Scalable clustering algorithm for N-body simulations in a shared-nothing cluster

YongChul Kwon[1], Dylan Nunley[2], Jeffrey P. Gardner[3], Magdalena Balazinska[4], Bill Howe[5], and Sarah Loebman[6]

University of Washington, Seattle, WA

{[1]yongchul,[2]dnunley,[4]magda,[5]billhowe}@cs.washington.edu

[3]gardnerj@phys.washington.edu

[6]sloebman@astro.washington.edu

***Abstract*— Science is increasingly becoming a data management problem. Scientists in many domains such as astronomy, biology, and oceanography are acquiring data at an unprecedented rate from large-scale sensor deployments, high-throughput laboratory equipment, and massive-scale computational simulations. With such massive datasets, data management and analysis tasks are becoming the new bottleneck for scientific research.**

In this paper, we focus on one common yet challenging data analysis problem from the astronomy simulation domain: massive-scale data clustering. We study the performance and scalability of a clustering algorithm called Friends-of-Friends. This algorithm is designed to cluster points in a multi-dimensional space and is commonly used on simulation data to study galaxy formation and evolution. We address two technical challenges in the scalability of this algorithm. First, we show how this algorithm can be parallelized and implemented in a MapReduce-style shared-nothing computational cluster. Second, we present an optimization to handle extremely dense regions that are common in these kinds of astrophysical simulations. We implement our solution in the Dryad parallel data processing system using DryadLINQ. We evaluate its performance and scalability using a real dataset comprised of 906 million points, and show that in a small 8-node cluster, our algorithm can cluster even a highly-skewed dataset in 70 minutes and offers near-linear scalability.

I. Introduction

Advances in high-performance computing technology and better access to large-scale compute clusters are changing the face of many scientific disciplines. One area where this change is particularly visible is the area of scientific simulations. Simulations are used to model the behavior of complex natural systems ranging from subatomic particles to biological proteins, climate change, and even the evolution of structure in the universe. Increased compute power and an increased ability to harness this power enable scientists to run simulations at an unprecedented scale. For example, by the end of 2011, a single astrophysics simulation of galaxy formation will generate several petabytes of data, with single snapshots ranging in size from 10s to 100s of TB.

While simulations are growing in size and complexity, scientists' ability to analyze the resulting data remains limited. The reason is not lack of expertise, but simple economics. A simulation code is typically used by a large number of researchers, and it is often in use for 10 years or more. Data analysis applications, however, are currently often unique to individual researchers and evolve much more quickly. Therefore, while it may be affordable for a science discipline to invest the time and effort required to develop highly scalable simulation applications using hand-written code in languages like Fortran or C, it is usually infeasible for each individual researcher to invest a similar effort in developing their own scalable, hand-written data analysis solution. Consequently, data analysis is becoming the bottleneck to knowledge discovery, and scientists often limit their simulations in scale and scope due to their inability to manage the results.

The state-of-the-art technologies for such large-scale analytic queries include parallel databases, such as Oracle [1], DB2 [2], Teradata [3], and Greenplum [4], and new types of massive-scale data processing platforms, such as MapReduce [5], Hadoop [6], and Dryad [7]. In addition, the database research community has initiated a collaborative effort to design and build a new data management system for complex analytical queries over massive scientific databases [8].

While analytical queries are helpful for data exploration, data clustering is another popular task for scientists. Clustering algorithms, however, are generally outside of the scope of conventional query systems and have traditionally motivated their own field of research [9]. In astrophysics, cluster identification programs, like most astrophysical data analysis applications, have historically been written as serial programs. Substantial effort is involved to rewrite them in a scalable manner for the parallel distributed-memory architectures commonly used to run simulations.

Contributions: To address the above limitation, we investigate scalable clustering techniques using a shared-nothing cluster. More specifically, our contributions are:

- We present a density-based distributed clustering algorithm expressed in a shared-nothing parallel programming framework, Dryad, (a generalized form of MapReduce [7]).
- We present optimizations for data partitioning and spatial indexing that together provide near-linear scalability.
- We implement the proposed algorithm using Dryad [7] and DryadLINQ [10] and evaluate its performance in a small-scale eight-node cluster using two real world datasets from the astronomy simulation domain. Each dataset comprises over 906 million objects in 3D space.

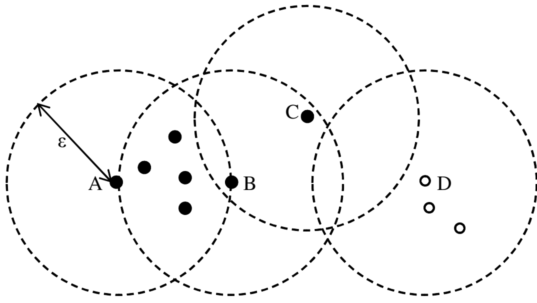

Fig. 1. **Friend of Friend relation.** Two particles are considered *friends* if the distance between them is less than a threshold ϵ. In the figure, A and B are friends and B and C are friends, but A and C are not because the distance between A and C is greater than ϵ. The friend relation is symmetric if the distance is symmetric. The Friend of Friend relation (FoF) is defined between two points if they are friends or if they are contained in the transitive closure of the friend relation (e.g., A and C are friend of friend via B). In the figure, the FoF relation induces a partition on the particles: all black filled points are in one cluster and all white unfilled points are in another.

Overall, we demonstrate that our approach can cluster a massive-scale simulation dataset in under 70 minutes and offers near linear scaleup and speedup.

II. Background

Cosmological simulations are used to study how structure evolves in the universe on distance scales ranging from a few million light-years to several billion light-years. In these simulations, the universe is modeled as a set of particles that interact with each other through gravity and hydrodynamics. These particles represent gas, dark matter, and stars. Certain particles can be created or destroyed as the simulations progresses: for example, a gas particle can form several star particles, eventually shrinking to zero mass and disappearing as all of its mass turns into stars. Every few simulation timesteps, the simulator outputs a snapshot of the universe, which lists all the particles, their locations, velocities, and other properties. State of the art simulations (e.g., Springel *et al.* 2005 [11]) use over 10 billion particles producing a data set size of over 200 GB per snapshot. Such a simulation requires a terabyte of RAM, and over 1 million CPU hours. In the near future, astronomers will use the full 1.6 million CPU cores of the upcoming NCSA/IBM Blue Waters system [12] which has the potential to generate volumes exceeding 10 PB of data per run and 10 to 200 TB per snapshot, depending on simulation parameters.

A. Friends of Friends Clustering Algorithm

The Friends-of-Friends (FoF) algorithm (*cf* [13] and references therein) has been used in cosmology for at least 20 years to identify interesting objects and quantify structure in simulations [11], [14]. FoF is the simplest algorithms used to identify clusters of objects in simulations. Thanks to its simplicity, however, FoF is one of only two algorithms to have been implemented in a distributed parallel fashion [15], [16]. The other is AMIGA[17].

The Friends-of-Friends algorithm is based on the notion of a *friend*. Two particles are friends if they are within a distance ϵ of each other. Two particles are *friend-of-friend* if they are either direct friends or if they can be reached through a series of intermediate friend relations. More formally:

Definition: (Friend relation). Let $\delta : R^k \times R^k \rightarrow R$ be a metric over R^k and let ϵ be a real number. For $x, y \in R^k$, $Friend(x, y, \epsilon)$ if and only if $\delta(x, y) < \epsilon$.

Definition: (Friend of Friend (FoF) relation). Let D be a set of points in R^k. Then $FoF_D(x, z, \epsilon)$ if and only if $Friend(x, z, \epsilon)$ or there exists $y \in D$ such that $FoF_D(x, y, \epsilon)$ and $Friend(y, z, \epsilon)$.

We illustrate the friend-of-friend relation in Figure 1. To identify clusters, the FoF algorithm simply computes these relations. All particles related by the friend-of-friend relation belong to the same cluster. Formally:

Definition: (FoF closure of x). The FoF closure of x over D, denoted by $FoF^+_D(x, \epsilon)$, is a maximal subset of D such that all points y in the subset satisfy $FoF_D(x, y, \epsilon)$. i.e., $FoF^+_D(x, \epsilon) = \{y \in D | FoF_D(x, y, \epsilon)\}$

Definition: (FoF algorithm). The FoF algorithm finds the FoF closure of each particle in a set D with distance threshold ϵ.

