Data-Parallel Halo Finding with Variable Linking Lengths

Wathsala Widanagamaachchi*
SCI Institute, University of Utah

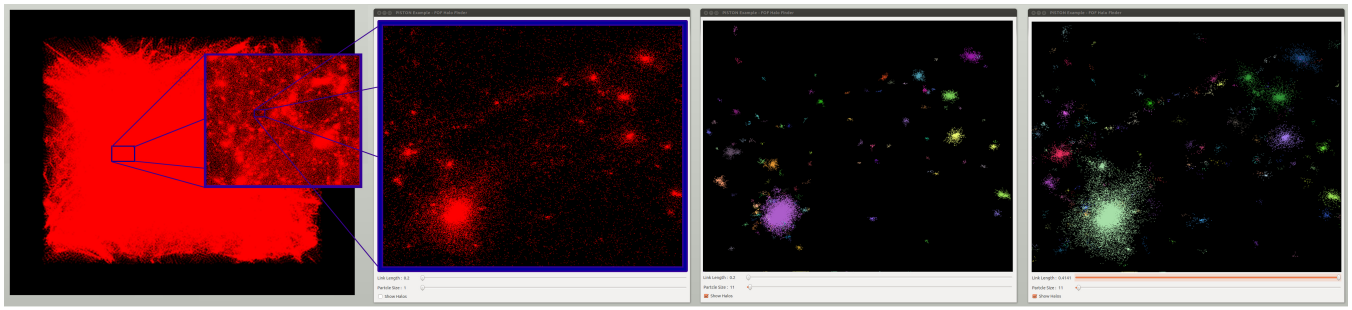
Peer-Timo Bremer†
SCI Institute, University of Utah
Lawrence Livermore National Laboratory

Christopher Sewell‡
Los Alamos National Laboratory

Li-Ta Lo§
Los Alamos National Laboratory

James Ahrens¶
Los Alamos National Laboratory

Valerio Pascucci‖
SCI Institute, University of Utah



(a) (b) (c)

Figure 1: Halos found on a set of particles from a cosmological simulation: (a) Dataset contains 16.7 million dark matter particles. (b) Halos with linking length 0.2 and halo size 11. (c) Halos after increasing the linking length to 0.4141. Each set of particles in the same color in (b) and (c) represents a single halo.

ABSTRACT

State-of-the-art cosmological simulations regularly contain billions of particles, providing scientists the opportunity to study the evolution of the Universe in great detail. However, the rate at which these simulations generate data severely taxes existing analysis techniques. Therefore, developing new scalable alternatives is essential for continued scientific progress. Here, we present a data-parallel, friends-of-friends halo finding algorithm that provides unprecedented flexibility in the analysis by extracting multiple linking lengths. Even for a single linking length, it is as fast as the existing techniques, and is portable to multi-threaded many-core systems as well as co-processing resources. Our system is implemented using PISTON and is coupled to an interactive analysis environment used to study halos at different linking lengths and track their evolution over time.

Index Terms: H.3 [INFORMATION STORAGE AND RETRIEVAL]: Information Search and Retrieval—Clustering; J.2 [PHYSICAL SCIENCES AND ENGINEERING]: Astronomy—

1 INTRODUCTION

Modeling and understanding the evolution of the Universe is one of the fundamental questions in cosmology. In particular, understanding how the observed, hierarchical distribution of dark matter forms is of significant interest. To explore different hypotheses, numerical simulations based on different structure formation models are used to evolve a dark matter distribution from some initial near-uniform configuration, and these results are compared to observations. To increase the fidelity of these models, ever more particles are used, and the state-of-the-art simulations often contain billions of dark matter particles. However, analyzing these datasets is becoming increasingly challenging and can require substantial computational resources. In particular, some of the baseline analyses, such as halo finding, require new scalable alternatives to existing techniques.

Finding dark matter halos is a preliminary step to a wide range of analysis tasks. In this context, a "halo" [18, 10] is defined as an over-dense region of dark matter particles and represents one of the common features-of-interest. The two most common definitions of a halo are based on either a friends-of-friends (FOF) [7] clustering or a spherical-overdensity (SO) measure [21]. The former combines all particles that are reachable through links shorter than a predefined distance (the *linking length*) to be in one halo, while the SO method estimates the mean particle density and grows spherical regions around local density maxima. While there exist other definitions, these are by and large derivatives or combinations of the two baseline approaches. Here, we concentrate on the FOF-based definition as it is less biased towards a particular shape and the preferred technique of many scientists.

To address a prior bottleneck, scientists at Los Alamos National Laboratory (LANL) in collaboration with the Hardware Accelerated Cosmology Codes (HACC) team developed a serial halo finding algorithm [13, 26] that currently serves as the technique for per-node computation in HACC cosmology simulations. This algorithm has also been included in the standard distributions of Paraview [12]. When simulated on a massively parallel machine, dark matter particles are distributed amongst the nodes using MPI with a sufficient overlap (the *ghost zones*) such that any halo resides entirely on at least one of the nodes. As a result the halo finding is typically performed first within a node followed by a clean-up to ensure each halo is recorded only once. Given a linking length, this algorithm reports all corresponding halos which are subsequently filtered by size. However, while there exists a default linking length,

*e-mail: wathsy@sci.utah.edu
†e-mail: bremer5@llnl.gov
‡e-mail: csewell@lanl.gov
§e-mail: ollie@lanl.gov
¶e-mail: ahrens@lanl.gov
‖e-mail: pascucci@sci.utah.edu

IEEE Symposium on Large Data Analysis and Visualization 2014
October 9–10, Paris, France
978-1-4799-5215-1/14/$31.00 ©2014 IEEE

it is well-known that the halo structure may change substantially for different values and as the structure formation in FOF halo finding is hierarchical these result in "halos-within-halos" (sub-halos). Consequently, analyzing halos for different parameters is of significant interest [11, 24]. Unfortunately, virtually all existing algorithms, including the implementation in Paraview, require a full re-run for each linking length, rendering a comprehensive exploration of the parameter space infeasible. In an in-situ environment where re-computation for different linking lengths is not feasible, current techniques are unable to provide the flexibility to explore the parameter space of the data.

