Treelogy: A Benchmark Suite for Tree Traversals

Nikhil Hegde, Jianqiao Liu, Kirshanthan Sundararajah, Milind Kulkarni
School of Electrical and Computer Engineering
Purdue University, West Lafayette
Email: {hegden, liu1274, ksundar, milind}@purdue.edu

***Abstract*—An interesting class of *irregular* algorithms is *tree traversal* algorithms, which repeatedly traverse various trees to perform efficient computations. Tree traversal algorithms form the algorithmic kernels in an important set of applications in scientific computing, computer graphics, bioinformatics, and data mining, etc. There has been increasing interest in understanding tree traversal algorithms, optimizing them, and applying them in a wide variety of settings. Crucially, while there are many possible optimizations for tree traversal algorithms, which optimizations apply to which algorithms is dependent on algorithmic characteristics.**

In this work, we present a suite of tree traversal kernels, drawn from diverse domains, called *Treelogy*, to explore the connection between tree traversal algorithms and state-of-the-art optimizations. We characterize these algorithms by developing an ontology based on their structural properties. The attributes extracted through our ontology, for a given traversal kernel, can aid in quick analysis of the suitability of platform- and application- specific as well as independent optimizations. We provide reference implementations of these kernels for three platforms: shared memory multicores, distributed memory systems, and GPUs, and evaluate their scalability.

I. Introduction

Applications in a number of computational domains including scientific computing [1], [2], computer graphics [3], data mining [4]–[7], and computational biology [8], are built around *tree traversal* kernels, which perform various computations by traversing trees that capture structural properties on input data. Because these tree traversals are time consuming, memory intensive, and complicated, there has been substantial interest in developing optimizations and implementation strategies that target improving the performance of tree traversal [4], [9]–[15].

While tree traversal kernels are widespread, they are also highly varied in terms of the types of trees that are used (e.g., octrees, kd-trees, ball trees, etc.), the traversal patterns of those trees, the number and variety of separate traversals that are performed, etc. Each new tree algorithm requires careful thought to determine which implementation strategies and optimizations are likely to be effective, and hence developers of new tree-based algorithms may struggle to devise efficient implementations.

Conversely, developers of optimizations targeting tree traversals also face challenges in determining how effective and applicable their optimizations are. Unfortunately, existing graph benchmark suites feature only a handful of tree traversal kernels [16]–[21], not providing enough variety to understand the generality and behavior of new optimizations. Tree traversals are a distinct subclass of graph algorithms, with their own unique challenges—traversals touch large portions of highly-structured tree data, unlike most graph kernels which operate on small, localized *neighborhoods* of the graph [22]—so evaluating implementation strategies and optimizations for tree kernels requires benchmarks that cover the breadth of tree traversal behaviors.

Interestingly, many optimizations depend on particular *structural* characteristics of tree traversal kernels to be effective: some optimizations target only top-down, pre-order traversal kernels [15], while others work for top-down traversals but not for bottom-up traversals [9]. Others rely on a fixed traversal order of the tree [12], while still others allow traversals to traverse the tree in any order [14]. However, there has not been any attempt to identify this set of characteristics in a manner that allows multiple optimizations to be targeted to particular tree traversal kernels. Much past work on tree traversals was aimed at efficient expression and optimization of specific tree traversal kernels [3], [4], [11], [13], [23]. Understanding these characteristics, and how they affect optimization opportunities, is critical to optimizing tree traversal kernels.

A. Contributions

To better understand traversal algorithms, and develop and understand optimizations for those algorithms, it is helpful to have a set of benchmarks that span a wide range of characteristics. To that end, this paper presents *Treelogy*, a benchmark suite and an ontology for tree traversal algorithms.

1) We present a suite of nine algorithms spanning several application domains: (1) Nearest neighbor [5]; (2) K-nearest neighbor [5]; (3) Two-point correlation [4]; (4) Barnes-Hut [1]; (5) Photon mapping for ray tracing [3]; (6) Frequent item-set mining [6]; (7) Fast multipole method [2]; (8) K-means clustering [7]; (9) Longest common prefix [8].
2) We develop an *ontology* for tree traversal kernels, categorizing them according to several *structural* attributes. Treelogy kernels span the ontology: for each category, Treelogy has at least two kernels of each type.
3) We present a mapping of existing tree traversal optimizations to the types of traversals described by our ontology, and show how the ontology can guide which optimizations can be applied to which kernels and vice-versa.
4) We evaluate the benchmarks in Treelogy with multiple types of trees, on real and synthetic inputs, and across multiple hardware platforms: GPUs, shared memory, and distributed memory. Using our evaluation framework,

978-1-5386-3890-3/17/$31.00 ©2017 IEEE

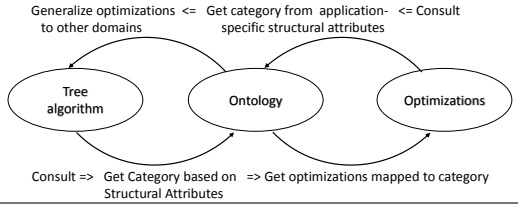


Fig. 1: Treelogy use case.

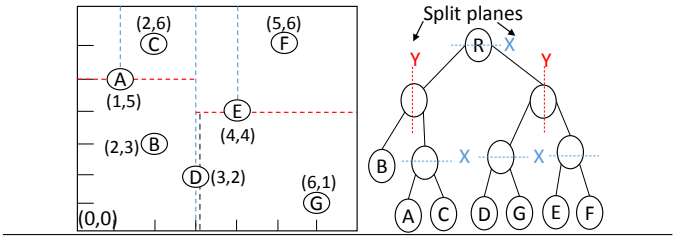


Fig. 2: Sample space (2D) and corresponding kd-tree.

we present results showing the scalability of reference implementations of the benchmarks, and demonstrate that traversal algorithms with certain tree types yield better performance compared to the "standard" tree.

Treelogy thus benefits both the designers of new tree algorithms as well as the developers of new tree traversal optimizations, as shown in Figure 1. If a designer devises a new tree algorithm, he or she can use Treelogy's ontology to quickly categorize the algorithm and determine which existing optimizations and implementation strategies are likely to yield an efficient implementation. If a developer produces a new optimization for tree traversals, he or she can use the ontology to help determine what structural properties are necessary for the optimization to apply, and use the benchmarks of Treelogy to evaluate his or her optimization against reference implementations. Treelogy is publicly available at: `https://bitbucket.org/plcl/treelogy`.

B. Outline

The remainder of this paper is organized as follows. Section II presents background on tree traversal algorithms, including a discussion of spatial trees and a general skeleton for tree traversal algorithms. Section III describes the kernels of Treelogy. Section IV presents our ontology for tree traversal algorithms, maps the kernels of Treelogy to the ontology, and discusses how this ontology can be used to determine the suitability of different optimizations. Section V evaluates the kernels of Treelogy on multiple inputs and on multiple platforms. Finally, Section VI summarizes related work and Section VII concludes.

II. Background

This section describes spatial indexing structures that organize the input data in the form of the trees that underpin the tree traversal algorithms. Next it discusses the structure of an example tree traversal kernel and optimizations that are common to most of Treelogy's benchmarks (i.e., those that do not depend on deeper structural characteristics).