Algorithm II.1 Friends-of-Friends (`fof`)

Input: $D \leftarrow \{(pid, x, y, z)\}$ // set of particles
  $\epsilon \leftarrow$ distance threshold
Output: $\{(pid, cid)\}$ // cluster assignments
1: $sidx \leftarrow$ `build_spatial_index`(D)
2: $R \leftarrow \phi$ // resulting cluster assignment
3: **for all** $p \in D$ **do**
4: **if** p not visited **then**
5: $newCid \leftarrow p.pid$
  // find FoF closure of p
6: $N \leftarrow FoF^+(p, \epsilon, sidx)$
  // generate cluster assignment tuples
7: $R \leftarrow R \cup \{(x.pid, newCid) | x \in N\}$
8: **end if**
9: **end for**
10: **return** R

Algorithm II.2 Friend-of-Friend Closure (FoF^+)

Input: $p \leftarrow$ seed of expansion
  $\epsilon \leftarrow$ distance threshold
  $sidx \leftarrow$ spatial index
Output: FoF closure of p
1: **if** p is already visited **then**
2: **return** ϕ
3: **else**
4: mark p visited
5: **end if**
6: $N \leftarrow sidx.$`getNeighbors`(p, ϵ)
7: $R \leftarrow \{p\} \cup N$
8: **for all** $q \in N$ **do**
9: $R \leftarrow R \cup FoF^+(q, \epsilon, sidx)$
10: **end for**
11: **return** R

Algorithm II.1 takes a set D of particles and a distance threshold ϵ as input. Each particle has a unique identifier (pid) and a position vector in R^3. The algorithm outputs a set of cluster assignments. Each cluster assignment is represented as a (particle, cluster) pair, (pid, cid).

The algorithm proceeds as follows. First, a spatial index $sidx$ over D is built to accelerate spatial range queries (i.e.,

finding friends). The index is an object with a single method $getNeighbors(p, \epsilon)$ that returns all particles within ϵ of a given particle p. Then, for each point in D, FoF finds its closure by repeatedly calling FoF^+ (Algorithm II.2). Each particle in the closure of p is assigned a cluster identifier, taken to be the identifier of p as shown in Algorithm II.1. Next, new cluster assignment tuples are generated for each particle in the closure. Once the algorithm visits all particles in D, it returns the final assignments.

FoF is a special case of the DBSCAN [18] clustering algorithm. In DBSCAN, only particles having more than $MinPts$ neighbors can seed a cluster. In FoF, however, the $MinPts$ parameter is zero, i.e., there is no noise. Thus, the final result of FoF will include a large number of clusters with few particles. To filter out such uninteresting clusters, FoF applies simple filter based on the number of points in the cluster. We omit this filtering from Algorithm II.1 for brevity.

The challenge of implementing FoF in a scalable manner is the processing of clusters that cross the domain boundaries of multiple distributed-memory nodes. A further challenge is that for astrophysical applications, there is no characteristic cluster size or mass. The universe is, to a large extent, self-similar in clustering at nearly all scales represented by the simulation. The largest cluster in the simulation will always comprise roughly a few percent of the total number of particles. This makes efficient load balancing across many nodes difficult, since it is hard to assign each node a domain with a statistically equivalent number of clusters.

The most scalable, parallel, distributed-memory FoF algorithm at the moment is *N*tropy [15], [16]. *N*tropy-FoF gives computational astrophysicists the ability to analyze their massive datasets using the same HEC platforms on which the data was generated. However, FoF is only one of many data clustering algorithms. Most are quite complex [19], [20], [21], and substantially more difficult to parallelize. Consequently, we would like to investigate the effectiveness of MapReduce in this regime. By examining an implementation of FoF in MapReduce, we expect to open the door to evaluating this paradigm for clustering problems in general.

III. Distributed Friends of Friends

In order to cluster the massive-scale datasets returned by an N-body simulation, we introduce dFoF, a FoF algorithm for MapReduce-style shared-nothing clusters.

There are two challenges in implementing FoF in a MapReduce-style system. First, the shared-nothing architecture precludes the use of a global index structure for spatial range queries. FoF heavily relies on a global spatial index to quickly retrieve the neighbors of a point. In a shared-memory system, such an index can be built in memory then accessed by multiple cores. To overcome this limitation, there have been proposals to build distributed spatial indexes including the work by Xu *et al.* [22]. In this paper, we investigate a radically different approach. Instead of trying to use a distributed index, we redesign the algorithm to better follow the shared-nothing, parallel query processing approach and not require a global index at all.

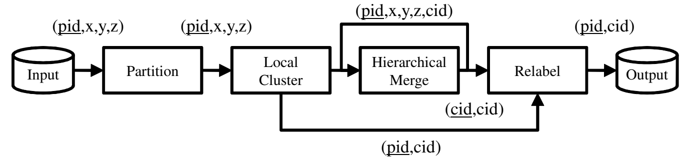

Fig. 2. **High-level algorithm, data flow and schema of dFoF.** dFoF runs in four phases. Each phase exchanges data in the form of standard relation or key-value pairs. Thus, dFoF can be easily implemented in existing data analysis platform for shared-nothing cluster.

Second, we must balance the load across the cluster. Load imbalances can negate the benefits of parallelism [23]. Worse, if a node runs out of memory, the whole MapReduce job fails. To ensure load balance, we must ensure that each partition of the same operation processes its input data in approximately the same amount of time.

In the following subsections, we first present the overall structure of the dFoF algorithm, then introduce two optimizations. The first improves load balance between nodes. The second significantly reduces the runtime of each individual node. We find that both optimizations are critical for the approach to scale.

A. Overall Framework

The basic idea behind dFoF is to partition the input data, perform as much work as possible within individual partitions, then merge the results hierarchically to find clusters that span multiple partitions.

dFoF thus runs in four phases: *Partition*, *Local cluster*, *Hierarchical merge*, and *Relabel*. In the first phase (*Partition*), dFoF hierarchically partitions the input data into chunks of approximately the same size. In the second phase (*Local cluster*), each node runs FoF locally on its assigned data chunk. In the third phase (*Hierarchical merge*), local clustering results are hierarchically merged using the pre-defined partition hierarchy to discover global structures. Finally, in the *Relabel* phase, the local clustering results are relabeled as per the identified global structures.

Figure 2 shows the overall data flow of the algorithm and the schema of the data exchanged between the different phases. We assume that phases exchange data in the form of standard relations or key-value pairs. Thus, the algorithm can be naturally expressed in various MapReduce-style frameworks [5], [6], [7], [10], [24], [25], [26].

We now present the different phases in more detail using a simple 2D example.

1) Partition: The data is first spatially partitioned and distributed. Each partition should be at most the size of available memory in each node in the cluster. The simplest partitioning algorithm is to partition on *space* rather than *data*. Given known boundaries of the simulated space and assuming a uniform data distribution, we can *recursively* split the space into increasingly fine cells until the estimated data size per cell fits in memory. We call these finest-resolution

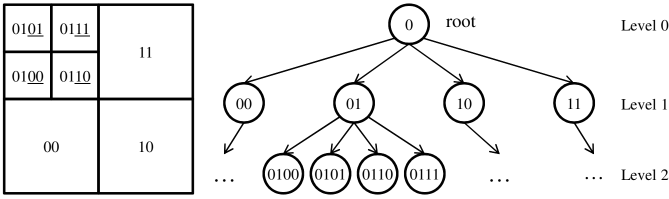

Fig. 3. **Uniform partitioning and Cell hierarchy.** Left figure shows uniform partitioning of 2D space in two levels. Each region at the same level has identical size and is assigned a unique identifier based on its relative position. The upper-left region is further partitioned to demonstrate id assignment. Right figure shows the hierarchy of cells in left figure. The structure of the hierarchy is identical to that of quadtree in 2D.

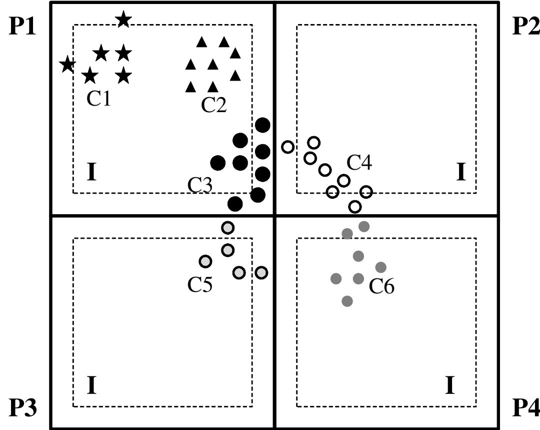

Fig. 4. **Example of merging two partitions.** Points with the same shape are in the same global cluster. Particles with different colors but with the same shape are in different local clusters (e.g., the circles at the middle). Each P_i shows the cell boundary and each I shows the interior region that are excluded during the *Hierarchical merge* phase. After merge, three cluster mappings are generated: (C4,C3), (C5,C3), and (C6,C3). Such mappings are used to relabel local clusters during the *Relabel* phase.