Instead, we present a new halo finding approach which, unlike the existing approaches, computes entire halo families by hierarchically encoding halos at different linking lengths. This allows us to analyze a range of parameters interactively, significantly increasing the flexibility overall. Our halo finding operator is implemented in PISTON [17], a cross-platform library of data parallel visualization and analysis operators. Since PISTON makes use of NVIDIA's Thrust library [1], it allows codes to be compiled to different backends, including CUDA, OpenMP, and Intel Threading Building Blocks (TBB). Nevertheless, using a hybrid strategy that combines GPU and multi-threaded CPU processing, the performance is comparable to the Paraview implementation for a single linking length query, and outperforms it with multiple linking lengths. The resulting data can be interactively explored in both space and time, providing a new level of post-processing capabilities previously not feasible. We demonstrate the results using data from a 256^3 cosmological simulation (containing 16.7 million particles) on Maverick at the Texas Advanced Computing Center (TACC) and provide detailed comparisons with the Paraview implementation. Furthermore, results from a 1024^3 cosmological simulation (containing 1.07 billion particles) tested on the Moonlight supercomputer at LANL are also presented.

2 Related Work

The FOF [7] and SO [21] methods are the first two halo finding algorithms found in the literature. Although many other halo finding algorithms have been introduced [14, 15, 19, 9, 20, 23, 26], almost all of them are derived from either one or both of these methods.

The FOF algorithm constructs halos of arbitrary shape by making use of two parameters: linking length and halo size. The underlying idea behind this algorithm is to carry out an extended neighbor search based on the linking length parameter specified. This linking length parameter is a fraction of the average inter-particle spacing and is considered as the radius for the neighbor search. It is usually set to ≈ 0.2 of the mean inter-particle separation. After connecting all pairs of particles which lie closer than the specified linking length, the FOF algorithm results in a network of linked particles. Each connected component found in the network is considered as a single FOF halo. Finally, the discovered halos are filtered based on the halo size parameter, where all the halos with fewer particles than the specified value are ignored (Figure 1). The relative ease in interpretation as well as in implementation and the fact that this algorithm avoids making any assumptions about the halo shape are some of the advantages of the FOF halo finding algorithm. However, one of its main disadvantages is the use of the linking length parameter, which if too large can merge two separate particle clusters together.

The SO method assumes halos to be spherical in shape. The idea is to identify spherical regions with an average density defined by a mean over-density criterion. The spheres are centered on density peaks and grown until they enclose the specified over-density value. Each such spherical region found is considered to be a halo. The main drawback of this algorithm is that the results are somewhat artificial due to the enforced halo shape. Therefore, the SO method can yield oversimplifications which could lead to unrealistic results.

As our halo finding operator focuses on the FOF algorithm, the rest of this section highlights some of the relevant related work found in this area. An extensive survey of the available halo finding algorithms can be found in [16].

The core of the Paraview FOF halo finding algorithm [13, 26] is a balanced kD-tree. Use of this balanced kD-tree reduces the number of operations needed for the halo finding process. Once the balanced kD-tree is built from the input particles, a recursive algorithm starts at the leaf nodes (single particles) and merges nodes into halos by checking if the particles are within a given linking length. Here, particle tests are reduced by using sub-tree bounding boxes as proxies for points. If a sub-tree bounding box is too distant, all of the points can be skipped. Conversely, if an entire bounding box is close enough all of the points can be added to a halo. To implement efficient parallelization for this algorithm, they have used a block structured decomposition which minimizes surface to volume ratio of the domain assigned to each process. This halo finding implementation is used for large-scale structure simulations, where the size of any single halo is much smaller than the size of the whole box. It uses the concept of "ghost zones" such that each process is assigned all the particles inside its domain as well as those particles which are around the domain within a given distance. After each process runs its serial version of the FOF finder, MPI-based "halo stitching" is performed to ensure that every halo is accounted for, and accounted for only once. Using this block structured decomposition to further split the input data space, this FOF halo finder can be made to utilize on-node parallelism. However, the initial data distribution and final clean up steps required create an overhead cost. In this paper, we compare the performance of our halo finding operator against this Paraview implementation.

The Ntropy algorithm [9] also uses a kD-tree data structure to speed up the FOF halo finding distance searches. It also employs an implementation of a parallel connectivity algorithm to link halos which span across separate processor domains. The advantage of this method is that no single computer node requires knowledge of all groups in the simulation volume. However, identifying large halos which span across many processors requires more time and computation. This causes the algorithm to scale poorly and to duplicate the work required to find these large halos.

In the MPI-based parallel Friends-of-Friends algorithm named Parallel FOF (pFOF) [22], particles are distributed in cubic sub volumes of the simulation and each processor deals with one cube. Within each processor, the FOF algorithm is run locally. When a halo is located closer to the edge of a cube, pFOF checks whether the particles belonging to the same halo are in a neighboring cube. This process is iteratively repeated until all halos extending across multiple cubes are merged. This merging is performed for each such halo and not implemented in a data-parallel manner as is ours.

The structure formation in FOF halo finding is hierarchical, such that each halo at one particular linking length contains substructure which contains halos at lower linking lengths. However, due to the definitions of the basic FOF algorithm, different linking lengths need to be specified to identify these "halos-within-halos" (sub-halos).

Rockstar [2] is a recent halo finding algorithm which makes use of these "halos-within-halos". This is a new phase-space based halo finder which is designed to maximize halo consistency across time-steps of a time-varying cosmological simulation. This algorithm first selects particle groups for a 3D FOF variant using a very large linking length. Then, for each main FOF group, this algorithm builds a hierarchy of FOF subgroups in phase space by progressively and adaptively reducing the linking length. Finally, it converts these FOF subgroups into halos beginning at the deepest level of the hierarchy. This algorithm is based on a different halo definition than ours; specifically it uses the adaptive hierarchical refinement of FOF groups in six dimensions.

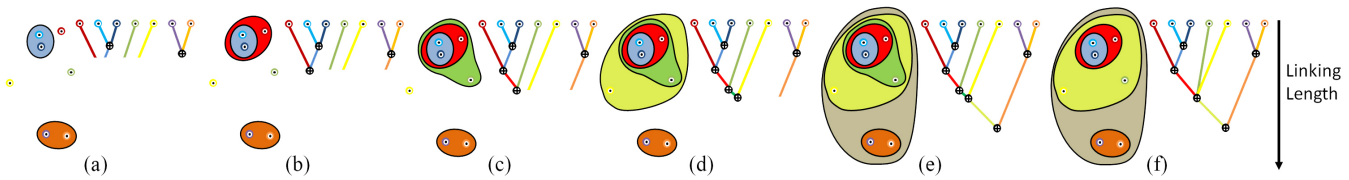


Figure 2: (a)-(e) Construction of hierarchical halo definitions by recording the clustering of dark matter particles as the linking length is swept top-to-bottom through the full value range. (f) Augmented hierarchy: obtained by removing branches which are shorter than a certain desired interval.

