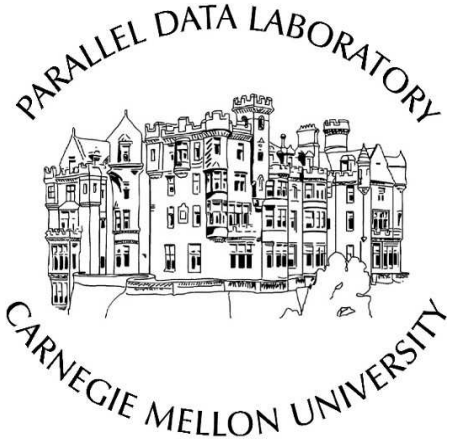

DiscFinder: A Data-Intensive Scalable Cluster Finder for Astrophysics

Bin Fu Kai Ren Julio López Eugene Fink
Garth Gibson
{binf, kair, jclopez, efink, garth}@cs.cmu.edu

CMU-PDL-10-104

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

DiscFinder is a scalable, distributed, data-intensive group finder for analyzing observation and simulation astrophysics datasets. Group finding is a form of clustering used in astrophysics for identifying large-scale structures such as clusters and superclusters of galaxies. DiscFinder runs on commodity compute clusters and scales to large datasets with billions of particles. It is designed to operate on datasets that are much larger than the aggregate memory available in the computers where it executes. As a proof-of-concept, we have implemented DiscFinder as an application on top of the Hadoop framework. DiscFinder has been used to cluster the largest open-science cosmology simulation datasets containing as many as 14.7 billion particles. We evaluate its performance and scaling properties and describe the performed optimization.

Keywords: Astrophysics, Data Analytics, Out-of-Core Algorithms, Data-Intensive Scalable Computing, DISC, eScience

1 Introduction

Today, the generation of new knowledge in data-driven sciences, such as astrophysics, seismology and bio-informatics, is enabled by the processing of massive simulation-generated and sensor-collected datasets. The advance of many fields of science is increasingly dependent on the analysis of these necessarily much larger datasets. For example, high-resolution cosmological simulations and large sky surveys are essential in astrophysics for answering questions about the nature of dark energy and dark matter (DE&DM) and the formation of large-scale structures in the universe. The analysis of these datasets becomes extremely challenging as their size grows. They become too large to fit in the aggregate memory of the computers available to process them. Enabling the scaling of science analytics to these much larger datasets requires new data-intensive approaches.

Frameworks, such as Hadoop and Dryad, are commonly used for data-intensive applications on Internet-scale datasets. These applications include text analysis, natural language processing, indexing the web graph, mining social networks and other machine learning applications. The natural question is then: *Can we leverage these data-intensive frameworks for science analytics?* In this work, we want to understand the requirements for developing new algorithms to enable science analytics using these systems. In the process, we should understand the advantages and limitations of these frameworks and how they should be enhanced or extended to better support analytics for science.

As part of our collaboration with domain scientists, we have developed DiscFinder: a new data-intensive distributed approach for finding clusters in large particle datasets. Group finding is a technique commonly used in astrophysics for studying the distribution of mass in the universe and its relation to DE&DM. DiscFinder is complementary to other existing group finding methods and is particularly useful for processing very large datasets on commodity compute clusters even in the case where the available aggregate memory cannot hold the entire dataset. Although, the mechanisms described here are specific to group finders for astrophysics, the principles are generally applicable to clustering problems in other fields of science.

DiscFinder is scalable, conceptually simple and flexible. DiscFinder scales up to process datasets with tens of billions of particles. It has been used to find clusters in the largest open-science simulation datasets. The DiscFinder design is compact and conceptually simple. DiscFinder employs a direct approach that leverages sequential group finder implementations to cluster relatively small subsets of the input particles. The main idea is to group and partition the input particle dataset into regions of space, then execute a sequential group finder for each partition, and finally merge the results together in order to join the clusters that span across partitions. The sequential finders are independently executed on relatively small subsets of the input in order to keep their running time low. The approach is implemented as a series of jobs on top of the Hadoop framework. DiscFinder relies on Hadoop for splitting the input data, managing and coordinating the execution of tasks and handling task failures. Finally, the DiscFinder strategy is flexible in the sense that it allows multiple implementation of the sequential group finder to be used. DiscFinder distributes

The work in this paper is based on research supported in part by the Betty and Gordon Moore Foundation, by the National Science Foundation under award SCI-0430781, by the Department of Energy, under award number DE-FC02-06ER25767, and by a Google research award. We also thank The McWilliams Center for Cosmology and the member companies of the PDL Consortium (including APC, Data-Domain, EMC, Facebook, Google, Hewlett-Packard, Hitachi, IBM, Intel, LSI, Microsoft, NEC, NetApp, Oracle, Seagate, Sun, Symantec, and VMware) for their interest, insights, feedback, and support.

the execution of the sequential finder in order to scale to large datasets. This strategy reduces the overall implementation complexity and benefits from the effort invested in the development of the sequential finders.

We are interested in determining the feasibility of the DiscFinder design and understanding its performance characteristics. In our evaluation, we first characterize the DiscFinder scalability with respect to the data size and find that it is possible to cluster very large datasets with this approach.

The main benefits of DiscFinder and future analysis applications implemented using similar approaches are their simplicity and potentially shorter development time. However, the simplicity of the implementation comes at a performance cost. We characterize the DiscFinder running time and apply a set of modifications to the implementation to improve its performance. There is clearly room for improvement both at the application and framework levels. We discuss various potential optimizations that can provide additional performance benefits. As the performance of these frameworks improves, they will be more widely used in data analytics for science.

The rest of this paper is structured as follows: Section 2 describes the motivating application; Section 3 summarizes previous related work; the DiscFinder design is explained in Section 4; the implementation details are presented in Section 5; Section 6 presents the performance evaluation of the approach.

2 Motivating Application

Analysis of Astronomical Datasets. Domain scientists are now commonly faced with the challenge of analyzing massive amounts of data in order to conduct their research. In particular, ever increasing large observation and simulations datasets abound in astrophysics. The advent of digital sky surveys, beginning with the Sloan Digital Sky Survey (SDSS), increased dramatically the scope and size of astronomical observational datasets [1]. The latest SDSS data release was over 30 TB in size. The field is moving towards large time domain astronomy surveys, such as Pan-STARRS [29, 23] and LSST [22], which will store many time slices of data. Pan-STARRS is producing in the order of 1TB/day of imagery. Both Pan-STARRS and LSST are projected to produce multi-TB datasets over the lifetime of the surveys. Similarly, state-of-the-art N-body cosmological simulation, such as the Bigben BHCosmo and Kraken DM simulations [9] produce multi-billion particle datasets with sizes in the orders of tens of terabytes, even after aggressive temporal sub-sampling. This down-sampling is needed to deal with the bandwidth and capacity limits of the storage system available in the supercomputers where these simulations execute.

