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. Scientists' ability to generate and collect massive-scale datasets is increasing. As a result, constraints in data analysis capability rather than limitations in the availability of data have become the bottleneck to scientific discovery. MapReduce-style platforms hold the promise to address this growing data analysis problem, but it is not easy to express many scientific analyses in these new frameworks. In this paper, we study data analysis challenges found in the astronomy simulation domain. In particular, we present a *scalable*, *parallel* algorithm for *data clustering* in this domain. Our algorithm makes two contributions. First, it shows how a clustering problem can be efficiently implemented in a MapReduce-style framework. Second, it includes optimizations that enable *scalability*, even in the presence of skew. 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 an 8-node cluster, our algorithm can process even a highly skewed dataset 17 times faster than the conventional implementation and offers near-linear scalability. Our approach matches the performance of an existing hand-optimized implementation used in astrophysics on a dataset with little skew and significantly outperforms it on a skewed dataset.

1 Introduction

Advances in high-performance computing technology are changing the face of science, particularly in the computational sciences that rely heavily on simulations. Simulations are used to model the behavior of complex natural systems that are difficult or impossible to replicate in the lab: subatomic particle dynamics, climate change, and, as in this paper, the evolution of the universe. Improved technology — and improved access to this technology — are enabling 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 individual snapshots in time ranging from 10s to 100s of terabytes. The challenge for scientists now lies in how to analyze the massive datasets output by these simulations. In fact, further increases in the scale and resolution of these simulations — to adequately model, say, star formation — are constrained not by limitations of the simulation environment, but by limitations of the data analysis environment. That is, data analysis rather than data acquisition has become the bottleneck to scientific discovery.

Ad hoc development of data analysis software to efficiently process petascale data is difficult and expensive [1]. As a result, only relatively few science projects can afford the staff to develop effective data analysis tools [2, 3]. An alternate strategy is to use either a parallel database management system (DBMS) or a MapReduce-style parallel processing framework [4, 5]. Both types of systems provide the following benefits: (1) they run on inexpensive shared-nothing clusters; (2) they provide quick-to-program, declarative interfaces [6–8]; and (3) they manage all task parallelization, execution, and failure-handling challenges. These frameworks thus hold the promise to enable cost-effective, massive-scale data analysis. However, porting complex scientific analysis tasks to these platforms is far from trivial [9, 10] and therefore remains relatively rare in practice.

Clustering algorithms in particular have been difficult to adapt to these shared nothing parallel data processing frameworks, for two reasons.

First, both parallel DBMSs and MapReduce-type systems support a dataflow style of processing where data sets are transformed by a directed acyclic graph of operators that do not otherwise communicate with each other. Additionally, the system controls the data placement and movement. In this setting, it is difficult to efficiently track clusters that span machine boundaries because such tracking typically requires frequent communication between nodes, especially when clusters may be arbitrarily large, as is the case in astronomy. Previous work on distributed clustering used a space-filling curve to partition the data, then built and queried a global distributed spatial index such that only adjacent partitions needed to communicate [11]. However, the same approach is difficult to implement in a MapReduce-style system that constrains communication patterns and controls data placement.

Second, the varying data density causes significant skew (i.e., inter-node load imbalance) in processing-times per partition. Significant skew can counteract the benefit of using a parallel system [12]. We find this effect to be significant in the astronomy simulation domain (see Section 6). Previous research proposed to use approximation to handle such dense regions [13, 14]. This approach, however, yields a different result than an exact serial computation.

In this paper, we address the above two challenges by describing dFoF, an algorithm for scalable, parallel clustering of N-body simulation results in astrophysics. Our algorithm is designed to run efficiently in a MapReduce-style data processing engine and is also optimized to deliver high performance even in the presence of skew in the data. We implement dFoF on the shared-nothing parallel data processing framework Dryad [15], and evaluate it on real data from the astronomy simulation domain. Dryad can be described as a generalization and refinement of MapReduce [4] allowing arbitrary relational algebra expressions and type-safe language integration [16].

The contributions of this paper are threefold. First, we develop dFoF, the first density-based parallel clustering algorithm for MapReduce-type platforms. Second, we show how to leverage data-oriented (rather than space-oriented) partitioning and introduce a spatial index optimization that together enable dFoF to achieve near-linear scalability even for data sets with massive skew. Third, we implement the proposed algorithm using DryadLINQ [16] 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. We show that our approach can cluster a massive-scale 18 GB simulation dataset in under 70 minutes (faster than the naïve version by a factor of 17) and offers near-linear scaleup and speedup. We also show that

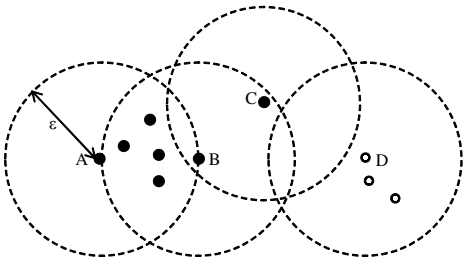

Fig. 1. Friends of Friends clustering algorithm. Two particles are considered *friends* if the distance between them is less than a threshold ϵ: A and B are friends and B and C are friends, but A and C are not. The friend relation is symmetric if the distance is symmetric. The Friend of Friend relation (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 a friend of friend pair via B). In the figure, the FoF relation induces a partition on the particles: all black points are in one cluster and all white points are in another.

our approach matches the performance of an existing hand-optimized implementation used in the astrophysics community on a dataset with little skew and improves running time by a factor of 2.7 on a skewed dataset.

