ParaTreeT: A Fast, General Framework for Spatial Tree Traversal

Joseph Hutter, Justin Szaday, Jaemin Choi, Simeng Liu, Laxmikant Kale
Department of Computer Science
University of Illinois
Champaign, IL
{jhutter3, szaday2, jchoi157, simengl2, kale}@illinois.edu

Spencer Wallace, Thomas Quinn
Department of Astronomy
University of Washington
Seattle, WA
scw7@uw.edu, trq@astro.washington.edu

2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 978-1-6654-8106-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/IPDPS53621.2022.00079

***Abstract*—Tree-based algorithms for spatial domain applications scale poorly in the distributed setting without extensive experimentation and optimization. Reusability via well-designed parallel abstractions supported by efficient parallel algorithms is therefore desirable. We present ParaTreeT, a parallel tree toolkit for state-of-the-art performance and programmer productivity. ParaTreeT leverages a novel shared-memory software cache to reduce communication volume and idle time throughout traversal. By dividing particles and subtrees across processors independently, it improves decomposition and limits synchronization during tree build. Tree-node states are extracted from the particle set with the *Data* abstraction, and traversal work and pruning are defined by the *Visitor* abstraction. ParaTreeT provides built-in trees, decompositions, and traversals that offer application-specific customization. We demonstrate ParaTreeT's improved computational performance over even specialized codes with multiple applications on CPUs. We evaluate how several applications derive benefit from ParaTreeT's models while providing new insights to these workloads through experimentation.**

***Index Terms*—N-body simulation, Tree traversals, Shared-memory models**

I. Introduction

Sophisticated tree-based algorithms are effective in reducing the complexity of many science applications from an unrealistic $O(N^2)$ to $O(N \log N)$ or lower. Prominent examples include tree-based N-body gravity calculations [1], [2] and k-nearest neighbor searches [3]. However, parallelization of such tree codes is a complex task, especially when combined with a modern mix of sophisticated algorithms, e.g., the Fast Multipole Method (FMM) [4], [5] and heterogeneous hardware, e.g., many-core shared memory nodes and GPGPU architectures. Nevertheless, significant success at scaling tree codes has been made starting from the time massively parallel machines became readily available [6]. More recently, pure tree codes successfully scaled, with appropriately large problem sizes, to hundreds of thousands of compute cores [7], [8]. GPGPUs have been used effectively at scale [9]; The FMM method was implemented on GPGPU nodes [10], efficiently scaling to thousands of nodes for very large (more than a trillion particles) problems.

These successes were achieved through expertise in both domain science and computer science. This level of knowledge is widely considered necessary to extend these benefits to other problems amenable to tree-based solutions. However, the use of good abstractions can significantly ease the implementation of algorithms. This is particularly true for parallel implementations where a fairly trivial algorithmic component requires significant implementation effort, e.g., visiting the next element in a tree structure might require communication with the processor on which it is located. For linear data structures like arrays or regular grids, parallel languages such as UPC [11], Chapel, [12] and Co-array Fortran [13] have demonstrated increased programmer productivity without sacrificing parallel performance. Tree-based algorithms could be implemented in these languages by linearizing the data structures, but only at the expense of code complexity, efficiency, and effort for the programmer, particularly for more sophisticated tree-based algorithms. Here we explore the use of abstractions specifically designed for tree algorithms with the same aim of increasing programmer productivity with minimal loss of parallel performance. We focus on spatial trees because of their wide applicability to physical science problems.

Spatial trees consist of a collection of nodes where each node represents a contiguous region of space and the particles within that region. The root node contains all the particles in the simulation "universe", and represents the region defined by a bounding box that contains all particles. Child nodes are defined by recursively subdividing this region along with the contained particles. The recursion finishes when a child node contains less than a specified number of particles. Such nodes are referred to as "leaf nodes" and their contained particles are referred to as a "bucket" of particles. Tree types are defined by the strategy used to subdivide the spatial regions. For example, in 3D simulations octrees are created by subdividing each node into eight regions of equal volume. On the other hand, k-d trees are created by subdividing each node along its longest dimension into two subregions such that each of the child nodes contains half the particles of the parent. The preferred tree type depends on the particle distribution and application. For example, octrees can become significantly imbalanced in representing highly non-uniform particle distributions, but each node always has a bounding box with an aspect ratio near one. On the other hand kd-trees are guaranteed to be balanced, but nodes can have very different aspect ratios.

Tree algorithms offer computational efficiencies when the traversal of a particular node's descendants can be replaced by an approximation based on the properties of the node.

1530-2075/22/$31.00 ©2022 IEEE
DOI 10.1109/IPDPS53621.2022.00079

For example, in the Barnes-Hut algorithm, the gravitational forces on a distant body due to all particles within a node may be approximated by a single force calculated from the center of mass of the node. If the approximation is sufficiently accurate, the traversal can be "pruned" at that node. Since the gravitational force approximation is more accurate for near-spherical regions, an octree with its bounding boxes with aspect ratios near one, tends to be preferred for the Barnes-Hut application. In contrast, k-nearest neighbor searches prefer nodes with children that are uniform in particle count. However, these preferences do not necessarily hold for all particle distributions, and the best choice of tree type for a given application and particle distribution is often best determined by experimentation.

Mapping tree traversals to a distributed setting requires dividing particles and the global tree across processors. Traditionally, the particle set is decomposed according to an octree or a space-filling curve (SFC, i.e. a continuous curve whose range fills an n-dimensional hypercube), and the tree is built around the decomposition. Octree-based decomposition performs a parallel octree build breadth-first until there are enough nodes, then particles are assigned to the processor that represents their node in the global octree. SFC-based decomposition maps particles to the number line using a space-filling curve, and then divides that number line into slices uniform in particle count [6], [14]. Starting with a set of assigned particles and an artificial root node, each processor recursively creates node children and assigns them particles until each leaf represents a bucket. Then those pieces of the global tree are merged to form one unified global tree.

ParaTreeT aims to make development of applications involving hierarchical tree-structured data easy while delivering high performance and scalability. Our specific contributions are as follows:

- a set of core abstractions and interfaces that define the fundamental nature of spatial tree traversals.
- a shared-memory tree-cache model for aggregating local data and received remote data with atomic read and write operations.
- a model for separating concerns of tree build from tree traversal that reduces synchronization and offers new combinations of decomposition and tree type.

After elaborating on these contributions, we assess their implications for a number of important algorithms in the spatial tree domain. We explain how those applications can be built in C++ with ParaTreeT, their computational performance, and the implied productivity. Finally, we discuss the impact of standardization on parallel tree algorithms.

II. ParaTreeT Framework

ParaTreeT builds up a layered set of abstractions that allow fluid transitions from spatial structures to tree structures and back again during a single iteration step. These abstractions work to unlock new insights and resultant improvements to how spatial tree traversals are conducted in multi-threaded and distributed settings.

A. Tree Traversal Abstractions

ParaTreeT establishes the core relationships among particles, tree nodes, and traversals. It offers users several powerful abstractions to build and traverse spatial trees efficiently. *Trees* and their *Data* live at the lowest level of abstraction and are interpreted by *traversals* and their *Visitors* to specify tree algorithms. These abstractions allow the user to focus on the application at hand, oblivious to parallelism.

Trees are built from the root down according to tree type, but data is aggregated from the leaves to the root according to the *Data* abstraction. The data consists of application-specific attributes that adorn internal nodes of the tree, summarizing the set of particles contained within that subtree in some fashion. Trees are traversed according to traversal type but are pruned according to the *Visitor* abstraction. These concepts are illustrated more concretely in Figure 1 for a universe of 5 particles represented by a k-d tree of maximum bucket size 2.

