SPIRIT: A Framework for Creating Distributed Recursive Tree Applications

Nikhil Hegde, Jianqiao Liu, Milind Kulkarni

School of Electrical and Computer Engineering, Purdue University
{hegden,liu1274,milind}@purdue.edu

ABSTRACT

An important set of applications, from diverse domains such as cosmological simulations, data mining, and computer graphics, involve repeated, depth-first traversal of trees. As these applications operate over massive data sets, it is often necessary to distribute the trees to process all of the data. In this paper, we introduce SPIRIT, a framework to ease the writing of distributed tree applications. SPIRIT automates the challenging tasks of tree distribution, optimizing communication and parallelizing independent computations. The common algorithmic pattern in tree traversals is exploited to effectively schedule parallel computations and improve locality. As a result, we identify systematic ways of exploiting pipeline parallelism in these applications and show how this parallelism can be complemented by selective application of data parallelism to provide greater speed-ups without requiring excessive data replication. SPIRIT is packaged into a set of application programming interfaces (APIs) that developers can use to create scalable applications. Evaluation of SPIRIT on various tree traversal algorithms shows a scalable system. We also find that SPIRIT implementations perform substantially less communication and achieve significant performance improvements over implementations in other distributed graph systems, and are competitive against state-of-the-art, hand-tuned, application-specific implementations.

CCS CONCEPTS

•Software and its engineering →Application specific development environments; •Computing methodologies →Distributed programming languages; Massively parallel algorithms;

ACM Reference format:

Nikhil Hegde, Jianqiao Liu, Milind Kulkarni. 2017. SPIRIT: A Framework for Creating Distributed Recursive Tree Applications. In *Proceedings of ICS '17, Chicago, IL, USA, June 14-16, 2017*, 11 pages. DOI: <http://dx.doi.org/10.1145/3079079.3079095>

1 INTRODUCTION

A common algorithmic pattern that arises in many applications in social networking, data mining, computer graphics, etc., is *tree traversal*. These applications often consist of repeated traversals of a tree—for example, repeated traversals of an object tree during ray

tracing to determine the color of each eye ray. These applications tend to possess abundant data parallelism: the tree traversals are independent, so each of the millions or even billions of tree traversals can be performed in parallel. As input sizes increase, the data sets can easily become too large to be processed on a single machine.

A natural approach to continuing to parallelize tree applications is to simply process the data across multiple machines: replicate the tree on multiple nodes, then have each node process a portion of the data set. Sadly, this approach does not really work, because the size of the *tree* being traversed typically increases commensurate with the number of traversals. As a result, it is infeasible to have the tree replicated at each node. Instead, the only possible approach is to distribute the tree across multiple nodes. Consequently, it is necessary to restructure the traversal application to handle traversals that can span multiple nodes.

Tree traversals on a distributed tree presents a host of challenges. First, because the tree is no longer replicated across all of the nodes, we cannot rely on data parallelism arising from processing multiple independent traversals in parallel to provide work for all nodes. We must find parallelism from other sources. One such source of parallelism is *pipeline parallelism*, where each traversal is broken into a pipeline of *sub-traversals*, and sub-traversals in different portions of the distributed tree can proceed in parallel, allowing all nodes to perform work even if they do not hold the entire tree.

Second, because individual traversals now span multiple nodes, we must schedule tree traversals to maximize locality while minimizing communication. In contrast to the kinds of applications targeted by most distributed and parallel graph frameworks [8, 10, 15, 17–19], each tree traversal spans large portions of the tree, rather than small regions of a graph. Hence, it is not practical to consider each tree traversal as a single (or even atomic) unit of computation for processing. Instead, traversals must be carefully split up and interleaved to achieve good performance.

Finally, we must ensure that the pipeline does not contain any bottlenecks due to sub-traversals overloading a particular subtree on a node. This requires the ability to identify and replicate bottleneck subtrees on multiple nodes. Because this replication trades off memory usage (more replication consumes more overall memory) for performance (more replication ameliorates more hotspots), an application should be able to tune its replication level to achieve the right balance.

Contributions

In this paper, we introduce SPIRIT, a *space-adaptive, distributed* tree traversal framework. SPIRIT distributes a tree across multiple nodes, and then carefully schedules multiple tree traversals across these subtrees to achieve parallelism, promote locality, and provide load balance. The chief features of SPIRIT are:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICS '17, Chicago, IL, USA

© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM. 978-1-4503-5020-4/17/06...\$15.00

DOI: <http://dx.doi.org/10.1145/3079079.3079095>

- (1) SPIRIT provides *pipeline parallelism* for traversals. It splits traversals into sub-traversals that end at subtree boundaries, allowing sub-traversals that are in different subtrees to execute simultaneously on different nodes.
- (2) SPIRIT carefully schedules the resulting finer-granularity tasks to improve locality: sub-traversals that are accessing the same portion of a subtree will execute in close succession. It also reduces communication: sub-traversals on a node that want to communicate with the same remote node aggregate their messages.
- (3) SPIRIT provides load balance by selectively replicating portions of the tree that are heavily loaded, allowing multiple nodes to take on the responsibility of handling those sub-traversals. This feature also allows SPIRIT to be *space-adaptive*: programmers can tune the amount of replication, providing a balance between pure pipeline parallelism, which requires no data replication, but may have limited parallelism in practice, and pure data parallelism, which has abundant parallelism, but requires complete replication of the tree at each node.
- (4) SPIRIT provides APIs that allows programmers to easily implement their tree traversal algorithms without considering the challenges of distribution or scheduling.

We evaluate SPIRIT on 5 tree traversal benchmarks and show that it achieves both strong and weak scaling across these challenging applications, even when resource limitations prevent fully replicating the tree. We demonstrate the advantages of SPIRIT’s space adaptivity—SPIRIT can smoothly trade off space usage for parallelism, and, when sufficient resources are available, a SPIRIT application is able to perform equivalently to a fully-replicated, data-parallel implementation. Finally, we compare SPIRIT implementations to prior work in two ways: we show that SPIRIT implementations far outperform implementations using existing distributed graph frameworks, and we show that SPIRIT is competitive with, and sometimes better than, existing custom implementations of traversal algorithms.

2 BACKGROUND AND MOTIVATION

In this section, we begin by discussing terminology and tree traversal characteristics common to the benchmark applications. Scheduling and locality opportunities in a tree traversal algorithm and their applicability in a distributed setting are discussed next. Finally, available approaches to distributed graph processing, example frameworks, and the challenges involved in using the frameworks to create distributed tree applications are explained.