2 Background and Related Work

Application domain. Cosmological simulations serve to study how structure evolves in the universe on distance scales ranging from a few million light-years to several billion light-years in size. In these simulations, the universe is modeled as a set of particles. These particles represent gas, dark matter, and stars and interact with each other through gravitational force and fluid dynamics. Particles may be created or destroyed during the simulation (e.g., a gas particle may spawn several star particles). Every few simulation timesteps, the program outputs a snapshot of the universe as a list of particles, each tagged with its identifier, location, velocity, and other properties. The data output by a simulation can therefore be stored in a relation with the following schema:

$Particles(id, x, y, z, v_x, v_y, v_z, mass, density, \cdots)$

State of the art simulations (e.g., Springel *et al.* 2005 [17]) use over 10 billion particles producing a dataset size of over 200 GB per snapshot. When the NCSA/IBM Blue Waters system [18] comes online in late 2010, it will support astrophysical simulations that generate 100 TB per snapshot and a total data volume of more than 10 PB per run.

Friends of Friends Clustering Algorithm. The Friends-of-Friends (FoF) algorithm (c.f.Davis *et al.* 1985 [19] and references therein) has been used in cosmology for at least 20 years to identify interesting objects and quantify structure in simulations [17, 20]. FoF is a simple clustering algorithm that accepts a list of particles (pid, x, y, z) as input and returns a list of cluster assignment tuples $(pid, clusterid)$. To compute the clusters, the algorithm examines only the distance between particles. FoF defines two particles as friends if the distance between them is less than ϵ. Two particles are *friends-of-friends* if they are reachable by traversing the graph induced by the friend relationship. To compute the clusters, the algorithm computes the transitive closure of

the friend relationship for each unvisited particle. All particles in the closure are marked as visited and linked as a single cluster. Figure 1 illustrates this clustering algorithm.

Related work. Thanks to its simplicity, FoF is one of only two algorithms to have been implemented in a distributed parallel fashion in the astrophysics community [21, 22] using the *N*tropy library [21, 22]. *N*tropy is an application-specific library that supports parallel kd-trees by combining "distributed shared memory" (DSM), a popular parallel data-management paradigm, with "remote procedure call" for workload orchestration.

FoF is a special case of the DBSCAN algorithm [23] corresponding to a $MinPts$ parameter of zero; there exists a large body of work on distributed DBSCAN algorithms [11, 13, 14, 24]. This prior work can be categorized into two groups. The first category of approaches is similar to Ntropy. These algorithms build a distributed spatial index on a shared-nothing cluster and use this index when merging local clustering results [11, 24]. The second category of approaches is to perform approximate clustering by using clustering on local models [13] or using samples to reduce the size of data or the number of spatial index lookups [14].

In contrast to these prior techniques, dFoF does not require any approximations. More importantly, it is designed and implemented to run on a data analysis platform such as Dryad [15] or MapReduce [4] rather than as a stand-alone parallel or distributed application. By leveraging an existing platform, dFoF automatically benefits from fault-tolerance, task scheduling, and task coordination. Further, dFoF does not rely on a global shared index but rather incrementally merges large clusters detected by different compute nodes.

Additionally, previous work on parallelizing DBSCAN has been evaluated against relatively small and often synthetic datasets [11, 13, 14, 24]. Their datasets have, at most, on the order of one million objects in two dimensions. In this paper, we evaluate the performance and scalability with real datasets of substantially larger scale.

Programming shared-nothing clusters has been gaining increased attention. Several distributed job execution engines have been proposed [5, 4, 15, 25], and several high-level job description languages have been defined [7, 16, 26–28]. However, complex scientific analysis tasks are only just beginning to be ported to these new platforms. In particular, Chu *et al.* investigated how to leverage such emerging platforms to run popular machine-learning algorithms and gain linear scalability [29]. There are several efforts to implement scalable machine learning algorithms in MapReduce-style framework [30–32]. Papadimitriou *et al.* implemented a co-clustering algorithm [31]. Panda *et al.* implemented a decision tree learning algorithm using MapReduce [32]. The Mahout project, inspired by Chu *et al.*, implements many machine learning algorithms in Hadoop including k-means and other clustering algorithms, but there is no density-based algorithm yet [30].

3 Basic Distributed Friends of Friends

In this section, we introduce dFoF, our distributed FoF algorithm for MapReduce-style shared-nothing clusters. We discuss critical optimizations that make this algorithm truly scalable in the following section.

The basic idea behind any distributed clustering algorithm is to (1) partition the space of data to be clustered, (2) independently cluster the data inside each partition, and

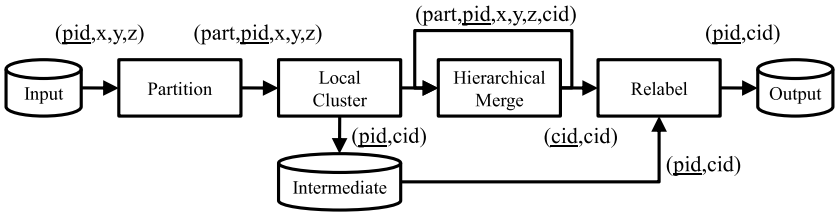

Fig. 2. Dataflow in dFoF algorithm. dFoF runs in four phases. Each phase exchanges data in the form of a standard relation or set of key-value pairs. Underlined attributes are the primary keys of the corresponding relations. part represents a partition id. pid represents a particle ID. x, y, and z correspond to the particle coordinates. cid is a cluster id. Phases execute in series but with intra-phase parallelism.

finally (3) merge clusters that span partition boundaries. There are several challenges related to implementing this type of algorithm on a MapReduce-style platform and in the context of astronomy data.

Challenges. First, in astrophysical applications, there is no characteristic cluster size or mass. The clustering of matter in the universe is largely scale-invariant at the size represented by the simulation. This means a cluster can be arbitrarily large and span arbitrarily many partitions. To identify such arbitrarily-large clusters from locally found ones, one cannot simply send to each compute node its own data plus a copy of the data at the boundary of adjacent partitions. Indeed, nearly all data would have to be copied to merge the largest clusters. Alternatively, one could try to use a global index structure, but this approach requires extensive inter-node communication and is therefore unsuitable for the dataflow-style processing of MapReduce-type platforms. 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. In this section, we present this algorithm, which we call dFoF.