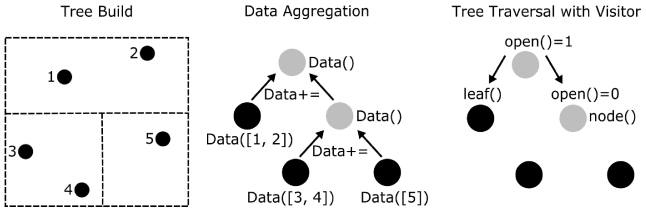

Fig. 1: Visual overview of the steps to spatial tree traversal with ParaTreeT. Left: the spacial extent of the leaf nodes of a kd-tree in a universe of 5 particles. Center: the accumulation of user *Data* from leaves to parent nodes. Right: a traversal which is pruned at the second child of the root node based on the value of *open()*.

1) Data Abstraction: ParaTreeT's traversals access user-created data stored at each tree node to let the user dynamically decide when to stop traversing. This state for each node typically summarizes characteristics of its subtree with constant space. For example, gravity applications need to know for each node the centroid (or higher-order moments) of all particles contained within its subtree's bounding box. This can be achieved by accumulating the moments from the leaves to the root.

The *Data* operators help the user extract state from the particle set into nodes of the tree. The user defines the state in a Data class and defines functions that extract data from particles. When the library assigns particles to leaves, the user implements `Data(Particle* particles, int n_particles)` to set the leaf's state. Parent nodes are initialized with the empty constructor and accumulate data from their children. The user defines this functionality by implementing `Data()` and `operator+=(const Data& child_data)`. As diagrammed in Figure 1, the library will use these functions to initialize the parent nodes and leaf nodes, then accumulate `Data` up towards the root.

2) Visitor Abstraction: ParaTreeT pre-packages the most common traversal types. Top-down traversals begin at the root and iterate depth-first, onto the local, unpruned children of each node. In the distributed setting, depth-first ordering is preferred but somewhat relaxed: traversals can bypass it by

evaluating local nodes before remote nodes. A second type of traversal, called up-and-down, does a top-down traversal iteratively from each node on the path from the leaf to the root. This traversal is usually reserved for pruning criteria that can change during the traversal, as with k-nearest neighbors. Users may implement their own traversal types using the *Traverser* interface, such as a priority-driven traversal for ray tracing.

The *Visitor* abstraction helps the user perform actions at each step of the traversal, including telling the library when to prune and stop traversing. Following the traditional terminology in the field, "opening" a node during traversal means deciding to continue the traversal underneath it. Not opening a node means the traversal will use the summary information attached to it instead. The user defines the boolean function `open()` to decide whether to traverse the source node's children. If not, the library invokes `node()`. When the traversal reaches a leaf, it invokes `leaf()`. This flow is shown in Figure 1. ParaTreeT separates out the open and leaf functions so that compilers can freely generate vectorized instructions in `node()` without restriction from the control flow in `leaf()`. Dual-tree traversals [15] require an additional function `cell()` to decide, when evaluating the interaction of two nodes with B children, whether to open both target and source (B^2 interactions), or keep the target and open the source (B interactions).

3) Performance with Generality: ParaTreeT is able to offer generality with no cost to performance by leveraging C++ techniques and building on its well-designed layers. The tree algorithms are not required to be shoe-horned into some inefficient general structure provided by the library. Rather, ParaTreeT exposes decisions each step of the way to help the user tune the program according to their application's needs. The more complex decisions, like finding splitters to recursively decompose the particle domain, are exposed via abstract classes that users can implement in addition to the prepackaged ones. The smaller decisions, like how to evaluate the interaction of two nodes, are exposed via class templates. Users write a class that defines a few member functions, and then those decisions can be compiled into the traversal code itself. Housing the frequently called functions in class templates eliminates the cost of virtualization. These static interfaces are tightly crafted so that users cannot easily add inefficiencies to the overarching tree algorithms. The program state is well-protected through read-only semantics enforced on functions executed in parallel to prevent the user from introducing race conditions. The large set of static and dynamic interfaces offers users a modular approach to development that lends itself to customization. These techniques limit the performance loss due to abstraction.

B. Shared-Memory Cache Model for Tree Data

Because spatial tree traversals comprise many simultaneous singular traversals, e.g. each particle can concurrently engage in a traversal to calculate the forces on it, a large number of tree nodes and particles must be communicated across processes each iteration. Caching these received remote data reduces the total communication volume, but the best design of this software cache remains an open problem. Past distributed tree traversal libraries have implemented the cache as a table of tree-node data hashed by tree-node key [6], [16]. Unfortunately, hash tables do not extend well to shared memory to take advantage of today's wider multicore chips. Because the software cache is constantly being updated as remote requests are filled, its parallel accesses by threads on the process must be thread-safe, which is a challenge for hash tables of dynamic size. One approach is to use a lock-free, no-delete hash table. In practice however, allocating a large enough table prior to traversal is impractical since the number of nodes traversed on a process is highly variable. A second approach is to flush thread-local hash tables to a process-wide hash table by locking. But this requires extra work, extra synchronization, and extra space for the thread-local hash tables.

ParaTreeT introduces a new shared-memory software cache model for the global tree that is wait-free (lock-free) and supports parallel reads and writes. We implement our software cache as a single tree per process, as opposed to a hash table over a collection of pointers, leveraging the intrinsically safe nature of the tree data structure to avoid locking. We extend beyond prior insights that threads need only contend on individual tree nodes, and remove contention entirely [17].

Parallel writing to the software cache can accelerate the tree traversal step, particularly towards the end. We insert into the cache directly so that other threads on the process can access previously received remote data without doing extra work to find them, i.e. as if local. Parallel cache writing can significantly reduce the length of a communication-bound critical path by enabling parallelism to cache insertions after remote request receipts. This is as opposed to assigning all cache inserts to a single thread, which is simpler than designing thread-safe cache insertions. ParaTreeT's shared-memory model permits multiple simultaneous readers and writers by using atomics with relaxed memory order load and store operations. We reduce the critical path by assigning remote request fill messages to the currently least busy worker thread on the process.

1) Implementation: The software cache setup begins during the tree build step. Pointers to the roots of the local subtrees are inserted into a process-level hash table, which uses locks for these inserts but not during the traversal step. This hash table is shown in the bottom left of Figure 2, where it is used in Step 3. Afterwards, the global root and a user-specified number of its descendants are shared with each process. Placeholder nodes represent remote data and are given an atomic flag indicating if they have been requested yet. When a traversal reaches a remote node (node 5), not yet requested, its thread sends a request to that node's home process (Step 0) and continues handling other nodes. Then the requested node and a user-specified number of its descendants, along with particles for any leaves, are serialized and sent to the requesting process (Step 1). The least busy worker on the requesting process will convert this collapsed array into Node objects and wire together the parent and child pointers (Step 2). In Step 3, it will

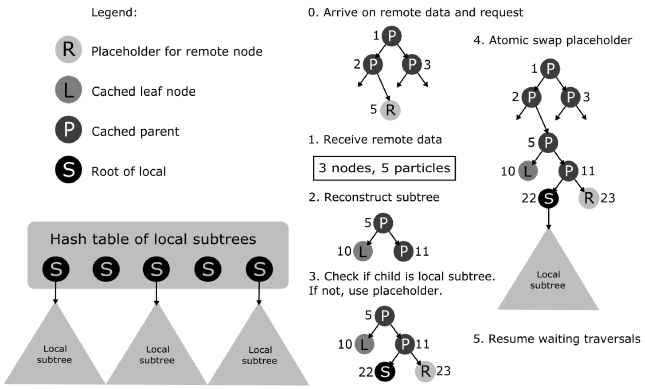

Fig. 2: Shared-memory model for our software cache of distributed tree traversals, shown by enumerating all the steps from cache miss of remote data (node 5) to resumption of the traversal, with a focus on the data structures at play.