A. Trees for accelerating computations

The use of trees to optimize different types of computations is common across many algorithms. These trees often take one of two forms: *spatial acceleration structures*, that organize data in an n-dimensional metric space, and *n-fix trees* (our term that covers both prefix and suffix trees) that organize sets of sequences according to similarity.[1]

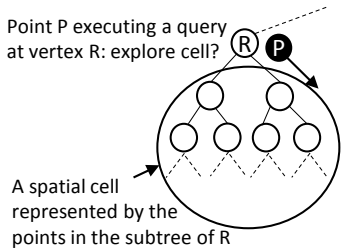
1) Spatial acceleration structures

As Gray and Moore argue, an effective, general way to speed up the computation of n-body algorithms is through the use of spatial trees [4]. An n-body computation, in its naïve implementation, requires comparing each of a set of items with every other item in a data set. Rather than performing this $O(n^2)$ process, spatial trees organize the items of the set into a tree structure to speed up the comparison process. These trees work for data that is embedded in a metric space (or a pseudo-metric space, where a distance can be determined between any pair of points).

For example, a kd-tree [24] organizes k-dimensional spatial data (hence the name kd-tree) by recursively splitting the set of points into subspaces by cutting the space using a split plane along one of the dimensions to divide the points in the current space in half. Each split subdivides the space into two, and the two subspaces are made children of the parent space in the kd-tree. In this way, the tree is built top down, with the root of the tree representing all of the points, and the leaves of the tree representing a single point (or a small set of points). Figure 2 shows an example of a kd-tree built in two dimensions over 7 points. This structure allows for a very fast proximity check between any point p and the entire subspace represented by a node in the kd-tree: if any part of the subspace is "close" to p, that means that the subspace may contain a point within that distance of p. Because most n-body codes are fundamentally concerned with the question of which points are close to each other (whether to compute a 2-point correlation, or estimate forces, etc.), this query allows the tree to be traversed, and in doing so, quickly eliminate entire subspaces without individually visiting points (see Figure 3(a)).

So, for example, to use a kd-tree to compute the two-point correlation of a point p, where the purpose is to determine how many points are within a radius r of p, the point p starts at the root of the kd-tree. As long as the current cell of the kd-tree overlaps with any portion of the (k-dimensional) sphere around p, the point traverses down the tree. Otherwise, that entire subtree (and hence subspace) can be truncated from the traversal. If p reaches a leaf of the tree, it compares with the point(s) in that leaf. In this way, p's 2-point correlation can be computed without comparing to every other point in the space.

[1] We do not mean to imply that these are the only types of trees; merely that they are among the most common.



(a) Query: is any point in the cell close enough to P? If no, don't explore cell

```
1  void Traverse(Vertex v, Point p, float radius)
2  {
3      if(distance(v, p) > radius)
4          return;
5      if (v.isLeaf())
6        foreach (Point pl : v)
7          if (distance(pl, p) < radius)
8            p.corr++;
9      else
10       Traverse(v.leftChild, p, radius);
11       Traverse(v.rightChild, p, radius);
12 }
```

(b) Two-Point Correlation

```
1  void Traverse(Vertex v, Point p){
2      status = ComputeVertex(v, p)
3      if(status is truncate)
4          return
5      if(status is traverse_left){
6          Traverse(leftChild, p)
7          Traverse(rightChild, p)
8      }
9      else{
10         Traverse(rightChild, p)
11         Traverse(leftChild, p)
12     }
13 }
```

(c) Nearest Neighbor

Fig. 3: Purpose of spatial acceleration structures and traversal pseudocodes.

Figure 3(b) shows pseudocode for this traversal. Other n-body computations can be performed in similar ways [4].

Kd-trees are one of many types of spatial-acceleration trees. Others include vp-trees (vantage point trees) [5] where subspaces are determined not by being on one side or another of an orthogonal split plane, but instead by being inside or outside hyperspheres. BSP trees (binary space partitioning trees) [25] use split planes to divide subspaces just like kd-trees, but those planes may not be orthogonal. Octrees [26] for 3-dimensional spaces, as used in Barnes-Hut, do not evenly divide subspaces by the distribution of points but instead by distance: when a cell is divided into subspaces, the subspaces are eight equally-sized subspaces. Quad-trees are similar to octrees and are for 2-dimensional spaces. Unlike kd-tree or octree, sibling subspaces in Balltree [27] are allowed to intersect and need not partition the whole space. As we see in Section V, the type of spatial tree used affects the behavior of each traversal, and hence performance.

2) n-fix trees

Another common type of tree is used to organize data that is not embedded in a metric space, but instead shares another form of similarity: common sequences. For example, prefix trees (tries) are a space-efficient structure for storing a set of strings, with nodes in the tree representing letters, and paths in the tree representing words. Two words that share a common prefix (e.g., "cat" and "car") share the same path in the tree for their common prefix ("ca-"), and then split into two children for their suffixes. Suffix trees are similar, but organized in the opposite direction. While these trees are often used for efficient representations of strings (suffix trees are especially common for representing sets of genomic sequences [28]), they also are used for other sets, such as in frequent item-set mining [6].

B. Traversal structure and optimizations

Efficiently implementing point correlation (Figure 3(b)) requires considering many issues. Note that a point's traversal touches the vertices of the tree in depth-first order resulting in no scope for intra-traversal parallelism if the traversal order is to be preserved. However, more often, multiple points traverse the tree, these points are independent of each other, and the tree is not modified during traversal. Hence, there exists ample coarse-grained parallelism. In such a scenario, when multiple traversals touch a set of vertices and perform computation at vertices in the set (referred to as vertex computation from now on), more than one traversal may touch the same vertex. Therefore, traversals can be reordered in such a way that those touching similar vertices can be scheduled in close succession so as to enhance temporal locality. Block scheduling [29] is a generic locality enhancement technique where a *block* of points traverses the tree rather than a single point, allowing multiple points to interact with a single vertex from the tree.

While the previous paragraph described the properties of the traversal kernel and optimizations possible for point correlation only, we find that many tree-based traversal kernels share these properties. In fact, we can systematically categorize tree traversal kernels so that kernels in the same category share similar properties and hence can be reasoned about collectively when they are the focus of optimizations. Section IV describes this ontology.

III. Treelogy Traversal Kernels

This section presents the 9 traversal kernels of Treelogy: *Nearest Neighbor* (NN), *k-Nearest Neighbor* (KNN), *Two-Point Correlation* (PC), *Barnes-Hut (BH)*, *Photon Mapping* (PM), *Frequent Itemset Mining* (FIM), *Fast Multipole Method* (FMM), *k-Means Clustering* (KC), and *Longest Common Prefix* (LCP).

1) ***Nearest Neighbor*** [5] is an optimization problem in data mining, image processing, and statistics, where the goal is to find closest point(s), based on some distance function, to a query point. Figure 3(c) shows pseudocode for an NN implementation. First, a set of points representing the *feature space* are organized into a metric tree. A query point then traverses the tree depth-first to find its closest neighbor(s). The traversal begins by guessing the distance to closest neighbor and sets it to a very large value. At every vertex on the traversal path, the query point determines whether the vertex's subspace *could* contain a point that is closer than the current guess. If so, the query point updates the current guess and the traversal proceeds to explore the vertex's children. If not, the traversal is truncated and proceeds to unexplored vertices of the tree. When the traversal reaches a leaf, point(s) in the leaf's subspace are inspected and the query point updates its guess for its closest neighbor. Typically, there exist several query points and hence offer scope for parallel execution. Treelogy includes NN implementations

using kd- and Vantage Point (vp) trees.