Second, the uncharacteristic clusters pose a challenge for load balancing — each node needs to hold a contiguous region of space but there is no *a priori* spatial decomposition that is likely to distribute the processing load evenly. Load imbalances can negate the benefits of parallelism [12]. To achieve load balance and improve performance, we must ensure that each partition of the same operation processes its input data in approximately the same amount of time. This requirement is more stringent than ensuring each node processes the same amount of data. Indeed, in the FoF algorithm, execution times depend not only on the number of particles in a partition but also their distribution: small dense regions are significantly slower to process than large sparse ones. We discuss extensions to our algorithm that address these challenges in Section 4.

Approach. Our basic dFoF algorithm follows the typical distributed clustering approach in that the data is first partitioned, then clustered locally, and finally the local clusters are reconciled into large ones. Our algorithm differs from earlier work primarily in the way it handles the last phase of the computation. Instead of relying on a distributed index, dFoF reconciles large clusters through a *hierarchical merge process* that resembles a parallel aggregate evaluation [33]. To keep the cost of merging low, only the *minimum amount of data* is propagated from one merge step to the next.

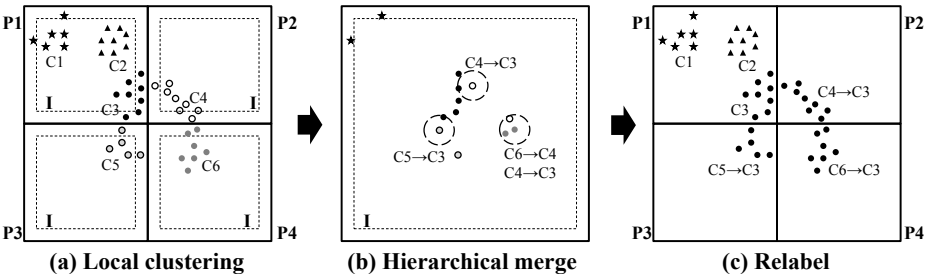

Fig. 3. Illustration of the dFoF algorithm. (a) shows the first local clustering phase. Data is partitioned into four cells. Points with the same shape are in the same global cluster. Particles with different shades but with the same shape are in different local clusters. Each P_i shows the cell boundary and each I shows the interior region that is excluded during the *Hierarchical Merge* phase. (b) demonstrates *Hierarchical Merge* phase. Note that only boundary particles in (a) are considered during the merge phase. After the 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 as illustrated in (c).

The rest of the data is written to disk before the merge. A final *relabeling* phase takes care of updating this data given the final merged output. dFoF thus runs in four phases: *Partition*, *Local cluster*, *Hierarchical merge*, and *Relabel*. Figure 2 shows the overall data flow of the algorithm, with each step labeled with the type of its output. We now describe the four phases in more detail using a simple 2D example.

Partition During partitioning, we assign each node a contiguous region of space to improve the probability that particles in the same cluster will be co-located on the same node. Figure 3 illustrates a 2D space split into four partitions P_1 through P_4. To determine these uniform regions, dFoF recursively bisects the space, along all dimensions, until the estimated number of particles per region is below the memory threshold of a node, such that local processing can be performed entirely in memory. We call the hierarchical regions *cells*, and the finest-resolution cells — the leaves of the tree — *unit cells*. The output of this phase is a partition of all data points (i.e., particles).

Local Cluster Once the data is partitioned, the original FoF algorithm runs within each unit cell. As shown in Figure 2, the output of this phase is written to disk and consists of a set of pairs: `(pid,cid)`, where `pid` is a particle ID and `cid` is a globally-unique cluster ID. Each input particle is labeled with exactly one cluster ID. Particles within distance ϵ of the boundary of each cell continue on to the next phase. They will serve to identify locally found clusters that need to be merged into larger ones.

Hierarchical Merge To identify clusters that span multiple cells, particles at cell boundaries must be examined. If particles in adjacent partitions are within distance threshold ϵ of each other, their clusters must be merged. Figure 3 illustrates the merge step for four partitions P1 through P4. The outer boxes, P_i, represent the cell boundaries. The inner boxes, I, are distance ϵ away from the corresponding edge of the cell. In Figure 3(a), the local clustering step identified a total of six clusters labeled C1 through C6. Each cluster comprises points illustrated with a different shape and shade of gray. However, there are only three global clusters in this dataset. These clusters are identified during the hierarchical merge process. Clusters C3, C4, C5, and C6 are merged because the points near the cell boundaries are within distance ϵ of each other. Only points in-