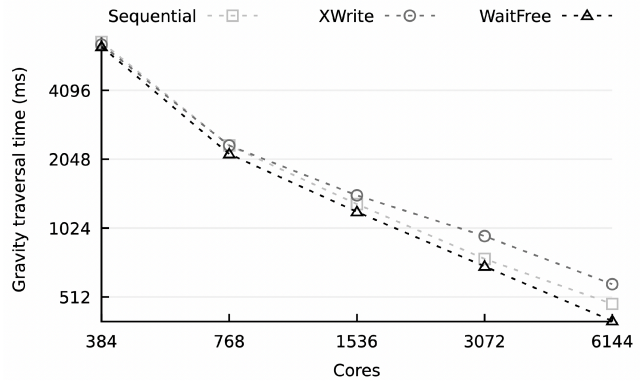

Fig. 3: Comparison of our shared memory cache 'WaitFree' against a single-threaded model 'Sequential' and an exclusive-write model 'XWrite' when performing Barnes-Hut gravity calculations on 80m particles. This was executed on Stampede2 with 24 cores to a process, one thread per core.

assign particles to the leaves (node 10) and create placeholders (node 23) for the nodes not shipped by the home process, checking the hash table first (node 22). Only then is the placeholder that represented the requested remote node (node 5) swapped out atomically from the tree (Step 4). Finally, in Step 5, the process enqueues tasks for its worker threads to resume the paused traversals. This wait-free model maintains the software cache in a valid state at all times.

2) Performance Benefits: We demonstrate the effect of the improved software cache on a gravity traversal by comparing against a per-thread software cache and an exclusive-write shared-memory cache, where every cache insertion is protected by a lock. Shown in Figure 3, we measure the runtime of the average Barnes-Hut gravity traversal on Stampede2's SKX partition for a *clustered* dataset of 80 million particles. Compared to our shared-memory model, the exclusive-write approach begins to degrade in scaling efficiency at around 1,536 cores, followed by the single-threaded approach at around 6,144 cores. Even at 1,536 cores, increased synchronization costs plague the exclusive-write model, since threads have to wait for permission to insert to the shared-memory cache. The single-threaded approach requires more communication volume and memory footprint than the two shared-memory approaches. But effective overlap of communication and computation hides the impact of this increased communication up until 6,144 cores, when the traversal's critical path appears to become communication bound. This experiment indicates that ParaTreeT's wait-free cache model performs better and scales more effectively than less sophisticated models.

C. Partitions-Subtrees Model

Tree build and traversal are made easier when particles are distributed across processes according to their position in the global tree. Unfortunately, distributing load according to the cubical shapes engendered by octrees can create imbalances in parallel execution. SFC (Space-Filling Curve) based decomposition balances load well, but requires building an octree with non-local ancestors. All such branch nodes, or tree nodes whose descendants are divided across multiple processing elements, require synchronization to merge their data. For example, if two particles A and B are in the same leaf of the octree, but SFC decomposition chooses a splitter key between their particle keys, then their leaf node and its entire path to the root must be duplicated across at least two processing elements. At the extreme end of strong scaling, wherein the SFC decomposition is very fine, merging these tree nodes will require a significant amount of communication. SFC decomposition can pair well with octrees using a mapping function from particle key to octree node key [6]; however, this kind of mapping does not exist naturally between all tree types and decomposition types.

We present the Partitions-Subtrees model, a solution to the problem of decomposing particles in a manner inconsistent with the tree structure. Tree decompositions serve dual purposes in traditional n-body codes: dividing work among processors, and acting as a distributed repository of hierarchically organized data. Our model separates these concerns by using two distributed data structures to provide multiple views of the same data. This allows ParaTreeT to offer seemingly contradictory tree and decomposition types at no cost to parallelism.

The crucial insight of the Partitions-Subtrees model is that at the boundaries of decomposed Partitions, only buckets need be split up, and not tree segments. Towards this end, we assign the division of particle buckets (i.e., load) to the Partitions, and the division of the tree (i.e., memory) to the Subtrees. More concretely, particles are held in Subtrees, and after the tree is built, buckets are passed by pointer or by value to Partitions. The Partitions-Subtrees model grants users several advantages over the traditional approach of splitting up subtrees. Firstly, communication volume is reduced – only split leaf nodes need to be communicated across processes, not their whole path to the root. Secondly, as early as the parallel tree build, users can locally access all the ancestors of local nodes. Thirdly, users of custom, application-specific tree types can write just one

implementation instead of one for each paired decomposition type. This is a great improvement to user productivity, and we showcase this by making an application-specific tree and decomposition type in our case study.

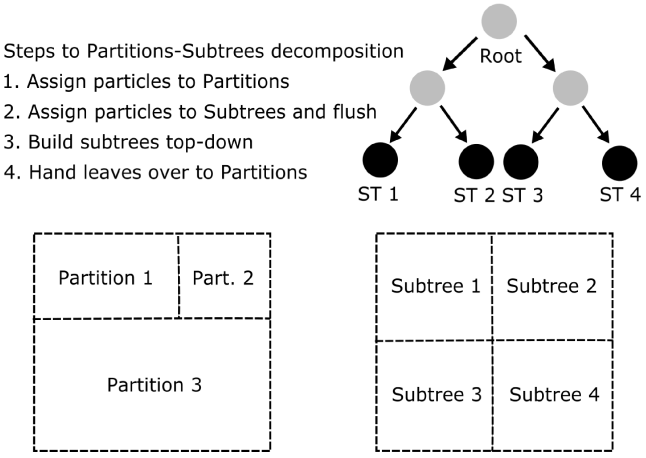

Fig. 4: Demonstration of the steps to decompose a spatial domain with the Partitions-Subtrees model: explicit steps (top left), a sample partition assignment of a region of space (bottom left), a sample subtree assignment (bottom right), and a picture of the associated global tree (top right).

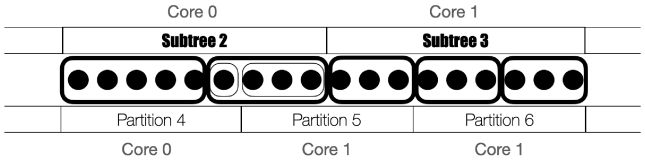

Fig. 5: A Partitions-Subtrees decomposition at the border of cores 0 and 1. The second bucket (thick) contains particles that span two Partitions. Partitions 4 and 5 each create a local bucket (thin) to accommodate for this.