The analysis of these datasets is essential for tackling key problems in astrophysics, such as the understanding the nature of dark energy (DE) and dark matter (DM), and how DE&DM controls the formation of large-scale structures in the universe [5], such as clusters and superclusters of galaxies. In order to better understand the role of DE&DM in the evolution of the universe, theoretical and observational astrophysicists analyze the aggregate properties of large-scale structures, such as the distribution of their size, volume and mass [40]. Finding the groups of particles that make up the large-scale structures is the first step towards carrying out this process. Once the groups have been identified, then their properties can be calculated and the groups can be decomposed into sub-halos for further analysis [33, 6, 10, 32, 19]

Group Finding. In astrophysics, group finding refers to a family of physics-based spatial clustering problems applied both to observation and simulation datasets. Their input is a set of celestial

objects such as stars, galaxies, gas, dark matter, etc. We will refer to those as particles, points or objects. A group finder separates the particles into groups that make up larger structures such as galaxies and clusters of galaxies. In very loose terms, we will refer to those structures as groups or clusters interchangeably. In slightly more formal terms, a group finder takes an input set of particles $P = \{p_0, p_1, p_2, ..., p_{n-1}\}$ and produces a set of m disjoint groups $\mathcal{G} = \{G_0, ..., G_{m-1}\}$ such that each group G_j comprises the subset of points from P (i.e., $\forall G_i \in \mathcal{G}\colon G_i \subset P$ and $\forall G_i \in \mathcal{G}, \forall G_j \in \mathcal{G}, i \neq j\colon G_i \cap G_j = \emptyset$). The criteria for determining the membership of a particle to a group may use physics-based computations that take into account particle properties such as position, mass and velocity. There are many variants of group finding approaches, both from the algorithmic point of view as well as the criteria used for the selection.

Friends-of-Friends (FOF). FOF is a simple algorithm proposed by Huchra and Geller to study the properties of groups of galaxies in observation surveys [7, 21]. Its group membership criteria is solely based on the Euclidean inter-particle distance. FOF is widely used and works in practice for many analysis scenarios. There are a large number of more sophisticated group finding approaches that are based on FOF.

FOF is driven by two parameters: a *linking length* (τ) and a *minimum group size* (`minGz`). The input is the set of particles P, in particular each particle is a tuple of the form $\langle \texttt{pid}_i, (x_i, y_i, z_i) \rangle$, where `pid` is the particle identifier and x_i, y_i, z_i are the particle's position coordinates in 3D space. The only group membership criteria is the following: two particles p_i and p_j belong to the same group G_l (are friends), if the distance between them (d_{ij}) is less than τ. This procedure is illustrated in two dimensions in Figure 1. Any other particle p_k also belongs to the group G_l, if the distance between p_k and any of the particles in G_l is less than τ. After applying this criteria, all the friends of p_i become friends of p_j and vice versa, hence the name of the approach. The output set $\mathcal{G}$ comprises the groups that contain a number of points larger that the `minGz` parameter. The output is represented as a list of tuples of the form $\langle \texttt{pid}, \texttt{groupId} \rangle$ where `groupId` is the group identifier.

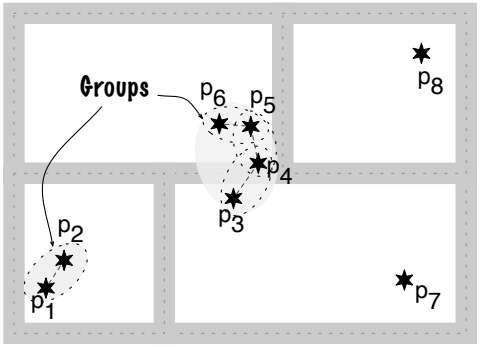

Figure 1: Friends of Friends (FOF) clustering. This dataset contains 8 particles $p_1, ..., p_8$. Shaded regions denote groups of particles. Notice that particles p_3 and p_6 belong to the same group although they are not close enough to each other. The reason is that they are indirectly linked through their *friends*, particles p_4 and p_5. In this example, `minGz` $= 2$, thus particles p_7 and p_8 do not belong to any group.

There are available sequential implementations of the FOF algorithm, such as the one from the "N-body shop" at the University of Washington [37]. These implementations rely on building a spatial indexing structure, such as a *kd*-tree, to speed up lookups to nearby particles [25]. However, sequential solutions are relatively slow to process billions of galaxies. There are also various

parallel group finders for distributed memory machines with low-latency networks that are often used in simulations (See Section 3).

Group finding implementations, whether sequential or parallel, are in-core and thus require large enough available memory to fit the dataset and internal data structures. This means that finding groups and analyzing large simulation datasets require supercomputers of comparable capacity as the one used to generate them. For the very large datasets these resources are not readily available. In the cases where there is enough aggregate memory to execute the group finding process, the end-to-end running time is dominated by the I/O required to load the input dataset and produce the results. We have developed an alternative data-intensive group finding approach named DiscFinder, which complements existing approaches. DiscFinder is useful for finding groups in datasets need to be loaded from external storage, such as it is often the case in the analysis of astronomy surveys and post-simulation analysis of synthetic datasets. DiscFinder makes it possible to find groups in datasets that are much larger than the memory of the available compute resources.

3 Related Work

Group finders used in astrophysics refers to a class of clustering approaches. The group membership criteria used in the basic FOF approach may not be appropriate for certain applications. Variations of the base algorithm are designed, among other reasons, to handle uncertainties in the input data, and to take into account other particle properties besides their positions [18, 38]. For example, Extended FOF (EXT-FOF) is a finder used in photometric datasets [3]; probability FOF (pFOF) is used to identify galaxy groups in catalogs in which the red shift errors have large dispersions [26]. Hierarchical FOF [18], SOF [39], SKID [38], DENMAX [15], IsoDEN [31], HOP [11] and Sub-halos [33], among others, are sophisticated group finders with selection criteria that take into account the density of the the neighborhood surrounding a particle and whether or not that particle is gravitationally bound to the larger structure. The approximate implementation from the University of Washington "N-body shop" (aFOF [36]) aims to overcome the severe slowdown experienced by the basic FOF implementation when the the linking parameter τ is relatively large.

Parallel group finders, such as pHOP [27], HaloWorld [30], Amiga Halo Finder (AHF) [17] and Ntropy [14], are implemented atop the Message Passing Interface (MPI) [28] and are designed to execute on parallel distributed memory machines with a fast low-latency interconnect between the processing nodes. These implementations are suitable for finding groups during the execution of the simulation. Ntropy is implemented as a framework atop MPI that provides a distributed *kd*-tree abstraction. One of its goals is to make it easy to implement analysis algorithms, such as FOF, on massively parallel distributed memory platforms. The user writes a function for reading the data that the N-tropy framework calls at run time to load the data into memory.

All the aforementioned approaches require the complete dataset to fit in memory to execute. Their reported performance exclude the time needed to load the data and assume that the data is already in memory, which makes sense for the case where these approaches are used in numerical simulations. Recently, Kwon et al. developed an approach atop Dryad that shares similarities with the work presented here [24]. They have shown results for clustering datasets with up to 1 billion particles using 8 nodes.

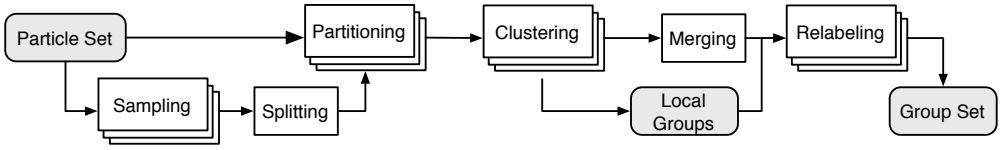

Figure 2: DiscFinder Pipeline. The DiscFinder computation flowgraph comprises the following stages: 1. Sampling (dist), 2. Splitting (seq), 3. Partitioning (dist), 4. Clustering (dist), 5. Merging (seq), and 6 Relabeling (dist).