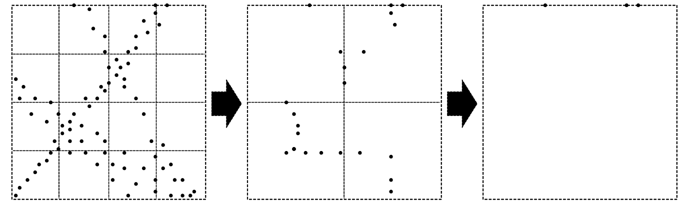

Fig. 5. **Hierarchical merging of SPEATH 8 dataset with uniform partitioning.** Cell boundaries are shown in dashed lines. Only points close to cell boundaries are kept and propagated to next level.

cells the *unit cells*. Overall, the hierarchy of cells is identical to quad-tree and its high-dimensional variants. Such 2D uniform partitioning scheme is illustrated in Figure 3.

Figure 3 also illustrates how an identifier is assigned to a cell. A cell identifier is constructed by concatenating the parent cell id with a 2-bit id determined by the relative position of the child cell within the parent cell. With this approach, the identifier of each cell is unique given the location of a cell within the cell hierarchy. Extending the partitioning to 3D space is straightforward; the hierarchy becomes octree and 3-bit is used to encode relative position.

2) Local Cluster: Once the data is partitioned into unit cells, the original FoF algorithm can run within each cell. Once the local clustering completes, the data is materialized on disk. Only particles at the boundary of each cell continue on to the next phase.

The *Local Cluster* phase takes the longest time in dFoF because FoF is CPU and memory intensive and processes the largest input of all the phases. We further analyze the performance of this phase in Section V.

3) Hierarchical Merge: To identify clusters that span multiple cells, particles near cell boundaries must be examined and merged if they are within distance threshold ϵ.

Figure 4 illustrates such a merging step for four partitions P1 through P4. The outer boxes, P_i, represent the cell boundaries. The inner boxes, I, are ϵ units away from the corresponding edge of the cell. The local clustering step identified a total of six clusters labeled C1 through C6. Each cluster comprises points illustrated with a different color and shape. However, there are only three clusters in this dataset. These clusters are identified during the hierarchical merge step. C3, C4, C5, and C6 are merged because the points near the cell boundaries are within distance ϵ. In Figure 4, C2 does not merge with any other cluster because all points in C2 are sufficiently far from P1's boundary. We can thus safely discard C2 before merging: These points are not needed during the merge phase. In general, we can discard all the points in the larger I regions before merging thus reducing the input data size to the merging algorithm. This reduction is necessary to enable nodes to look at increasingly large spaces or cells during the mergers without running out of memory.

We demonstrate hierarchical merging with uniform partitioning step-by-step in Figure 5. The space is first partitioned into 16 unit cells of equal size. Then, every set of four unit cells are merged to form a larger cell in the middle figure, and finally the larger four cells are merged again to cover the entire space. At each step, only points close to cell boundaries are kept, so only a small fraction of the data is kept after a merge.

At a high-level, the merging algorithm, `mergefof`, discovers merged clusters from closures. By the definition of closure in Section II-A, all particles in the same closure must be in the same cluster; therefore those clusters must be merged together. `mergefof` chooses a new cluster id for such merged clusters, then generates a (`old cluster id`, `new cluster id`) pair for each cluster.

Algorithm III.1 shows the pseudo code of `mergefof`. The algorithm takes the set of particles each labeled with cluster id and produces mappings that each represents a merge. The algorithm runs in three steps. First, Algorithm III.1 finds all closures in D by repeatedly calling FoF^+. Whenever a closure is found, the should-be-merged cluster ids are extracted from the closure and saved for the next step. At the end, we have a nested set of merged cluster ids M.

The first step does not discover all merged clusters because of discarded points like those in I in Figure 4. For example, the dataset in Figure 4 will have three sets,

Algorithm III.1 Merge result of FoF (`mergefof`)

Input: $D \leftarrow \{(pid, cid, x, y, z)\}$ // output from *Local Cluster* or *Hierarchical merge*
$\epsilon \leftarrow$ distance threshold
Output: $\{(old\ cid, new\ cid)\}$
1: $M \leftarrow \phi$// nested set to store cluster ids of closure
2: $R \leftarrow \phi$// output
3: $sidx \leftarrow$ `build_spatial_index`(D)
4: **for all** $p \in D$ **do** // find all closures
5: $N \leftarrow FoF^{+}(p, \epsilon, sidx)$
6: $C \leftarrow \{x.cid | x \in N\}$
7: $M \leftarrow M \cup \{C\}$
8: **end for**
9: **repeat** // find all should-be-merged local clusters
10: **for all** $C \in M$ **do**
11: $C^{+} \leftarrow \{X | X \in M, C \cap X \neq \phi\}$
12: **if** $|C^{+}| > 1$ **then**
13: $M \leftarrow M - C^{+}$
14: $C' \leftarrow \{x | x \in X, X \in C^{+}\}$
15: $M \leftarrow M \cup \{C'\}$
16: **end if**
17: **end for**
18: **until** M does not change
19: **for all** $C \in M$ **do** // produce output
20: $newCid \leftarrow \min C$
21: $R \leftarrow R \cup \{(cid, newCid) | cid \in C\}$
22: **end for**
23: **return** R

$\{\{C1\}, \{C3, C4, C5\}, \{C4, C6\}\}$, in M by the end of the first step. C6 is not merged with C3, C4, C5 because the particles of C4 bridging C6 to C3 are discarded before merging. We can infer such missing links by examining intersections between sets of merged cluster ids. For example, set $\{C3, C4, C5\}$ means C4 should be merged with C3 and C5. Also, $\{C4, C6\}$ tells us that C4 should be merged with C6. Thus, C3, C4, C5, C6 are all merged together because of C4. The second step of Algorithm III.1 does such reasoning by repeatedly finding non-disjoint sets in M and merging them until M contains only disjoint sets.

In the last step, the algorithm simply chooses the lowest original cluster id as the new id of the merged cluster. Finally, it produces outputs.

Algorithm III.1 executes every time child cells under the same parent are merged as we proceed up the cell hierarchy. After each execution, the mappings between clusters that are found are saved. They will be reused during the final *Relabel* phase.

4) Relabel: The last phase of dFoF is *Relabel*. In dFoF, there are two types of relabeling, intermediate and global. The intermediate relabeling occurs at the end of each Hierarchical Merge step. In the intermediate relabeling, cluster ids of all particles passed to the next merge step are relabeled according to the mapping produced by the last `mergefof`. This guarantees that there exists only one mapping for each local cluster id.

The global relabeling occurs at the end of dFoF. The global relabeling first determines the final cluster ids for each local cluster id based on the accumulated output of `mergefof`. It then updates the local cluster assignments from the first phase with the final cluster id information.

Algorithm III.2 Merging of cluster mappings (`mergeGroup`)

Input: $M \leftarrow$ output of `mergefof`
Output: map of *old cluster id* to *new cluster id*
1: sort M in ascending order of old cluster id
2: $M' \leftarrow \phi$
3: **for all** $m \in M$ **do**
4: **if** $m.new \in M'$ **then**
5: $M'[m.old] \leftarrow M'[m.new]$
6: **else**
7: $M'[m.old] \leftarrow m.new$
8: **end if**
9: **end for**
10: **return** M'

Algorithm III.3 Distributed Friends-of-Friends (dFoF)

Input: $D \leftarrow \{(pid, x, y, z)\}$ // set of particles
$\epsilon \leftarrow$ distance threshold
Output: $\{(pid, cid)\}$ // pair of particle id and cluster id
// Partition
1: $grid \leftarrow$ `compute_grid`(D)
2: $level \leftarrow grid.height$
3: $D \leftarrow$ `partition`$(grid[level], D)$
// Local Cluster
4: $L \leftarrow$ `fof`(D, ϵ)
5: $B' \leftarrow \{(pid, x, y, z, cid) | p \in D, q \in L, p.pid = q.pid)\}$
6: $B \leftarrow$ `on_surface`$(grid[level], B')$
7: $level \leftarrow level - 1$
// Hierarchical Merge
8: $M \leftarrow \phi$
9: **while** $level \geq 0$ **do**
10: $B' \leftarrow$ `partition`$(grid[level], B)$
11: $M' \leftarrow$ `mergefof`(B', ϵ)
12: $S \leftarrow$ `on_surface`$(grid[level], B')$
13: $B \leftarrow$ relabel S according to M // intermediate relabel
14: $M \leftarrow M \cup M'$
15: $level \leftarrow level - 1$
16: **end while**
// Relabel
17: $M \leftarrow$ `mergeGroup`(M)
18: $G \leftarrow$ relabel L according to M // global relabel
19: **return** $\{(x.pid, x.cid) | x \in G\}$

Algorithm III.2 shows the pseudo code of determining final cluster id for each local cluster. Because Algorithm III.1 always chooses the lowest identifier as a representative, a local cluster id always maps to a strictly smaller cluster id. Thus, by reversing the order, Algorithm III.2 determines the final cluster id in ascending order and indexes this mapping in memory. If a local cluster id x maps y which was already mapped to z, then x maps to z. Otherwise, x maps to y. The algorithm terminates when it determines all final cluster ids for all local cluster ids. Actual relabeling of data is just an outer join between the data and mapping on cluster id. Particles in non-merging clusters retain their original cluster ids.