1) Implementation: During the decomposition phase, Partitions are decomposed according to the decomposition type and Subtrees are decomposed in a manner consistent with the chosen tree type. The library first finds both sets of splitters, marks particles according to Partition assignment, and then forwards particles to their assigned Subtrees. Subtrees perform their usual tree build and cache initialization step. Next is the leaf-sharing step: Subtrees share their leaf pointers with local Partitions by following the particles' Partition assignments. These steps are visualized in Figure 4. Buckets whose particles are mapped to multiple Partitions will be split into local buckets as shown in Figure 5. Because particles are generally assigned to Partitions spatially and there are many buckets to a Partition, only a few buckets will need to be split this way. Subtrees whose leaves have particles belonging to remote Partitions will serialize those leaves and send them. In practice, this leaf sharing step takes only 0.1-0.4% of the total iteration time. ParaTreeT optimizes for the case when both sets of splitters are the same by binding Partitions and Subtrees by location. In this case, buckets are never split up and each particle's Partition and Subtree are on the same process.

D. Building ParaTreeT Applications

1) Using Charm++ runtime services: ParaTreeT builds on Charm++ [18] for its adaptive runtime system (RTS), implicitly migratable objects, and inherent asynchrony. A Charm++ program is specified in terms of interacting collection of objects, called *chares*. Each chare encapsulates a coarse-grained unit of work and associated data. Messages are addressed to specific chares, not to the processors. Chares are placed on processors and nodes of the system under the control of its RTS, and message delivery is under the control of Charm++ location management service. To leverage the Charm++ paragdigm, we divide the particle set into more partitions than processors, each partition implemented as a chare object. Since chares are implicitly migratable, ParaTreeT can easily redistribute work between iterations to balance loads. Charm++ has prewritten load balancing schemes and a simple interface for new schemes that offer users a clear path to achieving scale [19]. In addition, ParaTreeT rebuilds and reassigns partitions during a "flush" step if load ever becomes irreparably imbalanced.

2) Coding, configuring and running the application: To develop an application, a programmer customizes the data structures by defining appropriate subclasses. Within each iteration, the user customizes ParaTreeT's behavior by implementing `traversal()` and `postTraversal()`. The post-iteration function lets the user do work unrelated to the traversal but needed for the simulation, such as colliding particles together in planet formation or calculating kernel-weighted quantities in smoothed-particle hydrodynamics. To conduct a simulation with ParaTreeT, the user first defines a configuration object for initialization. With this object the user specifies various run and performance parameters. These include input file name, number of iterations, load balancing period, minimum number of Subtrees and Partitions, decomposition type, tree type, among others. Users can also tune other performance-specific hyperparameters: number of nodes fetched per request, number of branch nodes shared across all processors, and load balancing frequency.

3) Gravity Example: To illustrate further how applications are built on top of the ParaTreeT framework, we provide as an example our Barnes-Hut gravity code. Barnes-Hut [1] is an algorithm for calculating mutual gravitational forces among a collection of N particles in order $N \log(N)$ time by approximating the force from distant particles by grouping them into nodes of a tree. Here `gravApprox` and `gravExact` are the user's helper functions that perform Newton force calculations. In our Barnes-Hut gravity application, we devise a moment accumulator called CentroidData that implements the Data interface. We provide a simplified example of this code in Figure 6. Our more sophisticated gravity solver tracks higher order multipole expansions here as well [4]. The tree kernels that compute distances and call the gravity helper functions are shown in Figure 7. Those kernels are invoked by the library's

```
struct CentroidData {
  Vector3D moment;
  double sum_mass;
  Vector3D centroid() const {return moment / sum_mass;}
};
CentroidData() {
  moment = (0, 0, 0);
  sum_mass = 0;
}
CentroidData(Particle* particles, int n_particles) {
  for (int i = 0; i < n_particles; i++) {}
    moment += particles[i].mass * particles[i].position;
    sum_mass += particles[i].mass;
  }
}
CentroidData& operator+=(const Data& child_data) {
  moment += child_data.moment;
  sum_mass += child_data.sum_mass;
  return *this;
}
```

Fig. 6: Example tree *Data* implementation that stores in each tree node a centroid, used for computing gravitational interactions.

```
struct GravityVisitor {
  bool open(const SpatialNode<CentroidData>& source,
  SpatialNode<CentroidData>& target) {
    Sphere sphere {source.data.centroid, source.data.rsq};
    return Space::intersect(target.data.box, sphere);
  }
  void node(const SpatialNode<CentroidData& source,
  SpatialNode<CentroidData>& target) {
    for (int i = 0; i < target.n_particles; i++) {
      auto accel = gravApprox(source, target.particles[i]);
      target.applyAcceleration(i, accel);
    }
  }
  void leaf(const SpatialNode<CentroidData>& source,
  SpatialNode<CentroidData>& target) {
    for (int i = 0; i < target.n_particles; i++) {
      auto accel = gravExact(source, target.particles[i]);
      target.applyAcceleration(i, accel);
    }
  }
};
```

Fig. 7: Example tree *Visitor* implementation that uses centroid distances and helper functions to compute gravitational forces between tree nodes built on *CentroidData*.

tree traverser when it processes a node interaction. These codes are used statically through templates by the application driver, which is initialized, registered, and launched by the code in Figure 8.

Note that except for a few simple sequential functions encoding the numerics, and some elisions for simplicity, the code in these figures is the entirety of the code needed to define a full-fledged code for gravitational evolution! In total it is only 135 lines long.

III. Evaluation

We first focus the evaluation of our models and abstractions on the cosmology domain. Cosmology is an exemplar domain for several reasons. Firstly, it requires a large dynamic range in spatial scales to go from star formation distances (subparsec) to the size of the observable Universe (gigaparsecs), so an adaptive hierarchical data structure is necessary. Similarly, the range in timescales runs from thousands of years to the age of the Universe. Secondly, a variety of algorithms are used in the computation and analysis of cosmological datasets,

```
class GravityMain : public paratreet::Driver<CentroidData>
{
  using namespace paratreet;
  virtual void configure(Configuration& conf) {
    conf.input_file     = "...";
    conf.num_iterations = 10;
    conf.tree_type      = TreeType::eOct;
    conf.decomp_type    = DecompType::eSfc;
  }
  virtual void traversal(int iter) {
    partitions().startDown<GravityVisitor>();
  }
  virtual void postTraversal(int iter) {
    partitions().outputParticleAccelerations();
  }
};
```

Fig. 8: Example Barnes-Hut gravity application Driver that uses *CentroidData* and *GravityVisitor* as class templates to perform N-body simulations.

including gravity, k-nearest neighbors, and n-point correlation functions. Lastly, the sheer size of these datasets demands peak performance. A general framework that can address these challenges is likely to be successful in many other domains.

We demonstrate ParaTreeT's performance on three different architectures to explore a variety of parallel configurations. Their relevant characteristics are shown in Table I. Our work is targeted at scalable distributed memory parallelization of tree codes and the challenges encountered in that pursuit. Work used to accelerate local kernels via GPU is orthogonal, and has been addressed in past literature [9], [20], [21]. For that reason, we have chosen CPU architectures for this evaluation.

TABLE I: Relevant characteristics of supercomputers used.