4 DiscFinder

DiscFinder is a distributed approach for group finding in datasets that may be larger than the total aggregate memory of computers performing the computation. DiscFinder enables the use of stand-alone sequential group finders in a distributed setting, and thus leveraging the effort that has been put in the development of those sequential implementations. At a high level, the basic idea behind DiscFinder is to take a large *unordered* input particle set, then organize it and split it into multiple spatial regions and find groups in each region using the sequential group finder. The resulting groups from each region are merged to obtain the global set of groups. Figure 2 depicts the stages involved in the DiscFinder pipeline. As in many other computational sciences applications that deal with a spacial phenomena, DiscFinder uses spatial partitioning to split and distribute the computation across processing units. Unlike other applications, there is no explicit communication in the algorithms used at the application level. Instead, the needed data is moved by an underlying framework or through a distributed file system.

Particle Set. The input particle dataset contains tuples of the form $\langle \texttt{parId}, \texttt{pos}, \texttt{oAttr} \rangle$, where `parId` is the particle identifier, `pos` is the position of the particle in 3D space, and `oAttr` are other particle attributes such as mass, temperature, velocity, and so on. For the purpose of clustering using the FOF criteria, we are only interested in `parId` and `pos`. Other attributes can be safely ignored and not read at all.

Sampling. The objective of this stage is to sample the dataset in order to build a suitable partition of the data so the processing can be split into independent compute nodes. Since the particles positions in space is the primary criteria for determining the groups, the generated partitions correspond to regions of space. The simplest approach is to divide the space into equal-size disjoint regions, such as 3D cubes. We have a-priori knowledge that points in real-world astronomy datasets, whether generated by simulation or collected by observation, rarely follow a uniform distribution. This makes it undesirable to simply split the space into equal-size cubes, as this partitioning scheme generates partitions with very disparate number of points per partition, and thus becomes hard to balance the load. A tree-based data structure, such as a *kd*-tree, can be used to split the domain space into cuboid regions with equal number of points. This process is memory intensive and the time complexity of building such *kd*-tree using a median-finding algorithm is $O(n \log n)$, where n is the number of particles. The resulting cuboid regions are axis-aligned rectangular hexahedra, which we simply refer to as boxes.

To deal with the size of the input dataset, instead of building a *kd*-tree for the whole dataset, we build a much smaller *kd*-tree with randomly selected sample points. The sampling stage is performed in a fully data-parallel manner. The input dataset is divided into many disjoint pieces such as file offset ranges or subsets of files for datasets made up of many files. Then a worker task

can sample each of these subsets by choosing a small percentage of the records, e.g., 0.01%, and output the position of the sampled records. There is a sample set per worker task.

Splitting. This is a sequential step that creates the split boxes used for partitioning the particles. It takes the sample sets produced by the sampling tasks in the previous step and builds a *kd*-tree that can be used to partition the input particle set into boxes with roughly the same number of particles per box. The granularity of the boxes, and thus the depth of the tree, is chosen so the number of particles per box fits in the memory of a worker task in the cluster stage of the pipeline. This is a requirement for executing the sequential group finder in the cluster stage. This yields a number of partitions that is usually much larger than the number of available worker tasks for the pipeline. The number of worker tasks is proportional to the number of available compute nodes, e.g., 4 to 8 worker tasks per compute node. If the number of partitions is less than the number of worker tasks that means the pipeline can be easily executed with fewer computers.

Shell-based Partitioning. This stage takes the input particle set and sorts it according to the partitions produced by the splitting phase. Splitting the data in disjoint spatial partitions requires communication across neighboring partitions in order to determine whether a group crosses a partition boundary. For example, the domain shown in Figure 1 is divided into 4 rectangular partitions. Although, particles p_3 and p_4 are in separate partitions, they are close enough to belong to the same group. DiscFinder is designed so each partition can be processed independently and asynchronously. Partitions can be processed at different points in time. DiscFinder is targeted to execute atop data processing frameworks, such as MapReduce, Hadoop, or Dryad, that do not provide explicit communication primitives for the applications.

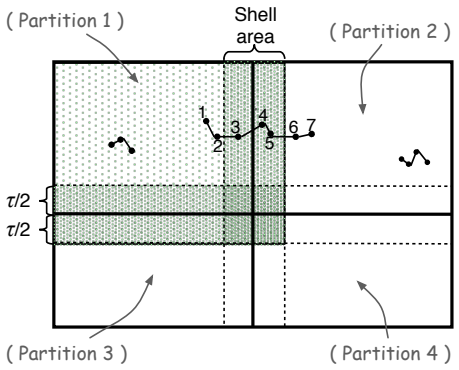

Figure 3: Shell-based Partitioning Four partitions are shown here in two dimensions. Each partition has an additional shell of length $\tau/2$ along the boundary. The shaded area at the top left of the figure represents the size of the partition once it has been extended to include its shell.

DiscFinder uses a *shell-based partitioning* scheme to avoid explicit communication at the cost of a small increase in the memory requirement for each partition. All the partitions are extended by a small factor $sl = \tau/2$ on each side, where τ is the linking length parameter for the group finder. The end result is that a partition has a shell around it that contains points shared with adjacent partitions. Shell-based partitioning enables the independent and asynchronous processing of each partition by decoupling the local computation of groups inside a partition from the resolution of groups that span across partitions.

To illustrate the approach, consider a 2D domain made of 4 partitions as shown in Figure 3. The non-overlapping partitions are delimited by the heavy continuous line. Each partition is extended

along each boundary by an additional length of $\tau/2$ (shaded area). The particles numbered from 1 to 7 in the figure form a group. Without the shell, two disjoint groups would be created. The group in the top-left partition would contain three of the points $(1,2,3)$ and the group in the top right partition would comprise the other four points in the group $(4,5,6,7)$. Joining these two non-overlapping groups requires communication across the tasks that process these partitions. Once the partitions are extended with the shell, the top-left partition finds a group $G_1 = 1,...,5$; the top-right partition contains a group with 5 points $G_2 = 3,...,7$. The members of these two groups overlap, which facilitates the unification of the groups in a separate merging stage down the pipeline.

The choice of the value for *sl* ensures that when two points in separate partitions are at a distance $d(p_i, p_j) < \tau$, then at least one of them is included in the shell of one of the partitions, which is enough to later find the groups that span multiple partitions. Note that shell-based partitioning increases the volume of each partition and potentially the number of particles to be processed per partition. This approach works under the assumption that the size of the shell is relatively much smaller than the size of the partition. In practice, this assumption holds because the chosen linking length parameter τ is relatively small compared to the edge length of each partition.

The partitioning stage is distributed across many worker tasks. Each task reads a chunk of the input particles and classifies them into buckets that correspond to the spatial partitions (including their shells). Each bucket contains a subset of the particles belonging to a spatial partition.

Distributed Clustering. In this stage, all the buckets corresponding to a partition are collected so a worker task has all the particles in a shell-extended partition. The partitions are distributed to worker tasks. Locally, each task executes a sequential group finder, such as the UW FOF or aFOF implementations [37, 36]. For each point in a partition, the local group finder generates a $\langle$`pointId`,`pGroupId`$\rangle$ tuple, where the group identifier `pGroupId` is local to each partition. The generated tuples are separated into two sets: *shell set (S)* and *interior set (I)*. The shell set contains points inside the shell region only. This set is passed to the merging stage. The interior set contains points that fall inside the original disjoint partition, but not in the extended shell of any partition. This set is stored and used again later in the relabeling stage. At the end of the distributed clustering stage there are P shell sets and P interior sets, where P is the number of partitions generated by the splitting phase.