2.1 Terminology and Traversal Characteristics

This paper considers distributing and parallelizing *tree traversal* applications. A tree traversal consists of a set (or multiple sets) of depth-first traversals of a tree (the vertices of the tree can have any number of children, though in practice, most trees are binary trees or octrees). *Points* are the entities that traverse the tree. Points may not perform a complete traversal of the tree: a traversal may be truncated at any vertex, leading the point to skip visiting the subtree rooted at that vertex. While the tree may be altered in the overall application execution scenario, during tree traversal, neither

```

1 void TraverseTree(Vertex v, Point p){
2   status = EvaluateVertex(v, p)
3   if(status is truncate) return
4   if(status is traverse_left){
5     TraverseTree(v.leftChild, p)
6     TraverseTree(v.rightChild, p)
7   } else {
8     TraverseTree(v.rightChild, p)
9     TraverseTree(v.leftChild, p)
10  }
}
```

Figure 1: Tree Traversal in Nearest Neighbor.

the tree structure nor the vertex properties are altered (in other words, in applications where the tree does change, the changes happen *between* different sets of traversals being performed on the tree). A *node* is a physical machine that houses a partition of the distributed tree and is involved in some computation, which is primarily traversal of points through the partition.

2.2 Tree traversal algorithms and locality

Figure 1 shows the pseudocode for a nearest-neighbor (NN) algorithm [27] implemented using a kd-tree traversal. The goal is to find closest point(s), based on some distance function, to a query point. First, a set points representing the *feature space* in N dimensions is organized into a kd-tree. The tree is built top-down, recursively partitioning the feature space into two subspaces along one of the N dimensions until a subspace contains a single or some predefined number of points. Each vertex of the tree hence represents a (sub)space of the feature space. The tree is then traversed depth-first to find the closest neighbor(s) of the query point.

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 (line 2). 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.

Locality. Nearest neighbor applications often have multiple query points and the traversal paths of query points have a set of vertices in common with other query points. This means that traversals of multiple query points can be scheduled to access the common set of vertices in close succession to improve temporal locality. As the queries are independent, the traversals can be interleaved, and multiple vertex computations (by multiple points) can be scheduled simultaneously. Point blocking [14] exploited the locality available at this finer granularity of vertex computations in the context of shared-memory systems.

Distributed traversal. A distributed tree traversal must check for the presence of remote vertices—vertices existing on remote nodes. If the vertex is remote, a message must be sent to the remote node for scheduling the *sub-traversal* on the subtree rooted at the remote vertex and the sub-traversal on the current node must be suspended until the exploration of remote subtree is complete. The current node can then schedule another traversal when available. Traversals of multiple points touching the same remote vertex could be aggregated on the current node before sending a single message to

the remote node to schedule their respective sub-traversals. Thus, a *pipeline* of sub-traversals together with aggregation, enable parallel execution of traversals, and reduce communication overhead in addition to improving locality of remote subtree accesses—SPIRIT is designed to leverage this insight.

2.3 Graph processing frameworks

Many graph processing frameworks allow programmers to write kernels that ignore the distribution of the graph, performing all of the necessary distribution and communication automatically and transparently [6, 8, 10, 15, 17–19]. However, these frameworks adopt a *vertex-centric* programming model¹: the kernels that they encourage programmers to write operate on a single vertex and its edges, or, at best, a small region in the graph. Each such kernel is considered a task, and the graph frameworks derive parallelism from processing numerous such tasks in a bulk-synchronous manner. The small size of each task makes it straightforward to schedule for improved locality, reduced communication, task migration for load balancing etc.

For a subset of tree traversal algorithms, the vertex-centric approach is simply not suitable. Traversing an *oct-tree* in Barnes-Hut [3], a *kd-tree* in ray tracing [24] or in computing two-point correlation [9], or a *vp-tree* in finding nearest neighbors [27], is based on depth-first order and hence, there is no parallelism at the vertex level in these traversal algorithms.

Also, the trees are traversed repeatedly without any dependency across traversals. Thus, though there exists plenty of opportunities for parallelism when a sequence of vertex accesses spanning the entire tree is considered, the vertex-centric approach misses this while focusing on an individual vertex or small region of the tree in identifying independent computations. Moreover, a vertex-centric approach can miss opportunities for better locality or reduced communication by exploiting similarities *across* traversals.

Distributed GraphLab (DGL) [17]. DGL adopts a vertex-centric approach to distributed graph processing. Algorithms in DGL are captured in an easy-to-program template of Gather-Apply-Scatter (GAS). A GAS operation on a vertex is essentially a vertex computation that is scheduled for parallel execution by the programmer. In the *gather* phase, results of vertex computations of adjacent vertices are aggregated. The *apply* phase uses the aggregated results to perform some computation at the current vertex. Based on the results of the *apply* phase, the *scatter* phase updates adjacent edges, thereby scheduling new vertex computations on adjacent vertices in the next iterative step. At the end of an iterative step, all the vertices are checked for a change in state to determine convergence. DFS traversals are inherently unsuitable to the GAS model of execution as they have low vertex-level parallelism.

Parallel BGL (PBGL) [10]. PBGL provides a more flexible abstraction to graph distribution and traversal: built-in skeletons are provided for creation and traversal of distributed graphs in addition to the data structures necessary for their manual creation. The traversal abstractions (*visitors*) of PBGL are primarily focused on distributing and parallelizing a *single* traversal (e.g., by processing

Figure 2: Tree partitioning and distribution in SPIRIT.

multiple vertices in a breadth-first search simultaneously). Unfortunately, in our tree traversal algorithms, traversals are depth-first, and hence have no opportunity for intra-traversal parallelism. Moreover, PBGL cannot exploit opportunities for *inter-traversal* locality, message aggregation, and load balance that are afforded by the execution of multiple traversals of a single tree.

Thus, to exploit new parallelism opportunities, to schedule traversals for optimizing communication and locality, and to address the challenges in distributed tree creation, we developed SPIRIT, a framework that exploits inter-traversal behaviors to improve performance and reduce communication. SPIRIT provides a *traversal-centric* set of abstractions for writing distributed tree applications, as described in the following sections.

3 SPIRIT

This section describes the design of SPIRIT, a framework for writing distributed, space-adaptive tree applications.

3.1 Tree partitioning and distribution

To distribute computation across multiple nodes, SPIRIT *partitions* the tree into subtrees, with each node assigned one or more subtrees. Figure 2 shows a tree partitioned into three subtrees of height, *SubtreeHeight*. The subtrees are labeled 1, 2, and 3, and created on different nodes. (Subtrees 4 and 5 are replicas of 1 and are explained in Section 3.4; for now, we ignore them). Each subtree’s root, except that of subtree 1, is called a *pseudo-root* (to distinguish them from the true root of the tree). Similarly, vertices whose children belong to a different subtree are called *pseudo-leaves* (and note that those children are, by definition, pseudo-roots).

SPIRIT includes tree building algorithms for three types of spatial trees: kd-trees, octrees, and vp-trees. Programmers can also implement their own tree building algorithms. Figure 3 shows the pseudocode of octree building.