To our knowledge, all existing FOF-based halo finding algorithms require re-computation of halos whenever the linking length or halo size parameters are changed. Since we compute a meta-representation which encodes all possible halos for a wide range of linking lengths and stores it using hierarchical feature definitions, our algorithm has the capability to quickly and efficiently compute halos for a wide range of linking length and halo size parameters. In an in-situ environment where re-computing for different parameters is not practically feasible, our algorithm provides scientists with significant additional flexibilities in exploring the parameter space of underlying data.

3 Data-Parallel Programming with PISTON

Data parallelism is achieved by independent processors performing the same operation on different pieces of data. With the increasing amount of data available today, data parallelism is considered to be an effective method to exploit parallelism on the state-of-the-art architectures. More details on the data-parallel programming model can be found in [3].

PISTON [17] is a cross-platform library of visualization and analysis operators that employ the data-parallel programming model. It allows operators to be written using only data parallel primitives (such as scans, transforms, sort etc.) and enables the same code to be compiled to multiple targets using architecture-specific backend implementations. Specifically, PISTON makes use of NVIDIA's Thrust library [1]. Therefore, it facilitates portability and avoids the need to re-write algorithms for different architectures, which is frequently required with other high performance parallel visualization and analysis operator tools. Visualization operators such as isosurface, cut-surface, and threshold have been implemented in PISTON. This framework is also being used to write physics code for simulations [8].

Thrust library provides a rich collection of data parallel primitives such as scan, sort, transform, and reduce for multiple backends including CUDA, OpenMP, and Intel TBB. It allows users to program using an interface which is very similar to the C++ Standard Template Library. Moreover, the two vector types provided, host vector and device vector (similar to the std::vector in STL), allow data to be copied efficiently between the host and device. Making use of these two vector types and developing operator code using only the data parallel primitives provided, efficient and portable code can be produced. Our halo finding operator is also implemented in such a way using Thrust. An illustration of a part of our algorithm, as implemented in Thrust, is shown in Section 5.

4 Hierarchical Feature Definitions

When dealing with large-scale data, to effectively understand their underlying patterns, scientists need to explore the parameter space. One of the most important steps towards enabling this exploration is providing the ability to quickly change feature parameters. For halo finding, this may mean adjusting the linking lengths and halo sizes. In most cases, given the expected data sizes, on-the-fly feature computation becomes practically infeasible and requires massively parallel computing resources. The alternative is to pre-compute features for a wide range of potential parameters and to store the results in an efficient look-up structure.

In this paper, we make use of hierarchical feature definitions [4, 25] which can encode a wide range of feature types and simplifications. To create this hierarchy, first features should be defined for each time-step in the data, and then those features must be grouped at different scales. By maintaining the notion of scale, this feature grouping naturally approximates a meaningful hierarchy. Since the structure formation in FOF halo finding is hierarchical by definition, hierarchical feature definitions yield an ideal fit for storing this resultant halo structure. For each time-step in the data, halos are extracted for a wide range of linking lengths and stored in this hierarchy to allow interactive extraction of halos for different parameters.

Figure 2(a)-(e) shows the clustering of seven dark matter particles as the linking length l is swept from top-to-bottom through the full linking length range r represented by a hierarchy. Each dark matter particle is represented by a leaf in the hierarchy. Each branch in the hierarchy represents a neighboring set of particles which are considered to be a halo at that level, and joining of branches indicates merging of halos. Given a linking length l within the full range of r, the corresponding halos can be found by "cutting" the hierarchy of r at l (Figure 3). This creates a forest of sub-trees, where each sub-tree represents a halo existing at l. The remainder of the paper uses this same 2D example input to illustrate the various details of our halo finding operator. The extension to 3D as used in our actual implementation is straightforward.

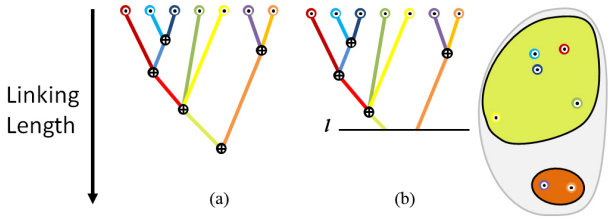


Figure 3: Extracting halos for a particular linking length l, obtained by cutting the hierarchy at l and ignoring all pieces below, shown (a) before and (b) after extraction. Each sub-tree in the forest of sub-trees is considered as a single halo. The halo particles are obtained by looking at the leafs of each sub-tree.

There exists a natural correspondence of leafs in the hierarchy to particles in space, and by storing this segmentation information one can easily extract the geometry of a sub-tree/feature as a union of branch segmentations. Along with the hierarchy, precomputed feature attributes such as first order statistical moments and/or shape characteristics can also be stored for each feature on a per-branch basis. In our case, we opted to store the halo id, size, position, and velocity information of each halo.

Although hierarchical feature definitions are highly efficient and flexible, they quantize the space of features depending on the function values involved. This implicit quantization can often be too