2) ***K-Nearest Neighbor*** finds the k nearest neighbors to each query point [30]. Instead of chasing the only nearest neighbor, KNN maintains a distance buffer to record the K closest neighbors. KNN is more robust to noisy training data than NN, and the k-neighbor buffer gives it a significantly different traversal pattern than NN. Treelogy includes KNN implementations using ball-trees and kd-trees.

3) ***Two-Point Correlation*** [4] The traversal algorithm in two-point correlation was introduced in section II-A1. This is an important algorithm in statistics, and data mining used to determine how clustered a data set is. As in the case with NN, different types of trees are possible for input representation. Treeogy includes PC implementations using kd- and vp-trees.

4) ***Barnes-Hut*** [1] is an efficient algorithm to predict the movement of a system of bodies that interact with each other. BH accelerates the computation of forces acting on a body due to its interaction with other bodies. It does this by building an octree (3-dimension) based on the spatial coordinates of those bodies. Every body then traverses the tree top-down to compute the force acting upon it. The force due to far away bodies can be approximated based on the center of mass of those bodies, allowing direct body-to-body interactions to be skipped. The algorithm runs for multiple time steps. At the end of each time step, the bodies' positions are updated based on the computed forces, and the tree is rebuilt using the updated positions. Treelogy includes BH implementations using octree and kd-trees. Our kd-tree implementation is based on the recursive orthogonal bisection method [31].

5) ***Photon Mapping*** [3] is an algorithm to realistically simulate interaction of light with objects in a scene. A kd-tree is used to accelerate the ray-object intersection tests. The objects in a scene are represented by triangle meshes. The triangle coordinates are then organized into a kd-tree structure. Each of a set of input rays traverses the kd-tree depth-first to determine the triangles (objects) that it intersects with. The algorithm proceeds in multiple phases, with each phase generating a set of reflected and refracted rays, which in turn traverse the tree in subsequent phases. The algorithm terminates when there are no reflected or refracted rays. Our implementation is adapted from HeatRay[2].

6) ***Frequent Item set Mining*** [6] is a data mining kernel typically employed in mining associations and correlations (e.g. finding correlation in the shopping behavior of customers in a supermarket setting). The input is a set of transactions, T, each containing a subset of items from the set B, which contains all the items that are available for purchase. Another input, $supp_{min}$, quantifies the term "frequent". The goal is to return all sets of items, I, that are "frequent" i.e. $\forall b_i \in I, |C(b_i)| \geq supp_{min}$. Where, the cover of b_i, $C(b_i) = T_i | (T_i \in T, b_i \in T_i)$

A naïve algorithm of generating all possible item sets ($candidates$) and then scanning the transaction set, T, is infeasible since it requires generation of $2^{|B|}$ item sets, which is impractical for large $|B|$ values, and T is often too large to fit in memory. Hence, modified prefix trees [6] are used to compactly represent T in memory. The tree is then systematically traversed in a bottom-up manner to generate the sets of all frequent items occurring in some combination in any transaction. This process involves generating additional *conditional* prefix-trees that are iteratively traversed. The FIM implementation in Treelogy is adapted from FPGrowth[3].

7) ***Fast Multipole Method*** [2] is an efficient algorithm to speed up the computation of particle interaction forces in an *n-body problem* e.g. computing potential of every particle in a system of charged particles. The spatial coordinates of particles are organized into an octree or quad-tree. The tree is then traversed in three steps (top-down, breadth-first in first step, bottom-up, depth-first in second and third steps) to compute the potential of all particles. We characterize the performance of top-down (TD) and bottom-up (BU) traversals separately in section V. Our quad-tree FMM implementation is based on "low-rank approximation of well-separated regions" [32].

8) ***K-means Clustering*** [7] is a popular cluster analysis method in data mining. It partitions a large number of data points into K different clusters. The algorithm works in an iterative way. First, every point computes the distance to these K clusters, and is assigned to the closest one. Then every cluster is updated to be the average position of the points that belong to it. This process repeats until all the points belong to the same clusters in two successive iterations.

While most implementations of K-Means are non-tree based, Treelogy implements a kd-tree based version of the algorithm [7]. For each iteration, a spatial tree is build to organize the set of clusters. Then the point traverses the tree to look for its nearest cluster. Because of the tree structure, a point can quickly filter out far away clusters to avoid unnecessary distance computations. The tree-based K-Means usually presents a better performance [7]. Our implementation is adapted from KdKmeans[4].

9) ***Longest Common Prefix*** [8] or longest common substring (LCS) problem is common kernel in bioinformatics and document retrieval. LCP finds the the longest string that is a common substring of two (or more) strings. Given two strings with length N and M, while the dynamic programming method typically takes O(N*M) time, a suffix tree-based traversal can solve the LCP problem in O(N+M) time. The longest common prefix of both strings can be found by building a generalized suffix-tree, and finding the deepest internal nodes that contain substrings from both input strings. Our implementation is adapted from Longest Common Substring[5].

IV. An Ontology for Tree Traversals

This section introduces an *ontology* for kernels in tree applications, identifying five key features that help categorize tree traversal kernels. We then explain how the categorization of a kernel according to these features can help direct which optimizations apply to a given kernel.

[2]https://github.com/galdar496/heatray/

[3]https://github.com/integeruser/FP-growth/

[4]https://github.com/vaivaswatha/kdkmeans-cuda

[5]http://www.geeksforgeeks.org/generalized-suffix-tree-1/

A. Ontology

1) **Top-down and Bottom-up:** Top-down traversals perform a traversal of the tree beginning from the root. Bottom-up traversals traverse the tree from leaves to the root. Top-down traversals can be preorder, inorder or postorder. There is a correspondence between postorder and bottom-up traversals in an algorithmic implementation: bottom-up traversals can be implemented as postorder. However, some optimizations only apply to top-down traversals [14], [33], while bottom-up traversals can avoid some of the recursion overhead of postorder traversals, making the choice of one or the other significant depending on the situation. In our experience, most tree traversal kernels are top-down.
2) **Unguided and Guided:** A top-down recursive traversal of a tree, whether preorder or postorder, effectively linearizes the tree. In many applications, every traversal of the tree creates "compatible" linearizations: while a single traversal may not visit the whole tree (due to truncation), there exists a single linearization of the tree where every traversal's linearization is a subset of that canonical linearization. For example, in a top-down traversal where every traversal visits a node's left child before visiting its right, all traversals have compatible linearizations. In other applications, though, one traversal might visit the left child before the right while another might visit the right child before the left. In this case, the two linearizations don't match: they are not subsequences of the same canonical linearization. Borrowing terminology from Goldfarb et al. [15], we call the former type of traversal *unguided* and the latter *guided*.
   Crucially, in an unguided traversal, the traversal order of the tree is not traversal dependent. In a guided traversal, however, not only is the traversal order of the tree dependent on properties of a given traversal, that order *could change based on computations performed during the traversal*. Figure 3(b) shows an unguided traversal: any point in 2-point correlation will traverse the tree in the same order. Figure 3(c) shows a guided traversal: which order a point traverses the tree depends on the results of `ComputeVertex`.
3) **Type of tree:** While the traversal kernels of Treelogy can be implemented with a variety of spatial and n-fix trees, this paper presents efficient implementations using oct-, k-dimensional (kd), quad-, ball-(bt), vantage-point(vp), suffix-, and prefix-trees. Various considerations, including memory usage, structural balance and input characteristics can influence the selection of tree types for an algorithm, and can lead to differing performance. Sometimes the tree type is constrained by the dimensionality of input space: 2-dimensional input in FMM requires a quad-tree while 3-dimensional input is represented with an octree.
4) **Iterative with tree mutation:** In iterative applications, the tree is traversed repeatedly until a terminating condition is satisfied. Traversal kernels in such applications modify the tree structure between successive iterations. However, within an iteration, the tree structure is not modified, thus allowing multiple traversals to execute simultaneously. Tree structure modification affects distributed-memory implementations, where tree building and distribution can consume a significant amount of time.
5) **Iterative with working set mutation:** In traversal kernels of iterative applications, the number of independent traversals executing simultaneously may vary across iterations. This set of independent traversals is referred to as the working set. This varying size of the working set can translate to opportunities in load-balancing, and often in minimizing parallel overheads due to synchronization.

