A Breadth-First, Ordered Parallel Tree Traversal

Joshua Haley
Undergraduate
University of Central Florida
joshua.haley@knights.ucf.edu

Benjamin Shaia
Undergraduate
University of Central Florida
shaia09@knights.ucf.edu

Joshua Subrahmanyam
Undergraduate
University of Central Florida
sjosh@knights.ucf.edu

Ebin Scaria
Graduate
University of Central Florida
ebinucf@outlook.com

Abstract—There has been a shift towards parallel computing in the computer science field over the past years, which has spurred interest in investigating the parallelization of classic sequential algorithms. Much research has been vested in graphs, particularly the application of breadth-first search. However, a subset of graphs, the tree data structure, has common use in various applications, yet focus tends to remain on graphs. A possible reason is that an effective graph algorithm could easily be applied to a tree, but a tree has a more distinct structure. Therefore, developing algorithms to exploit the specific design of a tree may improve efficiency over the application of a graph algorithm.

We present concurrent breadth first tree traversal algorithm for M-ary trees, where M is an upper bound to the number of child nodes that a given node may have. The algorithm was developed in Java for portability and differs from other approaches in that it shall provide a meaningful ordering of nodes, which is absent in breadth first graph traversals. The approach provides relative fairness in the work done by each thread to aid scalability and eliminates the use of transactional memory, which appears in other concurrent tree traversals that do provide ordering. However, the use of rollback in these methods introduces a higher level of non-determinism to the computation, which we attempt to avoid for a better analysis of the algorithm.

I. INTRODUCTION

The use of a breadth-first search (BFS) in programming has been a topic of interest for several decades. The current transition to parallel architectures has furthered this interest through attempts to effectively parallelize the classic sequential BFS algorithm. However, most research tends to investigate parallel applications related to large, undirected graphs [1][2]. Trees are a subset of graphs that have common applications in graphics (agglomerative clustering) [3], intelligent gaming (decision trees), and data storage (file hierarchies). Yet, little literature has been found on traversals for arbitrary trees. A standard parallel "tree walk" often applies to binary trees [2], and the Barnes-Hut algorithm is specific to oct-trees [4]. Furthermore, workload distribution for large trees may be handled by a message-passing model for multiple processors [5]; this application can have limitations for common multi-core systems utilizing a single CPU.

The structure of a tree, branching out from a source to form various levels, is well suited for a parallel BFS style traversal. So, utilizing current graph algorithms is a reasonable solution to traversing arbitrary trees. However, graphs can contain cycles, and vertices are often stored in an adjacency matrix. The checks for cycles and additional structure for vertices are unnecessary for a tree. Furthermore, parallel BFS algorithms for graphs typically do not provide or need any ordering.

Therefore, the development of a parallel tree traversal algorithm, for arbitrary trees, that takes advantage of current hardware and the tree structure would be beneficial.

II. CONTRIBUTION

We present a unique parallel tree traversal algorithm, which provides a breadth-first ordering of nodes. The existence of trees as a subset of graphs has led to parallel traversals utilizing graph based algorithms. Our solution seeks to eliminate overhead of these algorithms not applicable to trees by leveraging the structure of the tree itself to enhance parallelism. The meaningful ordering of nodes also provides advantages over parallel graph traversals, which are typically only concerned with visiting each node in an arbitrary order. Our major contribution is the generalization of the algorithm to trees of arbitrary height and branching factor. Traversal of arbitrary trees in parallel establishes a wider range of applications for our algorithm including intelligent gaming (decision trees) and file system traversals. Overall, the algorithm provides advantages of sequential traversals and seeks to provide a reasonably fair, balanced, and scalable solution with both commercial and academic applications.

III. RELATED WORK

A. Autoparallezation of Sequential Algorithms

The Galois project exists as an effort to simplify the reasoning of parallel code through the auto-parallelization of sequential algorithms [3]. This approach allows programmers to exploit potential areas of parallelism in an algorithm by dividing tasks into ordered and unordered sets [3]. Ordered sets consist of operations that must execute in a sequential manner, while unordered sets contain operations that can execute completely independent of one another [3]. The system can then execute a program by concurrently iterating through the operations, maintaining the partial ordering of tasks in an ordered set [3].

Conflicts in the Galois approach are resolved by the scheduler through the use of rollbacks, and a pre-defined set of commutative operations is used to detect the existence of conflicts [3]. So, conflict resolution is a two-stage process. The set of commuting operations is checked and conflicts are detected; the conflicting block is copied and rolled back to be executed when no conflict will occur [3].
The Galois approach was applied to a standard BFS, on an undirected graph, using a FIFO queue to store nodes [1]. The processing of nodes in the queue was defined as an ordered operation set, while the addition of nodes to the queue was defined as an unordered operation set [1]. Testing revealed enhanced performance over the sequential algorithm, but scalability was poor [1]. Analysis of Galois showed the scheduler generates a majority of overhead on the system during conflict resolution [1].