Merging As shown in Figure 3, a point inside a partition's shell is processed in two or more partitions. Such a point may belong to different groups across partitions. Figure 4 shows the resulting groups for two partitions (1 & 2) of Figure 3. Points $3,4,5$ are assigned to group 1 in partition 1 and to group 2 in partition 2. The purpose of the merging stage is to consolidate groups that span multiple partitions by using P shell sets (S) generated in the previous stage. The amount of data passed to this stage is relatively much smaller than the size of the particle datasets (since τ is very small), thus reducing both compute and I/O requirements for this stage.

The merging stage employs a *Union-Find* algorithm to merge subgroups that have common points into unique global groups [13, 12]. Union-Find uses a *disjoint-set* data structure and to maintain several non-overlapping sets, each one containing its own elements. This data structure supports the following two operations: `Union` and `Find`. `Union(A,B)` merges two sets A and B and replaces them with their union. `Find(e)` determines to which set an element e belongs. For the purpose of the group merging stage, a group corresponds to a set in the disjoint-set data structure, and a particle corresponds to an element. Initially, each particle belongs to its own group. The procedure iterates over all the particles and when it discovers that a particle belongs to two

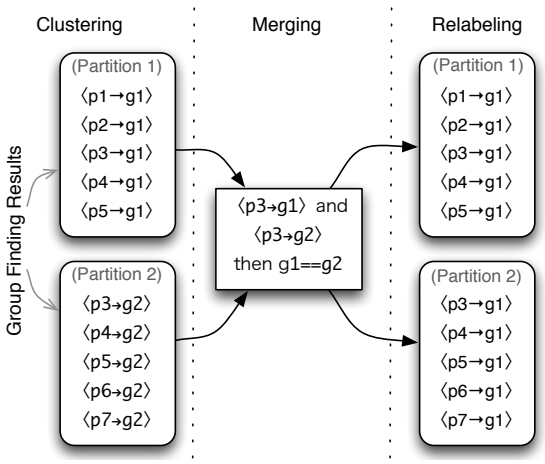

Figure 4: Merging Stage The shell sets for partitions 1 and 2 in Figure 3 contain groups that share common points. The Merging Stage consolidates the sub-groups that span multiple partitions.

different groups, then it merges those two groups using the `Union` operation. The output of the merging procedure is a list of *relabeling rules* that describe the equivalent groups, i.e., the groups that span across partitions. These rules are tuples of the form $\langle \texttt{oldGroupId}, \texttt{newGroupId} \rangle$. A set of relabeling rules is produced for each partition.

This data structure and the corresponding Union-Find algorithm can be implemented using hash tables. The complexity of the Union-Find algorithm is nearly linear in the number of input elements [34]. Since it is only applied to a subset of the particles, the ones inside the shells, its running time is relatively short compared to the rest of the pipeline. Since this algorithm is sequential, its scalability is limited by the memory available in a single processing node. We have used it to execute pipelines with close to 15 billion particles. Until now, its memory requirements have not been an issue.

Relabeling. This is the last stage in the pipeline. Its purpose is to apply the relabeling rules from the merging stage to each of the partitions generated in the clustering stage. The is a data-parallel process. The work is distributed to many tasks, each task applies the rules to a partition. This is done in a single pass. The partitions are expected to be evenly balanced, thus the running time of this step is proportional to N/m, where N is the total number of particles and m is the number of available processing units.

5 Pipeline Implementation

The DiscFinder algorithms are designed to be implemented using a MapReduce style of computation [8]. We implemented the DiscFinder pipeline atop Hadoop – an open-source, free implementation of the MapReduce framework [35]. Alternatively, the DiscFinder pipeline could be implemented as a collection of programs that use a programming interface such as MPI or OpenMP [4] and are glued together with scripts that execute with the help of a resource manager such as PBS [2]. Our objective is to explore how the MapReduce programming model, and frameworks such as Hadoop, can be used to tackle large-scale data-intensive analytics found in many science domains.

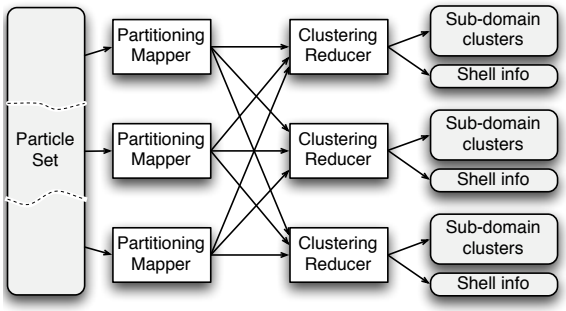

Figure 5: DiscFinder partition and clustering stages. This is the central MapReduce job in the DiscFinder pipeline

Hadoop. Hadoop is a distributed framework that has become very popular among the Internet services companies. It is widely used at companies such as Yahoo! and Facebook for their analytics applications for web graph and social network data. Applications for the framework use a MapReduce programming model. Hadoop has a few features that make it attractive for large-scale data-intensive processing: its support for processing datasets larger than the available memory of the compute resources; its ability to scale the processing to handle very large datasets; its resilience to individual component failures; its relatively simple programming model; and, its community-supported open-source implementation.

As a first approximation, Hadoop provides a mechanism for applying a user-defined procedure (*map function*) over a large dataset, then grouping the results by a key produced by the map function, and then applying another user-defined procedure (*reduce function*) to the resulting groups. The execution of these functions is distributed over a compute cluster. The framework handles the partitioning of the input data, the data communication, the execution of the tasks and the recovery from errors such as hardware failures and software crashes. For example, Figure 5 illustrates a MapReduce job that belongs to the DiscFinder pipeline. The input particle dataset is split across three map tasks that perform the partitioning stage by assigning a key (partition box id) to each input record (particle). The framework collects all the intermediate records that have the same key (box id) such that a reducer process receives all the records with the same box id. Separate distributed processes execute the reduce function that, in this case, perform the clustering pipeline stage. Each reducer produces a subset of the output dataset.

The map and reduce functions may execute in the same set of computers. The datasets are available to the tasks on a distributed file system (DFS). Hadoop works with a variety of DFSs including parallel file systems, such as PVFS2, and Hadoop's own HDFS. HDFS is modeled after the Google File System [16]. It is designed to aggregate the storage capacity and I/O bandwidth of the local disks in a compute cluster. This allows for scaling up the capacity and bandwidth by adding computers to the file system. In a typical Hadoop deployment, the computation is carried out in the same computer used for HDFS. The Hadoop job scheduler makes an effort to place the computation in the hosts that store the data.

DiscFinder Pipeline. The DiscFinder pipeline (Fig 2) is implemented as a succession of Hadoop MapReduce jobs and sequential programs written in Java. The sampling and splitting stages are implemented as one MapReduce job, the partitioning and clustering phases make up another job. The merging stage is implemented as a stand-alone Java program. The relabeling phase is a distributed map-only job.

The procedures shown below are the high-level driver for the DiscFinder pipeline. It takes as input the astrophysics dataset P and the grouping parameters (here τ) and produces as output a list with mappings from particles to groups. Remember that the input set P contains a set of tuples where each tuple corresponds to a particle in the dataset. The tuple contains information about the particle such as its identifier and other attributes such as its position and velocity among others.