Algorithm 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 ← ∅ // nested set to store cluster ids of merged clusters
2: R ← ∅ // output mappings
3: sidx ← build_spatial_index(D)
4: for all unvisited p ∈ D do // compute cluster ids to merge
5:    N ← friendclosure(p, ε, sidx) // find all friends of friends of p using the spatial index
6:    mark all q ∈ N as visited
7:    C ← {x.cid | x ∈ N} // set of all cluster ids found in N
8:    M ← M ∪ {C} // All cluster ids in N must be merged
9: end for
10: repeat // find additional clusters to merge
11:    for all C ∈ M do
12:       C+ ← {X|X ∈ M, C ∩ X ≠ ∅}
13:       if |C+| > 1 then
14:          M ← M − C+
15:          C' ← {x|x ∈ X, X ∈ C+}
16:          M ← M ∪ {C'}
17:       end if
18:    end for
19: until M does not change
20: for all C ∈ M do // produce output
21:    newCid ← min C // select the lowest identifier in C
22:    R ← R ∪ {(cid, newCid)|cid ∈ C}
23: end for
24: return R
```

side each P_i but outside each region I are needed to determine that these clusters must be merged. Figure 3(b) shows the actual input to the hierarchical merge following local clustering phase. This data reduction is necessary to enable nodes to process hierarchically larger regions of space efficiently and without running out of memory.

Algorithm 1, which we call `mergefof`, shows the detailed pseudocode of the merge procedure. At a high-level, the algorithm does two things. First, it re-computes the clusters in the newly merged space. Second, it relabels the cluster ids of those clusters that have been merged. The input is a set of particles, each labeled with a cluster id. The output is a set of pairs $(oldcid, newcid)$ providing a mapping between the pre-merge cluster ids and the post-merge cluster ids.

Lines 1-8 show the initial cluster re-computation whose output, M, is a nested set of clusters that must be merged. For example, for the dataset in Figure 3, M will have three elements, $\{\{C1\}, \{C3, C4, C5\}, \{C4, C6\}\}$. This set M, however, is not yet quite correct, as there are potentially members of M that should be further combined. To see why, recall that some particles — those in the interior of the merged regions — were set aside to disk before the merge process began. These set-aside particles may connect two otherwise disconnected clusters. In our example, C6 should be merged with C3, C4, and C5 but is not because the particles of C4 bridging C6 to C3 were set aside to disk before merging. We can infer such missing links by examining the pairwise intersections between sets of merged cluster identifiers. For example, since $\{C3, C4, C5\}$

and $\{C4, C6\}$ both contain C4, we infer that C3, C4, C5, and C6 are all part of the same cluster and can be assigned a single cluster id. The second step of `mergefof` (lines 10-19) performs this inference. In the last step, the algorithm simply chooses the lowest cluster id as the new id of the merged cluster (lines 20-23).

Algorithm `mergefof` executes every time a set of 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 to disk. They will be reused during the final *Relabel* phase.

Relabel In dFoF, there are two occasions for relabeling, *intermediate* and *global*. Intermediate relabeling assigns each particle used during the merge process a new cluster id based on the output of `mergefof`. This operation occurs once for each cell in the merge hierarchy. Global relabeling occurs at the end of dFoF. This operation 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 by reprocessing the data previously set aside to disk as shown in Figure 2.

4 Scalable Distributed Friends of Friends

The dFoF algorithm presented thus far is parallel but not scalable due to skew effects. Some compute nodes during *Local clustering* phase may run significantly longer than others, negating the benefits of parallelism. In this section, we discuss this problem and present two optimizations that address it. The first optimization significantly improves the performance of both local `fof` and `mergefof` algorithms. The second optimization improves load balance.

4.1 Pruning Visited Subtrees

With an ordinary spatial index implementation, each partition can spend a significantly different amount of time processing its input during the local clustering phase (i.e., FoF), despite having approximately the same amount of input data. We demonstrate this effect in Section 6, where we measure the variance in task execution times (in Figure 7, all plots except for non-uniform/optimized exhibit high variance). This imbalance is due to densely populated regions taking disproportionately longer to process than sparsely populated regions, even when both contain the same number of points.

To understand the challenge related to dense regions, recall that the serial FoF algorithm computes the transitive closure of a particle using repeated lookups in a spatial index. The average size of the closure, and therefore of the traversed part of the index, are proportional to the density of the region. These lookups dominate the runtime. Astronomy simulation data is especially challenging in this respect, because the density can vary by several orders of magnitude from region to region. To address this challenge, we optimize the local cluster computation as follows.

The original FoF algorithm constructs a spatial index over all points to speed up friend lookups. 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 that is set to $true$ when all points within the subtree rooted at the node have been returned as a result of previous friends lookups. The algorithm

Algorithm 2 Range search with pruning visited subtree

Input: $root \leftarrow$ search root node
$query \leftarrow$ center of the range search (i.e., querying object)
$\epsilon \leftarrow$ distance threshold
Output: set of objects within distance ϵ of $query$
1: **if** $root.visitedAll$ is **true then**
2: **return** $\emptyset$ // skip this subtree
3: **end if**
4: $R \leftarrow \cdots$ // normal range search for $root$
// update bookkeeping information
5: **if** entire branch under $root$ marked $visited$ **then**
6: $root.visitedAll \leftarrow$ **true**
7: **end if**