Summary: We put four phases of dFoF together in Algorithm III.3. In Algorithm III.3, there are three functions not described in this paper. We present brief descriptions of each.

`compute_grid` generates partitioning information, i.e., a tree of cells (e.g., Figure 3), given a set of points D. In case of uniform partitioning, it does not consult input data at all but uses prior knowledge of the size of the simulated space.

`partition` partitions data according to given partitioning

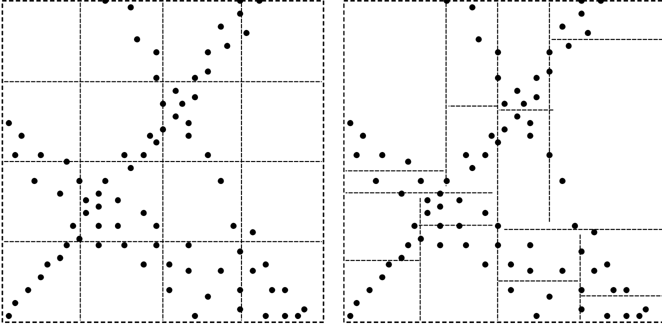

Fig. 6. **Uniform partitioning and Non-uniform partitioning of SPEATH 8 dataset.** Uniform partitioning would generate uneven workloads: two empty partition and five dense partition containing more than 5 points out of 80 points. Data-oriented partitioning, however, produces even workload: each partition is assigned exactly five points.

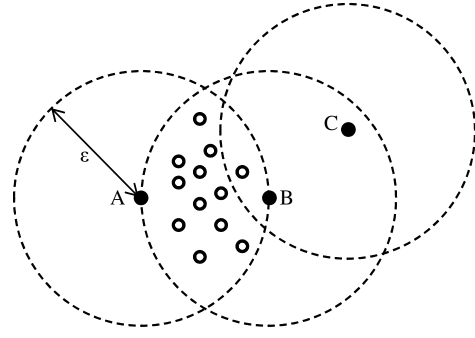

Fig. 7. **Spatial range search in dense region.** In sparse region, the number of objects (i.e., index traversal) is small. For example, C has only two neighbors and the range search is low-overhead. Conversely, the range search becomes high-overhead in dense regions. For example, compared to C, A and B must examine more entries in the index because they have more neighbors than C. Even worse, this high-overhead range search occurs many times while processing the shared neighbors of A and B (the white points).

information. In Algorithm III.3, we simply right shift the grid identifier 3 bits per level of hierarchical merging. Then we partition using the newly computed value.

`on_surface` filters all particles which do not participate in the merging process (i.e., select particles on the surfaces of grids at level i). For example, all particles in I in Figure 4 will be discarded by `on_surface`.

The two relabels at line 13 and line 18 are outer joins as described in previous section. Most of the steps in Algorithm III.3 are set oriented operations thus easily parallelizable in MapReduce style platform.

In following subsections, we discuss two optimizations to balance load and improve performance of local `fof` and `mergefof` using special index traversal.

B. Non-Uniform Data Partitioning

With the uniform space-based partitioning describe above, some nodes may be assigned too much data and may delay or even halt the overall job execution because uniform partitioning does not take into account the real distribution as shown in Table I. Such skew is critical for two reasons. First, the overloaded node delays the overall computation. Second, overloaded nodes may run out of memory causing the entire job to fail in existing distributed job execution environments. The only way to recover is for the system to restart the job using a smaller unit cell.

As an alternate solution, we propose to sample the data to determine the appropriate partitioning information. For this, we first scan the data and collect a random sample. We insert the sampled data into a kd-tree. A kd-tree is a multi-dimensional binary search tree. It is constructed by partitioning data along the median of alternating axes. In our case, we partition data until the estimated size of the data in leaf nodes fits into memory. We choose to use a kd-tree because it is easy to implement and its spatial partitioning nature is well-suited to the underlying shared-nothing architecture (i.e., it generates non-overlapping regions that are also easy to merge). In Figure 6, we compare the uniform and data partitioning schemes. Because we use samples instead of the entire dataset, there is some small discrepancy in the size of the partitions. Also, sampling requires an extra scan over the data, thus adding overhead to the entire job. However, it effectively reduces load skew and the overall task completion time as we show in Section V.

C. Pruning Visited Subtrees

The density-based partitioning described above ensures that each task receives approximately the same amount of data. Even with this optimization, however, each partition can spend a wildly different amount of time processing its input. We demonstrate this effect in Section V, where we measure the variance in task execution times (Figure 12, all plots except for non-uniform/optimized show high variance). This imbalance is due to dense regions taking disproportionately longer to process than sparse regions even when both contain the same number of points.

To understand the challenge related to dense regions, we must look back at the FoF algorithm shown in Figure II-A. A key component of this algorithm is the computation of the closure of a point (line 6, call to FoF^{+}). Computing the closure requires looking-up the friends of a point, then the friends of these friends, etc. In a dense region, the number of such friends at each step can be extremely high as shown in Figure 7. Even worse, the number of lookups is proportional to the density of the region. Astronomy simulation data is especially challenging in this respect compared to other datasets because the density of galaxies is often orders of magnitude greater than that of other regions. To address this challenge, we optimize the closure computation as follows.

The original FoF algorithm constructs a spatial index over all points to speed up friend look ups. We modify this data structure to keep track of the parts of the subtree where all data items have already been visited. For each node in the tree (leaf node and interior node), we add a flag which sets to $true$ when all points within the subtree rooted at the node have been returned as a result of one or more previous friends lookups. The algorithm can safely skip such flagged subtrees

Algorithm III.4 Range search with pruning visited subtree

Input: $root \leftarrow$ search root node
$target \leftarrow$ search origin
$\epsilon \leftarrow$ distance threshold
Output: set of objects within ϵ from $target$
1: **if** $root.visitedAll$ is **true then**
2: **return** ϕ // skip this subtree
3: **end if**
4: $R \leftarrow \cdots$ // normal range search for $root$
// update book keeping information
5: **if** entire data under $root$ marked $visited$ **then**
6: $root.visitedAll \leftarrow$ **true**
7: **end if**
8: **return** R

because all data items within them have already been covered by previous lookups. By nature of spatial indexes, points in a dense region are clustered under the same subtree and are thus quickly pruned. With this approach, the index shrinks as over time as the previously-visited subtrees are pruned.

Because this optimization requires only one flag per node in the spatial index, it imposes a minimal overhead. Furthermore, the flag can be maintained while processing range lookups. In Algorithm III.4, we show the modified version of range search in this modified index structure. Line 5 is dependent on the type of spatial index. For a binary tree, the condition can be evaluated by checking the flags of the child nodes and the data item assigned to the $root$ node. For R-tree-style indexes that does not have data items in the internal nodes, the flag of an internal node is set by checking flags of its child nodes and the flag of a leaf node is set by checking whether all data items in the leaf node have been visited.

We apply this optimization both during the local clustering and the merging phases.

IV. Implementation

We implemented dFoF in C# using DryadLINQ [10] the programming interface to Dryad [7]. Dryad is a massive-scale data processing system similar to MapReduce yet offers more flexibility because its vertices are not limited to map or reduce operations.

For the spatial index, we chose to use a kd-tree [27] because of its simplicity. We implemented both a standard version of the kd-tree and the optimized version presented in Section III-C. We used the kd-tree both for local FoF computations and also when planning the initial data partitions in the *Partition* phase of the dFoF algorithm.

The entire code base for the experiments is written in about 3000 lines of code including both the basic and optimized kd-tree implementations, data serialization methods, brief documentation, and boiler plate code.

Similar to Clustera [28] and Pig [24], DryadLINQ offers relational style operators such as filters and joins. As in Pig [24], queries are expressed using a combination of procedural and declarative statements. We leveraged DryadLINQ's pre-defined operators and standard API as much as possible. However, we had to implement `fof`(), `mergefof`(), and `mergeGroup`() as user-defined operators. These three UDFs are single threaded, and thus do not exploit the presence of extra cores during the execution.