Benchmark	Traversal order	Traversal guidance	Tree type	Tree mutation	Work set mutation
NN_kd	preorder	guided	kd	×	×
NN_vp	preorder	guided	vp	×	×
KNN_kd	preorder	guided	kd	×	×
KNN_bl	preorder	guided	bl	×	×
PC_kd	preorder	unguided	kd	×	×
PC_vp	preorder	guided	vp	×	×
BH_oct	preorder	unguided	oct	✓	×
BH_kd	preorder	unguided	kd	✓	×
PM_kd	preorder	unguided	kd	×	✓
FIM_pre	bottom-up	unguided	prefix	✓	✓
KC_kd	inorder	guided	kd	✓	×
LCP_suf	postorder	unguided	suffix	✓	×
FMM_qd	preorder and bottom-up	unguided	quad	✓	×

TABLE I: Classification of benchmarks

Ontology applied to Treelogy

The benchmark programs in Treelogy span the ontology: for each attribute type, Treelogy includes at least two benchmarks covering each possible value for that attribute (with the exception of tree type, as tree type is often independent of algorithm). Table I shows the benchmarks classified according to our ontology. Note that most of Treelogy's algorithms are top-down traversals of various kinds. This is consistent with our experience that most tree-traversal kernels are top-down.

B. Optimizations

Over the years, researchers have proposed numerous optimizations for tree traversal kernels. This section presents specific optimizations, and elucidates how the ontological characterization of a tree traversal can be used to determine whether an optimization is generalizable.

Locality: Traversals through the top part of the tree represent a negligible amount of work compared to traversals through the bottom part. In many top-down traversal kernels, the behavior of traversals in the top part of the tree can provide insight into behavior in the bottom part. Hence, by profiling traversals in the top part, better temporal locality can be achieved through optimized scheduling of traversals through the bottom part [14], [33]. Such profiling is ineffective in case of post-order or bottom-up traversals due to large volume of profiling data. For bottom-up traversals, a tiling optimization such as that in cache-conscious prefix trees [23] can be effective in enhancing spatial locality.

Vectorization: An optimization for Barnes-Hut [34] that is used in vectorized implementations involves a pre-processing step which linearizes traversals before doing any vertex computation. Similarly, many GPU implementations of tree traversals rely on pre-computing *interaction lists*: the tree is traversed to determine which nodes of the tree are touched by each traversal, and only afterwards are the actual computations performed [12]. These types of optimizations can only be applied for unguided traversals: they require either a single linearization of the tree, or that the linearization can be computed without performing the full traversal. These optimizations cannot be applied for guided traversals such as NN, since the next vertex to be visited depends on the result of current vertex computation. In such a scenario, in order to achieve vectorization, the tree structure needs to be modified with other techniques such as autoroping and lockstepping [15].

Input representation: Bottom-up traversal of suffix trees was shown to be efficient in performing multiple genome alignment [35]. Depth-first traversals through prefix trees in FIM beat breadth-first traversals through subset trees [36]. In NN, vp-trees were shown to be the best option compared to alternate input representation options in case of handling queries for finding similar patches in images [11].

In case of spatial trees, traversal properties through one type of tree can be very different when compared to traversals through other types: while vp-trees can facilitate faster truncation of a traversal—hence shorter traversals and faster performance—insertion and deletion is difficult, and hence may not be a good choice for a kernel involving frequent updates to the tree structure. A kd-tree typically offers a height-balanced structure, but at the cost of more vertices, while an octree represents the data more compactly but is not usually balanced. As we see in section V, NN, PC, and BH implementations with certain types of trees yield faster traversal performance for certain inputs.

Other optimizations: Locally essential trees (LET) [10] is a BH-specific optimization. When the input domain of cosmological bodies is decomposed and represented as a tree structure, nearby bodies in a spatial subdomain see (and traverse) a similar subtree structure because the fine- and coarse-grained interactions of these bodies with bodies in other subdomains are similar. In order to compute the total force acting on each body in that subdomain, the union of all such subtrees is needed. This union is referred to as an LET. Essentially, by replicating portions of the tree, the tree structure necessary to compute forces is made available locally to a processor's domain (in a distributed-computing scenario).

Distributed-memory implementations of kernels involving repeated, top-down traversals benefit from the generalization of this optimization: the top subtree (necessary for a subset of point-cell interactions) can be replicated on node-local memory to achieve scalability and minimize inter-node communication.

Estimating returns of optimizations: We provide a simple reuse distance analyzer tool [37] to estimate the efficacy of locality optimizations in tree traversals. Reuse distance is known to be a good predictor of cache performance, despite its many simplifying assumptions. Our tool analyzes locality at the granularity of vertices of a tree, so does not provide complete reuse distance analysis but an analysis tailored to our suite. Reuse distance analysis can help provide architecture-independent measures of locality: small reuse distances mean good locality, large reuse distances mean poor locality but, importantly, an opportunity to potentially improve locality. Hence, the analyzer helps us understand the potential for optimization in a benchmark and/or input. Section V provides a case study of using the tool to verify the effectiveness of locality optimization.

An ontological characterization of tree traversals provides better guidance to design code transformations and optimizations focusing on performance. We have presented a rigorous classification, well known classes of optimizations and a methodology to estimate the returns of a particular class of optimization for tree traversals. Next to support our design, we provide an evaluation of our benchmark suite in different aspects.

V. Evaluation

We now present the evaluation of our benchmark kernels.

A. Methodology

We evaluate the following variants of tree traversal kernel implementations:

1) **SHM**: shared-memory implementation that is run on a single compute node.
2) **DM**: distributed-memory implementation that is run on a cluster of nodes, using the approach of Hegde et al. [38].
3) **GPU**: GPU implementation, using the approach of Liu et al. [33].

Since the goal of this study is to characterize traversal kernels, we do not profile the entire application. Hence, the performance measurements show the traversal times only. Especially, for GPU implementations, we only measure the computation kernel runtime spent on GPU. All our baselines are single-threaded *SHM* versions that optimize scheduling of multiple independent traversals by executing *blocks* of traversals simultaneously [29]. All *DM* runs execute a data-parallel configuration of the kernel, where the tree is replicated on each node and the set of independent traversals are partitioned among the nodes of the cluster. Every configuration of a test is run until a steady state is achieved, which yields errors of 1.2% of the mean with 95% confidence.

In addition to the optimized variants we evaluate here, the Treelogy distribution includes unoptimized, single-threaded implementations for use as optimization targets.