The implementation of our algorithm utilizes wait-free auxiliary structures, which aid in the traversal of the tree. We effectively eliminate major areas of conflict, preventing any need for roll backs as a form of conflict resolution.

B. Use of Multithreaded Libraries

The MultiThreaded Graph Library (MTGL) is an ongoing project to develop a generic framework for graph algorithms as part of the C++ libraries [6]. The MTGL is based on the Boost Graph Library and relies mostly on the visitor pattern for graph analysis [6]. The framework allows programmers to develop a visitor object with methods to mark nodes, test edges, and traverse branches of graphs [6]. Additionally, MTGL offers a set of atomic primitives to aid in reading and writing values [6]. A fundamental aspect of MTGL is a design to determine when parallelization is necessary; a threshold value is set and the framework instructs the core machine to parallelize a loop if this value is met, otherwise execution proceeds in a serial manner [6]. Overall, MTGL leverages multithreading architecture to tolerate memory latency and encapsulate common concurrent programming issues, such as hot spotting and race conditions [6].

The framework for the MTGL has been utilized to develop concurrent graph algorithms for S-T Connectivity, Breadth-First Search, and other common graph computations [6]. Testing revealed reasonable reductions in runtime with increasing number of processors [6]. However, a point of concern with the BFS was contention on the tail of a queue due to its implementation as a global structure; a suggested fix, that has yet to be tested, was distributing the queue [6]. Overall, the MTGL helps demonstrate the usefulness of abstractions through generic framework to increase, or at least maintain, reasonable performance on multithreaded architectures [6].

The MTGL framework utilizes the standard queue of a sequential BFS to process nodes in a graph. Furthermore, if the parallelization threshold is not met then code can proceed in a sequential manner. Though our algorithm utilizes a queue, its primary purpose is to aid in the distribution of work for threads. In ideal cases, the queue that handles nodes would be unnecessary. Moreover, we attempt to ensure increased parallelism by leveraging the specific structure of a tree.

C. Asynchronous Graph Traversals

Pearce, Gokhale, and Amato detail the problems of scalability relating to graph algorithms as the graphs become arbitrarily large in size [7]. They cite memory access as the primary issue, arguing that the successive operations occurring on a large data structure that is non-contiguous in memory creates excessive latency [7]. The three also note that algorithms utilizing standard synchronization techniques can limit performance due to potential load imbalances at the synchronization points; collectively synchronous methods and data latencies largely contribute to the poor scalability of common concurrent graph algorithms [7]. The proposed solution was the development of asynchronous algorithms that utilized flash memory technologies to decrease memory latency [7]. Such asynchronous solutions for common graph computations were designed and tested, including Breadth-First Search [7].

However, the BFS was not developed as an independent algorithm but rather exploited the mechanics of a Single Source Shortest Path algorithm. Pearce, Gokhale, and Amato initially designed an asynchronous SSSP; the asynchronous BFS was then obtained by running the SSSP algorithm on graphs with weighted edges equal to 1 [7]. Regardless, the mechanics of each still invoke many of the same principles.

The key to their algorithm is the use of a visitor queue. The visitor queue consists of a conglomeration of priority queues, each of which is associated with an individual thread via a lock [7]. Furthermore, the visitor queue is utilized to hold adjacent vertices of visited nodes to be later processed; the order is decided by a hash function that will select a priority queue [7]. The algorithm proceeds by visiting a source node, enqueuing adjacent nodes, and processing these nodes in a nondeterministic manner [7]. As the nodes are processed, their values in the adjacency matrix are updated to reflect the current shortest path; these procedures repeat until the visitor queue is empty [7]. The primary drawbacks to the implementation is that the nondeterministic processing can cause multiple visits to the same vertex to ensure proper results, and graphs with few independently traversable pathways can reduce the execution to a serial Dijkstra's algorithm [7].

Regardless, testing on both large in-memory and semi-external memory graphs showed significant gains over a serial BFS, along with reasonable scalability as thread count increased [7]. Furthermore, the three showed that for graphs with V vertices, E edges, and p degrees of parallelism, the algorithm is bound by \( O \left( \frac{E}{p} \log \left( \frac{V}{p} \right) \right) \) [7]. The bound implies that with minimal parallelism, the asynchronous algorithm is still guaranteed to run in \( O \left( |E| \log |V| \right) \) time [7]. However, Pearce, Gokhale, and Amato have yet to run analysis on non-flash memory technologies [7].