Procedure DiscFinder(P, τ): Driver for the DiscFinder pipeline.

Input: P is a set of N points, where each point p_i is of the form `<parId, attributes>`
Input: τ is the linking distance parameter for FOF
Output: Mapping of particles to groups.

```
   // Run a map reduce job for the sampling phase
1  MapReduce( SamplingMap, SamplingReduce );
   /* Use map reduce for point partitioning points and running the group
      finder distributedly                                              */
2  MapReduce( PartitionMap, ClusteringReduce );
3  CollectShell() // Collect the shell data in a central location
4  UnionFind() // Merge groups across partitions
   // Finally, use a map-only job to relabel the points
5  MapReduce( RelabelMap, NullReducer );
```

The first step (Line 1) corresponds to the dataset sampling in the processing pipeline. It is carried out using a distributed MapReduce job. The map and reduce functions for this step are shown in detail below in the Procedures `SamplingMap` and `SamplingReduce`.

Procedure SamplingMap(Int Key, Particle Value)

Input: Key is a randomly generated integer value for each point.
Input: Value contains relevant payload values for the particle, such as `<parId, position>`
Input: c is a pre-defined prime integer to indicate sampling frequency. We use $c = 16001$ in our experiments.
Output: Sampled subset of particles

```
   /* The key is used to sample the input points, since the key is
      randomly generated                                            */
1  if Key % c == 0 then
2  |  EmitIntermediate(1, Value.position)
3  end if
```

Sampling and Splitting. The sampling stage is implemented as a map function in the first job. Each map task randomly selects a subset of particles. A single reducer in this job implements the splitting stage. In Line 5, `CubeSplit` is a sequential procedure that takes the output of the sampling to determine the partitioning scheme for the input dataset. Finally the partition information is written to HDFS.

Partitioning and Clustering. The second job executes the partitioning and clustering stages, which correspond to the the bulk of the processing in the pipeline. The partitioning stage is executed

Procedure SamplingReduce(Int Key, Iterator Values)

Input: Key as emitted by the SamplingMap function.
Input: Values contains the positions for the sampled particles.
Output: Partitioning scheme for the input dataset

```
1 List<Particle> particleList = null
2 foreach v in Values do
3 |  particleList.add(v)
4 end foreach
5 Output(CubeSplit(particleList))
  /* Use particleList to build a kd-tree and use τ to generate the
     boundary of each cube.                                        */
```

by the map tasks (See Figure 5). Each task takes a portion of the particle dataset and executes the function shown in Procedure `PartitioningMap`.This procedure is executed for each input particle. It receives as parameters the particle's identifier and position. First, it determines the partition in which this particle falls (Line 1) and emits a tuple with the partition (box.id) as the key, and the particle id and position as the values. Additional tuples are emitted for particles that fall in the shell of any other partition(s) (Lines 3–7).

Procedure PartitioningMap(pId, position)
Partition particles into overlapping boxes.

Input: pId → particle identifier.
Input: position → particle position (`<x,y,z>`).
Output: Tuple → ⟨key = `boxId`, `pId`, `position`⟩

```
1 box ←getPartitionBox( position )
2 EmitIntermediate( box.id, pId, position )
3 if position in shell(box) then
  |  // Emit tuples for adjacent boxes
4 |  foreach oBox: getAdjBoxes( position, box ) do
5 |  |  EmitIntermediate( oBox.id, pId, position )
6 |  end foreach
7 end if
```

The framework groups the tuples having the same box id and sends them to a reducer task and calls the ClusteringReduce procedure for each group of values that have the same key. This procedure executes the external group finder implementation (Line 1), specifically the UW FOF sequential implementation [37]. It involves creating an input file in the local file system for the external program, launching the executable and reading the output produced by the program from the local file system. The group membership information for particles that lie in the shell is stored in a separate file (Lines 4–6).

Merging. The merging stage is implemented as a sequential Java program that reads the files with the shell information, which were generated by the ClusteringReduce procedure, executes the union-find algorithm and writes out the relabeling rules only for groups that span across partitions.

Procedure ClusteringReduce(boxId, particles)
Apply the sequential group finder to a partition (box).

Input: boxId → partition identifier.
Input: particles → list of tuples ⟨`pId`, `position`⟩.
Output: Particle membership list ⟨`pId` → `gId`⟩. where `pId`: particle id, `gId`: group id.

```
// Run local group finder
particleToGroupMap = groupFind( particles )
foreach particle in particleToGroupMap do
    Emit( particle.id, particle.groupId )
    // Write in a separate file the group membership for particles in the
       shell
    if particle in shell(box) then
        Output( particle.id, particle.groupId )
    end if
end foreach
```

Relabeling. A map-only job (no reducer) implements the final relabeling stage. Each task reads the groups for a partition and the corresponding relabeling rules. For each tuple ⟨pid, gIdLocal⟩, a corresponding tuple ⟨pid, gIdGlobal⟩ is generated using the relabeling rules to map from local group (gIdLocal) to the equivalent global group (gIdGlobal).

6 Evaluation

The goal of our evaluation is to test whether DiscFinder is a feasible approach for clustering massive particle astrophysics datasets, and indirectly whether similar approaches can be used for other large-scale analytics in science. We want to measure and understand the overheads introduced by the DiscFinder algorithm and the Hadoop framework, and thus find ways to improve both the application and the framework. We conducted a set of scalability and performance characterization experiments. Below are the preliminary results.

Name	Particle count	Snap size	Snap count	Total size
BHCosmo	450 M	850MB	22	10.5 GB
Coyote Univ.	1.1 B	32GB	20	640 GB
DMKraken	14.7 B	0.5TB	28	14 TB

Figure 6: Cosmology simulation datasets

Datasets. In our evaluation we used snapshots (time slices) from the three different cosmology simulation datasets shown in fIgure 6. BHCosmo simulates the evolution of the universe in the presence of black holes [9]. Coyote Universe is part of a larger series of cosmological simulations

Procedure UnionFind(groupMapping): Execute the union-find algorithm for particles on the shell.

Input: groupMapping is a set of `<parId, groupId>` pairs.
Output: A set of group-transition rules.

```
1  parToGroupMap ←EmptyMap;
2  groupToGroupMap ←EmptyMap;
3  foreach (parId, groupId): groupMapping do
4      if groupId not in groupToGroupMap then
           // Add mapping:  groupId to groupId
5          groupToGroupMap.put( groupId, groupId )
6      end if
7      if parId not in parToGroupMap then
           // Add mapping:  parId to groupId
8          parToGroupMap.put( parId, groupId )
9      else
10         groupId* = parToGroupMap.get( parId );
11         while groupId not equal groupToGroupMap.get( groupId ) do
12             groupId = groupToGroupMap.get( groupId )
13         end while
14         while groupId* not equal groupToGroupMap.get( groupId* ) do
15             groupId* = groupToGroupMap.get( groupId* )
16         end while
17         if groupId not equal groupId* then
               // Merge (union) groups groupId and groupId*
18             groupToGroupMap.put( groupId, groupId* )
19         end if
20     end if
21 end foreach
22 foreach (groupId, groupId') : groupToGroupMap do
23     if groupId not equal groupId' then
24         while groupId' not equal groupToGroupMap.get( groupId' ) do
25             groupId' = groupToGroupMap.get( groupId' )
26         end while
27         Output <groupId, groupId'> to group-transition rule file.
28     end if
29 end foreach
```

carried out at Los Alamos National Laboratory [20]. DMKraken is a 14.7 billion dark-matter particle simulation carried out by our collaborators at the CMU McWilliams Center for Cosmology. These datasets are stored using the GADGET-2 format [32].