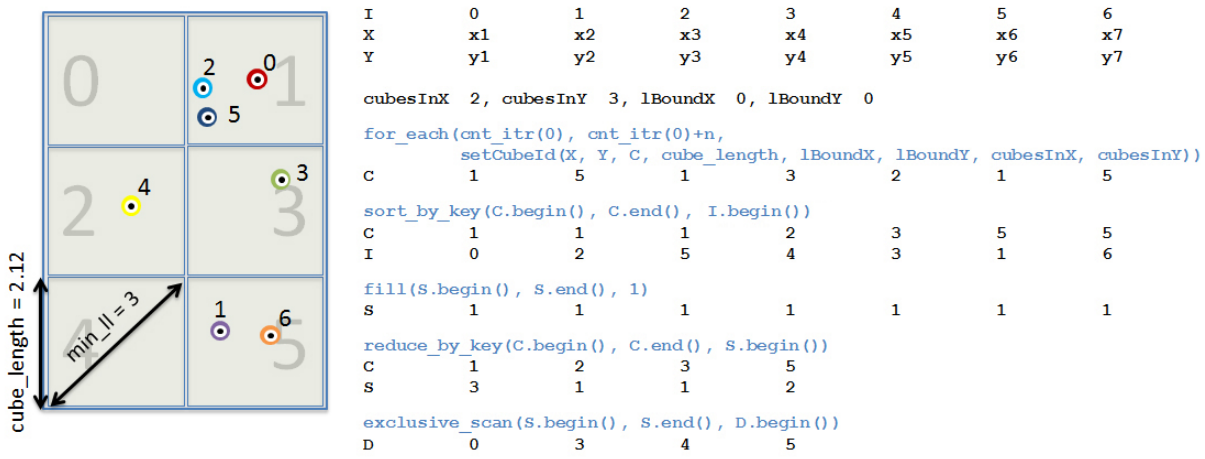


Figure 5: Space Partitioning: An illustration of the underlying algorithm for this phase. First, the cube id is set for each particle by using the '`setCubeId`' functor (see Listing 1) and the results are stored in the `C` vector. Then particles are sorted by their cube id using the `sort_by_key` operator. Finally, the non-empty cube ids, sizes, and offsets into the particle array (`I` vector) are computed and stored in `C`, `S` and `D` vectors. For example, cube 5 contains two particles and those particles can be found between indices 5 and 6 within the particle array.

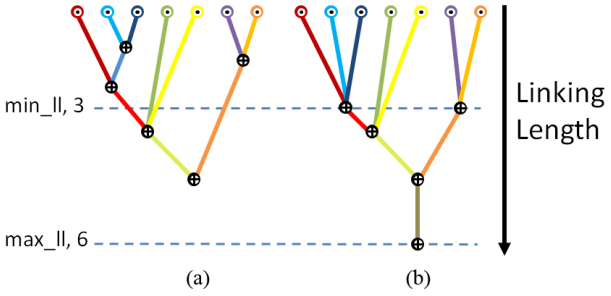


Figure 4: Storing the clustering hierarchy of dark matter particles for the range from *min_ll*=3 to *max_ll*=6: (a) Clustering hierarchy for the entire linking length range. (b) Clustering hierarchy for the range from 3 to 6. Note that any extra nodes outside the required range are removed to only store the necessary clustering information.

coarse or too fine to be practical. For halo finding, we found the quantization of the linking length to be too fine and chose to create an *augmented hierarchy* [6] by removing branches which are shorter than a certain desired interval (Figure 2(f)). For example, even for a small subset of the dataset we used (about 24000 particles), when considering a linking length range of 0.19 to 0.2, some particles in the final hierarchy contained $\approx$ 15000 parent nodes while others contained fewer than ten parent nodes. An augmented hierarchy reduces the granularity of the clustering found within the final hierarchy and makes interactive halo extraction possible. However, the amount of reduction in the hierarchy resolution is left up to the scientist to decide at run-time. In this manner, hierarchical feature definitions encode an entire feature family alongside its segmentation and relevant attributes in a compact and efficient manner.

5 FOF Halo Finding Operator

The naive implementation of a non-parallel FOF halo finder compares each and every particle pair, requiring $n^2/2$ operations. Here, n is the total number of particles in the data. Our goal is to find the clustering hierarchy of particles for a range of linking lengths with fewer particle pair operations than the naive approach and to implement the algorithm using only data-parallel primitives in order to achieve portability and performance. Given a minimum linking length value (*min_ll*), maximum linking length value (*max_ll*), and a set of dark matter particles (along with their attributes such as halo id, position, and velocity), our FOF halo finding operator computes the clustering hierarchy of the input particles for the range from *min_ll* to *max_ll* at a desired level of resolution (Figure 4). As mentioned earlier, along with the output halo hierarchy, several halo statistics (such as halo id, size, position, and velocity information) are stored on a per-halo basis.

Furthermore, while our algorithm can store the halo hierarchy for a range of linking lengths, it can also be used for just a single linking length. Simply by setting both *min_ll* and *max_ll* to the required linking length value, clustering information for only that linking length value will be stored in the final hierarchy.

5.1 Algorithm

The algorithm for our PISTON-based FOF halo finding operator can be divided into four phases: space partitioning, local hierarchy computation, edge computation and global hierarchy computation. More details on each of these phases are presented below.

5.1.1 Space Partitioning

First, the full spatial domain of the input particles is partitioned into equal sized cubes. Here, the diagonal length of each cube is set to the *min_ll* value. An id is set for each cube according to its row-major order and then that cube id is set for each of its particles. Then, the number of non-empty cubes in space and their particle sizes are obtained. The core algorithm for this phase, as implemented in Thrust, is illustrated in Figure 5 and Listing 1.

5.1.2 Local Hierarchy Computation

After partitioning, the local halo hierarchy of each cube is computed. The maximum distance between any two particles within a cube is its diagonal length. Therefore, by setting the diagonal length of each cube to *min_ll* value, we can be sure that all the particles within a cube lie at a distance less than or equal to the *min_ll* value. In other words, all the particles within a cube can be clustered together to be one halo at '*min_ll*'. Accordingly, the local hierarchy can be easily computed by inserting a single parent node at *min_ll* value (Figure 6). This parent node combines all particles within the cube to be in a single cluster at that linking length value. Since particle details (id, position, and velocity) are also available,

as this node is created, its details are also stored by combining its particle details.

```
// for a given particle, set its cube id
struct setCubeId : public thrust::unary_function<int, void>
{
  float *X, *Y;
  unsigned long long *C;

  float cubeLen;
  float lBoundX, lBoundY;

  unsigned long long cubesInX, cubesInY;

  __host__ __device__
  setCubeId(float *X, float *Y, unsigned long long *C,
    float cubeLen, float lBoundX, float lBoundY,
    unsigned long long cubesInX, unsigned long long cubesInY) :
    X(X), Y(Y), cubeLen(cubeLen), lBoundX(lBoundX), lBoundY(lBoundY),
    cubesInX(cubesInX), cubesInY(cubesInY) {}

  __host__ __device__
  void operator()(int i)
  {
    // get x,y coordinates for the cube
    unsigned long long x = (X[i]-lBoundX)/cubeLen;
    unsigned long long y = (Y[i]-lBoundY)/cubeLen;

    if(x>=cubesInX) x = cubesInX-1;
    if(y>=cubesInY) y = cubesInY-1;

    // set cube id
    C[i] = x + (y*cubesInX);
  }
}
```

Listing 1: Code for the '`setCubeId`' custom functor

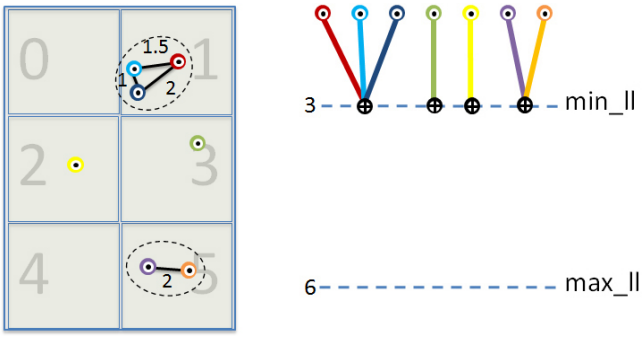


Figure 6: Local hierarchy computation: When *min_ll*=3 and *max_ll*=6, for each cube its local hierarchy is computed by inserting a single parent node at value 3. This combines all the particles within each cube to be in a single cluster at this value. For this example, four halos exist, one for each non-empty cube in the input domain. Each sub-tree in the forest of sub-trees (at right) represents a local hierarchy.