Our algorithm differs from the presented asynchronous traversals primarily by eliminating backtracking. The nondeterminism of the asynchronous algorithms can incur multiple visits to a node, which would be observed as backtracking in a tree. Our breadth-first traversal moves directly from the root down, without incurring such backtracking.

D. Improving Work Efficiency

Leiserson and Scharfi argue that the standard use of a FIFO queue for a parallel BFS actually restricts the possible parallelism of the algorithm; this limitation results from the
need to serialize operations on the queue [2]. They propose an algorithm that utilizes a unique data structure, called a bag, and layer synchronization, using a reducer object, to improve work efficiency in a parallel BFS [2].

The bag structure acts as an unordered set used to process nodes of a graph during the BFS: the elements of a bag consists of nodes stored in an auxiliary structure called a pennant [2]. A pennant is a simply a binary tree of 2k nodes (k is a non-negative integer) in which the root contains only a left child, and this child forms a complete tree of the other elements [2].

The reducer object aids in concurrency through the use of the mathematical principles of identity and associativity. A sequence of associative operations is divided into sets of independent tasks, and a temporary variable is initialized with the identity function [2]. Each task then proceeds independently, and the results are combined using the same associative function [2]. This result is stored back in the shared variable; the principle of associatively ensures this result is correct regardless of the task division and execution [2]. Leiserson and Schardi provide a BAG-CREATE (identity) and BAG-INSERT (associative) function for their reducer object [2].

The parallel BFS utilizing these changes proceeds by processing a graph by layers [2]. A reducer is created, then a layer is processed; the nodes at each layer are examined concurrently, and the reducer combines any necessary operations before adding a pennant to the bag [2]. The next layer can then be processed from the nodes added to the bag, and this procedure repeats itself until there are no more layers that can be processed [2].

The design of this algorithm does create some race conditions, yet the results are instances of redundant work [2]. So, the non-determinism of this parallel BFS does not affect correctness, but the redundancies make practical analysis difficult [2]. However, Leiserson and Schardi were able to determine a theoretical bound using a modified dag model to account for reducers [2]. Their proof yields the following results for their algorithm:

"For a graph \( G = (V, E) \) with diameter \( D \) and bounded out-degree, the algorithm runs in time \( O\left(\frac{V + E}{P + D + \rho^h} (\frac{1}{D})\right) \) on \( P \) processors, which means a near linear speedup if \( P \ll \frac{V + E}{D + \rho^h} (\frac{1}{D}) \)." [2]

Despite the argument presented by Leiserson and Schardi, we do still implement queues that aid in processing nodes. However, as mentioned, the primary use of the queue is distributing workload as threads proceed down the tree. As a result, we prevent the need for layer synchronization that is utilized in Leiserson and Schardi’s solution.

IV. THE ALGORITHM

A. Preliminaries

We have designed an algorithm to resolve the traversal of trees with arbitrary height \( h \) and branching-factor \( B \). We define branching-factor as the maximum number of children a parent node can have at any point in the tree. So, a binary tree has branching-factor 2, and an oct-tree has branching-factor 8. The height of the tree is derived from the number of child node layers that exists within the tree. We define the root has having height 0, and each subsequent level of child nodes increases the height by 1. Therefore, and arbitrary tree of height \( h \) and branching-factor \( B \) can have no more than \( B^h \) nodes at a single level, and the maximum nodes that a tree can contain is defined by the following equation:

\[
\sum_{i=0}^{h} B^i = \frac{B^{i+1} - 1}{B - 1}
\]

The Java programming language was used to prototype and implement the algorithm because of the threading and concurrency facilities provided by the standard Java API. To avoid possible hidden synchronization methods utilized by trees in the Java standard library, we chose to implement our own tree data structure. This structure is a standard linked tree with the exception that nodes are extended to include a "next" pointer.

The algorithm is designed to modify the tree structure so that it may be traversed like a list. The traversal is performed in parallel while a separate thread may handle outputting the node contents in order. This is accomplished by connecting all nodes on a given level in parallel, easing the traversal, as well as connecting the first node on a given level to the first node on the level beneath. This creates a type of in-place hyper-dimensional list within the levels of the tree, where the tree becomes a list of lists. We shall assume that modifications are done on the original tree, although if the structure of this tree is not allowed to be modified, we may simply make a copy of it before proceeding. Algorithm 1 shows the pseudocode.

