparison.** The randomised `COMPRESS` avoids costly pointer-jumping by using coin-flips to eliminate unary chains. In expectation, each round removes a constant fraction of nodes, achieving $O(\log n)$ contraction steps using only $O(n)$ processors. Unlike deterministic versions, it simplifies control logic and avoids bottlenecks in high-degree nodes.

7 Optimal Randomised Tree Evaluation

The optimal randomised evaluator improves over naive random contraction by shrinking the active node set in phases. Each phase reduces the number of nodes from x_i to $x_{i+1} = \lceil \alpha x_i \rceil$, where $\alpha = \frac{31}{32}$, until $x_i < n/\log n$.

The algorithm maintains a vector `nodes` of active (non-deleted) nodes and proceeds as follows:

- **Phased contraction:** While `nodes.size()` > 1, repeatedly:

  1. Perform one round of randomised contraction (Section 6).
  2. Filter out deleted nodes.
  3. If `nodes.size()` $> x_{i+1}$, randomly sample x_{i+1} of them to form the next active set.

- **Final reduction:** Once the active set is small enough, apply dynamic contraction until a single node remains.
- **Output:** The final remaining node holds the result in `getEval()`.

Comparison and Complexity

This multistage approach performs only $O(\log\log n)$ full-size contractions and a final cleanup of $O(n/\log n)$ nodes, reducing total expected work to $\Theta(n)$. It achieves $O(n/T)$ runtime on T processors while preserving the $O(\log n)$ step complexity of randomised PRAM algorithms. Compared to naive parallelism, it provides optimal work-efficiency and faster convergence in expectation.

8 Results

These tests were done on an Apple M1 (8 cores/8 threads) with 8 GB RAM, running macOS 15.4.1 (Build 24E263). Code was compiled using Apple Clang 17.0.0 with -O3 -std=c++17 -pthread and executed under the default macOS power profile, using all 8 hardware threads. All the tests on the parallelised algorithm were done with 2 threads as that was the value we found to be optimal in Part 4.

8.1 Full binary tree

The full-binary tree generation works by first creating exactly $n + 1$ numeric leaves, then repeatedly picking two existing subtrees at random, combining them under a new operator node, and putting that operator back into the pool—until only a single root remains. Concretely, we begin by generating $n + 1$ nodes whose strings are random numbers. While there is more than one node in our list, we pick two distinct indices at random, remove the corresponding nodes (call them "left" and "right"), choose a random operator (e.g. "+", "-", "*", "/"), build a new internal node whose left and right children are those two subtrees, and append this new operator node back into the list. Each such step reduces the list size by one. After exactly n merges, only one node is left: the root. Because each internal node always has exactly two children and we start with $n + 1$ leaves, the resulting tree is guaranteed to be a full binary tree of height $\log(n)$.

Table 4: Timing and Speedup for Different Evaluation Methods ($\max$ threads = 2)

2*N	**Serial**		**Parallel**		**Rand. Paral.**		**Opt. Rand.**	
	Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup
10 000	0.0167998	1.00×	0.0130838	1.28×	0.00637712	2.64×	0.0403439	0.42×
100 000	0.0379808	1.00×	0.0211285	1.80×	0.00963942	3.94×	0.111190	0.34×
1 000 000	0.407437	1.00×	0.307930	1.32×	0.101015	4.04×	1.118150	0.36×

Across all three sizes each method produces the same final result, confirming correctness. For $N =$ 10,000, basic parallel (2 threads) gives a modest $1.28\times$ speedup, while randomised-parallel achieves $2.64\times$; the optimal randomised variant is slower (0.42×) due to its heavier sampling overhead. At $N =$ 100,000, parallel yields $1.80\times$ and randomised-parallel $3.94\times$ gains, but optimal randomised again lags (0.34×). At the largest size $N =$ 1,000,000, randomised-parallel shines with over 4 speedup, basic parallel tops out at 1.32, and optimal randomised remains costly (0.36×). In summary, randomised contraction consistently accelerates evaluation—especially at scale—while the "optimal" scheme's additional bookkeeping incurs prohibitive overhead for these problem sizes.