Initially, all the processes collectively determine the bounding box of a 3-dimensional particle space and its center. The root vertex of the tree represents this box. Each particle is then inserted into this box and a vertex may be split into multiple children if the box contains more particles than a user-specified limit. The split results in creation of one or more child vertices corresponding to octant(s) within the enclosing box. If a child is a pseudo-leaf and needs to be split, the tree building procedure halts if the subtree of that pseudo-leaf is remotely owned—*GetSubtreeOwnerID* returns owner ID of the octants of the enclosing box. Every process now

¹Parallel BGL [10] and STAPL [6] support coarse-grained computation model in addition to the vertex-centric model.

```

1 BuildOctree(char* input, InputParser* parser){
2   <points, numRead>=DistributeParticles(input, parser)
3   <center, dia>=GetBoundingBox(points, numRead)
4   //Create root vertex and set bounding box
5   <points,numRead> = LoadParticles(input, parser)
6   while(numRead < totalPoints) {
7     for each p in points
8       BuildSubTree(root,p,center,dia,depth)
9   <points,numRead>= ReadData(input, BATCH)
10  }

12  for each pseudoleaf whose subtree is local
13    computeCofm(pseudoleaf)
14  //inform local pseudoleaf info to all processes.
15  for each pseudoleaf
16    all-to-all(pseudoleaf-cofm)
17  //Compute cofm of all vertices in top subtree.
18  ComputeCofm(rootVertex)
19 }

```

Figure 3: Distributed octree building.

has a top subtree of depth *SubtreeHeight*, connected to a set of local subtrees (in dark) as the tree structure in figure 3 shows.

An all-to-all communication of center-of-masses of local pseudo-leaves is performed so that every process has the same vertex data in all the pseudo leaves. The center-of-mass of the vertices in the top subtree is computed bottom up starting from a pseudo-leaf. Kd-tree building follows similar procedure except for the absence of the all-to-all communication. The bounding box and other vertex data needed are updated during particle insertion.

For large data sets resulting in a tree deeper than $2 \times \text{SubtreeHeight}$ levels, a fixed *SubtreeHeight* can result in over-fragmentation. SPIRIT avoids this by allowing variable height bottom subtrees. The structure of the octree in Barnes-Hut may change every time step based on the results of traversal from the previous time step. Currently, SPIRIT does not have optimizations in place to re-use parts of the tree that do not require restructuring; entire tree is rebuilt at the beginning of each step.

3.2 Tree traversals and pipeline parallelism

For repeated, independent, depth-first traversals of a tree, shifting focus from a point's vertex access to its entire traversal exposes the entire traversal as a target for parallel execution. Because there are numerous traversals in an application, this view of the computation exposes substantial data parallelism: traversals are independent, so the tree can be replicated across nodes, with each node performing a subset of the traversals. However, as the tree size increases with increasing input data size, it may no longer be feasible to replicate the entire tree on each node. As a result, we are forced to distribute the tree across nodes. Unfortunately, this seems to preclude data parallelism! Each traversal needs to visit multiple nodes to complete its work. Interestingly, though, distributing the tree exposes a *new* opportunity for parallelism: *pipeline parallelism*.

Building the pipeline. A point's traversal through the partitioned tree can then be viewed as a flow through a pipeline with several,

```

1 BuildSubtree(Vertex* v, Point* p, float[3] center, float dia, int depth){
2   if(!v){
3     //create a leaf vertex v, insert p into v.
4   } else {
5     //push p to a list cp
6     num=GetOctant(center, p.cord)
7     c=v.child[num]
8     if(c exists and is leaf){
9       if(numPoints contained by cell c <= MAX_POINTS_PER_CELL){
10        //insert p into c's cell
11        return c
12      } else { //split child vertex.
13        //Remove and push all points in c to cp
14        if(c.level == subtreeHeight) {
15          //make c pseudoleaf
16          owner=GetSubtreeOwnerID(c)
17          if(owner != self) return c
18          //otherwise continue recursive insertion of p
19        } }
20      for each point q in cp {
21        num=GetOctant(center, q.cord)
22        newcen = GetCenter(center, dia, num)
23        v.child[num]=BuildSubtree(c,q,newcen,dia*0.5,depth+1)
24      } }
25      return v
26 }

```

independent stages where each stage corresponds to a sub-traversal visiting one of the subtrees. Because each traversal visits different subtrees (and, indeed, some traversals visit subtrees in different orders), the pipeline stages unfold *dynamically* during traversal. Note that a single subtree can be visited multiple times during a traversal—every stage corresponds to visiting one subtree, but a subtree can be associated with multiple stages. If a point goes from a pseudo-leaf to a pseudo-root or vice-versa, the traversal moves from a pipeline stage associated with the first subtree to a pipeline stage associated with the second subtree. If the subtrees involved are placed on distinct nodes, the traversal then makes a *transition* from one stage to another when the point traverses from the first subtree to the second. For example, in Figure 2, a traversal can start at the root of the tree, in subtree 1, then move to subtree 2, return to subtree 1 before proceeding to subtree 3—executing four pipeline stages while touching three subtrees.

Exploiting pipeline parallelism. In principle, each (dynamic) pipeline stage is independent of the others, so multiple traversals that are in different pipeline stages can be processed in parallel. In particular, two points that are visiting different subtrees (which may be mapped to different processes) can be processed simultaneously. However, because each process has only one thread (see Section 4), in practice, two pipeline stages can only be executed simultaneously if they are in subtrees that are mapped to different processes.

3.3 Scheduling for locality and aggregation

Stages from different points that visit the same subtree can be co-scheduled to obtain improved temporal locality of vertex accesses. This is achieved through *blocking* of input data: allowing *blocks* of points to traverse the tree simultaneously (in essence, turning the pipeline stages associated with each of the points in the block into a single pipeline stage associated with *all* of the points in the block). Locality is improved by allowing multiple points to interact with each vertex in the subtree consecutively.

Figure 4: Block traversal through distributed tree.

Further, when a pipeline stage completes and the block of points is ready to move to another subtree, rather than initiating the next pipeline stage immediately, the block is aggregated in a buffer at the pseudo-leaf. Hence, multiple blocks can traverse a subtree before *all* of the points from those blocks are launched as a single pipeline stage on the next subtree. This *aggregation* further improves locality at the next stage, and also reduces messaging overhead.

Figure 2 also shows the blocking of traversal workload, buffers, and the subsets of points from multiple blocks being aggregated in buffers. Note that the buffers contain only subsets of the blocks that traverse the subtree, because some points truncate their traversal and do not arrive at the pseudo-leaf.

Managing block scheduling. When some of the points contained within the block truncate their traversals at a vertex, the block continues to traverse with “holes” left at positions of these points. These holes are tracked by keeping *hole maps* or *contexts*, vectors at each vertex, marking which holes are created at that vertex (i.e., which points truncate at that vertex). Figure 4 shows a block executing a stage associated with subtree 1, followed by a stage at subtree 2, and then executing a stage at subtree 1 again. The hole map on node 1 shows a hole created at position 3 at vertex *a*, at positions 4 and 5 at vertex *b*, and so on. No points truncate during the traversal of subtree 2, so the hole maps are empty.