B. The Code

We assume that the each node in the tree has both a next and below pointer, where the next pointer shall eventually point to an adjacent node on the level to which a given node belongs, and the below pointer shall point to the leftmost child of a given node. Either of these pointers may be set to null in the case that no such sibling or child node exists respectively.

First, two thread-safe queues are created (Traverse method, lines 1-2). One queue holds the threads that will be performing operations on the tree (threadQueue) and the other will hold nodes of the tree if necessary (nodeQueue). The loop in line 3 of the Traverse method then executes, initializing and enqueuing all working threads into the thread queue. Finally, the head of the thread queue is dequeued and dispatched on the root node of the tree in line 5 of the Traverse method. The thread queue runs in parallel with the threads modifying the tree; execution continues until all threads are once again in the thread queue and there are no nodes in the node queue. However, if there are nodes in the node queue in addition to available threads, then the loop in line 1 of the threadQueue method is used to dispatch available threads on these nodes in a FIFO ordering.
Algorithm 1 TAA Pseudocode

Traverse
1: nodeQueue = concurrentQueue(empty)
2: threadQueue = concurrentQueue(empty)
3: for i = 0 → numThreads do
4:   threadQueue.enqueue(new Thread())
5:   threadQueue.enqueue.dispatch(root)

dispatch
1:   linkChildren()
2:   doWork()

linkChildren
1: for i = 1 → numChildren do
2:   children[i−1].next = children[i]
3:   i = i − 1
4:   current = this.next
5:   while current ≠ null do
6:     if current.below ≠ null then
7:       children[i].next = current.below
8:     break
9:   current = current.next
10:   if current = null then
11:     children[i].next = null

doWork
1: for i = 0 → numChildren−1 do
2:   if threadQueue ≠ empty then
3:     threadQueue.dequeue.dispatch(children[i])
4:   else
5:     leftmostUnworked = i
6:     while i + 1 < numChildren do
7:       nodeQueue.enqueue(children[i + 1]
8:       i = i + 1
9:     this.dispatch(children[i + leftmostUnworked])
10:    threadQueue.enqueue(this)

threadQueue
1: while ¬this.full ∨ ¬nodeQueue.empty do
2:   if ¬nodeQueue.empty then
3:     this.dequeue.dispatch(nodeQueue.dequeue)

When a thread is dispatched on a given node, the loop in line 1 of the linkChildren method executes; the thread links the children of its current node from left to right using their respective next pointers, assuming that this node has children. The loop in line 5 of the linkChildren method is then used to connect the rightmost child to the next node on this child’s level if it exists, or null otherwise (linkChildren method, lines 10-11). Lines 1-3 of the doWork method are then executed to dispatch any available threads in the thread queue onto the children of the current node from left to right. If this thread is able to dispatch a thread on each of its children, then it will enqueue itself into the thread queue (doWork method, line 10). If there are more children to be visited than available threads to dispatch, then lines 6-9 of the doWork method are executed. The thread enqueues the remaining children into the node queue, excluding the leftmost unvisited child; the thread then dispatches itself on this child node.

If the current node does not have children, it shall link this node’s below pointer, to the leftmost child of its closest right-hand sibling with children, assuming that it has a right-hand sibling with at least one child. If this criterion is not met, this node’s below pointer shall remain null. The thread will then enqueue itself into the thread queue (doWork method, line 10).

When the linking is completed, the tree my be output in order. We start at the root node, output it, and travel to the node contained at the below pointer. When we arrive at a given level we save the below pointer at this leftmost node and move on to the node pointed to by the next pointer. When the next pointer is null, we then move to the below pointer and repeat this process.

The structure of the tree before and after the modifications are shown in Figure 1 and Figure 2, respectively.

V. Algorithm Analysis

A. Preliminary Assumptions

We make a few assumptions for the simplicity of our scalability and time-complexity analysis. When any thread has completed the linking of the children of the node on which it has been dispatched, we have a sufficient number of threads to be dispatched on these children. A perfectly balanced tree, though not necessary for correctness, also lends more scalability to our algorithm. One caveat is that we cannot only guarantee that a balanced tree is really our best-case structure when comparing two trees such that the unbalanced tree is of height less than or equal the height of the balanced tree. If we observe two trees such that the unbalanced tree is taller, we may find the bottleneck caused by earlier levels may be overcome more easily and lead to a greater efficiency than the shorter, balanced tree. If either of these assumptions is not met in a given case, we reason that either scalability or runtime efficiency may be reduced.