Data sets: We evaluate each benchmark on both real-world[6] and synthetic inputs. We use a publicly available tool for generating FIM data[7], and also provide synthetic data generators for each

[6]Source: https://www.ncbi.nlm.nih.gov/genome/viruses/, UCI Machine Learning Repository

[7]FIM: https://sourceforge.net/projects/ibmquestdatagen/

Input	Description	Benchmark		
		Name	$\|V\|$	$\|P\|$
Synthetic1	Uniformly distributed data in 7-dimensional space ($\mathbb{R}^7$)	PC_kd, NN_kd	2x10^7-1	10^7
		PC_vp, NN_vp	10^7	10^7
Mnist	Handwritten digits data with reduced dimension to $\mathbb{R}^7$	PC_kd, NN_kd, KNN_bt	2x10^6-1	10^6
		PC_vp, NN_vp	10^6	10^6
Plummer	Plummer model with initial position, velocity, and mass	BH_kd BH_oct	2x10^6-1 ≈1.5x10^6	10^6
Synthetic2	Uniformly distributed data, $\mathbb{R}^3$	BH_kd BH_oct	2x10^7-1 ≈15x10^6	10^7
Christmas	Wavefront .obj file	PM_kd	462,818	2.3*10^6
Dragon	Wavefront .obj file	PM_kd	22,532	1.4*10^6
Gazelle2	KDD Cup-2000 data set	FIM_pre	202,234	202,234
Synthetic3	IBM Quest Data Generator	FIM_Pre	23,301	23,301
Synthetic4	Uniformly distributed data, $\mathbb{R}^2$	KC_kd	128	204,800
Wiki	Word vectors from first 10^8 words of wikipedia, $\mathbb{R}^{100}$	KC_kd	128	71,291
Synthetic5	Uniformly distributed data with constant mass and potential, $\mathbb{R}^2$	FMM_qd	≈10^6	≈10^6
Synthetic6	Strings each of size 10^5 construted randomly using an alphabet of 7 characters	LCP_suf	296,007	NA
Genome	Origin of two viral genome sequences from NCBI	LCP_suf	10,132	NA

TABLE II: Data sets and attributes; |V|= Number of vertices, |P|=Number of traversals.

Benchmark	Input	Baseline time (s)			Average traversal length	C_v	L3 miss rate (%)	CPI
		SHM	DM	GPU				
PC_kd	Syn1	926.4	1066.1	29.5	13790	0.13	65.15	1.82
	Mni	79.5	80.2	2.6	2606	0.38	14.74	0.75
PC_vp	Syn1	553.8	422.6	199.0	296	0.14	70.68	3.84
	Mni	2.7	3.2	0.8	40	0.001	40.21	1.67
NN_kd	Syn1	1250.4	1580.9	60.2	3143	0.14	48.2	1.39
	Mni	124.3	180.4	19.4	6310	0.31	16.8	0.63
NN_vp	Syn1	2815.5	4512.8	280.0	3234	0.21	58.9	2.66
	Mni	87.1	107.5	8.6	2657	0.26	13.6	1.01
KNN_bl	Syn1	2424.1	3284.3	106.8	3808	0.74	39.81	1.3
	Mni	145.6	237.2	4.3	3955	1.74	15.3	0.82
BH_oct	Syn2	841.08	2200.7	1.4	1403/step	0.11	63.6	4.32
	Plu	57.86	149.9	2.9	2709/step	5.86	35.55	1.59
BH_kd	Syn2	278.4	953.7	0.3	818/step	0.1	47.52	1.88
	Plu	62.1	169.68	1.3	7704/step	1.92	21.1	0.64
PM_kd	Chr	30.4	49.3	NA	93/step	0.78	42.92	2.07
	Dra	5.1	9.54	NA	86/step	0.68	4.3	1.19
FIM_pre	Gaz	3.9	4.0	NA	2	7.3	2.37	1.17
	Syn3	23.9	24.1	NA	2	1.5	19.4	1.48
KC_kd	Syn4	25.3	16.9	3.7	11/step	0.09	6.42	0.67
	Wiki	98.03	91.9	41.5	128/step	0.02	21.6	0.58
LCP_suf	Gen	0.05	0.02	NA	10,132	NA	17.17	0.84
	Syn6	1.27	0.63	NA	296,007	NA	58.24	1.01
FMM_qd	Syn5 TD	5.7	5.3	0.03	64/step	0.19	0.7	0.46
	BU	0.6	0.2	NA				

TABLE III: Runtime characteristics of benchmarks.

benchmark that allow users to create inputs of the desired size and dimensionality, allowing them to evaluate the behavior of implementations at different input sizes. Table II shows the details of the inputs used.

1) Platform and Program Development Environment

A single compute node consists of 20 Xeon-E5-2660 cores with hyperthreading and runs RHE Linux release 6. Each core has 32KB of L1 data and instruction cache, and 256KB of L2 cache. The cores share 25MB of L3 cache, and 64GB of RAM. The *DM* experiments are run on a cluster of 10 such compute nodes. *DM* implementations are compiled using mpic++—a wrapper compiler over gcc 4.4.7—and linked with Boost Graph Library (BGL 1.55.0) and MPI libraries (MPICH2 1.4.1p1). *SHM* implementations are compiled using gcc 4.4.7, and linked with $pthread$ libraries. GPU implementations are compiled with NVCC 7.0.27, and evaluated on a server with 32 GB physical memory, two AMD 6164 HE processors and a nVidia Tesla K20C GPU card (5GB memory on board). The K20C deploys 13 Streaming Multiprocessor (SMX) and each SMX contains 192 single-precision CUDA cores.

2) Metrics

We first characterize the Treelogy benchmarks according to various architecture dependent and independent metrics. Table III shows the results.

Average traversal length: For each benchmark/input pair, we compute the *average traversal length* of the traversals in the kernel. The length of a traversal is measured as the number of vertices that the traversal touches in the tree (note that we count each vertex just once, even if a traversal performs work at a vertex during both pre-order and post-order portion of the algorithm). Longer traversal lengths means that more of the tree is being touched, and that traversals have larger working sets. This has implications for locality (larger working sets means more likelihood of cache misses) and for scheduling (longer traversals, especially in relation to the size of the tree, means that there is more likelihood that traversals overlap and hence can be scheduled together for better locality or smaller divergence).

The L3-cache miss rate results measured on the baseline SHM performance in table III reflect this: we see higher miss rates with longer traversal lengths and larger inputs. However, PC_vp in comparison with PC_kd is a contradiction; despite the traversals through vptree being much shorter (for the same input), we see an increase in miss rate. This is because of the guided traversal kernel of PC_vp offering more intra-block parallelism compared to the unguided, PC_kd kernel, which means that for the same block of input points, a larger portion of the tree is touched in case of PC_vp. All other cases of lower miss rates are due to very shorter traversal lengths (PM, FIM, FMM, KC) and smaller tree sizes (KC, PM_Dra).

The following observations from table III emphasize how input-dependent the kernels are: a) while kd-tree offers the best traversal performance for Syn1, vptree offers the best performance for Mni for NN benchmark. From another perspective, traversals through the kd-tree are the shortest for Syn1 but longest in case of Mni when compared to vptree and ball-tree traversals. b) we also see that while every traversal in KC_kd touched the entire tree for Wiki, only 11 vertices, on average, were touched in Syn4. In summary, the behavior of these benchmarks is highly dependent on the tree type and even input distribution.