Name	Cores/N	CPU Type	Clock Freq	Comm. Layer
Summit	42	POWER9	3.1 GHz	UCX
Stampede2	48	Skylake	2.1 GHz	MPI
Bridges2	128	EPYC 7742	2.25 GHz	Infiniband

A. Barnes-Hut Gravity

Before demonstrating the performance of our gravity traversal we profile its work at a macro scale during the largest portion of runtime. In Figure 9 we show the time spent in several functions of our parallel tree traversal for Barnes-Hut gravity. Utilization remains high until the traversals finish toward the end of the iteration. The first step, distributing the root and a few levels below, is the low utilization block at the beginning. Then, *local traversals* can begin, which take up a large fraction of the work because all subtrees on the process are local with the shared-memory cache. Throughout the traversal, remote requests are made, which are handled by *cache requests* and received by *cache insertions*. Then, those paused traversals are resumed and their metadata fetched in *traversal resumptions*, which will kick off the actual *remote traversal*. At this scale of 1536 cores, ParaTreeT's built-in load re-balancers can reduce this simulation's total runtime by 26%, either by mapping measured load to the space-filling curve and redistributing it in chunks, or by aggregating load and assigning it recursively in 3D space. Since the comparison applications have different load re-balancing strategies available to them, an apples-to-apples comparison is difficult. Thus load re-balancing is turned

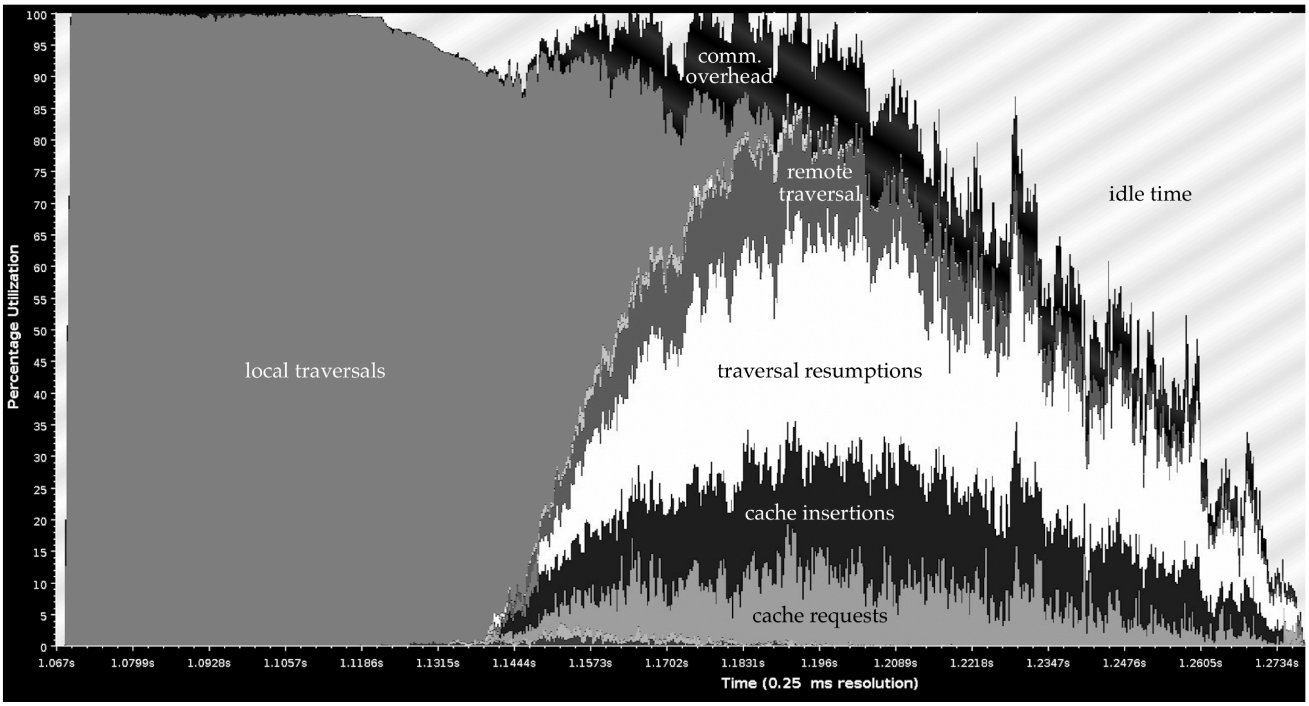

Fig. 9: Time profile of CPU utilization during parallel tree traversal for Barnes-Hut gravity collected on Stampede2 with 1536 CPUs using *Projections*. Labeled are the actions a group of processors is taking during each time interval. Due to node-wide tree aggregation and spatial decomposition, the bulk of time is spent in node-local traversals. The remaining time is spent fetching remote nodes by making cache requests, subsequent insertions, and resuming traversal.

off in our experiments, and only the first ten iterations are studied.

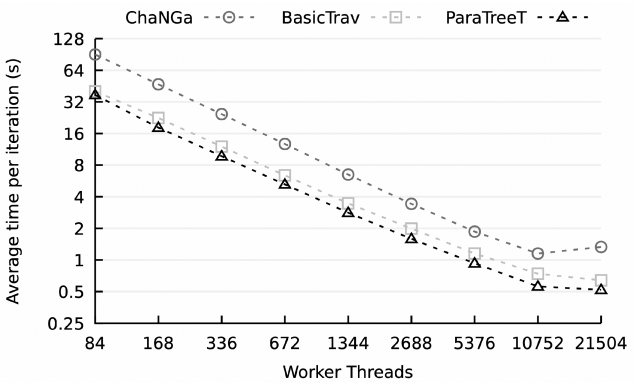

Fig. 10: Comparison of ChaNGa's and ParaTreeT's average iteration times for monopole Barnes-Hut gravity with SFC decompositions and octrees. To show the benefits of greater cache efficiency, ParaTreeT was also modified to use the standard DFS traversal style, here plotted as "BasicTrav." This was executed on Summit's POWER9 nodes for 80 million particles.

First, we benchmark the gravity application against ChaNGa's state-of-the-art distributed gravity solver [16]. ChaNGa is an n-body simulation library with a large community of users across a number of spatial applications. Studies on its excellent performance have been published in past computer science literature [8], [16], [22]. ParaTreeT can outperform ChaNGa both on a single node and at scale. Figure 10 shows a set of performance comparisons conducted on Summit at Oak Ridge National Laboratory using only CPU resources and 2-way SMT. The application here is Barnes-Hut gravity for 80 million particles in a *uniform* particle distribution representing a volume of the present-day Universe. ParaTreeT and ChaNGa return identical solutions and share the same computational work. ParaTreeT performs iterations 2-3x faster from 1 to 256 nodes, where there are 84 workers per node. At 256 nodes both applications stop strong scaling efficiently, but compared to 128, ParaTreeT's runtime improves slightly while ChaNGa's gets worse. ParaTreeT finds its improvements from reducing idle time: on a single node by reducing cache communication, and on multiple nodes by reducing inter-node communication as well.

ParaTreeT achieves higher sequential throughput by limiting the working set size and adopting the GPU style of traversal, also known as a locality-enhancing loop transformation. The *Data* abstraction drives a compact working set for each tree node, which provides most of the benefit between the two applications. Loop transformation offers ParaTreeT further improved cache efficiency: instead of traversing the tree for each bucket, it processes each bucket for each tree node [23]. This transposition works well on the GPU given its expensive global memory fetches and warp-level parallelism that can be used as thread-per-bucket [21]. While SIMT does not apply to multicore architectures, sequential tree traversal's pointer-chasing workload can be throttled by memory bandwidth. ParaTreeT adopts this transformation to achieve greater CPU

TABLE II: Comparison of ParaTreeT and ChaNGa's cache utilization statistics at the three levels of data cache storage for a gravity traversal of 100k particles. This was profiled on one of Stampede2's SKX nodes, where the L1D, L2, and L3 caches have capacities 32KB, 1024KB, and 33MB, respectively.