8: **return** R

can safely skip such flagged subtrees because all data items within them have already been covered by previous lookups. By the nature of spatial indexes, points in a dense region are clustered under the same subtree and are therefore quickly pruned. With this approach, the index shrinks 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 space overhead. Furthermore, the flag can be updated while processing range lookups. In Algorithm 2, we show the modified version of the range search using this modified index structure. Line 5 is dependent on the type of spatial index. For a kd-tree, the condition can be evaluated by checking the flags of the child nodes and the data item assigned to the $root$ node. For an R-tree, the condition can be evaluated similarly by checking child nodes and data items in a leaf node.

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

4.2 Non-Uniform Data Partitioning

While the above optimization solves the problem of efficiently processing dense regions, it does not solve all load imbalance problems. Indeed, with the uniform space-based partitioning described in Section 3, some nodes may be assigned significantly more data than others and may delay the overall execution or even halt if they run out of memory when the data is not uniformly distributed. The only way to recover is for the system to restart the job using a smaller unit cell. On the other hand, unit cells that are unnecessarily fine-grained add significant scheduling and merging overheads.

To address this challenge, we use a variant of Recursive Coordinate Bisection (RCB) scheme [34] to ensure that all partitions contain approximately the same amount of data (i.e., same number of particles). The original RCB repeatedly bisects a space by choosing the median value along alternating axis to evenly distribute input data across multi-processors. Since the input data does not fit in memory, we first scan the data, collect a random sample, and run RCB over the sample until the estimated size of the data for each bucket fits into the memory of one node. We use RCB because 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 compared to space filling curve). In Figure 4, we compare the uniform and data partitioning schemes. Because we use a sample instead of the entire dataset, there is some small discrepancy in the

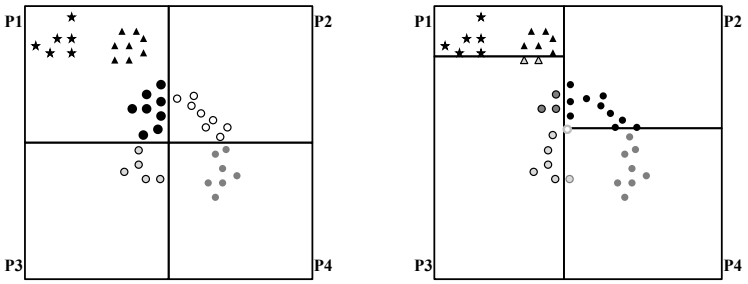

Fig. 4. Uniform partitioning and Non-uniform partitioning. Uniform partitioning would generate uneven workloads: P_1 contains 22 points while P_3 have only 5 points in it. Data-oriented partitioning, however, produces an even workload: each partition is assigned 10 or 11 points.

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, especially with the first optimization, and improves the job completion time as we show in Section 6.

5 Implementation

We implemented dFoF in approximately 3000 lines of C# code using DryadLINQ [16] the programming interface to Dryad [15]. Dryad is a massive-scale data processing system that is similar to MapReduce but offers more flexibility because its vertices are not limited to map or reduce operations. DryadLINQ is a Language-Integrated Query (LINQ) interface provider for Dryad. The LINQ offers relational-style operators such as filters, joins, and groupings and users can write complex data query succinctly and seamlessly in C#. At runtime, DryadLINQ automatically translates and optimizes the task written in LINQ expressions into a Dryad job which is a directed acyclic graph of operators with one process per operator. If possible, connected vertices communicate through shared memory pipes. Otherwise, they communicate through compressed files stored in a distributed file system. The job is then deployed and executed in the Dryad cluster.

We wrote the core `fof()`, `mergefof()` functions as user-defined operators. Because both functions have to see all data in the input data partition, we used DryadLINQ's `apply` construct which exposes the entire partition to the operator rather than a single data at a time. Other than the user defined operators, we used standard LINQ operators not only for the initial data partitioning and relabeling but also for the post-processing output of each phase. We also used the lazy evaluation feature of the LINQ framework to implement the iterative hierarchical merge phase. Thus, we only submit a single Dryad job for the entire dFoF task. Using MapReduce, we would have to schedule one MapReduce job for the local clustering, and also one for each iteration of the iterative hierarchical merge process. The entire job coordination is written in only 120 lines out of a total of 3000 lines. The final Dryad plan to process dataset in Section 6 consists of 1227 vertices with three hierarchical merges.

For the node-local spatial index used in the FoF computation (and also partitioning the data), we chose to use a kd-tree [35] because of its simplicity. We implemented both a standard version of the kd-tree and the optimized version presented in Section 4.1.

6 Evaluation

In this section, we evaluate the performance and scalability of the dFoF clustering algorithm using two real world datasets. We execute our code on 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 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 previously. Our goal is to show improvements in the relative numbers rather than try and show the best possible absolute numbers.

We evaluate our algorithm on data from a large-scale astronomy simulation currently running on 2048 compute cores of the Cray XT3 system at the Pittsburgh Supercomputing Center [36]. The simulation itself was only about 20% complete at the time of submission. Therefore we use two relatively early snapshots: S43 and S92 corresponding respectively to 580 million years and 1.24 billion years after the Big Bang. Each snapshot contains 906 million particles occupying 43 GB in uncompressed binary format. 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 is hash partitioned on the particle identifier attribute. Each partition is also compressed using the GZip algorithm. Dryad can directly read compressed data and decompresses it on-the-fly as needed.