B. Scalability

To analyze scalability we consider the progression of our algorithm as the threads move down the tree. We assume best-case constraints for simplicity, which include balance for this analysis. We start at the root of the tree; the thread dispatched to this node does its work, dispatches available threads on the children of this node, and enqueues itself into the thread queue to do work. The dispatched threads can then work in parallel on these children, and dispatch other available threads on the children of their respective nodes. Completing the work at a given node before proceeding to the next level, rather than dispatching the threads and making them wait for a given constraint (which introduces more overhead), limits the number of threads that can be running at any given time.
Figure 1. The original tree before modifications by the traversal algorithm.

Figure 2. The tree after initial modifications from the traversal algorithm. Each leaf node has a `below` pointer that points to `null`, but these have been removed to simplify the above image.
At the root node, only one thread may run. The next level allows \( B \) threads to run, where \( B \) is our branching factor and corresponds to the number of children on this level, given our perfect balance assumption in this case. Each of these \( B \) threads may generate \( B \) more threads on the next level, and this process continues as the scope of the algorithm progresses down the tree. The maximum number of threads at a given level is therefore equal to \( B^L \), such that \( L \) is the level we are considering. If \( L \) is maximized, then the number of threads that can run at any point in our algorithm is maximized. If our tree is perfectly balanced, the maximum number of threads that can improve our speed is equal to the number of leaf nodes belonging to our tree, given that this is the base case of our algorithm. This represents the "highest" level that these threads can visit and therefore a maximization of \( L \) in a given case.

Given that our threads will likely be located at different levels of the tree as time progresses, we can observe this behavior in a more realistic setting. We take a snapshot of our algorithm at a given point, and observe the "highest" level that any running thread occupies, where we count from the root starting at \( 0 \). The maximum number of threads that can be running at this point is equal to the number of nodes on this level, which is also the maximum number of nodes that can exist at this level. This result occurs because threads running at one level "lower" than the thread running at the current "highest" level will dispatch a number of threads equal to the branching factor of our tree as they move to this level. While moving back "down" the tree this process continues, where each level we move in this direction brings the maximum number of threads that we could potentially run at any given point down by a factor of \( B \). The "highest" level that our threads may occupy is the one that contains the leaf nodes of our tree, which is where the maximum number of threads may run. Therefore, the algorithm has the potential to scale reasonably well with increased threads and tree size.

C. Best-Case

To analyze the best-case execution of our algorithm, we assume while moving down the tree that threads working in parallel on a given level finish their work at approximately the same time, otherwise runtime efficiency may be reduced. For an optimal traversal, the thread working on the root will finish, and then dispatch \( B \) threads on the children of this node. These threads will finish at approximately the same time; they will then dispatch \( B \) threads each on the children of the nodes on which they are working. This process continues, with threads working on every node on a given level of the tree finishing at the same time. We can then say that in an optimal case, with enough threads, our best-case behavior is a runtime that is equal to the time spent at a given node multiplied by the height of the tree.

D. Average-Case

In an actual runtime scenario, there are two considerations that must be made. Firstly, it is highly unlikely that we will have access to the number of threads specified in our best-case requirements. Secondly, in practice, spawning large numbers of threads can be computationally expensive, and may actually lead to reduced efficiency. This means that these constraints may lead to a better theoretical complexity, while we may wish for a more conservative number of available threads. With these considerations in mind, we would also like to analyze a more realistic scenario, and analyze our algorithms performance under these constraints.

The number of operations that must be performed on a given tree is independent of the number of threads that must work on this tree. This implies that an increase in the number of threads will not result in a higher number of dispatches on the nodes of the tree; rather an increase in the number of enqueues and dequeues in both the node and thread queues may occur.

This means that the deciding factor in efficiency of our algorithms is the efficiency of the underlying node and thread queues. Therefore, detection of significant performance issues may be the result of the underlying behavior of these queues and require adjustments within these auxiliary structures.

E. Worst-Case

We will assess our algorithms performance when working with only a single thread, as that can be considered the worst-case runtime. In this case, the single thread will be dispatched on the root, and will link the rootOs children. It then enqueues all the children of the root node into the node queue, except for the leftmost node. The thread then dispatches itself on the leftmost child of the root node. Once the thread reaches the bottom of the tree, it enqueues itself into the thread queue, where it is then dispatched on the first node in the node queue. This corresponds to the second node from the left on the level just below the root, assuming a balanced tree.

We can simplify this analysis, by observing the behavioral equivalences to a serial breadth-first tree traversal compared with additional components used to leverage potential concurrency. Again, we see that the same number of dispatches is needed, which simply link the children. The exact same linking behavior is needed by both the serial and concurrent traversals. The additions to the concurrent approach are the use of the thread and node queues. So, the algorithm still relies on the performance of the underlying queues yet behaves nearly identical to the serial breadth-first traversal when utilizing a single thread.