The contexts must be tracked across multiple subtrees: when the block finishes traversing subtree 2 and returns to subtree 1 at vertex *c*, the hole information for that block must be recalled to ensure that points 1, 2, 4, and 5 are restored to the block so they can all visit vertex *f*. Tracking contexts efficiently can be challenging: The memory overhead in storing a context is $O(b + d)$, where *b* is the size of the block and *d* is the depth of the subtree.

Because a context can be quite large, SPIRIT does not send it along with the message when the block transitions from one subtree to another. Instead, a block’s context is distributed through the tree: when a block traverses a subtree, the context associated with the vertices in that subtree are kept in node-local storage. SPIRIT ensures that when a block returns to this subtree later in its traversal, it returns to the replica on the same node, allowing the context to be retrieved from node-local storage.

3.4 Load balance and Space adaptivity

Pipelining is critical to achieving good performance. However, the pipelined design as described in previous sections clearly has a bottleneck: all the blocks of input data execute the entry and exit stages associated with a single subtree (top subtree) when they begin and end their traversals at the root vertex. In addition, stages

of this subtree are executed whenever a traversal makes a transition from a subtree below. For this reason, the node containing the top subtree remains heavily loaded. To reduce the burden on this node, SPIRIT *replicates* the top subtree across a set of nodes. This *selective replication* ameliorates the pipeline bottleneck without requiring the replication of the entire tree. Figure 2 also shows the replicated top subtree on nodes 4 and 5.

Top subtree replication can improve performance greatly as blocks of points can enter the pipeline from any of the nodes where the top subtree exists, allowing multiple nodes to share the burden of processing those blocks.

Space adaptivity. In addition to replicating the top subtree, the user can tune the amount of replicated data by also replicating other bottom-level subtrees. Choosing to replicate a subtree effectively eliminates the subtree: the subtree is merged with its (replicated) parent subtree, and the associated pseudo-leaves and pseudo-roots are removed. In this way, the load on that subtree is distributed, and the messaging overhead of communicating across that subtree boundary is eliminated.

Note that if all subtrees are replicated, the entire tree is essentially replicated across all nodes. This immediately yields the data-parallel implementation of a distributed tree-traversal: the nodes are entirely independent of each other, and each node processes a subset of all the traversals on its locally-replicated tree. In this way, SPIRIT is able to smoothly transition between a pipeline parallel implementation (no replication) to a data parallel implementation (full replication) by choosing which subtrees to replicate.

Discussion. Given the dynamic nature of traversals, it is difficult to identify any bottleneck subtrees beyond the top subtree. Interestingly, we find that if subtrees beyond the top subtree are replicated, it is not especially important *which* subtrees are replicated: targeting the most heavily loaded subtrees does not perform appreciably better than replicating random subtrees. Section 5.3 explores this behavior in more detail.

4 IMPLEMENTATION

This section describes some of the implementation decisions in SPIRIT and presents an abstraction that can be used to create distributed tree applications.

SPIRIT system. Figure 5 shows the architecture, interface, and an example of NN implementation in SPIRIT. SPIRIT’s implementation uses the wrappers for MPI communication APIs available in BGL. The traversal kernel in SPIRIT is modeled upon BGL’s *visitor* pattern. However, a visitor in SPIRIT is designed to take advantage of inter-traversal locality, and message aggregation opportunities, and hence implements a traversal of the tree for a *block* of points, rather than just one. The visitor helps us to separate the details of the distributed traversal of a block from its state; an application programmer overrides a method *EvaluateVertex*, which specifies the vertex computation.

EvaluateVertex returns a status indicating the order in which a traversal wants to visit child vertices in a binary tree. For an octree, we assume that this order is always left-child first and plan to adapt our API to arbitrary orders in future work. Two additional methods need to be overridden: *SetContext* and *GetContext*. *GetContext*

Figure 5: SPIRIT Architecture, API, and example program

gets the result of local computation and *SetContext* updates local state with the result of remote computation.

The *optimizations* module accepts user-configurable values of block size, buffer size, and replication percentage to tune optimizations corresponding to blocking scheduling, aggregation, and replication respectively. Determining an optimal configuration of these values is hard. However, a value of block size chosen based on vertex size, subtree height, and available cache size can give good performance [14].

Threads vs. Processes. SPIRIT exploits multiple hardware execution contexts (cores or processors) on a single node by mapping multiple independent processes to each node. This design decision allows multiple subtrees to be mapped to a single node, improving load balance through overdecomposition. The processes are distributed round-robin, first across nodes and then to multiple cores within nodes. For e.g. on 12 processes with a 10-node cluster, we use all 10 nodes for the first 10, map 11th process to 1st node, 12th process to 2nd node and so on. Processes can either be chosen in round-robin (default) or random order to map them to subtrees, and adjacent subtrees whose pseudo-roots have the same parent are mapped to the same process. Blocked (default) or block-cyclic distribution of workloads to top-level subtrees is supported.

In order to alleviate the memory pressure arising from the need for each process to maintain its own storage for input data, SPIRIT uses shared memory segments: all the processes mapped to a node share the memory segment. Due to OS limitations, on some systems we cannot allocate all the points in a single shared memory segment to share between processes. In such situations, we process the input points in batches. While this introduces overhead that limits scaling

(as we must impose a barrier between the batches), this is purely an artifact of OS limitations and not a limitation of SPIRIT's design.

An alternative to exploiting multiple cores is to parallelize the processing of a single subtree within a process, handing out multiple blocks (or sub-blocks) to multiple threads. This multi-threading approach would have worse load balance compared to the multiple-process approach, and implementation-wise, would require another level of parallelism to be uncovered. SPIRIT uses the multi-process approach both because load balance is a substantial problem in pipeline parallelism and because it is easier to scale-up with existing implementation infrastructure to utilize more cores within a node.

5 EVALUATION

This section evaluates SPIRIT on five tree applications from the domains of scientific computing, data mining, and computer graphics. We begin by presenting strong-scaling and weak-scaling numbers for each of these benchmarks (Section 5.2), demonstrating the effectiveness of SPIRIT's combination of pipelining and selective replication. We then evaluate the space-adaptive capabilities of SPIRIT, showing how SPIRIT can smoothly trade off space usage (i.e., replication amount) for performance. Next, we present a performance breakdown consisting of the impact of subtree height, aggregation, and process utilization. Finally, we compare SPIRIT implementations of our benchmarks to implementations in two general graph frameworks—distributed Graph Lab and PBGL—as well as against existing, state-of-the-art distributed implementations of specific applications.