Experimental Setup. We carried out the experiments in a data-intensive compute facility that we recently built, named the DISC/Cloud cluster. Each compute node has eight 2.8GHz CPU cores in two quad-core processors, 16 GB of memory and four 1 TB SATA disks. The nodes are

Procedure RelabelMap(Int key, Int value) Relabel groups and particles for each partition. A mapper operates on a set of particles in a cube partition. Each entry in this set is of the form `<parId, groupId>`

Input: Group-transition rules produced by the Union-Find procedure.
Input: Key is the particle ID (`parId`).
Input: Value is the group ID (`groupId`).
Output: Relabeled particle set.

```
1 if exists mapping from groupId to groupId' then
      // Relabel by changing groupId to groupId'
2     Emit(parId, groupId')
3 else
      // Emit original result
4     Emit(parId, groupId)
5 end if
```

connected by a 10 GigE network using Arista switches and QLogic adapters at the hosts. The nominal bi-section bandwidth for the cluster is 60 Gbps. The cluster runs Linux (2.6.31 kernel), Hadoop, PVFS2 (2.8.1) and Java (1.6). The experiments were carried out using Hadoop 0.19.1. The compute nodes serve both as HDFS storage servers and worker nodes for the MapReduce layer. A separate set of 13 nodes provide external RAID-protected storage using PVFS2. Hadoop is configured to run a maximum of 8 simultaneous tasks per node: 4 mappers and 4 reducers.

Datasets Pre-processing The original input data is in the GADGET-2 format, which is a binary format used by astrophysicists. To fit it naturally to the Hadoop framework and speed up implementation, we converted all the input data to a simple text format. The whole conversion costs significantly (e.g. 11 hours for the 14.7 Billion point dataset), although each dataset only needs to be processed once.

6.1 Scalability Experiments.

In this subsection We want to answer the question that whether DiscFinder can be used to cluster datasets that are much larger than the aggregate memory of the available compute resources, and how its running time changes as we vary the available resources. Similar to the evaluations of compute-intensive applications, we performed weak and strong scaling experiments. However, the results are presented in terms of number of compute nodes, as opposed to number of CPUs. This is more representative of the evaluated application, where the memory capacity and I/O bandwidth are the bottlenecks. Due to the nature of how tasks are managed in Hadoop, there is no one-to-one mapping between tasks and number of CPU cores. The results reported below are the average of three runs with system caches flushed between runs. Some of the running times include partial task failures that were recovered by the framework and allowed the job to complete successfully.

In the weak scaling experiments, the work per host is kept constant and the total work grows proportional as compute resources are added. In the strong scaling experiments, the total work is kept constant and the work per host changes inversely proportional to the number of compute nodes. We varied the number of compute hosts from 1 to 32. The same nodes were used for HDFS as well. This reflects a Hadoop/HDFS deployment in a cluster with the given number of nodes.

The results are shown in figures 7 and 9. The X axis in these figures is the cluster size (number of worker nodes) in log scale.

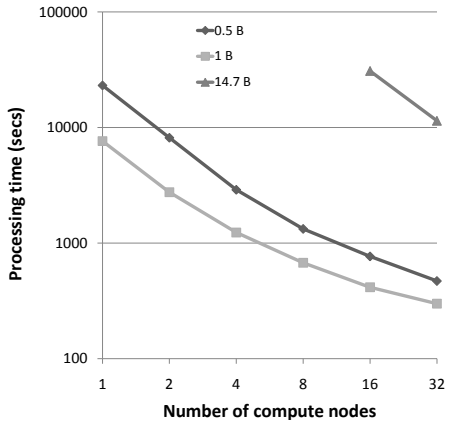

Figure 7: Strong scaling.

Nodes	0.5 B	1.1 B	14.7 B
1	23107	7625	n/a
2	8149	2754	n/a
4	2890	1234	n/a
8	1326	675	n/a
16	767	415	30816
32	470	299	11429

Figure 8: Strong scaling values in seconds.

Strong Scalability. Figure 7 shows the strong scalability of the DiscFinder approach. The Y axis is the running time in log scale. The curves correspond to different dataset sizes of 14.7, 1 and 0.5 billion particles. Notice that the 14.7 billion dataset is larger than the memory available for the experiments (16 and 32 nodes). The same applies for various scenarios in the cases for 1 and 0.5 billion particles. Linear scalability corresponds to a straight line with a slope of -1, where the running time decreases proportional to the number of nodes. The DiscFinder running time is not linear. With a small number of nodes, the running time suffers because each node performs too much work, and with a larger number of nodes each node performs too little work and the framework overhead dominates.
Weak Scalability. In order to perform the weak scaling experiments, we extracted sub-regions of the different datasets to match the appropriate size needed for the experiment The Y-axis in the weak scaling graph (Figure 9) is the elapsed running time in seconds (linear scale). The curves correspond to different problem sizes of 64 (top) and 32 (bottom) million particles per node. The right most point in each curve corresponds to a total size of 1 billion particles.

The best running time to compute nodes has a sweet spot around 4 to 8 nodes for datasets smaller than 1 billion particles. We expect non-negligible overheads, introduced by the framework and the approach, due to the incurred I/O, data movement and non-linear operations such as the shuffle/sort in the job that performs the partitioning and clustering stages. A non-negligible amount

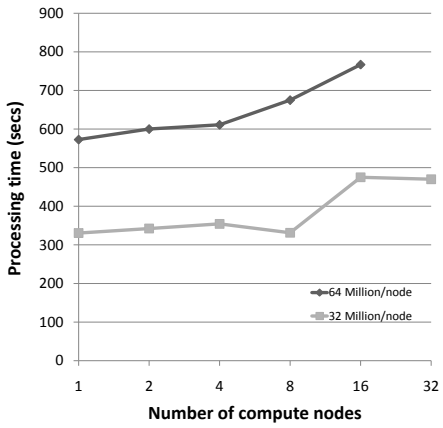

Figure 9: Weak scaling.

Nodes	60 M/n (sec)	32 M/n (sec)
1	573	331
2	600	342
4	611	354
8	675	331
16	767	475
32	n/a	470

Figure 10: Weak scaling values in seconds.

of memory in each node is used for the HDFS processes. The shuffle/sort operation also uses large amounts of memory per reducer task when the data sizes are large. The running time in all cases exhibits a non-linear trend, especially for larger number of nodes where the framework overheads dominate the running time. Even with all the aforementioned overheads and task failures, it was possible to process and cluster particle datasets that were much larger than the available memory.

6.2 Performance Characterization.

We are interested in gaining insight about the DiscFinder running time. We performed additional experiments to break down the total running time into the time spent in each stage of the pipeline. In addition, we split the running time of the second job (Partitioning and Clustering) into the phases corresponding to finer-grained operations in that job. For this set of experiments we used 32 nodes and focused on the largest (14.7 billion particles) dataset. The phases for the running time breakdown are the following.

1. *Sampling*: This corresponds to the the splitting stage of the pipeline.
2. *Splitting*: Stage that builds a kd-tree to spatially partition the space.
3. *Loading input data*: This refers to reading the particle dataset from HDFS. This step is performed in the map phase of the main MapReduce job (Partitioning and Clustering).