As a result of setting the diagonal length of each cube to *min_ll* value, we avoid making particle pair comparisons across particles within a cube and end up with a set of halos (one for each non-empty cube) existing at *min_ll* value. These local hierarchies are created using the information available for each cube, and are later modified in subsequent phases, when they are merged to produce the final hierarchy.

5.1.3 Edge Computation

In this phase, distances across the aforementioned set of halos are computed, to be used for constructing the final halo hierarchy. The distance between two halos is determined by the shortest distance across their particles. Therefore, to determine these distances particle pair comparisons need be performed across the halos. The naive approach requires comparing each halo with every other halo, but we compute particle pair comparisons with only halos lying within a specific range. For each cube, the range of neighboring cubes that should be checked (in each direction of each axis) is computed using *ceil(max_ll / cube_length)*. Since the final hierarchy is computed only for the range between *min_ll* and *max_ll* values, this still renders correct results and avoids making unnecessary particle pair comparisons (Figure 7).

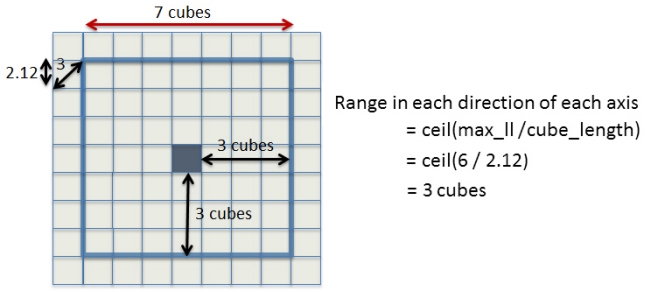


Figure 7: Range of neighboring cubes to be checked in each direction of each axis. Given *min_ll*=3 and *max_ll*=6, the length of a cube is 2.12. Then for each cube, three neighboring cubes (in each direction of each axis) should be checked. In other words, each cube would compare with 7*7-1=48 neighboring cubes (in 2D). However, by avoiding duplicate comparisons these comparisons can be reduced by half (to 24 cubes).

Once the range is found, each cube computes the shortest distance to each of the neighboring cubes within this range and each result is stored as an *Edge(SrcParticle, DestParticle, shortestDist)*. This shortest edge is found by performing particle pair comparisons across the two cubes, requiring $a * b$ operations (where a and b are the number of particles in each cube) and all the edges found for cubes are stored in a single array, to be used in the next phase. Here, only a shortest edge with a weight less than or equal to the *max_ll* is stored, and if the edge weight is less than the *min_ll*, it is stored as a *min_ll* weight. As an optimization, when performing the $a * b$ operations, once a particle pair is found with a distance less than the *min_ll* value, the rest of the comparisons for those two cubes are skipped and an edge with *min_ll* weight is stored. The reasoning is that when such an edge is found we can be sure that those two cubes are merged to be in one halo for the *min_ll* value and therefore the rest of the comparisons are unnecessary.

To find this shortest edge, instead of the brute force comparisons, more sophisticated approaches using particle sorting (by their physical location) were also exploited. However, as the number of particles in a cube was small, due to the small cube size, the brute force method worked best.

As previously mentioned, in order to decrease the granularity of the clustering we opted to create an augmented halo hierarchy. For our tests, we chose to store the edge weight up to three decimal places. Therefore, for a linking range of 0.1 to 0.2, 100 discretized linking length values (0.1, 0.101, 0.102, ... 0.2) could be extracted from the augmented halo hierarchy. This granularity within the halo hierarchy is a user-defined parameter and can be modified at runtime by just changing the precision of the edge weight.

5.1.4 Global Hierarchy Computation

The final halo hierarchy is created by merging the local halo hierarchies according to the computed edges. Starting from the initial set of cubes, we iteratively combine k cubes, until only one cube is left. Here, any k cubes in the input domain can be combined, for simplicity we sort the set of cubes by their cube id and merge each consecutive k cubes. At each merge step, we consider the edges of those k cubes and all the edges where the *SrcParticle* and *DestParticle* are within the k cubes are used to update the hierarchies of those k cubes. For example, in Figure 8(a) when cube 2 and 3 are merged only the edge indicated in black is considered.

The updating of the halo hierarchies is achieved using a similar procedure as the streaming merge tree construction algorithm by Bremer et al. [5]. As these hierarchies are updated along with the

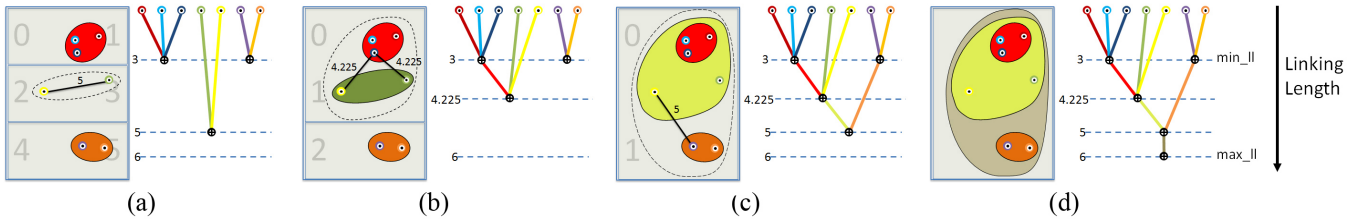


Figure 8: Global Hierarchy Computation: Final halo hierarchy is constructed by iteratively merging *k* cubes until only one cube is left. At each step, the local halo hierarchies of the *k* cubes are merged according to the computed edges. In this example, where *k*=2, (a)-(c) illustrates each iteration of the merging process. At each iteration, the edges used to update the hierarchies are indicated in black and the resulted hierarchy is presented in right. (d) Final halo hierarchy of the input particles for the range from *min_ll*=3 to *max_ll*=6.

halo ids, their position and velocity details are also updated so that the final halo hierarchy has the necessary halo details along with the clustering information. Once the final halo hierarchy is constructed, given a linking length value, halos can be extracted by cutting the hierarchy at that value. Then, the halo size details stored within the hierarchy are used to filter halos by the required halo size value.