5.1 Methodology

We evaluate four versions of SPIRIT:

- 1) **Spirit_{DP}**: The data-parallel version of SPIRIT.
- 2) **Spirit**: The space-adaptive version of SPIRIT replicating only the top subtree.
- 3) **Spirit_{PO}**: **Spirit** with pipeline-parallelism only (no subtree replication).
- 4) **Spirit_{RO}**: Same as **Spirit**, but no pipeline parallelism (i.e., a subtree traversed by a block is “locked out” until that block completes its traversal of the entire tree; some parallelism is still available because there are multiple replicas of the top subtree).

5.1.1 Benchmarks. The benchmarks are taken from the Treelogy suite [12]. Spirit is publicly available under the distributed-memory implementations of Treelogy benchmarks at: <https://bitbucket.org/plcl/treelogy>.

Barnes-Hut (BH) [3]: is an efficient algorithm to simulate interaction of gravitational bodies. In our tests, we present the performance of an octree implementation run for 10 simulation steps.

Nearest Neighbor (NN): is described in section 2.

2-Point Correlation (PC) [9]: is used to determine how clustered a data set is. The goal is to find the number of pairs of points that are within a specified distance, R , from each other. SPIRIT includes kd-tree implementation of PC.

Vantage Point (VP) [27]: is a nearest neighbor algorithm using vp-trees, where the partitioning plane is a hyper-sphere instead of a hyper-plane as in a kd-tree.

Photon Mapping (PM) [7]: is an algorithm to realistically simulate interaction of light with objects in a scene. The objects in a scene are

Input	Description	Benchmark		
		Name	$ V $	$ P $
Mnist	Handwritten digits data with reduced dimension to $\mathbb{R}^7$	PC, NN, VP	$2 \times P - 1$	10^6
Plummer	Data in plummer model, $\mathbb{R}^3$	BH	$\approx 1.5 \times P $	10^6
Christmas	Wavefront .obj file	PM	462,818	10^6
Synthetic	uniformly distributed data in $\mathbb{R}^3$ (BH), $\mathbb{R}^2$ (PC,NN,VP)	PC,NN, BH,VP	$= 2 \times P - 1$ $= P (\text{VP})$	$\{16, 32, 64, 128\} \times 10^6$

Table 1: Data sets; $|V|$ = Vertex set size, $|P|$ =Number of points.

represented as triangle mesh, whose coordinates are then organized into a kd-tree. Ray-object intersection tests are accelerated through the depth-first traversal of the kd-tree.

5.1.2 Workloads and distribution. Table 1 describes the data sets (real [16] and synthetic) used in our experiments. PC and BH workloads are inherent to the data set sizes. For other benchmarks, workloads are due to batching of multiple user requests: NN and VP workload sizes are half the data set sizes and PM’s workload is due to tracing 1 million rays through a 480x620 pixel scene. The workloads are not pre-processed with techniques such as sorting. For the largest inputs used, the virtual memory requirement for the trees ranged from 10GB(VP)-20GB(PC), and for the total workload (including traversals) ranged from 32GB-48GB. In all the evaluated versions, message aggregation is turned on since the performance is better than that without aggregation. Block size is fixed at 4096, and the buffer size is set to the block size. Unless otherwise noted, all performance measurements are taken on tree traversal times (exclusive of tree building time, as the traversal time is the dominant cost), and the runtimes are measured using wall-clock time. Every configuration of a test is run until a steady state is achieved, which yields errors of $\pm 1\%$ of the mean with 95% confidence. The locality optimized baselines represent single node runtimes and employ block scheduling [14]. The block size that gave the best performance (not necessarily the same block size in each configuration) is chosen; these implementations hence represent the best-available single-process implementations of each benchmark.

5.1.3 Platform and program development environment. All experiments are run on a 10-node cluster with gigabit ethernet interconnect. Each node contains two 10-core, Intel Xeon-E5-2660-v3 processors and runs RHE Linux release 6. A core has 32KB L1 data and instruction cache, and 256KB of L2 cache. The cores share 25MB L3 cache and 64GB main memory. All benchmarks are implemented using C++ and Boost Graph Libraries (BGL 1.55.0)². The programs are compiled using mpic++, a wrapper compiler for gcc 4.4.7, and linked with mpich2-1.4.1p1³ and the corresponding BGL wrapper library functions.

5.2 Scalability

When SPIRIT is given enough space to replicate the tree, it essentially runs in data-parallel mode, and achieves excellent scaling (*Spirit_{DP}* plots). The purpose of our scalability experiments is to explore the behavior when SPIRIT *cannot* replicate the tree across all inputs. In such a setting, SPIRIT’s pipelining allows it to achieve speedup in both strong-scaling and weak-scaling settings. While this speedup is far less than when SPIRIT is able to fully replicate the

trees, SPIRIT’s performance is on par with existing hand-written, optimized, application-specific implementations (Section 5.5), despite its generality.

Strong-scaling. We evaluate strong scaling in SPIRIT with both synthetic and real inputs but present here only representative results with real inputs due to lack of space. The input sizes used here are constrained by inputs that allow single-process runs to complete in a reasonable amount of time.

Figures 6 shows the results. Overall, *Spirit* scales well and outperforms *Spirit_{PO}*, and *Spirit_{RO}*. *Spirit_{DP}* evaluations of PM at 128 and 256 did not run because of memory constraints. Across all the benchmarks, the geomean speedup in *Spirit* is 16.3 and that in *Spirit_{RO}* is 7.7. Recall again that while this speedup seems small, it is in a setting—distributed trees—where, absent SPIRIT’s pipelining and load balancing, we would not expect speedup at all.

The *Spirit_{PO}* plots confirm the proposition of section 3.4 that pipeline-parallelism alone is not sufficient to obtain good performance and that selective data replication is necessary. It may seem that the top subtree replication enabling data-parallel execution of sub-traversals is the only reason for *Spirit*’s superior performance. However, comparison between *Spirit_{RO}* and *Spirit* plots sheds further light: pipeline-parallelism yields, on an average, 1.54x increase in the speedup performance of *Spirit_{RO}*: the combination of pipelining and replication provide the best performance.

As the input size is fixed, the per-process number of blocks of input decreases (for a fixed block size) with increasing number of processes. As a result, available pipeline-parallelism decreases. Hence, the performance difference between *Spirit_{RO}* and *Spirit* reduces as the number of processes increase. Note that in *Spirit_{RO}*, sub-traversals execute in parallel only on the replicated top subtrees and there can only be one block in flight on bottom level subtrees. However, for an application like NN or VP, there still exists some pipeline-parallelism in *Spirit_{RO}* because of the existing intra-block parallelism: a block may split into two sub-blocks at every vertex, each of which can be independently scheduled. This makes a single block traversing through the top subtree spread into multiple bottom subtrees, each of which execute the resulting sub-blocks in parallel. In other words, even with pipelining ostensibly turned off in *Spirit_{RO}*, pipeline parallelism is still exploited.