F. Concurrent Properties

Our algorithm is inherently wait-free [8], [9], [10], [11], [12] and by extension, lock-free. These properties are obtained because the work performed by a dispatched thread is done independently of any other thread. The only contention that can occur is in the use of the queue, but these are also lock and wait free in our implementation.

To verify this independent work case we may simply list those data elements that a given thread must modify and ensure that no other running thread will make additional modifications. We must also ensure that if a given data element
must be in a particular state, this state must be set by a thread that has completed its work at previous level of the tree in which it was assigned this task. If this can be proven, we will also prove that this lock-free and wait-free classification relies on the classification of the queue being used.

The data elements that a thread may need to access on a given node consists of all the next and below pointers belonging to its siblings in the final state that these pointers may take. Additionally, all the next pointers belonging to its children in their initial state are needed. All of the below pointers throughout the tree will be set on the creation of the tree, so no conflict will occur when trying to access these pointers that belong to the siblings of any given node. All of the next pointers belonging to the siblings of a given node will be set by the thread that worked on the parent of these children, and this setting will constitute the finishing of the thread Os work at that level. So, there will also be no conflict caused by this data element. Even though a thread may do later work after completion at a given node, it is considered to be a separate assignment; the critical task of the thread has already been completed. This may be made more obvious by the fact that we could just as easily allow this thread to die after completing the work at a given node and instead spawn a new thread, but this would cause more overhead. Our last consideration will be the child nodes, and we are guaranteed that only the working thread will need to set these pointers by virtue of the fact that only one thread will ever be dispatched on any given node.

VI. TESTING

Our algorithm was implemented in Java to allow for easy multi-threading and rapid development. The algorithm was tested against a sequential implementation of BFS to assess correctness and efficiency. The hardware used to test the system was a 2.0 GHz Intel i7 hyper-threaded Quad Core laptop with 12GB of 1600 Mhz DDR3 memory. To test correctness, unit test functionality was integrated that would generate a tree and run serial BFS and our algorithm against the tree. The results are then compared to each other to ensure the correctness of our algorithm. To test the performance, trees of different sizes were generated and then had a standard Breadth-First Search run and timed. This is compared against our algorithm, which was run for various numbers of threads upon the same tree. Each run was done five times and averaged. The trees generated are of a branching factor two and are various heights, until the maximum height allowable by the memory limitations of our hardware platform was reached. This experimental method allows us to judge the correctness of our algorithm, gauge average runtimes, and assess trends as numbers of threads and graph size are manipulated.

VII. RESULTS

Testing verified correctness of the parallel tree traversal when compared the serial BFS implementation. However, initial results showed mildly erratic performance, which averaged to reduced efficiency over the sequential algorithm. To gain further insights of the underlying reasons for our observations, we plan to use the experimentation methodology for non-blocking data structures proposed by Izadpanah et. al [13]. Examining our implementation, we reasoned that our algorithm was performing memory accesses for most of its operations, yet real applications would incur more operational complexity. We ran the test again, adding such complexity, to better assess overall efficiency. The results are shown in Figure 3 and Figure 4.

Figure 3 shows the runtime for an increasing number of threads and tree size. As the number of nodes to process increased, runtime increased linearly as expected. Furthermore, we observe that the use of 3 or more threads provides increased performance over the sequential algorithm. However, the performance gains begin to flatten as the number of threads increases.

Figure 4 better illustrates this result, comparing runtime to the number of threads used in our concurrent solution. The graph shows that the performance increase given by adding more threads of execution is a diminishing return that appears to follow AmdahlOs law. After approximately eight threads are added, which is the maximum number of execution units on our test machine, the algorithm actually begins to perform worse due to thread contention for execution units and the overhead introduced by excessive context switching.

Overall, we observed that our traversal algorithm provides increased performance over the use of a sequential BFS in the context of increased operational complexity for locating child nodes. Such complexity can be observed in applications using decision trees and traversals of file systems [14], [15], [7], [16]. Therefore, the algorithm has potential for use in commercial and academic applications; resolving memory latency issues in future work may also further expand its use.

VIII. PROBLEM ANALYSIS AND POTENTIAL SOLUTIONS

Despite performance gains, we must not ignore the potential problems leading to performance reduction during initial testing. We suspect several underlying issues may have contributed to the reduced performance. Following, we present a detailed interpretation of such problems and propose possible solutions for future work.