5.2 Implementation

The FOF halo finding algorithm described above is implemented using data-parallel Thrust primitives (such as `for_each`, `sort`, and `exclusive_scan`) along with some custom functors. Consequently, the same code was run using both OpenMP and CUDA backends.

In the partitioning phase, the input domain is divided into equal sized cubes, and the cube ids for all particles are computed in parallel. Then, all particles are sorted by their cube ids, and the range of particles within each cube is computed. The core algorithm for this phase and the associated functor are illustrated in Figure 5 and Listing 1. Next, in parallel, each cube computes its local hierarchy and edges using custom functors. The computed edges are stored in a single array such that the edges for each cube are located in consecutive locations. Furthermore, when creating the complete halo hierarchy, the merging process is performed in parallel for the set of cubes at that iteration. Finally, for a linking length query, each leaf in the hierarchy is traversed in parallel to extract the halos and their details.

Furthermore, as the *min_ll* and *max_ll* values used are usually very small relative to the size of the spatial domain, the number of empty cubes (without any particles) is often very large. Therefore, we chose to store details only for the non-empty cubes and parallelized over non-empty cubes instead of all cubes.

We further implemented a hybrid version combining GPU and multi-threaded CPU processing. Here, instead of computing all operations in either OpenMP or CUDA, the global hierarchy computation phase was performed using OpenMP and the other steps using CUDA. We observed that while the first three phases ran faster on the GPU than the CPU, the global hierarchy computation was not well-suited to the GPU due to many random data accesses and divergent branches within the functor implementation. The hybrid algorithm allowed us to match each phase to the architecture on which it performed best. The only difference with the hybrid code was that before the start of the global hierarchy computation phase, the data required for that phase was copied to the CPU, and then at the end of the hierarchy computation the resulting halo hierarchy was copied back to the GPU. The hybrid version can also help with GPU memory limitations. For example, in the edge computation phase, the edges can be computed in chunks and then copied directly into CPU memory, thus not requiring all the edges to be in GPU memory at once. Furthermore, with Thrust 1.7 target architectures were directly specified for individual function calls using specifiers such as `thrust::omp` or `thrust::cuda`.

6 Results

In this section, we analyze the performance of our halo finding operator, and compare it against the Paraview implementation on a single node, and within a multi-node run. The Paraview algorithm can be run either serially or in parallel across multiple MPI processes.

The first timing results presented here were run on a single node of the Maverick system at TACC using a cosmology dataset containing 16.7 million particles (from a 256^3 cosmological simulation). This single node in Maverick contained two 10-core Intel Xeon E5-2680 v2 processors and an NVIDIA Tesla K40 GPU. Similar results were obtained on the Darwin cluster at LANL and on several other computational resources available at the Scientific Computing and Imaging Institute in University of Utah.

First, the PISTON halo finding operator was compiled to the Thrust OpenMP backend and then the same code was compiled to the Thrust CUDA backend to demonstrate portability of the algorithm. The hybrid version of the halo finder was compiled using both Thrust OpenMP and CUDA backends. Various tests were performed for a single linking length as well as for a range of linking lengths. Then the serial formulation and the MPI-based parallel variant of the Paraview implementation were run on the same node.

Here, all the results are presented on realistic parameter values. The default linking length value in which the cosmologists are most interested is 0.2; therefore, the linking length ranges around this value were selected and the halo size was set to one. Also, in order to obtain an augmented halo hierarchy, the edge weights were stored up to three decimal places. When presenting our halo finding results, the timing results include both the hierarchy construction time and halo finding time. The hierarchy is constructed only once during the first query and as multiple queries are performed only a traversal within the already constructed hierarchy is needed to find halos.

After comparing the performance of the OpenMP and CUDA backends of our halo finding operator (Figure 9), it is clear that the first three phases are faster on the GPU, but the global hierarchy computation is faster on the CPU. Accordingly, having a hybrid implementation enables one to fully exploit the advantages of both the CPU and the GPU, and also to outperform the OpenMP and CUDA versions for both a single linking length and for a range of linking lengths. Due to the data copying necessary, timing for the global hierarchy computation phase was slightly longer in the hybrid version than in OpenMP.

As shown in Figure 10, the performance and scaling of our algorithm constructed for a single linking length are better than the Paraview implementation run serially and comparable when run with multiple MPI processes. As the linking length range is increased, the total hierarchy construction time increases since it requires more work (as indicated by the increased run-times for longer linking length ranges in Figure 10). Nevertheless, when performing multiple linking length queries our implementation clearly has an advan-

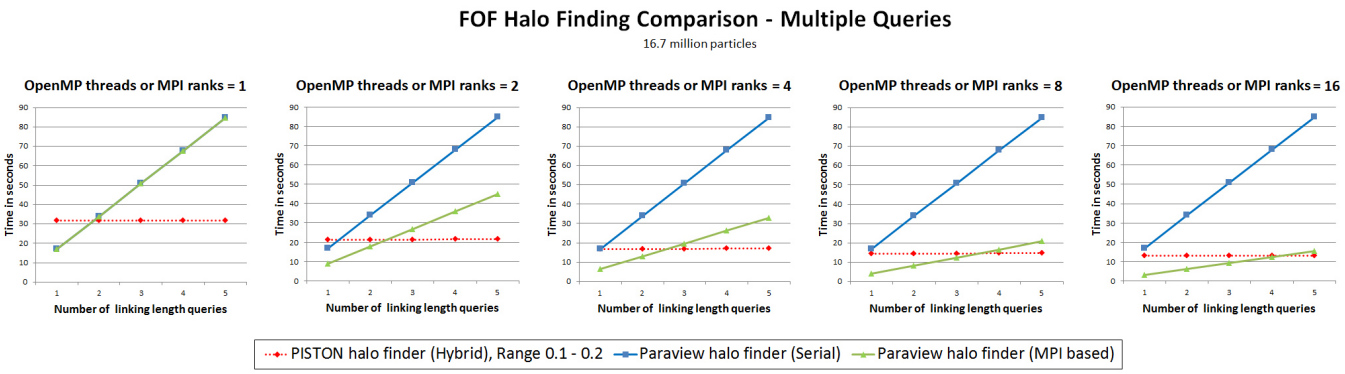


Figure 11: Comparing timing results of PISTON-based hybrid halo finding algorithm, for subsequent queries, against the Paraview implementation run serially and with multiple MPI processes. At each query, Paraview implementation always computes halo details for just one linking length, whereas our algorithm produces a hierarchy with 100 discretized linking length values for the range from 0.1 to 0.2. This hierarchy is created during the first query and is used throughout the rest of queries.