There is more intra-block parallelism, hence pipeline-parallelism, in NN than VP (traversals in VP are much shorter). As a result, *Spirit_{RO}* does not yield as much pipeline parallelism in VP, and turning on both replication and pipelining produces a big difference in performance. Because of the lower communication overhead in VP, the absolute speedup numbers are greater than those in NN while producing the same result. BH offers the maximum benefits due to pipeline-parallelism because of the presence of more pseudo-leaves for a given top subtree height. We find that the difference between *Spirit_{RO}* and *Spirit* is even more with larger data sets used in the weak scaling experiments.

Weak-scaling. Because our target applications are irregular, it is challenging to ensure that the amount of work done per node is fixed as the input and system size scale up. Instead, we scale the input size with the number of nodes—this results in more traversals and larger trees. Note that these benchmarks are $O(n \log n)$ applications, so a linear increase in the input size results in a larger

²<http://www.boost.doc/org/libs>; PBGL version is same as BGL as PBGL libraries are an extension of BGL.

³<http://www.mcs.anl.gov/mpi/mpich2>

Figure 6: Strong scaling in SPIRIT.

Figure 7: Weak scaling in SPIRIT.

increase in the amount of work; in this experiment, we expect the overall execution time to increase.

As figure 7 shows, we essentially see that behavior. Here, the input sizes are too big to fully replicate the tree, so only a distributed approach such as in SPIRIT suffices to execute these inputs. Every process performs 0.5 million traversals. The tree size varies approximately from 16 to 128 million vertices. Tree traversal times normalized against 16-process runtimes are measured.

VP shows the best weak scaling. This is expected because of the shortest average traversal length causing minimal communication overhead. PC shows the worst weak scaling. This is because of a fixed radius value: the need for finding all pairs of points within a fixed radius causes disproportionate increase in traversal length. PM also shows flat scaling. This is due to the tree size in PM experiments being constant. Due to the unavailability of inputs (scenes) with large number of object triangles and the difficulty in generating synthetic input specifying geometry and other properties of objects in scenes, we scale-up the rays being traced from 0.5 million to 4 million while keeping the tree size constant.

5.3 Space-adaptive evaluation

Figure 8 shows the space-adaptive capability of SPIRIT. Given a simple strategy of replicating *random* subtrees (in addition to the top subtree), SPIRIT supports space-adaptivity by varying the *number* of such random subtrees. Selecting more subtrees to replicate gives more replication, and hence better performance, at the cost of more space usage. The x-axis represents the user-specified percentage of total vertices of the tree that are replicated. The left-most data point corresponds *Spirit* configuration and the right-most is *SpiritDP*, which involves no communication and hence, represents the best performance. Partial replication beyond the *Spirit* configuration in PM was not possible in case of 256 processes and hence, the corresponding plot is not shown. As we see, in all cases, increasing the amount of replication improved performance.

Replication serves two purposes: mitigating load-imbalance in the pipeline and reducing messaging overhead. The top subtree is by far the biggest bottleneck in the pipeline, as all traversals must visit the top subtree but, due to truncation, lower-level subtrees may

not be visited by all traversals. Hence, the load balancing benefits of replication come primarily from replicating the top subtree. Further replication primarily serves to eliminate pseudo-leaves in the top subtree and hence messaging: if a lower-level subtree is replicated, then there is no need to perform communication when transitioning to its (former) pseudo-root, as both the subtree and its parent are on the same node.

An interesting consequence of this dichotomy (replicating the top subtree eliminates the bottleneck, while replicating other trees primarily targets reduced communication) is that it does not particularly matter *which* subtrees (beyond the top one) are replicated: any replicated subtree eliminates a pseudo-leaf, and while some subtrees are visited more than others, this is a second-order effect.

We confirm this behavior by evaluating a version of SPIRIT that replicates *bottleneck* subtrees, rather than random subtrees: we perform a profiling run to determine which subtrees are the most heavily loaded, and replicate those. We find that the performance difference between this profile-driven strategy and the random strategy is negligible. We also find that dynamic subtree replication strategies perform worse compared to replication during tree construction. It turns out that SPIRIT’s simple strategy is sufficient.

Overall, we note that we get the best performance with fully data-parallel implementations. It is important to note, though, that this requires replicating the entire tree across every node, which may consume too much memory. Overall, these experiments demonstrate that SPIRIT is able to adapt its space usage to the resources at hand, and use pipeline parallelism to mitigate the loss of data parallelism when full replication is not possible.

5.4 Performance breakdown

Process utilization. Table 2 shows the process utilization measured as the ratio of computation time over total traversal time. We show the results for the smallest and largest of the synthetic inputs. Computation time (VT) is obtained from the average of computation times of all processes. The processes are either performing local computation or incurring overhead: either from actively communicating with other processes to handle remote traversals or polling for messages that indicate the completion of remote traversals. In other words, VT represents the “real” computation performed by SPIRIT, as opposed to overheads incurred due to distribution. We expect to see two trends: first, as more processes are added for a fixed input size, we expect VT to decrease as a percentage of traversal time, as more time must be spent in handling distribution. Second, if the input is increased for the same number of processes, we would expect relatively more time to be spent in VT, as each process holds a larger portion of the tree, and hence performs more computation. Table 2 confirms both of these trends.

Figure 8: Space adaptivity in SPIRIT with varying amount of replication

Dataset Size	Processes	PC			NN			VP			BH			PM (Christmas dataset)		
		TT	VT	D	TT	VT	D	TT	VT	D	TT	VT	D	TT	VT	D
16M	16	167.6	74.6	291.1	61.3	9.6	693.6	9.2	34.6	122.8	40.6	31.6	617.6	18.3	8.4	307.5
	32	88.6	61.4	302.8	42.9	7.2	720.6	5.8	30.3	128.1	25.5	25.8	640.3	11.6	6.7	316.2
	64	56.3	41.8	307.9	39.1	4.1	733.8	4.3	22.3	130.5	17.5	18.6	650.7	8.4	4.9	320.6
	128	37.9	31.1	310.5	34.3	2.6	740.3	3.9	12.4	131.6	15.5	10.9	655.9	7.2	3.3	323.1
128M	16	3671.5	79.3	2467.6	317.2	25.3	2923.5	98.3	42.8	942.9	458.7	43.8	4940.7	87.1	18.5	1218.7
	32	2277.5	76.1	2565.8	196.1	21.5	3046.4	58.2	40.5	984.2	260.8	36.3	5122.4	47.5	16.5	1289.2
	64	1291.5	70.1	2609.4	148.1	15.4	3102.8	38.9	32.6	1002.2	179.1	26.2	5206.1	29.5	13.5	1420.5
	128	1099.5	41.8	2630.8	97.3	12.3	3130.3	32.2	20.5	1010.9	145.2	16.3	5247.6	42.6	11.1	1749.6

Table 2: Process utilization: TT=Total traversal time (seconds), VT=per-process computation time as a percentage of total traversal time, D=aggregate volume of data exchanged in MBytes.

Figure 9: Load distribution among 128 processes.

Figure 10: Impact of a) Subtree height. b) Aggregation.