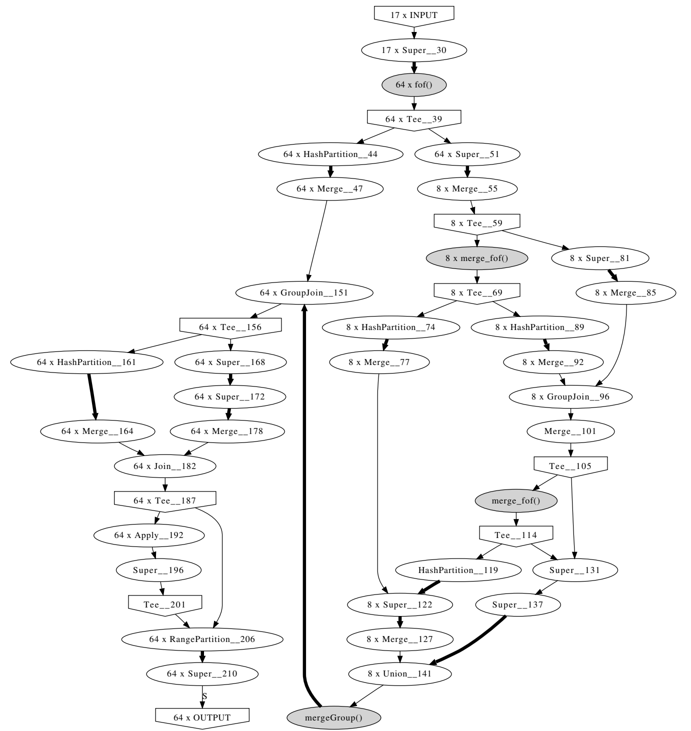

Fig. 8. **Example Query Plan.** Nodes in grey color are running user-defined functions. *Partition* and *Merge* vertex pair represents repartitioning of data. *Tee* vertices replicate the input data and feed them to multiple down stream vertices. *Super* vertices are a chain of operators which could be pipelined in memory.

In runtime, DryadLINQ automatically generates a dryad query plan of dFoF then deploys the plan to the underlying Dryad cluster. An example static plan with two levels of hierarchical merging is shown in Figure 8. In the query plan, each vertex represents a computational task processing data in parallel. Like the original MapReduce, this task is implemented as a process at the operating system level. The connected vertices communicate through a compressed file stored in a distributed file system. The prefix number in each vertex label of this graph represents the number of partitions; i.e., the maximum degree of parallelism of that vertex. For example, the first user-defined vertex `fof()` could run completely in parallel in a 64 nodes cluster while `mergeGroup()` always runs on a single node.

Unlike the original MapReduce where a job is a coordinated pair of map, reduce primitives, a job in Dryad is the entire query plan (Figure 8). The equivalent program would require many map-reduce jobs.

V. Evaluation

In this section, we evaluate the performance and scalability of the dFoF clustering algorithm using two real world datasets. We use an eight-node cluster running Windows Server 2008 Datacenter edition Service Pack 1. All nodes are connected to the same gigabit ethernet switch. Each node is equipped

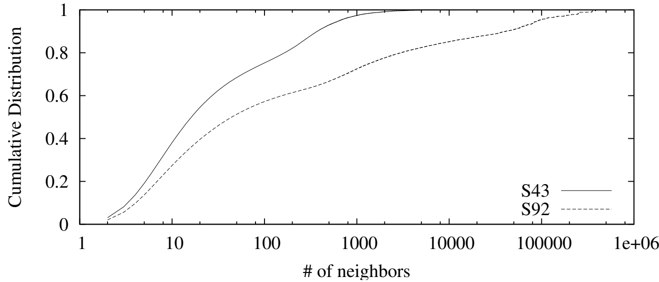

Fig. 9. **Cumulative distribution of the number of neighbors.** S92 is significantly skewed than S43. In S92, 28% of the particles have over 1k neighbors and 15% have over 10k, and 5% have 100k. The maximum number of neighbors for S43 and S92 is 10494 and 387136 respectively. Note that x axis is in log scale.

Percentile	25	50	75	90	99	99.9	100
S43	3803	5000	8517	16118	58442	288533	381440
S92	4000	5903	11514	27065	254968	1065458	5642100

TABLE I
Size of cluster at given percentile.

with dual Intel Xeon E5335 2.0GHz quad core CPU, 8GB RAM, and two 500GB SATA disks configured as RAID 0. Each Dryad process requires 5GB RAM to be allocated or it is terminated. This constraint helps quickly detect unacceptable load imbalance. Note that we tuned neither the hardware nor software configurations other than implementing the algorithmic optimizations that we described. Our goal is to show improvements in the relative numbers rather than try and show the best possible absolute numbers.

A. Dataset

We take two snapshots, S43 and S92, from a large-scale astronomy simulation. Each snapshot contains 906 million particles occupying 43 GB in uncompressed binary format. The simulation volume is a periodic box roughly 80 million light-years on a side and models the evolution of cosmic structure from about 100,000 years after the Big Bang to the present day. The simulation itself was only about 20% complete at the time of submission. Therefore we use two relatively early snapshots: S43 and S92 respectively correspond to 580 million years and 1.24 billion years after the Big Bang. This particular simulation will model the evolution and interaction of several thousand galaxies with unprecedented accuracy and spatial resolution. The entire run will take of order 10 million core-hours of computing, and is currently running on 2048 compute cores of the Cray XT3 system at the Pittsburgh Supercomputing Center [29].

Each particle has a unique identifier and 9 to 10 additional attributes such as coordinates, velocity vector, mass, gravitational potential stored as 32-bit real numbers. The data is preloaded into the cluster and hash partitioned on the particle identifier attribute. Each partition is also compressed using the GZip algorithm. Dryad can directly read compressed data and decompress on-the-fly as computational task reads.

For this particular simulation, astronomers set two parameters for FoF: the distance threshold (ϵ) is 0.000260417, and the cluster size threshold is 3000. Both datasets require at least two levels of hierarchical merging.

As the simulation progresses, the Universe becomes increasingly structured (i.e., more galaxies and stars are created over time). Thus, S43 has fewer clusters than S92: 890 and 3496 clusters respectively. Figure 9 shows the cumulative distribution of the number of neighbors for particles in clusters comprised of more than 3000 particles (i.e., the number of returned particles per spatial index lookup). The figure shows that S92 has significantly denser regions than S43. The densest region in S43 has 10494 particles. In S92, more than 10 million particles are in denser region than the densest region in S43. We also show the cluster size at specific percentiles for the datasets in Table I. Compared to S43, S92 contains more larger clusters and some of them are gigantic (consisting of over 5 million particles!). Ideally, the structure of data should not affect runtime of the algorithm so that scientists can examine and explore snapshots taken at any time of simulation in near constant time.

In following subsections, we evaluate how the partitioning scheme and spatial index implementation affect the performance of dFoF algorithm. Then, we evaluate scalability of the dFoF algorithm by varying the number of nodes in the cluster and the size of the input data.

B. Performance

In this section, we use the full eight-node cluster and vary the partitioning scheme and spatial index implementation. For the partitioning scheme, we compare deterministic uniform partitioning (Uniform) described in Section III-A.1 and dynamic partitioning (Non-uniform) described in Section III-B. We also compare an ordinary kd-tree implementation (Normal) to the optimized version (OPT) described in Section III-C. We repeat all experiments three times except the Uniform partitioning using Normal kd-tree implementation because it takes over 20 hours to complete. For Non-uniform partitioning, we use a sample of size 0.1%. We show the total runtime including sampling and planning times. There is no reason for using small sample except avoiding high overhead of planning. As the results in this section show, even such small samples work well for this particular datasets.

Figure 10 shows a summary of average total run times for each variant of the algorithm and each dataset. For dataset S43, which has less skew in the particle distribution, all variants complete in 70 minutes. However, if there is high skew (i.e., more structures as in S92), the normal kd-tree implementation takes over 20 hours to complete even though the optimized version still runs in 70 minutes. The missing bar for Uniform-OPT in S92 is because a node ran out of memory while processing a specific data partition and caused the failure of the entire job. We discuss this case in more detail shortly. Overall, the dynamic partitioning with optimized kd-tree performs the best on both datasets. It thus has good

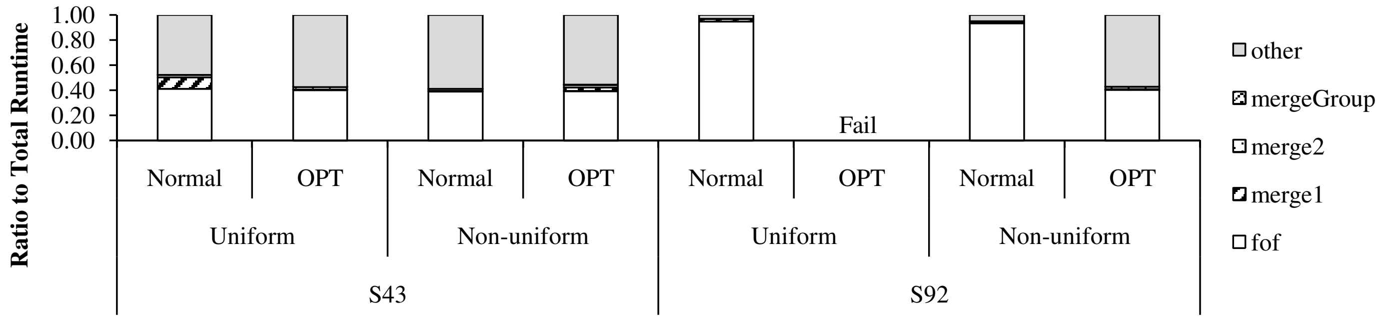

Fig. 11. **Ratio of UDF phase runtime to total runtime.** Average of three executions except jobs take longer than 20 hours. The initial `fof()` takes more than 40% of total runtime. All other UDFs took less than 4% of total runtime. Other represents time to take to run all white vertices in Figure 8. Overall, `fof()` is the bottleneck and completely dominates when there is high-skew in data and ordinary kd-tree is used.