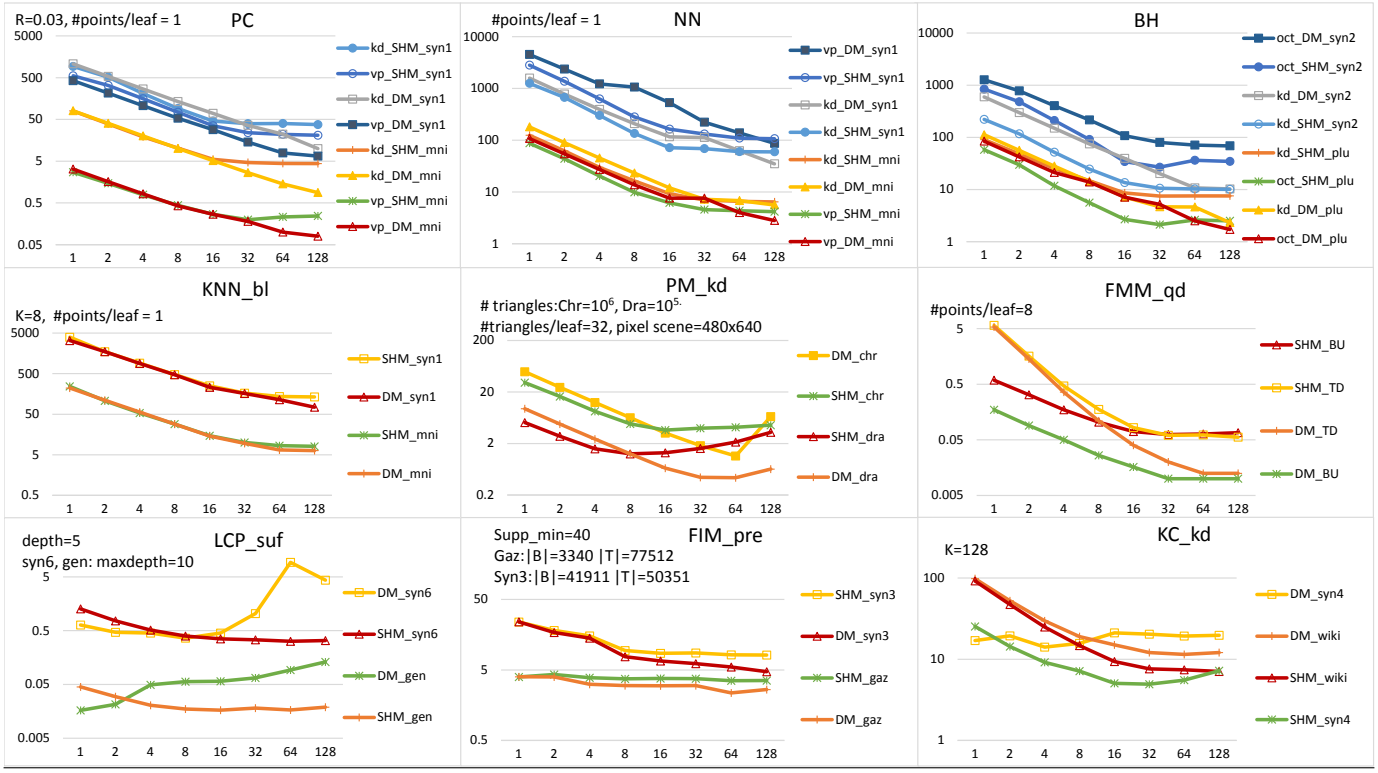


Fig. 4: Scaling in Treelogy Benchmarks. x-axis=Thread count (SHM)/Process count (DM), y-axis (log scale) =Runtime (s).

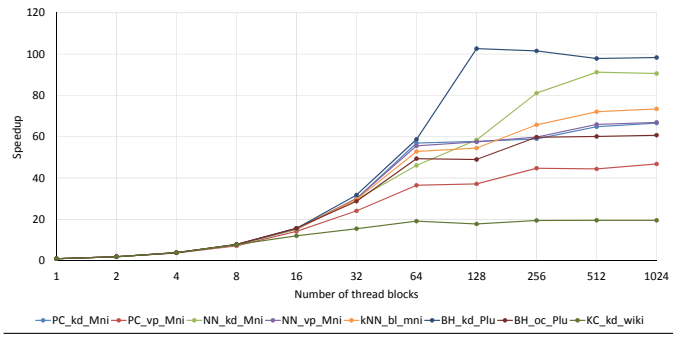


Fig. 5: GPU scalability

Load-distribution: To determine how the work of a traversal algorithm is distributed across the tree (useful for distributed memory load balancing), we measure *load-distribution*. We partition the tree into subtrees such that the resultant tree is two subtrees deep (i.e. the tree is logically sliced at half the maximum depth). The amount of work done by a subtree is the load, which is roughly equivalent to the sum of the size of all blocks executed at all vertices of the subtree. Since the lone top subtree has the maximum load because of all traversals beginning and ending at the root vertex, we skip this subtree from our analysis to get a clearer picture of load on other subtrees. For measuring load-imbalance, we calculate $C_v = \sigma/\mu$, where μ is the average load on a subtree and σ is the standard deviation. C_v, the coefficient of variation, indicates the presence of subtrees that are heavily loaded. C_v value for LCP is not measured as this is a single postorder traversal kernel. While a very small value of C_v indicates uniform load, a large value indicates the presence of bottleneck subtrees. Load distribution matters for implementations of traversal benchmarks that distribute the trees [13], [38], since different sub-trees may have different computational loads, necessitating load balance strategies such as replication of bottleneck subtrees.

We expect that the overall tree structure and the input distribution influence the load on a subtree. The C_v values in table III reflect this. The higher values of C_v for BH with plu input is because of the clustered nature of the data set. In case of KNN, PM and FIM, the higher value is mainly because of the resulting fragmented tree struture. The fragmentation is especially severe in FIM because the tree is typically extremely wide (e.g. >25k vertices at level 4 for Gaz).

B. Scalability

Figure 4 shows the strong scaling results of traversal kernels. As expected, *DM* scales better compared to *SHM* for all the benchmarks except KC and LCP. This is because in *DM* runs, as more compute nodes are added, more hardware execution contexts become available and hence, the nodes are able to utilize all of the available parallelism with minimal inter-node synchronization, thus yielding better scalability. However, scaling in *SHM* stops from 64 threads due to sharing of processing elements (cores) beyond 40 threads.

In case of KC, all the threads/processes synchronize at the end of every iteration to reconstruct the tree. Therefore, we see

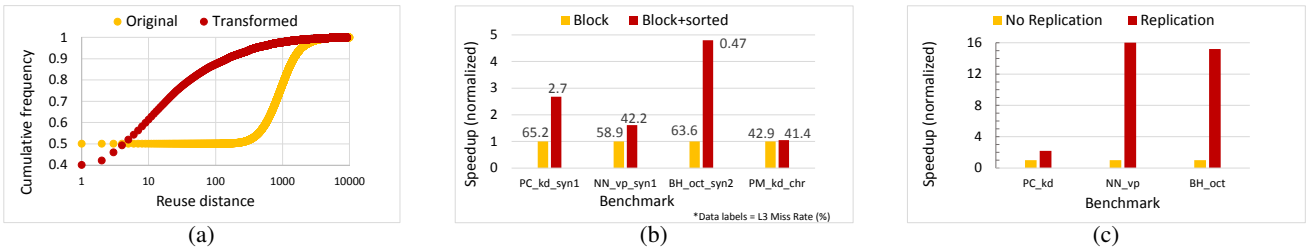


Fig. 6: Case studies: a) estimating locality benefits using reuse distance. b) improving locality through reordered traversal schedule. c) improving load-balance through subtree replication.

degraded performance of *SHM* versions beyond 32 processes (this behavior is observed in case of multiple-step runs of other iterative kernels such as BH, PM as well). Also, due to the increased overhead of inter-process synchronization, *DM* runs perform worse than *SHM* versions. Scaling is poor in case of LCP (both *SHM* and *DM*) because of low available parallelism and the extremely small traversal kernel: there is just a single post-order traversal to begin with and the child nodes of a vertex can be processed in any order before processing a parent vertex. When the traversal reaches a certain $depth$, specified as a tunable parameter, the child vertices are processed in parallel. We observe that even with the synthetic data of sufficiently long input strings, there are not enough child vertices to be processed in parallel. Hence, the parallel overhead due to large number of threads/processes brings down the performance.