For this particular simulation, astronomers set the distance threshold (ϵ) to 0.000260417 in units where the size of the simulation volume is normalized to 1. Both datasets require at least two levels of hierarchical merging.

As the simulation progresses, the Universe becomes increasingly structured (i.e., more stars and galaxies are created over time). Thus, S92 has not only more clusters (3496) than S43 (890) but also has denser regions than S43. The following table shows the distribution of the number of particles within distance threshold (i.e., density of data):

Percentile	25	50	75	90	99	99.9	100
S43	6	16	97	373	1,853	8,322	10,494
S92	8	44	1,370	41,037	350,140	386,577	387,136
S92:S43	1.33	2.75	14.12	110.02	188.96	46.45	36.89

Ideally, the structure of data should not affect the runtime of the algorithm so that scientists can examine and explore snapshots taken at any time of simulation in a similar amount of time.

Evaluation summary. In the following subsections, we evaluate our dFoF Dryad implementation. First, we process both snapshots using an eight-node Dryad cluster while varying the partitioning scheme and the spatial index implementation. These experiments enable us to measure the overall performance of the algorithm and the impact of our two optimizations. Second, we evaluate dFoF's scalability by varying the number of nodes in the cluster and the size of the input data. Finally, we compare dFoF to the existing OpenMP implementation that the astronomers use today. Overall, we find that dFoF exhibits a near linear speedup and scaleup even with suboptimal hardware and software configurations. Additionally, dFoF shows consistent performance regardless of skew in the input data thanks to the optimization in Section 4.

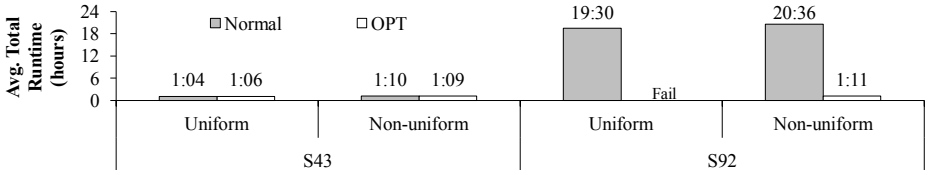

Fig. 5. Average time to cluster entire snapshot. Average of three executions except for jobs that took longer than 20 hours. Missing bar is due to a failure caused by an out-of-memory error. As the figure shows, with both optimizations enabled, both snapshots are processed within 70 min.

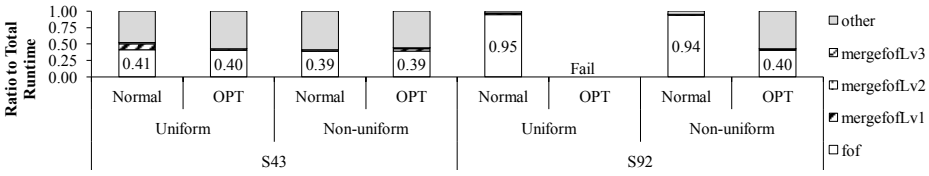

Fig. 6. Runtime breakdown across phases. Average of three executions except jobs that take longer than 20 hours. The initial `fof()` takes more than 40% of total runtime. The three-level hierarchical merges, `mergefof()`, took less than 4% of total runtime. "Other" represents time to take to run all standard vertices such as filter, partition, joins to glue the phases. Overall, `fof()` is the bottleneck and completely dominates when the data is highly skewed and an ordinary kd-tree is used.

6.1 Performance

In this section, we use the full eight-node cluster, varying the partitioning scheme and spatial index implementation. For the partitioning scheme, we compare deterministic uniform partitioning (Uniform) described in Section 3 and dynamic partitioning (Non-uniform) described in Section 4.2. We also compare an ordinary kd-tree implementation (Normal) to the optimized version (OPT) described in Section 4.1. We repeat all experiments three times except for Uniform partitioning using the Normal kd-tree because it took 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 special reason for using a small sample except to avoid a high overhead of planning. As the results in this section show, even small samples work well for this particular dataset.

Figure 5 shows the total run times for each variant of the algorithm and each dataset. For dataset S43, which has less skew in the cluster-size distribution, all variants complete within 70 minutes. However, when there is high skew (i.e., more structures as in S92), the normal kd-tree implementation takes over 20 hours to complete while the optimized version still runs in 70 minutes. Uniform-OPT over snapshot S92 did not complete because it reached full memory capacity while processing a specific data partition, causing the failure of the entire query plan as we discuss in more detail below.

Figure 6 shows the average relative time taken by each phase of the algorithm. The hierarchical merge occurs in order of mergeLv1, mergeLv2, and mergeLv3. As the figure shows, local clustering, `fof`, takes more than 40% of total runtime in all cases and 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

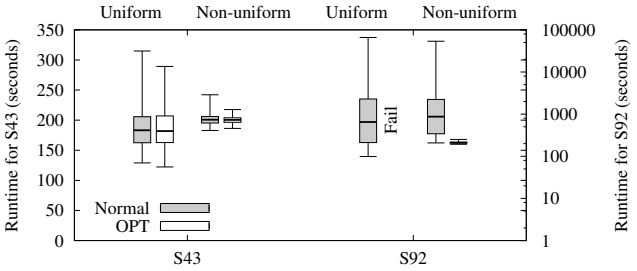

Fig. 7. Distribution of FoF runtime per partition. Uniform partitioning yields higher variance in per partition runtime than Non-uniform partitioning. FoF with optimized index traversal runs orders of magnitude faster than FoF with normal implementation in S92 dataset. Note that the y-axis for S92 is in log scale.

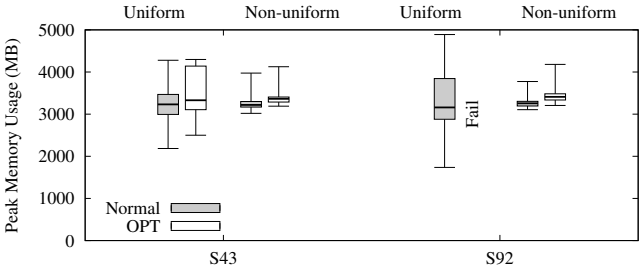

Fig. 8. Distribution of FoF peak memory utilization per partition. Uniform partitioning yields a higher variance in peak memory utilization per partition than Non-uniform partitioning. This high variance caused the crash of one of the partitions with optimized index traversal. The optimized kd-tree index traversal has a higher memory footprint than the normal implementation.

other standard operators account for over 50% of total runtime, but the total is the sum of more than 30 operators including repartitions, filters, and joins to produce intermediate and final result for each level of the hierarchical merge. 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 Figures 7 and 8, we measure the *per-node* runtime and peak memory utilization of the local `fof` phase. We plot the quartiles and minimum and maximum values. 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. In both Figures 7 and 8, Non-uniform partitioning shows a tighter distribution in runtime and peak memory utilization than uniform partitioning. With uniform partitioning, the worst 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 query plan to fail. This does not happen with normal kd-tree and uniform partitioning because the optimized kd-tree has a larger memory footprint as discussed in Section 4.1. Non-uniform partitioning is therefore worth the extra scan over the entire dataset.

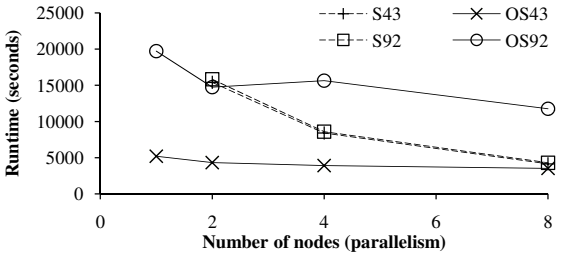