(ParaTreeT/ChaNGa)		Cache Accesses (billion)		Load Miss Rate (%)			Store Miss Rate (%)	
CPUs	Runtime (s)	L1D Load	L1D Store	L1D	L2	L3	(L1D & L2)	L3
1	9.2 / 16	27 / 47	9.0 / 21	3.4 / 1.5	1.9 / 3.5	19 / 9.2	0.036 / 0.020	62 / 26
2	5.2 / 8.0	24 / 38	6.7 / 15	3.8 / 1.9	1.0 / 3.0	32 / 8.1	0.050 / 0.030	48 / 19
4	2.8 / 4.3	20 / 36	4.1 / 12	4.4 / 2.1	1.5 / 2.9	44 / 19	0.12 / 0.046	55 / 35
8	1.6 / 2.5	18 / 32	3.2 / 11	4.4 / 2.3	2.1 / 3.7	32 / 18	0.24 / 0.091	43 / 29
16	1.1 / 1.6	18 / 30	3.0 / 10	3.7 / 2.5	3.6 / 4.6	26 / 22	0.33 / 0.13	43 / 32

cache efficiency through temporal locality. This is reflected in the cache utilization statistics in Table II. In this experiment a spatial tree representing 100k uniformly distributed particles is traversed by multiple CPUs on the same process. The set of buckets in a Partition fits in the L2 cache and the tree traversed for that set fits in the L3 cache. ParaTreeT sees lower runtime due to fewer cache accesses by not walking the tree once per bucket. This does yield higher miss rates for all cache levels and access types except L2 loads as expected. ParaTreeT's Traverser interface is exposed to the user so that they can write their own additional optimizations.

When running at scale, ParaTreeT has smaller communication overheads than those of ChaNGa due to its shared-memory cache and Partitions-Subtrees model, improving the traversal and tree build respectively. During traversal, ChaNGa often makes the same remote fetch for multiple worker threads within the same process. The costs of these extra requests and responses are especially noticeable on wider multicore chips and with SMT enabled. During tree build, because this simulation uses SFC decomposition for an octree, the data aggregation step requires merging many non-local ancestors. The Partitions-Subtrees model reduces the synchronization cost of this action and thus improves the tree build time.

B. Smoothed-Particle Hydrodynamics

Smoothed-particle hydrodynamics (SPH) is a Lagrangian method for astrophysical fluid simulations that also leverages spatial tree traversal. The basic idea of SPH is to construct a continuous field using the properties of nearby particles as discrete tracers [24], [25]. For many astrophysical applications, the density, internal energy and pressure fields are calculated and evolved. Gradients in the latter field produce a net force on the surrounding fluid.

Each iteration of SPH starts with a k-nearest neighbors traversal for each particle to find its principal contributors of density. Each neighbor's mass and distance is summed and weighted with a smoothing kernel to determine the density of the target. This neighbor list is then used to model the pressure field surrounding each particle. A pressure force, which is determined by the gradient of this field, is then applied to pairs of particles [26], [27].

In Figure 11 we compare ParaTreeT's performance to that of Gadget-2 [28], a well-established gravity and SPH application. Here we are just performing SPH computations without gravity. On Stampede2's SKX nodes, ParaTreeT yields a ∼10x speedup from 48 to 3072 cores, where both applications are doing the same SPH computations on an octree with SFC decomposition. While both are built on the MPI layer, Gadget-2 relies on the Message Passing Interface entirely, and does not leverage shared memory. ParaTreeT achieves most of this speedup by fetching a fixed number of neighbors using the k-nearest neighbors algorithm, as opposed to Gadget-2's more parallelizable but less efficient algorithm of converging on a smoothing length for each particle by doing a number of fixed-ball searches. ParaTreeT is able to implement this more efficient algorithm within its abstractions.

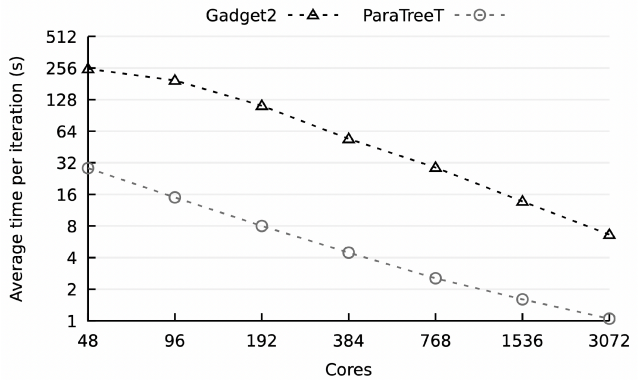

Fig. 11: Comparison of Gadget2's and ParaTreeT's average iteration times for smoothed particle hydrodynamics with octrees. This was executed on Stampede2's SKX nodes for a cosmological volume of 33 million particles.

C. Programmer Productivity

ParaTreeT's powerful abstractions and simple interface combine to achieve next-generation productivity in spatial tree simulations. Our gravity application, shown above to outperform state-of-the-art code, uses only 135 lines of user code including the numerical calculations. (Our smoothed particle hydrodynamics code is a bit longer at 250 lines of user code.) For comparison, the code specific to the Barnes-Hut application in ChaNGa totals roughly 4500 lines of code. Table III provides a breakdown of our code below. A simplified version of CentroidData.h was shown in Figure 6, GravityVisitor.h in Figure 7, and GravityMain.C in Figure 8.

Beyond the number of lines of code, which is just one simple metric for code complexity, the productivity benefits of ParaTreeT accrue mainly from the *separation of concerns*

TABLE III: Line counts of user code in ParaTreeT gravity application.

Filename	Line count	Use
CentroidData.h	50 lines	Define optimized Data functions
GravityVisitor.h	45 lines	Define Visitor functions
GravityMain.C	40 lines	Specify config, define traversal

it affords. A computational cosmologist can focus on specifying the numerical and algorithmic aspects of their model, while leaving the tedium of coding traversals, communication, memory management, caching of remote data, load balancing, and so on to ParaTreeT.

IV. Case Study: Planet-Forming Disks

While the previous applications dealt only with cosmological simulations, here we demonstrate a simulation of planet formation at unprecedented resolution. Protoplanetary simulations represent a different spatial layout (mostly 2D) and a different algorithm (collision detection) from traditional applications. This makes for an interesting case study of ParaTreeT's adaptibility and performance, especially when compared to ChaNGa.

A. Collisions Between Planetesimals

In this simulation, a disk of 10 million planetesimals and a Jupiter-sized planet orbits a star to mimic the conditions of the early Solar System. At only 50 km in size, the particles here represent actual individual planetesimals, rather than collections of bodies. Gravitational interactions are tracked between all particles. Additionally, the planetesimals are modeled as solid objects with a finite radius and are tested for collisions at each step. In regions of the disk where the orbital period matches an integer ratio of Jupiter's orbit (a resonance), strong nonlinear perturbations occur, making this an interesting problem to study with N-body methods.

Using the regular memory partition on the Pittsburgh Supercomputing Center's Bridges2, we evolved the disk for 2,000 years, which corresponds to roughly 150 orbits of the perturbing planet. Because the dynamical effects of the resonances cannot be built into the initial conditions, no collisions were recorded for the first 1,200 years of the simulation. This time interval corresponds to the libration period of the 2:1 resonance, located near the midpoint of the disk, and allows the phase space distribution of bodies to reach equilibrium.

In Figure 12, we show the resulting planetesimal collision profile as a function of both orbital period and distance from the central star. Here, the orbital period associated with a collision corresponds to the orbital period of one of the two bodies at the moment of impact. Gravitational perturbations from the planet are visible as gaps in the disk. In the collision profiles, the 3:1, 2:1 and 5:3 resonances (from left to right) are marked with vertical dashed lines. In total, 258 collisions were recorded, most of which are associated with high eccentricity particles near the 2:1 resonance at 3.27 AU.