We also see a super-linear speedup for the top-down traversals of FMM. This is because of a great reduction in the lower-level cache accesses (e.g. on moving from 1 thread to 4 threads in *SHM* version, total L2 cache accesses reduced by 97.8% and L3 accesses reduced by 62.3% resulting in a speedup of 12.4x). As seen from table III, the top-down traversal step in FMM was nearly 9.5 times slower than the two bottom-up steps combined.

DM configurations mapped multiple processes to a single node of the cluster when a free node was not available. This, along with the data-parallel execution (entire tree replication on every process) is the cause for reduced performance at large numbers of processes and large input sizes. The *DM* run for PM with chr input (DM_chr) failed to execute with 128 processes due to memory limitations. Hence, we ran this configuration with the tree partitioned across all the nodes of the cluster and replicating only the top subtree (this is the pipelining strategy outlined by Hegde et al. [38]). As a result, due to the added communication overhead, we see the performance going down w.r.t. 64 process data-parallel run.

The GPU platform has the similar hardware structure as the *SHM*: all threads share the same limited hardware. Thus the scalability test result is also close to *SHM* (using real-world inputs). We assign each thread block 192 threads (6 warps) to take full use of a SMX and evaluate the scalability on the granularity of thread block. In Figure 5, before the number of thread blocks reaches 64[8], all benchmarks scale well and show a linear speedup as the blocks increase. For most benchmarks, scaling stops after 64 thread blocks, as the GPU resources are exhausted.

In summary, we find that these traversal kernels scale well, when taking advantage of ontology-driven optimizations. While the performance of BH with octree is better with clustered input, BH with kd-tree performs better with uniformly distributed data. However, BH with octree is known to perform better at larger scales [13]. We also find that PC with vptrees attractive compared to PC with kd-trees because of vptrees facilitating efficient truncation, resulting in shorter average traversal lengths and hence better performance.

C. Case studies

Figure 6(a) shows an example of using dual-trees and recursion twisting optimizations [39] to improve locality in PC with the help of our reuse distance analyzer tool. In this experiment, we evaluate PC_kd with a subset of mnist data set, since the average traversal length with mnist is the longest (see table III). The cumulative frequency of bigger valued reuse distances is higher for the transformed code than the original code. Hence, we can infer that the transformation has resulted in better temporal locality.

Figure 6(b) shows the effect of application-specific sorting optimization on temporal locality and the resulting improvement in performance in top-down kernels with independent traversals. These results have been published earlier by Jo et.al. [29] but are presented here for completeness. The $Block$ configuration represents the *SHM* runtime values of table III. The $Block + sorted$ configuration represents a reordered schedule of traversals in $Block$ to obtain improved locality. These experiments are run with a block size of 4096. As expected, the performance improvement due to sorting is reflected in a commensurate decrease in cache miss rate. Note that $Block$ and $sorted$ are independent optimizations and can be applied to bottom-up kernels with independent traversals as well. In fact, $Block + sorted$ is equivalent to the tiling optimization in FIM by Ghoting et.al. [23].

Figure 6(c) shows the effect of top subtree replication as a way of generalizing the LET optimization. These experiments are run with 128 processes and the tree is partitioned into subtrees of fixed height because the entire tree could be replicated. For these inputs, $|V|$ ranged from 80x10^6 to 256x10^6, and $|P|$ was 64x10^6. As expected, the top subtree

[8]64 blocks contain 64*192 = 12,288 threads

presents a bottleneck when a large number of independent top-down traversals exist, and the node housing this subtree remains heavily loaded. Besides balancing the load, top subtree replication reduces communication overhead and hence, results in far better performance as seen from the figure.

VI. Related Work

There are a wide variety of benchmark suites that focus on graph applications [16]–[18], [40], and several of these suites contain a handful of Treelogy benchmarks, such as Barnes-Hut, Frequent Itemset Mining, K-Means, and Fast Multipole. However, because these suites focus on a broader class of graph algorithms, they do not feature enough tree traversal algorithms to cover the wide variety of traversal structures that arise in such kernels. Graph-based, irregular application kernels have also been studied extensively in other works [19]–[21], [41]. However, analysis of tree based applications is not as extensive. Treelogy provides a way of characterizing the structural properties of tree algorithms, and ensures that its benchmarks span this space. Moreover, Treelogy provides reference implementations of these algorithms on multiple platforms, instead of focusing on single platforms as many prior suites do.

Irregular, tree-based applications have been mapped to massively parallel and distributed hardware platforms [3], [13], [15], [42], [43]. However, customized problem representations and transformations for applications have been proposed, analyzed, and optimized in a stand-alone context. While the parallelism profile of Barnes-Hut is presented in Lonestar [18], traversal properties are not studied. Performance of Nearest Neighbor search with multiple trees has been studied from a purely query response time perspective [11]. Treelogy differs from these works as it considers a broader class of tree traversal kernels, and multiple hardware platforms and tree type combinations. Treelogy can be used to inform generalizations of the many application-specific optimization strategies that prior implementations use.

VII. Conclusions

In this work, we introduced an ontology for tree traversal kernels and presented a suite of tree traversals, Treelogy, with at least one kernel in each category. We evaluated multiple implementations of these traversal kernels using different types of trees and on multiple hardware platforms. We presented scalability analysis on different platforms, and an analysis of locality and load-balance with the help of metrics. We also presented case studies showing the effectiveness of certain optimizations. While the interest in tree based kernels comprising of traversals from various domains is increasing, we hope to expand the suite with more kernels and analyze their performance. Treelogy, by providing a wide variety of tree traversals, presents a useful target for developing and evaluating optimizations, and its ontology helps understand where and when those optimizations can be applied.

Acknowledgment

The authors would like to thank the anonymous referees for their comments. The authors would also like to thank Jad Hbeika for FMM related discussions and the GPU code. This work was supported in part by an NSF CAREER award (CCF-1150013) and a DOE Early Career award (DE-SC0010295).

References

[1] J. Barnes and P. Hut, "A hierarchical o(n log n) force-calculation algorithm," *nature*, vol. 324, p. 4, 1986.

[2] V. Rokhlin, "Rapid solution of integral equations of classical potential theory," *Journal of Computational Physics*, vol. 60, no. 2, pp. 187–207, 1985.

[3] T. Foley and J. Sugerman, "Kd-tree acceleration structures for a gpu raytracer," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, ser. HWWS '05, 2005, pp. 15–22.

[4] A. G. Gray and A. Moore, "N-body problems in statistical learning," in *Advances in Neural Information Processing Systems (NIPS) 13*, December 2001, pp. 521–527.