8.2 Most unbalanced tree

The "most unbalanced" constructor works by starting with a single leaf at the very bottom and then repeatedly wrapping it in a new operator node so that each new level adds exactly one operator above the entire existing subtree and one fresh numeric leaf as its right child. In practice, when you request height $= n$, the code first creates one leaf node. Then for level $= 2$ through n, it picks a random operator (e.g. "+", "-", "*", "/"), makes a new right-child leaf, and builds a parent whose left child is the entire subtree constructed so far and whose right child is that new leaf. By the time you reach level n, you have a single root whose left branch is $n-1$ operators deep (each operator's left child is the previous operator) and whose right child at every operator is just a leaf. The result is a "chain" of operators stretching down the left side to height n, with every right child being a number. The generated tree is therefore as unbalanced as possible.

2***Height**	**Serial**		**Parallel**		**Rand. Paral.**		**Opt. Rand.**	
	Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup
100	2.4666e-05	1.00×	2.05458e-04	0.12×	1.37750e-04	0.18×	1.17977e-02	0.002×
500	1.30833e-04	1.00×	2.43084e-04	0.54×	1.38833e-04	0.94×	—	—
1000	3.93875e-04	1.00×	3.92708e-04	1.00×	2.93208e-04	1.34×	—	—

Table 5: Timing and Speedup for Left-Deep Trees (Height 100, 500, 1000)

Across these left-deep trees, all methods compute the same result, confirming correctness. At height 100 the serial walk (25 μs) is fastest—the plain parallel version (205 μs) and randomised-parallel (130 μs) both run slower (speed-ups 1) since thread overhead dominates; the optimal randomised scheme is especially costly (11.8 ms). At height 500, randomised-parallel nearly matches serial (0.94$\times$) while plain parallel remains slower (0.54$\times$); the optimal variant again failed to complete. At height 1000, randomised-parallel outperforms serial by 1.34×, as sufficient work accumulates to amortise overhead, while plain parallel merely matches serial. The optimal randomised algorithm did not finish for heights 500 or 1000, indicating that there is a bug in the code.

8.3 Random tree

The random tree generator works by growing the tree from the root down to a maximum depth of n. At each node (starting at depth 1), we flip a weighted coin whose probability of "branching" slowly decreases as depth increases—specifically, we branch with probability $0.8 \times (1 - \frac{\text{depth}-1}{n-1})$, so early on there's an 80% chance of creating an internal operator node with two children. If we branch, we pick a random operator ("+", "-", "*", or "/") and recursively build both left and right subtrees at the next depth. Otherwise (or whenever we reach depth n), we stop and make a leaf holding a random numeric string.

Table 6: Timing and Speedup for Left-Deep Trees (Heights 10, 50, 100)

Height	**Serial (s)**	**Parallel (s)**	**Speedup**	**Rand. Paral. (s)**	**Speedup**	**Opt. Rand. (s)**	**Speedup**
10	2.25e-06	8.6792e-05	0.026×	2.39792e-04	0.009×	2.2667e-05	0.10×
50	2.45709e-04	3.37583e-04	0.73×	2.25375e-04	1.09×	1.06241e-02	0.02×
100	2.59066e-02	2.25064e-02	1.15×	1.17778e-02	2.20×	1.73605e-01	0.15×

Across these left-deep trees, all variants compute the same final value, confirming correctness. At height 10, plain parallel (87 μs) and randomized-parallel (240 μs) both underperform serial (2.3 μs), while the optimal randomised scheme (23 μs) achieves only 0.10× because of its overhead. At height 50, randomized-parallel (225 μs) slightly outpaces serial (246 μs, 1.09×), but plain parallel remains slower (0.73×) and optimal randomised is prohibitively slow (0.02×). At height 100, randomized-parallel delivers a 2.20× speedup (12 ms vs 26 ms), plain parallel gives modest 1.15×, and the optimal randomised variant still lags (0.15×). Overall, only randomised contraction on large, deep trees yields net benefits; the optimal scheme's extra bookkeeping outweighs gains on these narrow structures.

9 References

[Cpp] std::thread::hardware_concurrency. https://en.cppreference.com/w/cpp/thread/thread/hardware_concurrency. [Online; accessed 7-June-2025].

[Gee18] GeeksforGeeks. Lambda expression in c++. https://www.geeksforgeeks.org/lambda-expression-in-c/, 2018. [Online; accessed 7-June-2025].

[MR85] G. L. Miller and J. H. Reif. Parallel tree contraction and its application. In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 478–489, 1985. [Online]. Available: https://www.cs.cmu.edu/~guyb/paralg/papers/MillerReif89.pdf.

[Sta14] Stack Overflow community. Reusing thread in loop c++. https://stackoverflow.com/questions/26516683/reusing-thread-in-loop-c, 2014. [Online; accessed 7-June-2025].