Load imbalance. Figure 9 shows the load distribution with respect to both synthetic and real inputs. The figure shows, for each process id in the 128-process run, the relative amount of time spent in computation vs. messaging overhead (in other words, a higher number means more time spent performing real computation). If there were load imbalance, we would expect to see that some processes spend significantly more time in computation (vs. messaging overhead) than others.

As can be seen from the figure, SPIRIT's subtree based partitioning and distribution scheme results in uniform load among all processes mostly. However, BH_lambb is an exception (presence of peaks) as lambb is a clustered data set. Hence, some subtrees of the octree are visited more frequently during the force calculation stage. The current distribution scheme of allocating all the sibling subtrees to a single process results in overloading this process when the sibling subtrees are frequently visited. For this particular input, we found that distributing the tree differently resulted in better load balance (nevertheless, all of the results shown in this section use SPIRIT's default distribution).

Subtree size and message aggregation. Figure 10(a) shows the impact of subtree size and aggregation on the performance of 128-process runs. As *SubtreeHeight* is a rough indicator of subtree size, we measure the performance with varying subtree heights. As *SubtreeHeight* is increased, the number of pseudo-leaves increases, which increases pipeline-parallelism and also the messaging overhead. So, we expect the performance to improve initially and then degrade as the overhead outweighs parallelism benefits. The figure shows this trend for all the benchmarks except in case of PM. The kd-tree distribution in PM results in decreasing number of pseudo-leaves with increasing height due to the long and narrow tree structure. This improves performance due to more replication and less overhead. Figure 10(b) shows the speedup improvement due to message aggregation, over non-aggregated runs with all inputs used. Overall, aggregation yields a 1.6x speedup improvement. NN and VP show higher benefits compared to other benchmarks because of the improved pipeline-parallelism resulting from increased intra-block parallelism.

Figure 11: SPIRIT vs. ChaNGa.

5.5 SPIRIT, DGL, PBGL, and reference software

In this section, we compare *Spirit* with implementations using one traversal-centric and one vertex-centric framework: PBGL and DGL. We also compare *Spirit* with ChaNGa [13], which is the state-of-the-art distributed implementation of Barnes-Hut (reference software for distributed implementations of PM and NN were not available). Since *Spirit* is specialized for distributed tree processing, we expect *Spirit* to perform better in comparison with generic distributed graph processing systems like PBGL and DGL. However, since *Spirit* lacks the application-specific customizations that reference software systems are able to exploit, we might expect *Spirit*'s performance to lag behind that of ChaNGa.

The DGL model is not the right programming fit for any tree benchmark involving depth-first traversals: the executions continue to run beyond 4 hours—due to excessive synchronization (once after every vertex computation)—and hence time out. However, the traversals in DGL are implemented easily as state transitions encapsulated in GAS template.

PBGL traversals employ asynchronous message-passing and avoid excessive synchronization. However, the traversals visit more vertices than necessary, and do not optimize communication. Implementing PBGL traversals requires a little more effort than DGL: defining a *DFSVisitor* class and overriding its methods *discover_vertex* and *finish_vertex*, and passing an instance of the class and other bookkeeping arguments (for tracking vertex visits) to the *tsin_depth_first_visit* API. The API allows a programmer to traverse a distributed tree without worrying about the underlying tree distribution. However, the API does not handle multiple traversals efficiently, and has no support for enforcing the order of vertex visits—which ruled out PM implementation. These factors cause more vertex visits and hence, more work to be done. Also, the API does not take advantage of increased data replication when available and this creates a pipeline bottleneck as in *SpiritPO*.

Spirit, due to efficient truncation handling, optimizing communication, and avoiding the pipeline bottleneck, outperforms PBGL significantly (>150x geomean speedup). Also, *Spirit* implementations have the least average lines of code (90) compared to DGL (200) and PBGL (192) implementations as they encapsulate coarse-grained computations (entire traversal) similar to PBGL and fine-grained computation (vertex computation) similar to DGL.

SPIRIT vs. ChaNGa. Figure 11 compares the strong scaling performance of SPIRIT and ChaNGa. These tests are conducted on real inputs [22]. Overall these experiments show that SPIRIT is competitive with application-specific implementations.

SPIRIT's treecode computes only Newtonian gravity forces while ChaNGa is more general. In order to ensure that both ChaNGa and SPIRIT compute the same result, the performance impact due

to softening force calculations in ChaNGa is mitigated by setting very low eps values. ChaNGa contains many application-specific optimizations for incremental computation that allow it to run faster over multi-step simulations by avoiding re-computing forces that have not changed since the previous time step. However, the first computation step represents an apples-to-apples comparison between ChaNGa and SPIRIT: both fully compute the forces for each body. In this first step, SPIRIT is actually 1.2 to 2.6 times faster than ChaNGa for all inputs at smaller number of processes.

We attribute the faster performance of SPIRIT to its model of moving computation to data, rather than ChaNGa's model of moving data to computation—fetching and caching remote subtrees on node-local storage when a traversal visits them. At larger number of processes and larger scales, the overhead due to batch processing of input traversals as mentioned in section 4 brings down SPIRIT's performance. In addition, SPIRIT's default subtree distribution policy incurs some load imbalance issues that could be addressed with a different distribution (see Section 5.4) We plan to address these issues in the future.

6 RELATED WORK

Data partitioning, optimizing communication, overlapping computation with communication, and exploiting locality is critical to achieving high-performance in distributed irregular applications. While the performance of such applications has been analyzed in detail [5, 11, 20], creating high-performance distributed-memory implementations remains a challenge. Distributed-memory graph programming frameworks partially automate and hence, simplify the creation of such applications. Frameworks based on the vertex-centric model [8, 15, 17–19] are widely used in implementing such applications. However, they are not suitable for certain tree applications due to the 'bulk-synchronous' model of computation causing excessive communication and the absence of vertex-level parallelism in the tree applications. Frameworks with a flexible programming model [6, 10, 26] allow a coarser-granularity of computation to be captured and employ asynchronous message-passing to minimize the communication overhead. In fact, Nguyen et. al. [20] showed that bulk-synchronous model applied to vertex-centric formulation of some algorithms does not always give high performance and asynchronous scheduling along with algorithm-specific optimizations are necessary. While some of these frameworks do not aggregate messages [10, 17] some do [6, 19]. However, existing locality opportunities due to coarse-level computation granularity are missed. SPIRIT exploits locality, optimizes communication, and automates traversals in a subset of tree applications. In doing so, SPIRIT adopts a traversal-centric approach, aggregates messages, and employs asynchronous scheduling in creating high performance implementations of distributed tree applications.

High performance, distributed implementations of irregular applications [2, 4, 13, 21] exploit application-specific knowledge such as how the tree changes from iteration-to-iteration in Barnes-Hut [13], and the range of bin size in two-point correlation [4]. SPIRIT being general, currently does not have these optimizations incorporated. However, these application-specific optimizations could be incorporated into existing implementations.