A. Available Cores

Our testing was performed on a quad-core i7 Intel processor capable of hyper-threading. This means that we can only safely implement four threads while hyper-threading allows for the use of "eight" threads. Any tests that we run with more than eight threads are guaranteed to be at the mercy of the scheduler, potentially skewing results. This limitation in hardware also makes scalability more difficult to interpret through quantitative tests. We would like to run our program on a computer with more cores that can leverage a greater number threads to adequately test scalability.

B. Test Case Size

We have already observed that while moving down the tree, bottleneck is incurred, so it follows that we may not
Figure 3. Comparison of multithreaded algorithm against serial BFS for increasing tree size

Figure 4. Runtime of algorithm compared against number of threads
drastically enhance performance in small trees. One issue that we encountered is that the highest tree we could generate was height 23 with a branching factor of 2. While this has a high node count, this is really not an extreme test case. Both algorithms consistently finished this test case in less than five seconds, which may not be sufficient to overcome the overhead caused by threading.

C. Java Implementation

We chose to implement our algorithm in Java primarily due to the ease of use, which aided in determining the correctness of our algorithm. However, we have concluded that the language did not grant us enough control to ensure that we are doing similar operations in both our algorithm and the serial implementation. The garbage collector may have also interfered with the performance of both algorithms, as we encountered a number of cases in which the runtimes varied greatly from the norm. We would like to implement our algorithm in C++ as we will be given more control of the underlying processes. We can thereby reason better about our algorithm’s behavior and performance, ensuring transparency in our operations.

D. Use of Wait-Free Queues

As we have stated previously, our algorithm relies on the use of wait-free queues and we do not have an easy way to verify their performance. When our algorithm performed poorly, we were only able to hope that this was not caused by our use of these queues. To pinpoint these problematic areas we would like to utilize profiling tools, such as TLA/TLC for Java [17], to better assess the concurrent model of our algorithm.

Furthermore, we discovered that reverting to an earlier iteration of our algorithm reduced the average time each thread spent in the thread queue. This was done by making each thread responsible for dispatching available threads on the node queue before it has completed its work and before dispatching threads on its children. This is due to the fact that all threads are responsible for the node queue, rather than just one.

We would like to further experiment with breaking up the public node and thread queue, exchanging it for a set of local queues within each thread. We can implement a standard queue for the nodes off which a thread continues to work and allow a thread that has terminated to enqueue itself into the local thread queue belonging to a given running thread. This running thread would then dispatch available threads from its local thread queue onto its local node queue, which we may find to reduce overhead caused by the use of wait-free queues.

E. Serialized Memory Access

A major issue that we suspected was hindering performance increase was the limitation in testing to only the linking of children as the operation done at each node. Real world application would likely incur operations to be performed on the contents of each node. Linking requires memory reads and writes, and we only have a single memory bus, so the CPU was performing all these operations serially. Furthermore, we incurred the overhead introduced by the use of threading and wait-free queues, so it appeared as if our algorithm was much less efficient.

IX. Conclusion

The current paradigm shift to parallel computing has led to increased interest in the parallelization of classic sequential algorithms. In this work, we present a unique solution to multithreaded tree traversals, which enforces a breadth-first ordering of nodes. The algorithm is generalized to traverse arbitrary trees of height $h$ with branching factor $B$, providing advantages of current parallel tree walks designed for a specific type of tree data structure. The ordering of nodes and elimination of rollbacks also provides advantages over parallel graph traversals that may be applicable to the tree data structure. Furthermore, we demonstrated performance gains over a sequential BFS when run on binary trees of varied height.

We also examined areas of concern relating to memory latency that we would like to address in future work. We reasoned that hardware limitations and the restrictions of the Java language mitigated our ability to provide a more in-depth assessment of efficiency. In future work, we would like to implement the algorithm in C++ to provide more control over memory management, allowing us to better isolate latency concerns. We also intend to investigate various ways to implement the thread and node queues to better enhance parallelism within the algorithm. Moreover, the use of model checkers in these future endeavors will greatly aid in isolating, and improving, issues related to concurrency.

Overall, we accurately developed a parallel traversal, applicable to arbitrary trees. The algorithm provides a breadth-first ordering of nodes and reasonable performance enhancements over a sequential BFS. Furthermore, future developments of the algorithm can have strong potential in both commercial and academic applications.

In our future work we plan to employ the use of transactional data structures [18], [19], [20], [21], [22], [23] to support more complex operations and also verify the correctness of our non-blocking implementation using tools relying on formal techniques, such as CCSpec [24] and TxADT [23].