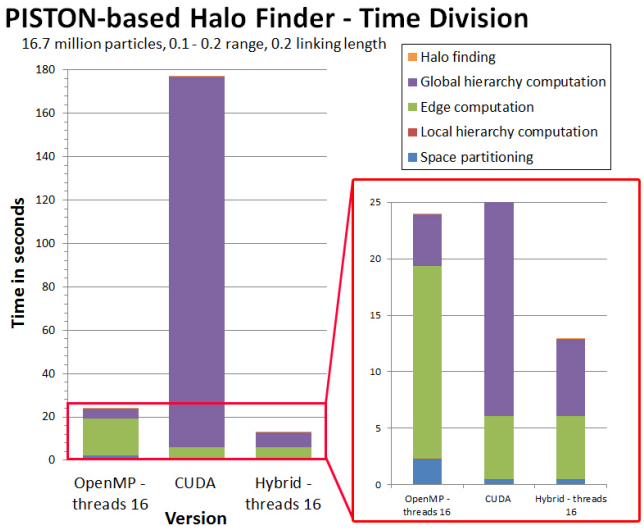


Figure 9: Time division for PISTON-based halo finding algorithm.

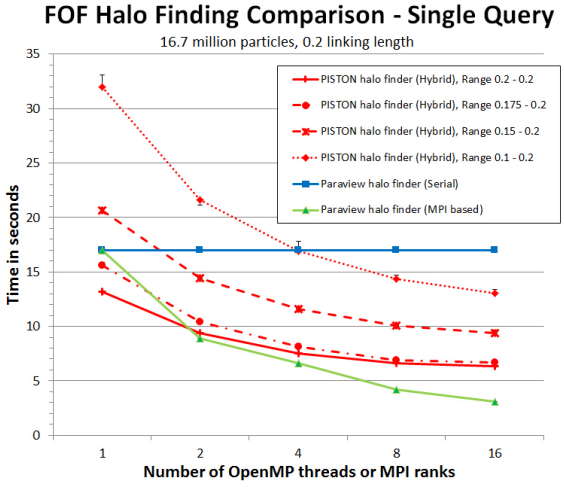


Figure 10: Comparing timing results of PISTON-based hybrid halo finding algorithm, for a single query, against the Paraview implementation run serially and with multiple MPI processes.

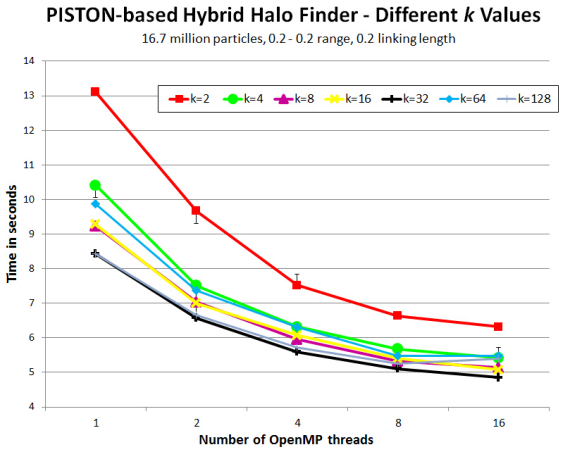


Figure 12: Timing results for PISTON-based hybrid halo finding algorithm when different k values are used in the global hierarchy computation phase.

tage over the Paraview halo finder even when run with multiple MPI processes (Figure 11). In our algorithm, the halo hierarchy needs to be computed once at the start of the program and only a traversal within the hierarchy is needed (which takes very little time relative to the construction time) to obtain halo information for multiple linking lengths queries, whereas for the Paraview implementation a full re-run of the algorithm is needed for every linking length change. Therefore, depending on the number of cores, it becomes more efficient to use our hierarchy when more than a minimum of 2-5 queries are performed.

As mentioned in the previous section, the global hierarchy computation phase iteratively combines k cubes in the input spatial domain. For simplicity, throughout the paper we used k=2. However, using different k values better results can be obtained (Figure 12). Specifically, changing the k value directly impacts the run time of the global hierarchy computation phase. As k increases, number of iteration within that phase is reduced, thus resulting in faster run times. However, from our experiments we noticed that the optimal number for k is dependent on the density of the input dataset. As k increases, if the data density is high, the amount of work which needs to be performed at one iteration is much higher and requires longer completion times. Also, instead of merging each consecu-

tive k cubes, more sophisticated methods like merging cubes by x, y and z axes can also be used to gain better results.

We also integrated our hybrid halo finding algorithm in place of the serial Paraview algorithm within the parallel halo finder and tested it on 128 nodes of the Moonlight supercomputer at LANL with a 1.07 billion particle data set (from a 1024^3 cosmological simulation). For the same linking length range 0.1-0.2, the halo hierarchy construction and halo finding took 75s (of which less than 0.2s was for querying the hierarchy for halos at a specific linking length). This compares to 43s for the Paraview implementation with one rank per node, and 16s with 16 ranks per node. The "crossing point" for how many queries it takes before our algorithm is more efficient than the 16 rank-per-node Paraview implementation is thus slightly less than 5, which is consistent with the single-node results (i.e., the last graph in Figure 11).

7 Conclusion

In this paper, we have presented a new data-parallel FOF halo finding operator which unlike the traditional approaches computes entire halo families by hierarchically encoding halos at different linking lengths. This allows us to interactively extract a set of halos for a range of parameters, significantly increasing the flexibility overall. In an in-situ environment where re-computing halos for different parameters is not feasible, our algorithm provides the capability to explore the parameter space of underlying data. Using a hybrid strategy which combines GPU and multi-threaded CPU processing, we show that the performance of our approach is comparable to the Paraview implementation for a single linking length query, and outperforms it with multiple linking lengths.