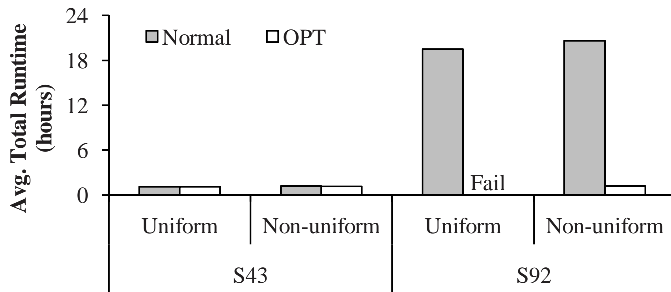

Fig. 10. **Average Job Completion Time.** Average of three executions except jobs take longer than 20 hours. Missing bar is due to failure caused by out-of-memory error at initial `fof()`.

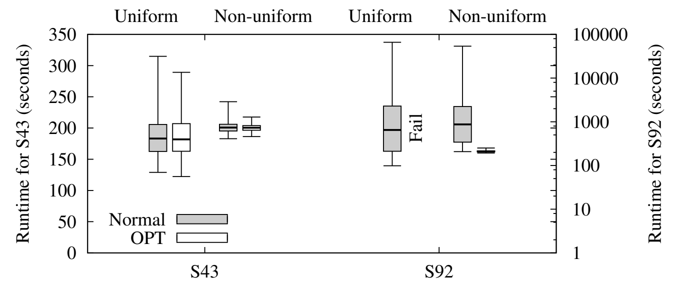

Fig. 12. **Distribution of FoF runtime per partition.** Uniform partitioning yields higher variance in runtime than Non-uniform partitioning. FoF with optimized index traversal runs orders of magnitude faster than FoF with normal implementation in S92 dataset. Note that the y axis for S92 is in log scale.

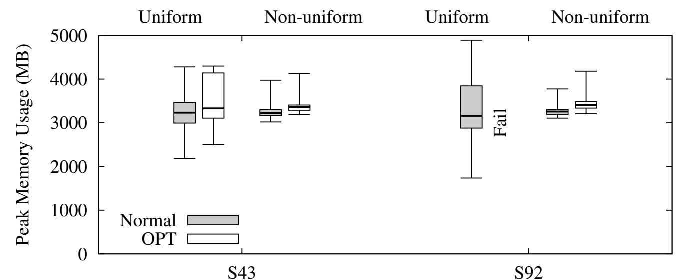

Fig. 13. **Distribution of FoF peak memory utilization per partition.** Uniform partitioning yields higher variance in peak memory utilization than Non-uniform partitioning. Such high variance caused crashing one of the partition with optimized index traversal. The optimized kd-tree index traversal has higher memory profile than normal implementation.

performance independent of the underlying structure of the data.

In Figure 11, we further break down the time into the runtime of four vertices in Figure 8 while running three user-defined functions (`fof,mergefof,mergeGroup`) and other standard DryadLINQ operators. We show ratio of their runtime to total runtime in Figure 10. The local clustering, `fof`, takes more than 40% of total runtime in all cases and even completely dominates when there is high-skew in the data and a normal kd-tree is used. All other user-defined functions account for less than 4% of total runtime. All other standard operators account for over 50%, but the number is a sum of more than 30 operators. Thus, the local clustering phase is the bottleneck of dFoF algorithm.

In the following subsections, we report results only for the dominant `fof` phase of the computation and analyze the impact of different partitioning schemes and different spatial index implementations.

In Figure 12 and Figure 13, we measure runtime and peak memory utilization of the `fof` phase for each partitioning scheme and plot the quartiles as well as the min and max values. We choose runtime and peak memory utilization because low variance in runtime represents a balanced computational load and low variance in peak memory represents balance in both computation and data across different partitions.

1) Partitioning Scheme: In both Figure 12 and Figure 13, Non-uniform partitioning shows a tighter distribution in runtime and peak memory utilization than uniform partitioning. With uniform partitioning, the worst case scenario happens when we try the optimized kd-tree implementation. Due to high data skew, one of the partitions runs out of memory causing the entire job to fail. This does not happen with normal kd-tree and uniform partitioning because the optimized kd-tree has a larger memory footprint. This specific result is somewhat of a coincidence because the maximum per-partition memory utilization for snapshot S92 is almost exactly 5 GB causing OPT to crash while NON-OPT completes. In general, however, a uniform data distribution is not scalable and could not serve

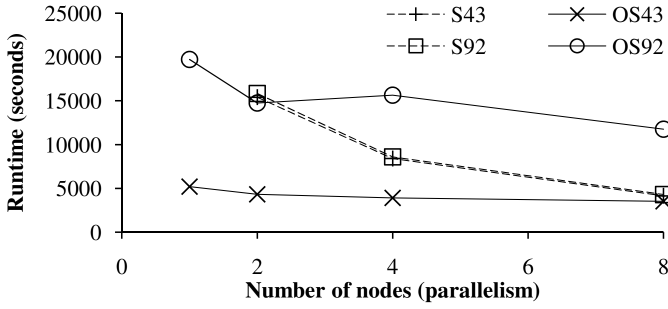

Fig. 14. **Speedup.** dFoF runtime for each datasets with varying number of nodes . dFoF speedup almost linear as the number of nodes increases. OS43 and OS49 are the result of OpenMP implementation of FoF with varying degree of parallelism.

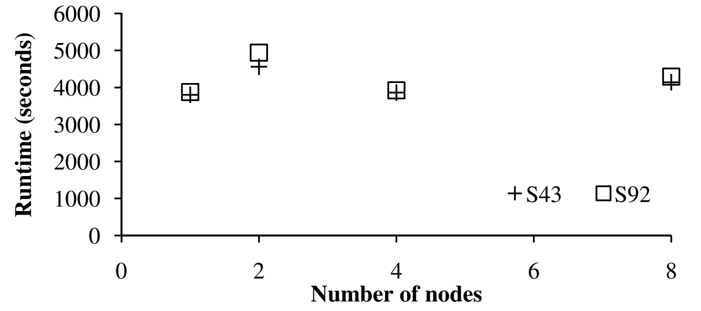

Fig. 15. **Scaleup.** Runtime of dFoF as increasing data size proportional to the number of nodes. Except two node case where scheduling overhead pronounced, dFoF scales up in linear.

to cluster snapshots later than 92 (there are roughly 30 more snapshots following S92 in this simulation).

In contrast, the distribution of peak memory utilization with Non-uniform partitioning does not significantly fluctuate as data distribution changes in Figure 13. We expect that with higher sampling rate, we could even further reduce the variation at the cost of modestly increased planning time.

In summary, Non-uniform partitioning is worth the extra scan over the entire dataset.

2) Optimized Index Traversal: As Figure 12 shows, dFoF with the optimized index (Section III-C) significantly outperforms Normal implementation especially when there is significant skew in the particle distribution. Thanks to pruning of visited subtrees, the runtime for S92 remains almost the same as that for S43. However, the optimization does not come for free. Due to extra flag tracking, the optimization requires slightly more memory than the ordinary implementation as shown in Figure 13. The raised memory requirement could be alleviated by more efficient implementation techniques such as keeping a separate bit vector indexed by node identifier or implicitly constructing kd-tree on top of an array rather than keeping pointers to children per node.

Summary: Both partitioning and spatial index influence performance of the algorithm. Sampling-based non-uniform partitioning more evenly distributes data and computation across the cluster. Such advantage is more pronounced when the data has large skew and the extra scan for sampling pays off. The optimized index traversal greatly improves performance when there is a large skew while the speed-up would depend on the degree of skew in the cluster sizes in general.

C. Scalability

We evaluate the scalability of the algorithm by measuring speedup and scaleup. In this section, we only use non-uniform partitioning and optimized kd-tree. We vary the number of nodes and redistribute input data only to the participating nodes. All reported results are averages of three runs. The standard deviation is less than 1%.

Figure 14 shows runtime of dFoF for each dataset by increasing the number of nodes from 2 to 8. Speedup measures how much faster a system can process the same data if it is allocated more nodes. Ideal speedup is identical to the number of nodes N, i.e., N times faster than single-node execution. For some reason, Dryad does not schedule individual vertices after 6 hours for a single-node case[1]. Thus, we do not present the number for the single-node case. For both datasets, the runtime of the dFoF halves as we increase the number of nodes.