4. *Loading box index*: Time spent loading the index that contains the partition information. This is done in the map phase of the second job.

5. *Box search*: This is the time required to search the spatial partition information in the map phase of the second job.

6. *Shuffling / Sorting*: This is the time required to move data from the mappers to the reducers in the second job.

7. *FOF data generation*: This is the time required in the reducer to generate the input data for the the external group finder program.

8. *FOF execution*: This is the time needed to run the external group finder.

9. *Merging*: Pipeline merging stage.

10. *Relabeling*: Pipeline relabeling stage.

We conducted extra experiments from the start of the pipeline to each of above phase. Using the time difference between these experiments we have acquired the running time of each phase. For instance, the running time of step4 is measured by the difference between experiment of phases 1-4 and experiment of phases 1-3.

The detailed breakdown of the running time for each of these steps is shown in Figure 11. Columns 2 and 3 show the absolute (in seconds) and relative time for the unmodified implementation of the pipeline. The relative time is expressed as a percentage of the total running time. The data shows that the Sampling/Splitting, Box search and Shuffle/sort steps account for about 80% of the running time. The time spent in the sampling/splitting step was unexpectedly high. This is a performance bug introduced by an early optimization. The long time spent in the box search step was due to an inefficient implementation of the search. An incorrect setting of a Hadoop parameter caused the shuffle/sort step to take longer than necessary.

Step	Base		Improved		Speedup
	Sec	Rel %	Sec	Rel %	X
Sampling	1555	13.4%	859	22.5%	1.8
Splitting	62	0.5%	62	1.6%	1.0
Load particles	229	2.0%	232	6.1%	1.0
Load box idx	3	0.0%	12	0.3%	0.3
Box search	3422	29.5%	122	3.2%	28.0
Shuffle / sort	4363	37.6%	1000	26.2%	4.4
FOF data gen.	762	6.6%	320	8.4%	2.4
FOF exec	576	5.0%	584	15.3%	1.0
Merging	151	1.3%	137	3.6%	1.1
Relabeling	486	4.2%	482	12.7%	1.0
Total	11609	100.0%	3810	100.0%	3.0

Figure 11: Breakdown of the DiscFinder elapsed time needed to find groups in a snapshot of the DMKraken 14.7 billion particle dataset on 32 worker nodes.

Performance Improvements. We performed a set of optimizations to improve the overall performance of the pipeline. The breakdown of the running time for the improved version is shown

in columns 4 and 5 of Figure 11. Column 4 has the absolute time for each step in seconds and Column 5 contains the relative time with respect to the total running time of the improved version. Column 6 shows the speedup for that step relative to the baseline. The overall speedup is 3.1X, However, do not read too much into this result. What it really means is that it is very easy to introduce performance bugs that may lead to inefficient implementations. Below are the anecdotes of our debugging experience.

- Fixing the Sampling/splitting Phase. During the early development stages, a "performance optimization" was introduced in the sampling phase. The idea was to avoid executing the sampling/splitting stages in a separate job, and instead integrate them in the map phase of the second job. Since the sampling is expected to read a relatively small portion of the input dataset (in the order of 0.01%), it made sense to have all the tasks perform the full sampling in parallel and construct the spatial partitioning locally. This can be achieved by ensuring that all tasks sample the same particles, e.g., by using the same seed for the random number generator. Since the Hadoop framework handles the reading of the records, it is not straightforward to choose what records get read. The initial sampling implementation would simply emit a very small fraction of the records and discard the rest. This was acceptable for the intended purpose, which was to reduce the number of particles used in the construction of the kd-tree. However, this meant that all the map tasks were reading the complete dataset in order to do the sampling. The current solution consists of performing the sampling outside Hadoop in a separate stage as originally intended. The sampling time is still relatively high. We are working in alternate solutions to further decrease the sampling time using some of the sampling facilities available in more recent versions of Hadoop (0.20.x).

- Speeding up Box Lookups. The performance of the box lookup was affected by the initial implementation choice. Since the number of partitions is relatively small, we used a simple linear search mechanism for this structure. However, this box lookup is performed at least once for every particle, and in some cases more than once when the particle is in a shell region. On aggregate, the lookup time becomes significant. Replacing the lookup mechanism with a routine that uses a $O(\log n)$ algorithm, where n is the number of partitions, provided major benefits.

- Adjusting Number of Reducers. In this set of experiments the particles are split into 1024 spatial partitions. At first, it was only natural for the domain application developer to set the total number of reduce tasks equal to the total number of partitions, such that, a reduce task would process a partition in the same way that in computational sciences applications the number of partitions matches the number of processes and processing units. During the execution of the original DiscFinder implementation, we noticed in our cluster monitoring tool that the network bandwidth consumption had a cyclic behavior with periods of high utilization followed by idle periods. This effect was caused by multiple waves of reduce tasks fetching data produced by the map tasks. By setting the number of reducers for the job to $(\text{numberOfNodes} - 1) * \text{reducersPerNode} = 124$, all the reduce tasks are able to fetch and shuffle the data in a single wave, even in the presence of a single node failure. In this case, each reduce task processes multiple domain partitions. This adjustment required no code changes, as this is the normal mode of operation for the framework.

Although, the running time significantly decreased after these modifications, there is clearly room for additional improvement, both at the algorithmic level in the application and in terms of efficiency in the framework.

6.3 Future optimizations

- Fine tunning the size of the memory buffers used for the shuffle/sort step in Hadoop jobs.
- Re-designing the pipeline so the external FOF can run on the map phase as opposed to the reduce phase in order to make more efficient use of the memory, which would allow for higher parallelism and more efficient use of the memory.
- Reading binary data directly: Loading the data using a binary Hadoop reader to avoid pre-processing step.
- Better sampling and box lookup implementation.
- Shuffle: Text vs. binary shuffle transfers. Using a binary representation for transferring data between map and reduce during the shuffle phase.
- Shuffle: Compressing the intermediate tuples, to reduce the network I/O.
- Memory management issues related to overcommitted virtual memory.

7 Conclusion

The analysis of state-of-the-art and future astrophysics datasets is difficult due to their size. Group finding and other analysis techniques need to be scaled to operate on these massive datasets. DiscFinder is a novel data-intensive group finder that scales to datasets with tens of billions of particles. DiscFinder has enabled the analysis of the largest state-of-the-art cosmology datasets. Nevertheless, its first implementation has relatively high overheads introduced by application algorithms and the framework implementation. We described different approaches to improve its performance. As the analysis of this truly very large datasets requires a data-intensive approach, there are opportunities and needs to improve the performance and extend the programming models to better support analytics applications for science.

References

[1] ABAZAJIAN, K., ET AL. The Seventh Data Release of the Sloan Digital Sky Survey. *Astrophysical Journal Suplement (ApJS) 182* (June 2009), 543–558.

[2] BAYUCAN, A., HENDERSON, R. L., LESIAK, C., MANN, B., TOMPROETT, AND TWETEN, D. *Portable Batch System (PBS), External Reference Specification*. MRJ Technology Solutions, 2672 Bayshore Parkway, Suite 810, Mountain View, CA 94043, 1999.

[3] BOTZLER, C. S., SNIGULA, J., BENDER, R., AND HOPP, U. Finding structures in photometric redshift galaxy surveys: an extended friends-of-friends algorithm. *Monthly Notices of the Royal Astronomical Society (NMRAS) 349* (Apr. 2004), 425–439.