Acknowledgements

We are thankful to Salman Habib and Katrin Heitmann at Argonne National Laboratory for the cosmology datasets and to Paul Navratil at Texas Advanced Computing Center for the time on Maverick. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program, through the SDAV Institute. This work was also performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344, LLNL-CONF-659054 and Los Alamos National Laboratory LA-UR-14-23700. This work is supported in part by BNSF CISE ACI-0904631, NSG IIS-1045032, NSF EFT ACI-0906379,DOE/NEUP 120341, DOE/-Codesign P01180734, DOE/SciDAC DESC0007446, and CCMSC DE-NA0002375.

References

[1] Thrust library. project webpage: http://code.google.com/p/thrust/, Accessed 2014.

[2] P. S. Behroozi, R. H. Wechsler, and H.-Y. Wu. The rockstar phase-space temporal halo finder and the velocity offsets of cluster cores. *The Astrophysical Journal*, 762(2):109, 2013.

[3] G. E. Blelloch. *Vector models for data-parallel computing*, volume 356. MIT press Cambridge, 1990.

[4] P.-T. Bremer, G. H. Weber, V. Pascucci, M. S. Day, and J. B. Bell. Analyzing and tracking burning structures in lean premixed hydrogen flames. *IEEE Transactions on Visualization and Computer Graphics*, 16(2):248–260, 2010.

[5] P.-T. Bremer, G. H. Weber, J. Tierny, V. Pascucci, M. S. Day, and J. B. Bell. Interactive exploration and analysis of large-scale simulations using topology-based data segmentation. *IEEE Transactions on Visualization and Computer Graphics*, 17(9):1307–1324, 2011.

[6] H. Carr, J. Snoeyink, and U. Axen. Computing contour trees in all dimensions. In *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, pages 918–926. Society for Industrial and Applied Mathematics, 2000.

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

[8] M. M. Francois, L.-T. Lo, and C. Sewell. Volume-of-fluid interface reconstruction algorithms on next generation computer architectures. In *Proceedings of the ASME 2014 4th Joint US-European Fluids Engineering Division Summer Meeting and 12th International Conference on Nanochannels*, Microchannels, and Minichannels, Chicago, IL, USA, 2014. To appear.

[9] J. P. Gardner, A. Connolly, and C. McBride. Enabling knowledge discovery in a virtual universe. In *Proceedings of the 2007 TeraGrid Symp*, 2007.

[10] S. Ghigna, B. Moore, F. Governato, G. Lake, T. Quinn, and J. Stadel. Dark matter haloes within clusters. *Monthly Notices of the Royal Astronomical Society*, 300(1):146–162, 1998.

[11] S. Gottlöber, A. Klypin, and A. V. Kravtsov. Merging history as a function of halo environment. *The Astrophysical Journal*, 546(1):223, 2001.

[12] A. Henderson. ParaView Guide, A Parallel Visualization Application. Kitware Inc., 2007.

[13] C.-H. Hsu, J. P. Ahrens, and K. Heitmann. Verification of the time evolution of cosmological simulations via hypothesis-driven comparative and quantitative visualization. In *PacificVis*, pages 81–88. IEEE, 2010.

[14] A. Klypin and J. Holtzman. Particle-mesh code for cosmological simulations. *arXiv preprint astro-ph/9712217*, 1997.

[15] A. Klypin, A. V. Kravtsov, O. Valenzuela, and F. Prada. Where are the missing galactic satellites? *The Astrophysical Journal*, 522(1):82, 1999.

[16] A. Knebe, S. R. Knollmann, S. I. Muldrew, F. R. Pearce, M. A. Aragon-Calvo, Y. Ascasibar, P. S. Behroozi, D. Ceverino, S. Colombi, J. Diemand, K. Dolag, B. L. Falck, P. Fasel, J. Gardner, S. Gottloeber, C.-H. Hsu, F. Iannuzzi, A. Klypin, Z. Lukic, M. Maciejewski, C. McBride, M. C. Neyrinck, S. Planelles, D. Potter, V. Quilis, Y. Rasera, J. I. Read, P. M. Ricker, F. Roy, V. Springel, J. Stadel, G. Stinson, P. M. Sutter, V. Turchaninov, D. Tweed, G. Yepes, and M. Zemp. Haloes gone mad: The halo-finder comparison project. 2011.

[17] L.-t. Lo, C. Sewell, and J. Ahrens. Piston: A portable cross-platform framework for data-parallel visualization operators. In *Eurographics Symposium on Parallel Graphics and Visualization*, 2012.

[18] J. Navarro and S. D. White. The structure of cold dark matter halos. In *SYMPOSIUM-INTERNATIONAL ASTRONOMICAL UNION*, volume 171, pages 255–258. KLUWER ACADEMIC PUBLISHERS GROUP, 1996.

[19] M. C. Neyrinck, N. Y. Gnedin, and A. J. Hamilton. Voboz: an almost-parameter-free halo-finding algorithm. *Monthly Notices of the Royal Astronomical Society*, 356(4):1222–1232, 2005.

[20] S. Planelles and V. Quilis. Asohf: a new adaptive spherical overdensity halo finder. *arXiv preprint arXiv:1006.3205*, 2010.

[21] W. H. Press and P. Schechter. Formation of galaxies and clusters of galaxies by selfsimilar gravitational condensation. *Astrophysical Journal*, 187:425–438, 1974.

[22] Y. Rasera, J. Alimi, J. Courtin, F. Roy, P. Corasaniti, A. Fuzfa, and V. Boucher. Introducing the dark energy universe simulation series (deuss). *arXiv preprint arXiv:1002.4950*, 2010.

[23] P. Sutter and P. Ricker. Examining subgrid models of supermassive black holes in cosmological simulation. *The Astrophysical Journal*, 723(2):1308, 2010.

[24] M. White, L. Hernquist, and V. Springel. The halo model and numerical simulations. *The Astrophysical Journal Letters*, 550(2):L129, 2001.

[25] W. Widanagamaachchi, C. Christensen, P.-T. Bremer, and V. Pascucci. Interactive exploration of large-scale time-varying data using dynamic tracking graphs. In *IEEE Symposium on Large Data Analysis and Visualization (LDAV), 2012*, pages 9–17, Oct. 2012.

[26] J. Woodring, K. Heitmann, J. Ahrens, P. Fasel, C.-H. Hsu, S. Habib, and A. Pope. Analyzing and visualizing cosmological simulations with paraview. *The Astrophysical Journal Supplement Series*, 195(1):11, 2011.