Figure 15 shows scaleup by increasing the number of nodes from 1 to 8. Scaleup measures how a system handles a data size that increases in proportion with the number of nodes. Ideal scaleup is flat line of 1. To vary the size of data, we subsample the S43 and S92 dataset. For 4 nodes and 8 nodes, the scale up is close to ideal, 0.99 and 0.91 respectively, while 2 nodes marked 0.83 and 0.78. We investigate the 2 node case and found that the size of the subsampled dataset is near border line of requiring additional hierarchical merging. Thus, each partition is underloaded and completes very quickly. Such fast completion eventually overloads the job scheduler and yields poor scaleup for two nodes.

Overall, considering suboptimal hardware configuration, the scalability of dFoF is reasonable. We expect that adding more disk spindles controlled by the MapReduce system rather than the RAID controller would improve the shape of both speedup and scaleup curves.

D. Compared to OpenMP implementation

Astronomers currently use a serial FoF implementation which has been moderately parallelized using OpenMP [30], a means of scheduling computation across multiple threads that all share the same address space. OpenMP was designed as a means of parallelizing programs that were originally written in serial, and is an example of "agenda" parallelism. The two biggest drawbacks of OpenMP are 1) non-trivial serial portions of code are likely to remain, thereby limiting scalability by Amdahl's Law; 2) the target platform must be shared memory. The serial aspects of this program are state-of-the-art in terms of performance — they represent an existing program that has been performance-tuned by astrophysicists for over 15 years. It uses an efficient kd-tree implementation to perform spatial

[1]At the time of submission, the development team is investigating this issue.

searches, as well as numerous other performance enhancements. The OpenMP aspects are not performance-oriented, however. They represent a quick-and-dirty way of attempting to use multiple processing cores that happen to be present on a machine with enough RAM to hold a single snapshot.

The CSE cluster nodes represent a common cost-efficient configuration for modern hardware: roughly 8 cores per node and one to two GB of RAM per core. Our test dataset is deliberately much larger than what can be held in RAM of a single one of these nodes. The astrophysics FoF application must therefore be run on an unusually large shared-memory platform. In our case, the University of Washington Department of Astronomy owns a large shared-memory SGI Altix system with 128 GB of RAM, 16 Opteron 880 single-core processors running at 2.4 GHz, and 3.1 TB of RAID 6 SATA disk. This system has 8 GB per processor core. Furthermore, to purchase a shared-memory system with 128 GB of shared memory would be much more expensive than 16 of our 8 GB CSE cluster nodes. Consequently, our goal is to at least match the performance of the astrophysics FoF running on the Altix with our Dryad version running on 8 CSE nodes (i.e., 64 GB of total RAM — just barely large enough to fit the problem in memory). If we do this, then we have demonstrated that the MapReduce paradigm is an effective means of leveraging cheaper distributed-memory platforms for clustering calculations.

In order to normalize serial performance, we ran the existing astrophysics FoF application on a smaller dataset on both the Altix system and our CSE cluster. The dataset was small enough to fit completely into RAM on a single CSE node. The Altix took 61.4 seconds to perform the same analysis that required 34.8 seconds on a CSE cluster node excluding I/O. We do not include I/O in our normalization because the Altix's storage hardware is still representative of the current state-of-the-art, only its CPUs are dated.

Running the astronomy FoF algorithm on the Altix for our test dataset step S43 (with the same parameters as our CSE cluster runs) took 5202 seconds in total—only 1986 of this was actual FoF calculation, the rest was I/O. In comparison, our Dryad version would likely have taken an estimated 30,000 seconds, extrapolated from our optimized Dryad 2-node run assuming ideal scalability. Also, remember that a core in our Dryad cluster is also 76% faster than a processor in the Altix, which means that this difference in performance is even more dramatic. We interpret this differential as arising from the difference in RAM between the two systems. The Altix has more than enough RAM to hold the entire dataset in memory, so it is not surprising that this capability allowed it to beat the Dryad implementation for S43.

However, things get much more interesting for snapshot S92. The particle distribution in S92 is more highly clustered than S43, meaning that the clusters are larger on average and there are more of them. In this case, the astrophysics FoF takes quite a bit longer: 16763 seconds for the FoF computation itself and 19721 for the entire run including I/O. This is vs. roughly 30,000 seconds for a serial Dryad run of the same snapshot.

One can also see the effect of S92's higher clustering on the OpenMP scalability. The OpenMP version is not efficient for snapshots with many groups spanning multiple thread domains. This is because multiple threads may start tracking the same group. When two threads realize they are actually tracking the same group, the one gives up entirely but does not contribute its already-completed work to the survivor. While this is another optimization that could be implemented in the OpenMP version, astronomers have not yet done so. This effect can be seen in Figure 14.

Since our Dryad version performed similarly on both snapshots, this seems to indicate that it is limited mostly by I/O. The I/O is, however, scalable. The advantage of our implementation can be seen when we run on higher numbers of nodes. This allows us to match the performance of the astrophysics code on S43 and to substantially outperform it for S92. Consequently, we have achieved our goal of reducing time-to-solution while at the same time using less expensive hardware than the current state-of-the-art in astrophysics.

VI. Discussion

In this section, we discuss our experience of developing clustering algorithm in a MapReduce style analysis platform. Our experience of using MapReduce system, Dryad, generally agrees to Pavlo *et al.* [31].

Once we designed dFoF, the implementation of the algorithm was quite straightforward. Our user-defined codes are nicely integrated into the standard API of DryadLINQ. The semi-declarative way of defining data processing greatly reduces the amount of code to coordinate the distributed execution of dFoF. Also, well-defined standard operators help us find bugs early in development. However, we spent most of our development time trying to understand the non-standard features (i.e., non-relational APIs) provided by DryadLINQ. It is also tedious to debug program crashes while running on the cluster. Overall, however, it is quite pleasant to use MapReduce system and we expect that the documentation and debugging support will become more mature in near future.

We wished for the following features to the MapReduce style system while implementing and evaluating dFoF.

First, an efficient data sampling mechanism should be provided. Several existing systems already leverage sampling while sorting and partitioning [10], [24]. We found that such sampling is more than necessity to implement custom partitioning algorithms for complex data types such as spatial data. Sampled data could be treated as a metadata, cached at loading time and shared by many other jobs [32].

Second, we would like better support for iterative tasks. Clustering algorithms typically runs in multiple stages. A common practice for these types of algorithms is to have loop coordination code in MapReduce client programs. We call such client program driver program. In existing implementations, the support of this type of application is limited. The driver program may leverage runtime statistics of a job to optimize execution in many ways. For example, if the size

of data becomes small enough to fit in a single machine, the remaining job could run on a single node rather than paying the high initiation cost for the remaining short tasks. Another example is testing loop termination conditions. Instead of spawning extra MapReduce jobs to evaluate termination conditions, MapReduce runtime may provide a way to embed such tasks as part of loop-body execution. If the termination condition is defined as the size or number of tuples to process, they could be easily piggy-backed in final runtime statistics of loop-body execution.

VII. Related Work

A. Scientific Data Analysis

Scientists usually implement analysis tasks in high-level languages such as IDL, C/C++ and Fortran and parallelize them using OpenMP [30] and/or MPI [33]. They also prefer large shared-memory platforms over a shared-nothing cluster because of performance concerns and well as ease of programming. In this paper, we take a completely different path to implement scientific data analysis using a shared-nothing cluster and a different programming model. Programming shared-nothing clusters has been gaining more attention in both academia and industry. Over the past few years, several distributed job execution engines have been proposed [5], [6], [7], [28] followed by high-level job description languages [10], [24], [25], [26], [34].

Chu *et al.* investigated how to leverage such emerging platforms to run popular machine-learning algorithms and gain linear scalability [35]. This research initiated an open-source project Mahout [36], a library of machine learning algorithms running on Hadoop. However, Mahout does not implement any density-based clustering algorithms yet. Papadimitrious *et al.* implemented co-clustering algorithm using Hadoop and evaluated performance and scalability [37]. All algorithms including dFoF shares the same high-level software architecture proposed by Chu *et al.*.

B. Distributed DBSCAN

The friends-of-friends (FoF) [38] clustering algorithm is one of the two available distributed algorithms to analyze N-body simulation results. FoF is a special case of the DBSCAN algorithm [18] where its density threshold parameter, $MinPts$ is zero.

There is a huge body of research work in parallelized and distributed DBSCAN. We briefly summarize previous work in three approaches. The first approach is to build a distributed spatial index on a shared-nothing cluster and use the index when merging local clustering results [22]. However, dFoF avoids building such a global index to fully leverage features of the MapReduce framework. The second approach is approximation by using clustering on local models [39] or using samples to reduce the size of data or the number of spatial index lookups [40]. dFoF processes data in its entirety and produces an identical result to DBSCAN. The last approach is to increase the throughput of spatial index lookups via distributed asynchronous protocols [41]. dFoF increases the throughput by pruning spatial index based on the unique constraint of the problem.