Dust generated by these collisions in planet-forming disks should be visible through sub-millimeter observations and

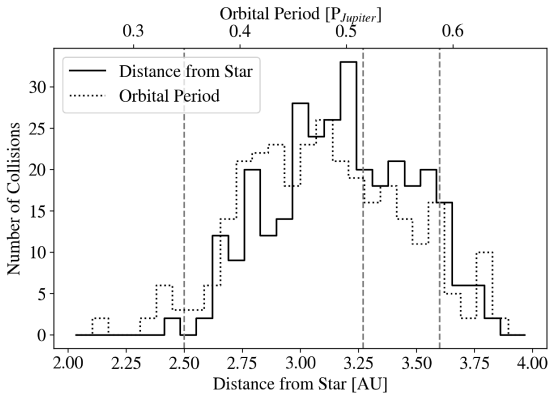

Fig. 12: For a planetesimal disk consisting of 10 million particles evolved with ParaTreeT, the number of planetesimal collisions detected as a function of distance from the star. Collisions as a function of orbital period about the central star are plotted with a dotted line. Vertical dashed lines indicate the location of resonances with the planet. Dust generated by these collisions will follow a profile similar to the solid curve.

can be used to infer the properties of the perturbing planet [29]. The solid line of Figure 12 shows the radial profile of the collisions. Assuming that the dust generated is well-coupled to any gas that is present, this should closely describe the sub-millimeter brightness profile of the disk. Although a comprehensive exploration of the dynamics contributing to the collision profile seen here is beyond the scope of this work, we have shown that this problem is thoroughly tractable with ParaTreeT.

B. Domain Decomposition for Disks

ParaTreeT allows the user to write their own decomposition type by implementing `findSplitters()` and their own tree type by implementing `findChildsLastParticle()` and setting the branch factor. The customization of these interfaces offers finer control over how the library dissects the particle domain in decomposition and tree build. This is especially useful for this simulation because the planetary space is a mostly two-dimensional disk. Dissecting all three dimensions equally makes for useless tree branching and poor decomposition, yielding weaker performance and scalability. A longest-dimension tree solves this problem by branching at the median but always in the longest dimension of the current subspace.

In Figure 13 we compare the average iteration time when using the longest-dimension tree and decomposition schemes against when using basic octree. The iteration step includes tree building, calculating gravitional forces, and detecting collisions. We also use this opportunity to compare ParaTreeT's performance to ChaNGa's on a different application and supercomputer than previously shown. This was conducted with Stampede2's SKX partition, for a planetesimal disk of 50 million particles. With octree decomposition, load imbalance towards nodes around the disk is significant enough to cancel the benefits of scaling for unfortunate configurations, like at

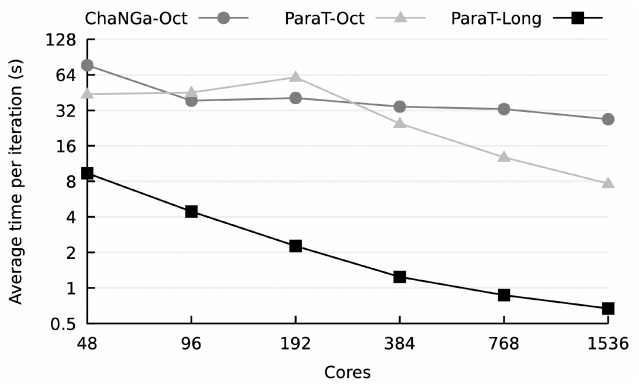

Fig. 13: Comparison of average iteration time for longest-dimension tree and decomposition against that of ParaTreeT and ChaNGa's octree implementations in simulating evolution of a protoplanetary disk. This was executed on Stampede2's SKX nodes for 50 million particles.

192 cores. The longest-dimension tree has better load balance and can achieve greater performance, especially at scale. A further improvement on this decomposition might be to divide the disk radially into sectors. With ParaTreeT's customizable modules, users can develop performant codes for even highly irregular applications.

V. Related Work

The performance and productivity demonstrated here is enabled by the abstractions implemented in ParaTreeT which is in turn built on the Charm++ runtime system. Since ChaNGa is another publicly available tree-based application built on Charm++, we were able to easily adopt some of its techniques. Both ChaNGa and ParaTreeT take advantage of overdecomposition for communication/computation overlap and load balancing [30]. More specific to tree algorithms, we adopt ChaNGa's use of a space-filling curve-based load balancing scheme that maps particles to their positions along a space-filling curve but weights them according to their current load as measured by the Charm++ runtime. Weighted sections of this curve can be used to remap processor assignments to achieve better load balance. Additionally, ChaNGa's process-wide local data aggregation step prior to traversal was an important first step in the development of the shared-memory cache [16]. We also adopted many of the mathematical kernels at the heart of gravity, hydrodynamics, and collision detection. While it satisfies performance benchmarks, ChaNGa is primarily an astrophysical simulation code, and the published development effort has been mainly focused on the implementation of astrophysics modules. These are all built around an octree data structure, and extending ChaNGa to explore new algorithms would be difficult.

There have been several past efforts to generalize the spatial tree domain for the purposes of productivity. One such project, called the Framework for Developing Particle Simulators (FDPS) [31], offers sufficient generality in the steps of tree-based simulations, but relies too heavily on particle-particle relationships. The abstractions therein are based on those relationships instead of particle-to-tree and tree-to-traversal. Trees are still built on each process in FDPS but their use is only in the creation of interaction lists, which are then shared back to the requesting process. These lists are then processed with SIMD or SIMT to achieve some further parallelism. These bulk steps of collecting and processing interaction lists requires more synchronization and larger communication bursts [32]. The approach of traversing the global tree iteratively through remote requests prevents these issues. Furthermore, with rapidly increasing problem sizes, sharing and maintaining these large sets of interaction lists can put a strain on pinned and virtual memory resources.

The SPIRIT library [22] is a more recent effort in the direction of generality. SPIRIT targets tree-based computing in the general case and is not spatially-driven. It offers pipeline parallelism for tree traversals using a scheduling system that targets temporal locality. It enforces tree-based decomposition of particles, relying on adaptive load balancing to accommodate load imbalance, and lacks in simulation infrastructure for N-body simulations. SPIRIT has proven itself for a number of tree-based algorithms, including two-point correlation and ray tracing. Unfortunately, it struggles at scale on spatial applications because of a focus on tree-based computations instead of bucket-based computations which offer more natural parallelism. Pipeline parallelism also requires heavily front-loading or back-loading work as distributed tree computations launch and complete.

VI. Conclusion

This work presents evidence that the standardization of tree-based algorithms is both feasible and advantageous for spatial domain applications. Creative solutions for the general case can yield greater improvements than overspecialization and incremental refinement for a particular case. We demonstrated such improvements by adopting strategies that favor newer computer architectures. As the shift towards higher thread counts continues (i.e., wider multicore chips and SMT/SIMT), the benefits of our shared-memory cache will become increasingly pronounced. Rapidly growing problem sizes demand greater scalability in the form of smarter decomposition strategies and load-adaptive techniques [33]. Only through productive experimentation will these enhancements be discovered, enabling previously infeasible simulations and analyses.