REFERENCES

[1] M. A. Hassaan, M. Burtscher, and K. Pingali, “Ordered and unordered algorithms for parallel breadth first search,” 2010.
[2] C. E. Leiserson and T. B. Schardi, “Work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers),” 2010.
[3] M. Kuilkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and P. L. Chew, “Optimistic parallelism requires abstractions,” September 2009.
[4] A. Y. Grama, V. Kumar, and A. Sameh, “Scalable parallel formulations of the barnes-hut method for n-body simulations.”
[5] Y. Xu, “Scalable algorithms for parallel tree search,” October 2007.
[6] P. Pearce, M. Gokhale, and N. M. Amato, “Multithreaded asynchronous graph traversal for in-memory and semi-external memory,” 2010.
[7] J. W. Berry, B. Hendrickson, S. Kahan, and P. Konecny, “Software and algorithms for graph queries on multithreaded architectures,” January 2007.
[8] S. Feldman, P. LaBorde, D. Dechev, and C. M. level Arrays, “Wait-free extensible hash maps,” in Proceedings of the 13th IEEE International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, (Samos, Greece), July 2013.

[9] S. Feldman, P. LaBorde, and D. Dechev, “Tervel: A unification of descriptor-based techniques for non-blocking programming,” in Proceedings of the 15th IEEE International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, (Samos, Greece), July 2015.

[10] A. Barrington, S. Feldman, and D. Dechev, “A scalable multi-producer multi-consumer wait-free ring buffer,” in Proceedings of the 30th ACM/SIGAPP Symposium on Applied Computing, (Salamanca, Spain), April 2015.

[11] S. Feldman, P. LaBorde, and D. Dechev, “A wait-free multi-word compare-and-swap operation,” International Journal of Parallel Programming, vol. 43(4), pp. 572–596, February 2015.

[12] S. Feldman, C. Valeraleon, and D. Dechev, “An efficient wait-free vector,” IEEE Transactional on Parallel and Distributed Systems, vol. 27, pp. 654–667, May 2016.

[13] R. Izadpanah, S. Feldman, and D. Dechev, “A methodology for performance analysis of non-blocking algorithms using hardware and software metrics,” in Proceedings of the 19th IEEE International Symposium on Object/component/service-oriented Real-time Distributed Computing, (York, UK), May 2016.

[14] S. Feldman, A. Bhat, P. LaBorde, Q. Yi, and D. Dechev, “Effective use of non-blocking data structures in a deduplication application,” in Proceedings of the 4th ACM Conference on Systems, Programming, Languages and Applications: Software for Humanity, (Indianapolis, IN), October 2013.

[15] D. Dechev and B. Stroustrup, “Scalable nonblocking concurrent objects for mission critical code,” in Proceedings of 24th International Conference on Object-Oriented Programming Languages, and Applications, (Orlando, Florida), October 2009.

[16] D. Dechev, P. Pirkelbauer, N. Rouquette, and B. Stroustrup, “Semantically enhanced containers for concurrent real-time systems,” in Proceedings of the International Symposium and Workshop on Engineering of Computer Based Systems, pp. 48–57, 04 2009.

[17] L. Lamport and Y. Yu, “Tlc - the tla+ model checker,” June 2011.

[18] D. Zhang and D. Dechev, “Lock-free transactions without rollbacks for linked data structures,” in Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, (Pacific Grove, CA), July 2016.

[19] D. Zhang, P. LaBorde, L. Lebanoff, and D. Dechev, “Lock-free transactional transformation,” ACM Transactions on Parallel Computing, vol. 5, June 2018.

[20] P. LaBorde, L. Lebanoff, C. Peterson, D. Zhang, and D. Dechev, “Wait-free dynamic transactions for linked data structures,” in The 10th International Workshop on Programming Models and Applications for Multicores and Manycores, (Washington DC, USA), February 2019.

[21] V. Cook, Z. Painter, C. Peterson, and D. Dechev, “Read-uncommitted transactions for smart contract performance,” in Proceedings of the 39th IEEE International Conference on Distributed Computing Systems, (Dallas, TX), July 2019.

[22] L. Lebanoff, C. Peterson, and D. Dechev, “Check-wait-pounce: Increasing transactional data structure throughput by delaying transactions,” in Proceedings of the 19th International Conference on Distributed Applications and Interoperable Systems, (Lyngby, Denmark), June 2019.

[23] C. Peterson and D. Dechev, “A transactional correctness tool for abstract data types,” ACM Transactions on Architecture and Code Optimization, vol. 14, December 2017.

[24] C. Peterson, P. LaBorde, and D. Dechev, “Cspec: A correctness condition specification tool,” in the 27th IEEE/ACM International Conference on Program Comprehension, (Montreal, QC, Canada), May 2019.