Another key difference of dFoF from aforementioned works is that dFoF is designed and implemented to run on a data analysis platform rather than stand alone parallel or distributed application.

Finally, previous works have been evaluated against relatively small and often synthetic datasets; their datasets have, at most, roughly one million objects in two dimensions. In this paper, we evaluate the performance and scalability with real datasets of substantially larger scale and more extreme skew.

VIII. Conclusion

Science is rapidly becoming data management problem. Scaling existing data analysis techniques is very important to expedite the knowledge discovery process. In this paper, we design and implement a standard clustering algorithm to analyze astrophysical simulation output using a popular MapReduce style data analysis platform. Through extensive evaluation using two real datasets and a small eight-node lab-size cluster, we show that our proposed dFoF algorithm achieves near-linear scalability and performs consistently regardless of data skew. To achieve such performance, we describe non-uniform data partitioning techniques based on sampling and optimized spatial range queries. As a future work, we will extend dFoF to the more general DBSCAN algorithm and scale up non-trivial scientific data analysis algorithms.

Acknowledgments

It is a pleasure to acknowledge the help we have received from Tom Quinn, both during the project and in writing this publication. Simulations “Cosmo25” and “Cosmo50” were graciously supplied by Tom Quinn and Fabio Governato of the University of Washington Department of Astronomy. The simulations were produced using allocations of advanced NSF–supported computing resources operated by the Pittsburgh Supercomputing Center, NCSA, and the TeraGrid.

We are also grateful to the Dryad and DryadLINQ teams, MSR-SV, and Microsoft External Research for providing us with an alpha release of Dryad and DryadLINQ and for their support in installing and running this software.

This work was funded in part by the NASA Advanced Information Systems Research Program grants NNG06GE23G, NNX08AY72G, NSF grants IIS-0713123, CNS-0454425, and gifts from Microsoft Research. Magdalena Balazinska is also sponsored in part by a Microsoft Research New Faculty Fellowship.

References

[1] “Oracle,” http://www.oracle.com/database.
[2] “Db2,” http://www.ibm.com/db2/.
[3] “Teradata,” http://www.teradata.com/.
[4] “Greenplum database,” http://www.greenplum.com/.
[5] J. Dean and S. Ghemawat, “MapReduce: simplified data processing on large clusters,” in *Proc. of the 6th OSDI Symp.*, 2004.
[6] “Hadoop,” http://hadoop.apache.org/.

[7] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, “Dryad: Distributed data-parallel programs from sequential building blocks,” in *Proc. of the 2007 EuroSys Conf.*, 2007, pp. 59–72.

[8] M. Stonebraker, J. Becla, D. DeWitt, K.-T. Lim, D. Maier, O. Ratzesberger, and S. Zdonik, “Requirements for science data bases and SciDB,” in *Fourth CIDR Conf. (perspectives)*, 2009.

[9] R. Xu and I. Wunsch, D., “Survey of clustering algorithms,” *Neural Networks, IEEE Transactions on*, vol. 16, no. 3, pp. 645–678, May 2005.

[10] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey, “DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language,” in *Proc. of the 8th OSDI Symp.*, 2008.

[11] V. Springel, S. D. M. White, A. Jenkins, C. S. Frenk, N. Yoshida, L. Gao, J. Navarro, R. Thacker, D. Croton, J. Helly, J. A. Peacock, S. Cole, P. Thomas, H. Couchman, A. Evrard, J. Colberg, and F. Pearce, “Simulations of the formation, evolution and clustering of galaxies and quasars,” *NATURE*, vol. 435, pp. 629–636, June 2005.

[12] “About the Blue Waters project.” [Online]. Available: http://www.ncsa.illinois.edu/BlueWaters/

[13] M. Davis, G. Efstathiou, C. S. Frenk, and S. D. M. White, “The evolution of large-scale structure in a universe dominated by cold dark matter,” *Astrophysical Journal*, vol. 292, pp. 371–394, May 1985.

[14] D. Reed, J. Gardner, T. Quinn, J. Stadel, M. Fardal, G. Lake, and F. Governato, “Evolution of the mass function of dark matter haloes,” *Monthly Notices of the Royal Astronomical Society*, vol. 346, pp. 565–572, Dec. 2003.

[15] J. P. Gardner, A. Connolly, and C. McBride, “Enabling rapid development of parallel tree search applications,” in *Proc. of the 2007 CLADE Symp.*, 2007.

[16] ——, “Enabling knowledge discovery in a virtual universe,” in *Proc. of the 2007 TeraGrid Symp.*, 2007.

[17] S. R. Knollmann and A. Knebe, “AHF: Amiga’s Halo Finder,” *Astroph. J. Suppl.*, vol. 182, pp. 608–624, June 2009.

[18] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, “A density-based algorithm for discovering clusters in large spatial databases with noise,” in *Proc. of the 2nd KDD Conf.*, 1996, pp. 226–231.

[19] D. H. Weinberg, L. Hernquist, and N. Katz, “Photoionization, Numerical Resolution, and Galaxy Formation,” *Astroph. J.*, vol. 477, pp. 8–+, Mar. 1997.

[20] “An Overview of SKID,” http://www-hpcc.astro.washington.edu/tools/skid.html.

[21] J. M. Gelb and E. Bertschinger, “Cold dark matter. 1: The formation of dark halos,” *Astroph. J.*, vol. 436, pp. 467–490, Dec. 1994.

[22] X. Xu, J. Jäger, and H.-P. Kriegel, “A fast parallel clustering algorithm for large spatial databases,” *Data Min. Knowl. Discov.*, vol. 3, no. 3, pp. 263–290, 1999.

[23] D. DeWitt and J. Gray, “Parallel database systems: the future of high performance database systems,” *Communications of the ACM*, vol. 35, no. 6, pp. 85–98, 1992.

[24] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, “Pig latin: a not-so-foreign language for data processing,” in *Proc. of the SIGMOD Conf.*, 2008, pp. 1099–1110.

[25] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou, “Scope: easy and efficient parallel processing of massive data sets,” in *Proc. of the 34th VLDB Conf.*, 2008, pp. 1265–1276.

[26] “Cascading,” http://www.cascading.org/.

[27] V. Gaede and O. Günther, “Multidimensional access methods,” *ACM Comput. Surv.*, vol. 30, no. 2, pp. 170–231, 1998.

[28] D. J. DeWitt, E. Paulson, E. Robinson, J. Naughton, J. Royalty, S. Shankar, and A. Krioukov, “Clustera: an integrated computation and data management system,” in *Proc. of the 34th VLDB Conf.*, 2008, pp. 28–41.

[29] “Bigben,” http://www.psc.edu/machines/cray/xt3/.

[30] L. Dagum and R. Menon, “Openmp: An industry-standard api for shared-memory programming,” *Computing in Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[31] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. R. Madden, and M. Stonebraker, “A comparison of approaches to large scale data analysis,” in *Proc. of the SIGMOD Conf.*, 2009.

[32] A. Silberstein, B. F. Cooper, U. Srivastava, E. Vee, R. Yerneni, and R. Ramakrishnan, “Efficient bulk insertion into a distributed ordered table,” in *Proc. of the SIGMOD Conf.*, 2008, pp. 765–778.

[33] M. Snir and S. Otto, *MPI-The Complete Reference: The MPI Core*, 1998.

[34] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan, “Interpreting the data: Parallel analysis with Sawzall,” *Scientific Programming*, vol. 13, no. 4, 2005.

[35] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun, “Map-reduce for machine learning on multicore,” in *NIPS*, B. Schölkopf, J. Platt, and T. Hoffman, Eds., 2007, vol. 19.

[36] “Apache Mahout,” http://lucene.apache.org/mahout/.

[37] S. Papadimitriou and J. Sun, “Disco: Distributed co-clustering with map-reduce: A case study towards petabyte-scale end-to-end mining,” in *Proc. of the 8th ICDM Conf.*, 2008, pp. 512–521.

[38] M. Davis, G. Efstathiou, C. S. Frenk, and S. D. M. White, “The evolution of large-scale structure in a universe dominated by cold dark matter,” *Astroph J*, vol. 292, pp. 371–394, May 1985.

[39] E. Januzaj, H.-P. Kriegel, and M. Pfeifle, “Scalable density-based distributed clustering,” in *Proc. of the 8th PKDD Conf.*, ser. LNCS, vol. 3202, 2004, pp. 231–244.

[40] Z. Aoying, Z. Shuigeng, C. Jing, F. Ye, and H. Yunfa, “Approaches for scaling dbscan algorithm to large spatial databases,” *Journal of Computer Science and Technology*, pp. 509–526, 2000.

[41] D. Arlia and M. Coppola, “Experiments in parallel clustering with dbscan,” in *Euro-Par 2001 Parallel Processing*, ser. LNCS, vol. 2150, 2001, pp. 326–331.