We have extended ParaTreeT to a variety of unique spatial applications, and will continue to expand its usability. Our underlying abstractions capture the nuances of tree building and traversal with minimal user direction. With ParaTreeT, scientists can focus their efforts on asking and answering scientific questions without a floor-to-ceiling redesign each time. This lowers the barrier of entry to conducting groundbreaking research in astrophysical simulations, and maintains momentum in the direction of progress. This paper did not focus on issues of dynamic load balancing, which is especially important for long-running applications. We intend to further develop scalable strategies that are specialized to

ParaTreeT and our *Partitions-Subtrees* model. Through our commitment to delivering excellent performance, we hope that standardization through ParaTreeT will concentrate the scientific community's effort on the frontier of tree-based simulations.

VII. Acknowledgments

We thank Ajay Tatachar and Pritish Jetley for invaluable contributions to ParaTreeT in its infancy. We also thank the developers and maintainers of the project's predecessor ChaNGa and its runtime system Charm++. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1548562. ParaTreeT used compute resources at the Texas Advanced Computing Center, the Pittsburgh Supercomputing Center, and the Oak Ridge National Laboratory throughout its development. This project was made possible by funding from the National Science Foundation with awards #1906892 and #1910428. ParaTreeT is hosted by GitHub at https://github.com/paratreet/paratreet.

References

[1] J. Barnes and P. Hut, "A hierarchical O(N log N) force-calculation algorithm," *Nature*, vol. 324, no. 6096, pp. 446–449, Dec. 1986.

[2] A. W. Appel, "An efficient program for many-body simulation," *SIAM J. Sci. and Stat. Comput.*, vol. 6, no. 1, pp. 85–103, 1985.

[3] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[4] L. Greengard and V. I. Rokhlin, "A fast algorithm for particle simulations," *Journal of Computational Physics*, vol. 73, 1987.

[5] W. Dehnen, "A hierarchical $O(N)$ force calculation algorithm," *Journal of Computational Physics*, vol. 179, pp. 27–42, 2002.

[6] M. Warren and J. Salmon, "A parallel hashed oct-tree n-body algorithm," in *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, 12 1993, pp. 12– 21.

[7] M. S. Warren, "2hot: An improved parallel hashed oct-tree," *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, Nov 2013. [Online]. Available: http://dx.doi.org/10.1145/2503210.2503220

[8] H. Menon, L. Wesolowski, G. Zheng, P. Jetley, L. Kale, T. Quinn, and F. Governato, "Adaptive techniques for clustered N-body cosmological simulations," *Computational Astrophysics and Cosmology*, vol. 2, p. 1, Mar. 2015.

[9] J. Bédorf, E. Gaburov, M. S. Fujii, K. Nitadori, T. Ishiyama, and S. Portegies Zwart, "24.77 pflops on a gravitational tree-code to simulate the milky way galaxy with 18600 gpus," in *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014, pp. 54–65.

[10] D. Potter, J. Stadel, and R. Teyssier, "PKDGRAV3: beyond trillion particle cosmological simulations for the next era of galaxy surveys," *Computational Astrophysics and Cosmology*, vol. 4, no. 1, p. 2, May 2017.

[11] U. Consortium, "Upc language specifications, v1.2," Lawrence Berkeley National Lab, Tech. Rep. LBNL-59208, 2005.

[12] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the chapel language," *The International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007. [Online]. Available: https://doi.org/10.1177/1094342007078442

[13] R. W. Numrich and J. Reid, "Co-array fortran for parallel programming," *SIGPLAN Fortran Forum*, vol. 17, no. 2, pp. 1–31, 1998.

[14] H. Sagan, *Space-filling curves*, ser. Universitext. New York: Springer-Verlag, 1994.

[15] A. G. Gray and A. W. Moore, "'n-body' problems in statistical learning," in *Proceedings of the 13th International Conference on Neural Information Processing Systems*, ser. NIPS'00. Cambridge, MA, USA: MIT Press, 2000, p. 500–506.

[16] P. Jetley, F. Gioachin, C. Mendes, L. Kalé, and T. Quinn, "Massively parallel cosmological simulations with changa," in *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, 04 2008, pp. 1–12.

[17] M. Burtscher and K. Pingali, "An efficient cuda implementation of the tree-based barnes hut n-body algorithm," *GPU Computing Gems Emerald Edition*, 12 2011.

[18] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, and E. Totoni, "Parallel programming with migratable objects: Charm++ in practice," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, vol. 2015, 11 2014, pp. 647–658.

[19] G. Zheng, E. Meneses, A. Bhatele, and L. V. Kale, "Hierarchical load balancing for charm++ applications on large supercomputers," in *2010 39th International Conference on Parallel Processing Workshops*, 2010, pp. 436–444.

[20] 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*, 06 2016, pp. 1–12.

[21] J. Liu, M. Robson, T. Quinn, and M. Kulkarni, "Efficient gpu tree walks for effective distributed n-body simulations," in *Proceedings of the ACM International Conference on Supercomputing*, 06 2019, pp. 24–34.

[22] N. Hegde, J. Liu, and M. Kulkarni, "Spirit: a framework for creating distributed recursive tree applications," in *Proceedings of the 2016 International Conference on Supercomputing*, 06 2016, pp. 1–11.

[23] 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: Association for Computing Machinery, 2011, p. 463–482. [Online]. Available: https://doi.org/10.1145/2048066.2048104

[24] L. B. Lucy, "A numerical approach to the testing of the fission hypothesis." *Astronomical Journal*, vol. 82, pp. 1013–1024, Dec. 1977.

[25] R. A. Gingold and J. J. Monaghan, "Smoothed particle hydrodynamics: theory and application to non-spherical stars." *Monthly Notices of the Royal Astronomical Society*, vol. 181, pp. 375–389, Nov. 1977.

[26] Y. R. López and D. Roose, "A parallel sph implementation on shared memory systems," in *9th SPHERIC International Workshop*, 2014.

[27] R. Davé, J. Dubinski, and L. Hernquist, "Parallel treesph," *New Astronomy*, vol. 2, no. 3, p. 277–297, Aug 1997. [Online]. Available: http://dx.doi.org/10.1016/S1384-1076(97)00019-5

[28] V. Springel, "The cosmological simulation code gadget-2," *Monthly Notices of the Royal Astronomical Society*, vol. 364, no. 4, p. 1105–1134, Dec 2005. [Online]. Available: http://dx.doi.org/10.1111/j.1365-2966.2005.09655.x

[29] S. C. Wallace, T. R. Quinn, and A. C. Boley, "Collision rates of planetesimals near mean-motion resonances," *Monthly Notices of the Royal Astronomical Society*, Mar. 2021.

[30] J. Choi, D. F. Richards, and L. V. Kale, "Achieving computation-communication overlap with overdecomposition on gpu systems," in *2020 IEEE/ACM Fifth International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*, 2020, pp. 1–10.

[31] M. Iwasawa, A. Tanikawa, N. Hosono, K. Nitadori, T. Muranushi, and J. Makino, "Implementation and performance of fdps: a framework for developing parallel particle simulation codes," *Publications of the Astronomical Society of Japan*, vol. 68, no. 4, pp. 54–54, Aug 2016.

[32] Y. Drougas and V. Kalogeraki, "Accommodating bursts in distributed stream processing systems," in *2009 IEEE International Symposium on Parallel Distributed Processing*, 2009, pp. 1–11.

[33] M. P. Katz, A. Almgren, M. B. Sazo, K. Eiden, K. Gott, A. Harpole, J. M. Sexton, D. E. Willcox, W. Zhang, and M. Zingale, "Preparing nuclear astrophysics for exascale," 2020.