Fig. 9. Speedup. dFoF runtime for each dataset with varying number of nodes. dFoF speedup is almost linear. OS43 and OS49 are the result of OpenMP implementation of FoF with varying degree of parallelism. Note that S43 overlaps with S92.

As Figure 7 shows, dFoF with the optimized index (Section 4.1) significantly outperforms Normal implementation especially when there is significant skew in the cluster-size distribution. Thanks to the pruning of visited subtrees, the runtime for S92 remains almost the same as that for S43. However, the optimization is not free. Due to the extra tracking flag, the optimization requires slightly more memory than the ordinary implementation as shown in Figure 8. The higher memory requirement could be alleviated by a more efficient implementation such as keeping a separate bit vector indexed per node identifier or implicitly constructing a kd-tree on top of an array rather than keeping pointers to children in each node. Overall, however, the added memory overhead is negligible compared with the order-of-magnitude gains in runtime.

6.2 Scalability

In this section, we evaluate the scalability of the dFoF algorithm with non-uniform data partitioning and the optimized kd-tree. In these experiments, we vary the number of nodes in the cluster and redistribute the input data only to the participating nodes. All reported results are the average of three runs. The standard deviation is less than 1%.

Figure 9 shows the runtime of dFoF for each dataset and increasing number of compute nodes. *Speedup* measures how much faster a system can process the same dataset if it is allocated more nodes [12]. Ideally, speedup should be linear. That is, a cluster with N nodes should process the same input data N times faster than a single node. For both datasets, the runtime of the dFoF is approximately half as long as we double the number of nodes, showing a close-to perfect linear speedup. We do not present the number for the single-node case due to an unknown problem in the Dryad version we use; the system did not schedule remaining operators if currently running operator takes too long to complete.

Figure 10 shows the scaleup results. Scaleup measures how a system handles data size that has increased in proportion to the cluster size. Ideally, as the data and cluster size increase proportionally to each other, the runtime should remain constant. To vary the data size, we subsample the S43 and S92 datasets. For 4-node and 8-node configurations, the scaleup is close to ideal: the ratio of runtimes to the single-node case are 0.99 and 0.91 respectively. The 2-node experiment showed a scaleup of only 0.83 and 0.78. We investigated the 2-node case and found that the size of the subsampled dataset was near the borderline of requiring one additional hierarchical merge. Thus, each partition

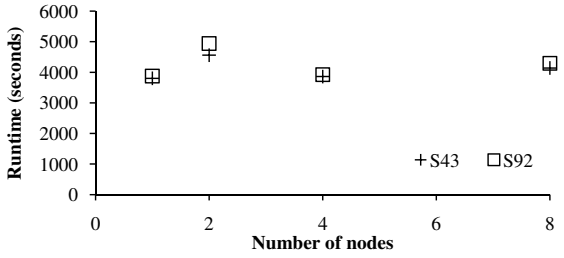

Fig. 10. Scaleup. Runtime of dFoF with increasing data size proportional to the number of nodes. Except for the two node case where scheduling overhead pronounced, dFoF scales up in linear.

was underloaded and completed quickly, overloading the job scheduler and yielding a poorer scaleup.

Overall, considering our suboptimal hardware configuration, the scalability of dFoF is excellent.

6.3 Compared to OpenMP implementation

Astronomers currently use a serial FoF implementation that has been moderately parallelized using OpenMP [37] as a means of scheduling computation across multiple threads that all share the same address space. OpenMP is often used to parallelize programs that were originally written serially. 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 searches, as well as numerous other performance enhancements. The OpenMP aspects are not performance-oriented, though. 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 shared-nothing cluster that we used for the previous experiments represents 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 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 disks.

At the scale of 128 GB of RAM, it is now cheaper to buy a single shared-memory system than to distribute the same 128 GB across 16 nodes. However, this cost-savings breaks down at the scales beyond 128 GB. For example, it is not possible to find symmetric multiprocessing systems (SMPs) with 1TB of RAM. At this scale, certain vendors offer systems with physically distributed memory that share a global address space ("Non-Uniform Memory Access" or "NUMA" systems), but these are generally more

expensive than building a cluster of distributed-memory nodes from commodity hardware. Furthermore, the ostensible advantage of the shared-nothing architecture over a large, shared-memory system is the scalability of I/O.

Consequently, our goal is to achieve competitive performance with the astrophysics FoF running on the shared-memory system with our Dryad version running on the shared-nothing cluster (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 utilizing cheaper distributed-memory platforms for clustering calculations at scales large enough to have economic impact.

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

Running the astronomy FoF algorithm in serial on the shared-memory system for our test dataset S43 (with the same parameters as our 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, as extrapolated from our optimized Dryad 2-node run assuming ideal scalability. However, since we do not actually know the serial runtime of Dryad on this dataset, it is difficult to compare a parallel Dryad run directly to our serial FoF implementation, since there is undoubtedly parallel overhead induced by running Dryad on more than one node.

The runtime comparisons are much more interesting for 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, compared to roughly 30,000 seconds for a serial Dryad run of the same snapshot. The OpenMP implementation still wins, but the difference is smaller than for S43.

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 limitation is because multiple threads may start tracking the same group. When two threads realize they are actually tracking the same group, 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 9.