[5] P. N. Yianilos, "Data structures and algorithms for nearest neighbor search in general metric spaces," in *SODA*, vol. 93, no. 194, 1993, pp. 311–321.

[6] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," in *ACM Sigmod Record*, vol. 29, no. 2. ACM, 2000, pp. 1–12.

[7] K. Alsabti, S. Ranka, and V. Singh, "An efficient k-means clustering algorithm," 1997.

[8] K. Toru, L. Gunho, A. Hiroki, A. Setsuo, and P. Kunsoo, "Linear-time longest-common-prefix computation in suffix arrays and its applications," in *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching*, A. Amir, Ed. Springer-Verlag London, UK, 07 2001, pp. 181–192.

[9] X. Zhang and A. A. Chien, "Dynamic pointer alignment: Tiling and communication optimizations for parallel pointer-based computations," in *ACM SIGPLAN Notices*, vol. 32, no. 7. ACM, 1997, pp. 37–47.

[10] M. S. Warren and J. K. Salmon, "Astrophysical n-body simulations using hierarchical tree data structures," in *Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*. IEEE Computer Society Press, 1992, pp. 570–576.

[11] N. Kumar, L. Zhang, and S. Nayar, "What is a good nearest neighbors algorithm for finding similar patches in images?" in *Computer Vision–ECCV 2008*. Springer, 2008, pp. 364–378.

[12] T. Hamada, K. Nitadori, K. Benkrid, Y. Ohno, G. Morimoto, T. Masada, Y. Shibata, K. Oguri, and M. Taiji, "A novel multiple-walk parallel algorithm for the barnes–hut treecode on gpus – towards cost effective, high performance n-body simulation," *Computer Science - Research and Development*, vol. 24, no. 1, pp. 21–31, 2009.

[13] P. Jetley, F. Gioachin, C. Mendes, L. V. Kale, and T. Quinn, "Massively parallel cosmological simulations with changa," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, 2008, pp. 1–12.

[14] Y. Jo and M. Kulkarni, "Automatically enhancing locality for tree traversals with traversal splicing," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, ser. OOPSLA '12. New York, NY, USA: ACM, 2012, pp. 355–374.

[15] M. Goldfarb, Y. Jo, and M. Kulkarni, "General transformations for gpu execution of tree traversals," in *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 10:1–10:12.

[16] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," in *ACM SIGARCH computer architecture news*, vol. 23, no. 2. ACM, 1995, pp. 24–36.

[17] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. IEEE, 2009, pp. 44–54.

[18] M. Kulkarni, M. Burtscher, C. Casçaval, and K. Pingali, "Lonestar: A suite of parallel irregular programs," in *Performance Analysis of Systems*

and Software, 2009. ISPASS 2009. IEEE International Symposium on. IEEE, 2009, pp. 65–76.

[19] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," *Cray User?s Group (CUG)*, 2010.

[20] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on gpus," in *Workload Characterization (IISWC), 2012 IEEE International Symposium on*. IEEE, 2012, pp. 141–151.

[21] M. Ahmad, F. Hijaz, Q. Shi, and O. Khan, "Crono: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores," in *Workload Characterization (IISWC), 2015 IEEE International Symposium on*. IEEE, 2015, pp. 44–55.

[22] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo *et al.*, "The tao of parallelism in algorithms," in *ACM Sigplan Notices*, vol. 46, no. 6. ACM, 2011, pp. 12–25.

[23] A. Ghoting, G. Buehrer, S. Parthasarathy, D. Kim, A. Nguyen, Y.-K. Chen, and P. Dubey, "Cache-conscious frequent pattern mining on modern and emerging processors," *The VLDB Journal*, vol. 16, no. 1, pp. 77–96, 2007.

[24] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, pp. 509–517, September 1975.

[25] H. Fuchs, Z. M. Kedem, and B. F. Naylor, "On visible surface generation by a priori tree structures," in *ACM Siggraph Computer Graphics*, vol. 14, no. 3. ACM, 1980, pp. 124–133.

[26] D. J. Meagher, *Octree encoding: A new technique for the representation, manipulation and display of arbitrary 3-d objects by computer*. Electrical and Systems Engineering Department Rensseiaer Polytechnic Institute Image Processing Laboratory, 1980.

[27] O. STEPHEN, M., "Five balltree construction algorithms," International Computer Science Institute, Tech. Rep. 89-063, November 1989.

[28] P. Bieganski, J. Riedl, J. V. Cartis, and E. F. Retzel, "Generalized suffix trees for biological sequence data: applications and implementation," in *System Sciences, 1994. Proceedings of the Twenty-Seventh Hawaii International Conference on*, vol. 5, Jan 1994, pp. 35–44.

[29] Y. Jo and M. Kulkarni, "Enhancing locality for recursive traversals of recursive structures," in *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, ser. OOPSLA '11. New York, NY, USA: ACM, 2011, pp. 463–482.

[30] N. Bhatia and Vandana, "Survey of nearest neighbor techniques," in *International Journal of Computer Science and Information Security*, ser. IJCSIS, vol. 8, 2010, pp. 302–305.

[31] M. D. Dikaiakos and J. Stadel, "A performance study of cosmological simulations on message-passing and shared-memory multiprocessors," in *Proceedings of the 10th international conference on Supercomputing*. ACM, 1996, pp. 94–101.

[32] L. Ying, "A pedestrian introduction to fast multipole methods," *Science China Mathematics*, vol. 55, no. 5, pp. 1043–1051, 2012.

[33] J. Liu, N. Hegde, and M. Kulkarni, "Hybrid cpu-gpu scheduling and execution of tree traversals," in *Proceedings of the 2016 International Conference on Supercomputing*, ser. ICS '16. New York, NY, USA: ACM, 2016.

[34] J. Makino, "Vectorization of a treecode," *Journal of Computational Physics*, vol. 87, no. 1, pp. 148–160, 1990.

[35] M. Höhl, S. Kurtz, and E. Ohlebusch, "Efficient multiple genome alignment," *Bioinformatics*, vol. 18, no. suppl 1, pp. S312–S320, 2002.

[36] C. Borgelt, "Frequent item set mining," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 2, no. 6, pp. 437–456, 2012.

[37] C. Ding and Y. Zhong, "Predicting whole-program locality through reuse distance analysis," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, ser. PLDI '03. ACM, 2003, pp. 245–257.

[38] N. Hegde, J. Liu, and M. Kulkarni, "Spirit: A runtime system for distributed irregular tree applications," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '16. New York, NY, USA: ACM, 2016, pp. 51:1–51:2.

[39] K. Sundararajah, L. Sakka, and M. Kulkarni, "Locality transformations for nested recursive iteration spaces," in *Proceedings of the Twenty Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: ACM, 2017.

[40] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 2008, pp. 72–81.

[41] S.-H. Lim, S. Lee, G. Ganesh, T. C. Brown, and S. R. Sukumar, "Graph processing platforms at scale: Practices and experiences," in *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*. IEEE, 2015, pp. 42–51.

[42] J. Gunther, S. Popov, H.-P. Seidel, and P. Slusallek, "Realtime ray tracing on gpu with bvh-based packet traversal," in *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, ser. RT '07, 2007, pp. 113–118.

[43] M. Burtscher and K. Pingali, "An efficient CUDA implementation of the tree-based barnes hut n-body algorithm," in *GPU Computing Gems Emerald Edition*. Elsevier Inc., 2011, pp. 75–92.