Distributed kd-trees are constructed bottom-up [1, 21], as against top-down, to avoid touching all points at every level. The domain-decomposition is customized for distributed octree construction in BarnesHut [13]. SPIRIT adopts a top-down, *iterative* approach for kd- and octree construction to avoid touching all the points in input data set at once. In the Hadoop implementation of distributed kd-tree by Aly et. al. [1], the top subtree height is limited by the number of nodes. SPIRIT exploits pipeline parallelism by making the height user-configurable. In replicating the top subtree, SPIRIT generalizes a Barnes-Hut specific optimization of locally essential trees [25] (replicating the top subtree captures only a subset of the interaction lists), and that of subtree caching of ChaNGa.

There has also been much work on application-specific and independent [2, 14, 23] scheduling approaches to exploit locality in tree traversals. While these techniques are proposed in the context of shared-memory systems and rely on data-parallel formulations, SPIRIT adopts an application-independent technique [14] in a distributed-memory setting.

7 CONCLUSIONS

We presented SPIRIT, a framework for creating distributed tree traversal applications. SPIRIT provides algorithms for tree distribution, supports distributed execution of applications when the input cannot be replicated on all nodes. It provides pipeline parallelism to expose parallelism in the distributed application. SPIRIT uses a block-scheduling scheme and message aggregation to maximize locality and reduce messaging overheads. Finally, it uses selective replication to avoid bottlenecks in the pipeline, allowing programmers to tune the amount of replication in an application to trade off performance and space usage. Our evaluation showed that SPIRIT can (strong- and weak-)scale across five benchmarks, and that its space adaptivity allows for increasing performance when more space is available for replication. Finally, we showed that SPIRIT implementations far outperform implementations in generic graph frameworks, and that SPIRIT provides competitive performance compared to reference software.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous referees for their helpful comments and suggestions. This work was supported in part by the following grants: NSF CCF-1150013, NSF CCF-1439126, and DOE DE-SC0010295.

REFERENCES

- [1] Mohamed Aly, Mario Munich, and Pietro Perona. 2011. Distributed kd-trees for retrieval from very large image collections. In *Proceedings of the British Machine Vision Conference (BMVC)*.
- [2] Margarita Amor, Francisco Argüello, Juan López, O Plata, and Emilio I. Zapata. 2001. A data parallel formulation of the barnes-hut method for n-body simulations. In *Applied Parallel Computing. New Paradigms for HPC in Industry and Academia*. Springer, 342–349.
- [3] Josh Barnes and Piet Hut. 1986. A hierarchical O (N log N) force-calculation algorithm. *Nature* (1986).
- [4] Jatin Chhugani, Changkyu Kim, Hemant Shukla, Jongsoo Park, Pradeep Dubey, John Shalf, and Horst D Simon. 2012. Billion-particle SIMD-friendly two-point correlation on large-scale HPC cluster systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 1.
- [5] Marios D Dikaiakos and Joachim Stadel. 1996. A performance study of cosmological simulations on message-passing and shared-memory multiprocessors. In *Proceedings of the 10th international conference on Supercomputing*. ACM, 94–101.
- [6] Adam Fidel, Nancy M Amato, Lawrence Rauchwerger, and others. 2012. The stapl parallel graph library. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 46–60.
- [7] Tim Foley and Jeremy Sugerman. 2005. KD-tree acceleration structures for a GPU raytracer. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware (HWS '05)*. 15–22.
- [8] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, Berkeley, CA, USA, 17–30.
- [9] A. G. Gray and A.W. Moore. 2001. N-Body Problems in Statistical Learning. In *Advances in Neural Information Processing Systems (NIPS) 13*. 521–527.
- [10] Douglas Gregor and Andrew Lumsdaine. 2005. The Parallel BGL: A generic library for distributed graph computations. *Parallel Object-Oriented Scientific Computing (POOSC) 2* (2005), 1–18.
- [11] Suyash Gupta and V. Krishna Nandivada. 2015. IMSuite: A benchmark suite for simulating distributed algorithms. *J. Parallel and Distrib. Comput.* 75, 0 (Jan. 2015), 1 – 19.
- [12] Nikhil Hegde, Jianqiao Liu, Kirshanthan Sundararajah, and Milind Kulkarni. 2017. Treelogy: A Benchmark Suite for Tree Traversals. *presented at the IEEE International Symposium on Performance Analysis of Systems and Software* (2017), Santa Rosa, CA, USA.
- [13] Pritish Jetley, Filippo Gioachin, Celso Mendes, Laxmikant V Kale, and Thomas Quinn. 2008. Massively parallel cosmological simulations with ChaNGa. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, 1–12.
- [14] Youngjoon Jo and Milind Kulkarni. 2011. Enhancing Locality for Recursive Traversals of Recursive Structures. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '11)*. ACM, New York, NY, USA, 463–482.
- [15] Milind Vidyadhar Kulkarni. 2008. *The galois system: optimistic parallelization of irregular programs*. Ph.D. Dissertation. Cornell University.
- [16] M. Lichman. 2013. UCI Machine Learning Repository. (2013).
- [17] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. 2012. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proc. VLDB Endow.* 5, 8 (April 2012), 716–727.
- [18] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*. ACM, New York, NY, USA, 135–146.
- [19] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. 2014. Grappa: A latency-tolerant runtime for large-scale irregular applications. In *International Workshop on Rack-Scale Computing (WRSC w/EuroSys)*.
- [20] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 456–471.
- [21] Jacopo Pantaleoni, Luca Fascione, Martin Hill, and Timo Aila. 2010. PantaRay: fast ray-traced occlusion caching of massive scenes. *ACM Transactions on Graphics (TOG)* 29, 4 (2010), 37.
- [22] Thomas Quinn. 2016. Personal correspondence. (2016).
- [23] Jaswinder Pal Singh, Chris Holt, Takashi Totsuka, Anoop Gupta, and John Hennessy. 1995. Load balancing and data locality in adaptive hierarchical N-body methods: Barnes-Hut, fast multipole, and radiosity. *J. Parallel and Distrib. Comput.* 27, 2 (1995), 118–141.
- [24] Bruce Walter, Kavita Bala, Milind Kulkarni, and Keshav Pingali. 2008. Fast Agglomerative Clustering for Rendering. In *IEEE Symposium on Interactive Ray Tracing (RT)*. 81–86.
- [25] Michael S Warren and John K Salmon. 1992. Astrophysical N-body simulations using hierarchical tree data structures. In *Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*. IEEE Computer Society Press, 570–576.
- [26] Jeremiah James Willcock, Torsten Hoefler, Nicholas Gerard Edmonds, and Andrew Lumsdaine. 2010. AM++: A generalized active message framework. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. ACM, 401–410.
- [27] Peter N Yianilos. 1993. Data structures and algorithms for nearest neighbor search in general metric spaces. In *SODA*, Vol. 93. 311–321.