Since our Dryad version performed similarly on both snapshots, we conclude that our methodology achieves scalability in both computational work and I/O. The advantage of our implementation can be seen when we run on more nodes. This advantage allows us to match the performance of the astrophysics code on S43 (3513 seconds vs. 4141 seconds) and to substantially outperform it for S92 (11763 seconds vs. 4293 seconds). This idea is in keeping with the MapReduce strategy: We employ a technique that may be less than optimally efficient in serial, but that scales very well. Conse-

quently, we have achieved our goal of reducing time-to-solution on platforms that offer an economic advantage over current shared-memory approaches at large scales.

7 Conclusion

Science is rapidly becoming a data management problem. Scaling existing data analysis techniques is very important to expedite the knowledge discovery process. In this paper, we designed and implemented a standard clustering algorithm to analyze astrophysical simulation output using a popular MapReduce-style data analysis platform. Through experiments on 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 leverage non-uniform data partitioning based on sampling and introduce an optimized spatial index approach. An interesting area of future work, is to extend dFoF to the DBSCAN algorithm.

Acknowledgment

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 CAREER award IIS-0845397, CNS-0454425, and gifts from Microsoft Research. Magdalena Balazinska is also sponsored in part by a Microsoft Research New Faculty Fellowship.

References

1. Becla, J., Lim, K.T.: Report from the SciDB meeting (a.k.a. extremely large database workshop). `http://xldb.slac.stanford.edu/download/attachments/4784226/sciDB2008_report.pdf` (2008)
2. Sloan Digital Sky Survey, `http://cas.sdss.org`
3. The Large Hadron Collider, `http://lhc.web.cern.ch/lhc/`
4. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: Proc. of the 6th OSDI Symp. (2004)
5. Hadoop, `http://hadoop.apache.org/`
6. Hadoop Hive, `http://hadoop.apache.org/hive/`
7. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig latin: a not-so-foreign language for data processing. In: Proc. of the SIGMOD Conf. (2008) 1099–1110
8. ISO/IEC 9075-*:2003: Database Languages - SQL. ISO, Geneva, Switzerland
9. Cohen, J., Dolan, B., Dunlap, M., Hellerstein, J.M., Welton, C.: Mad skills: new analysis practices for big data. Proc. VLDB Endow. **2**(2) (2009) 1481–1492
10. Stonebraker et. al: Requirements for science data bases and SciDB. In: Fourth CIDR Conf. (perspectives). (2009)

11. Xu, X., Jäger, J., Kriegel, H.P.: A fast parallel clustering algorithm for large spatial databases. Data Min. Knowl. Discov. **3**(3) (1999) 263–290
12. DeWitt, D., Gray, J.: Parallel database systems: the future of high performance database systems. Communications of the ACM **35**(6) (1992) 85–98
13. Januzaj, E., Kriegel, H.P., Pfeifle, M.: Scalable density-based distributed clustering. In: Proc. of the 8th PKDD Conf. Volume 3202 of LNCS. (2004) 231–244
14. Aoying, Z., Shuigeng, Z., Jing, C., Ye, F., Yunfa, H.: Approaches for scaling dbscan algorithm to large spatial databases. Journal of Comp. Sci. and Tech. (2000) 509–526
15. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: Distributed data-parallel programs from sequential building blocks. In: Proc. of the 2007 EuroSys Conf. (2007) 59–72
16. Yu et. al: DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In: Proc. of the 8th OSDI Symp. (2008)
17. Springel et. al: Simulations of the formation, evolution and clustering of galaxies and quasars. NATURE **435** (June 2005) 629–636
18. About the Blue Waters project, `http://www.ncsa.illinois.edu/BlueWaters/`
19. Davis, M., Efstathiou, G., Frenk, C.S., White, S.D.M.: The evolution of large-scale structure in a universe dominated by cold dark matter. Astrophysical Journal **292** (May 1985) 371–394
20. Reed et. al: Evolution of the mass function of dark matter haloes. Monthly Notices of the Royal Astronomical Society **346** (December 2003) 565–572
21. Gardner, J.P., Connolly, A., McBride, C.: Enabling rapid development of parallel tree search applications. In: Proc. of the 2007 CLADE Symp. (2007)
22. Gardner, J.P., Connolly, A., McBride, C.: Enabling knowledge discovery in a virtual universe. In: Proc. of the 2007 TeraGrid Symp. (2007)
23. Ester, M., Kriegel, H.P., Sander, J., Xu, X.: A density-based algorithm for discovering clusters in large spatial databases with noise. In: Proc. of the 2nd KDD Conf. (1996) 226–231
24. Arlia, D., Coppola, M.: Experiments in parallel clustering with dbscan. In: Euro-Par 2001 Parallel Processing. Volume 2150 of LNCS. (2001) 326–331
25. DeWitt et. al: Clustera: an integrated computation and data management system. In: Proc. of the 34th VLDB Conf. (2008) 28–41
26. Chaiken et. al: Scope: easy and efficient parallel processing of massive data sets. In: Proc. of the 34th VLDB Conf. (2008) 1265–1276
27. Cascading, `http://www.cascading.org/`
28. Pike, R., Dorward, S., Griesemer, R., Quinlan, S.: Interpreting the data: Parallel analysis with Sawzall. Scientific Programming **13**(4) (2005)
29. Chu et. al: Map-reduce for machine learning on multicore. In Schölkopf, B., Platt, J., Hoffman, T., eds.: NIPS. Volume 19. (2007)
30. Apache Mahout, `http://lucene.apache.org/mahout/`
31. Papadimitriou, S., Sun, J.: 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) 512–521
32. Panda, B., Herbach, J., Basu, S., Bayardo, R.: Planet: massively parallel learning of tree ensembles with mapreduce. Proc. of the VLDB Endowment **2**(2) (2009)
33. Yu, Y., Gunda, P.K., Isard, M.: Distributed aggregation for data-parallel computing: interfaces and implementations. In: Proc. of the 22nd SOSP Symp. (2009)
34. Berger, M., Bokhari, S.: A partitioning strategy for nonuniform problems on multiprocessors. Computers, IEEE Transactions on **C-36**(5) (1987)
35. Gaede, V., Günther, O.: Multidimensional access methods. ACM Comput. Surv. **30**(2) (1998) 170–231
36. Bigben, `http://www.psc.edu/machines/cray/xt3/`
37. Dagum, L., Menon, R.: Openmp: An industry-standard api for shared-memory programming. Computing in Science and Engineering **5**(1) (1998) 46–55