[4] CHANDRA, R., MENON, R., DAGUM, L., KOHR, D., MAYDAN, D., AND MCDONALD, J. *Parallel Programming in OpenMP*. Morgan Kaufmann, 2000. ISBN 1558606718.

[5] COLBERG, J., AND DI MATTEO, T. Supermassive black holes and their environments. *Monthly Notices of the Royal Astronomical Society (NMRAS)*, 387 (2008), 1163.

[6] COUCHMAN, H. M. P., PEARCE, F. R., AND THOMAS, P. A. Hydra code release, 1996.

[7] DAVIS, M., EFSTATHIOU, G., FRENK, C., AND WHITE, S. The evolution of large scale structure in the universe dominated by cold dark matter. *Astrophysical Journal 292* (1985), 371–394.

[8] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. In *Symp. on Operating System Design and Implementation (OSDI)* (San Francisco, CA, Dec 2004).

[9] DI MATTEO, T., ET AL. Direct cosmological simulations of the growth of black holes and galaxies. *Astrophysical Journal (ApJ)*, 676 (2008), 33.

[10] DIKAIAKOS, M. D., AND STADEL., J. A performance study of cosmological simulations on message-passing and shared-memory multiprocessors. In *ICS Conference* (1996).

[11] EISENSTEIN, D. J., AND HUT, P. Hop: A new group finding algorithm for N-body simulations. *Astrophysical Journal (ApJ) 498* (1998), 137–142.

[12] GALIL, Z., AND ITALIANO, G. F. Data structures and algorithms for disjoint set union problems. *ACM Comput. Surv. 23*, 3 (1991), 319–344.

[13] GALLER, B., AND FISCHER, M. An improved equivalence algorithm. *Communications of the ACM (CACM) 7*, 5 (1964), 301–303.

[14] GARDNER, J. P., CONNOLLY, A., AND MCBRIDE, C. A framework for analyzing massive astrophysical datasets on a distributed grid. In *Proc. conf. on Astronomical Data Analysis & Software Systems (ADASS XVI)* (2006).

[15] GELB, J. M., AND BERTSCHINGER, E. Cold dark matter 1: The formation of dark halos. *Astrophysical Journal*, 436 (1994), 467–490.

[16] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google File System. In *Proc. 19th Symposium on Operating Systems Principles (SOSP'03)* (New York, NY, USA, 2003), ACM, pp. 29–43.

[17] GILL, S. P. D., KNEBE, A., AND GIBSON, B. K. The evolution of substructure - I. A new identification method. *Monthly Notices of the Royal Astronomical Society (NMRAS) 351* (June 2004), 399–409.

[18] GOTTLOEBER, S. Galaxy tracers in cosmological n-body simulations. In *Proceedings of the Potsdam Cosmology Workshop: Large Scale Structure: Tracks and Traces* (Sept 1997).

[19] HEITMANN, K., LUKIĆ, Z., FASEL, P., HABIB, S., WARREN, M. S., WHITE, M., AHRENS, J., ANKENY, L., ARMSTRONG, R., O'SHEA, B., RICKER, P. M., SPRINGEL, V., STADEL, J., AND TRAC, H. The cosmic code comparison project. *Computational Science and Discovery 1*, 1 (Oct. 2008), 015003–+.

[20] HEITMANN, K., WHITE, M., WAGNER, C., HABIB, S., AND HIGDON, D. The coyote universe i: Precision determination of the nonlinear matter power spectrum, 2008.

[21] HUCHRA, J. P., AND GELLER, M. J. Groups of galaxies. I - Nearby groups. *Astrophysical Journal (ApJ) 257* (June 1982), 423–437.

[22] IVEZIC, Z., ET AL. LSST: from science drivers to reference design and anticipated data products. http://www.lsst.org/lsst/science/overview, 2008.

[23] KAISER, N., ET AL. Pan-STARRS: A Large Synoptic Survey Telescope Array. In *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series* (Dec. 2002), J. A. Tyson & S. Wolff, Ed., vol. 4836 of *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference*, pp. 154–164.

[24] KWON, Y., NUNLEY, D., GARDNER, J. P., BALAZINSKA, M., HOWE, B., AND LOEBMAN, S. Scalable clustering algorithm for n-body simulations in a shared-nothing cluster. Tech. Rep. UW-CSE-09-06-01, University of Washington, June 2009.

[25] LEE, D., AND WONG, C. Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees. *Acta Informatica*, 9 (1977), 23–29.

[26] LIU, H. B., HSIEH, B. C., HO, P. T. P., LIN, L., AND YAN, R. A new galaxy group finding algorithm: Probability friends-of-friends. *Astrophysical Journal (ApJ) 681*, 2 (2008), 1046–1057.

[27] LIU, Y., KENG LIAO, W., AND CHOUDHARY, A. Design and evaluation of a parallel hop clustering algorithm for cosmological simulation. *Proc. Parallel and Distributed Processing International Symposium* (April 2003), 8 pp.

[28] MPI FORUM. MPI: A Message Passing Interface. In *Proc. Supercomputing '93* (Portland, OR, Nov 1993), ACM/IEEE, pp. 878–883.

[29] PANSTARRS CONSORTIUM. Panoramic survey telescope & rapid response system (PanSTARRS). `http://pan-starrs.ifa.hawaii.edu`.

[30] PFITZNER, D. W., AND SALMON, J. K. Parallel halo finding in N-body cosmology simulations. In *2nd Conference on Knowledge Discovery and Data Mining (KDD-96)* (1996), AAAI Press, pp. 26–31.

[31] PFITZNER, D. W., SALMON, J. K., AND STERLING, T. Halo World: Tools for parallel cluster finding in astrophysical N-body simulations. *Data Mining and Knowledge Discovery 1*, 4 (December 1997), 419–438.

[32] SPRINGEL, V. The cosmological simulation code gadget-2. *MNRAS 364*, 4 (2005), 1105–1134.

[33] SPRINGEL, V., WANG, J., VOGELSBERGER, M., LUDLOW, A., JENKINS, A., HELMI, A., NAVARRO, J. F., FRENK, C. S., AND WHITE, S. D. M. The Aquarius Project: the subhaloes of galactic haloes. *Monthly Notices of the Royal Astronomical Society (NMRAS) 391* (Dec. 2008), 1685–1711.

[34] TARJAN, R. E. Efficiency of a good but not linear set union algorithm. *J. ACM 22*, 2 (1975), 215–225.

[35] THE APACHE FOUNDATION. The Hadoop Project. `http://hadoop.apache.org`.

[36] UNIVERSITY OF WASHINGTON N-BODY SHOP. aFOF: Approximate group finder for N-body simulations. `www-hpcc.astro.washington.edu/tools/afof.html`.

[37] UNIVERSITY OF WASHINGTON N-BODY SHOP. FOF: Find groups in N-body simulations using the friends-of-friends method. `www-hpcc.astro.washington.edu/tools/fof.html`.

[38] UNIVERSITY OF WASHINGTON N-BODY SHOP. Skid: A tool to find gravitationally bound groups in n-body simulations. `www-hpcc.astro.washington.edu/tools/skid.html`.

[39] UNIVERSITY OF WASHINGTON N-BODY SHOP. SOF: Spherical overdensity group finder for N-body simulations. `www-hpcc.astro.washington.edu/tools/so.html`.

[40] WHITE, M. The mass function. *The Astrophysical Journal 143* (2002), 241.