LA-UR-15-22202

Title: Utilizing Many-Core Accelerators for Halo and Center Finding within a Cosmology Simulation

Author(s): Sewell, Christopher Meyer
Lo, Li-Ta
Heitmann, Katrin
Habib, Salman
Ahrens, James Paul

Intended for: IEEE Symposium on Large Data Analysis and Visualization, 2015-10-25 (Chicago, Illinois, United States)

Utilizing Many-Core Accelerators for Halo and Center Finding within a Cosmology Simulation

Christopher Sewell* Los Alamos National Laboratory
Li-ta Lo† Los Alamos National Laboratory
Katrin Heitmann‡ Argonne National Laboratory
Salman Habib§ Argonne National Laboratory
James Ahrens¶ Los Alamos National Laboratory

ABSTRACT

Efficiently finding and computing statistics about "halos" (regions of high density) are essential analysis steps for N-body cosmology simulations. However, in state-of-the-art simulation codes, these analysis operators do not currently take advantage of the shared-memory data-parallelism available on multi-core and many-core architectures. The Hybrid / Hardware Accelerated Cosmology Code (HACC) is designed as an MPI+X code, but the analysis operators are parallelized only among MPI ranks, because of the difficulty in porting different X implementations (e.g., OpenMP, CUDA) across all architectures on which it is run. In this paper, we present portable data-parallel algorithms for several variations of halo finding and halo center finding algorithms. These are implemented with the PISTON component of the VTK-m framework, which uses Nvidia's Thrust library to construct data-parallel algorithms that allow a single implementation to be compiled to multiple backends to target a variety of multi-core and many-core architectures. Finally, we compare the performance of our halo and center finding algorithms against the original HACC implementations on the Moonlight, Stampede, and Titan supercomputers. The portability of Thrust allowed the same code to run efficiently on each of these architectures. On Titan, the performance improvements using our code have enabled halo analysis to be performed on a very large data set (8192^3 particles across 16,384 nodes of Titan) for which analysis using only the existing CPU algorithms was not feasible.

Index Terms: D.1.3 [Software]: Programming Techniques—[Concurrent Prgm.]

1 INTRODUCTION

The identification of halos (regions with a high density of dark matter particles) is one of the most important analysis steps in an N-body simulation. The halos carry information about structure formation, galaxy formation, and the content of the universe. Accurately determining the halo centers is critical for computing mass functions and for comparing simulation results with observations.

Computing halos and their statistics (especially their centers) can be very computationally demanding, and can be the limiting factor for the number of particles used in a simulation. In order to achieve ever higher levels of accuracy, current cosmology simulations may use hundreds of billions of particles. While major cosmology simulation codes such as HACC (Hybrid/Hardware Accelerated Cosmology Code) [9, 10] have long had the ability to parallelize such computations by partitioning particles across MPI ranks, and are designed as MPI+X codes, they do not currently take advantage of many-core accelerators for analysis computations, due in large part to the difficulty of porting such code across architectures.

The need to support the cross-product of multiple variants of halo and center finding definitions with all current and emerging multi-/many-core architectures is a major burden for a cosmology simulation code. Abstracting the architecture-specific optimizations from the algorithms using a portable, data-parallel framework reduces this code growth to scale linearly with the number of algorithms and architectures rather than quadratically with their product.

The primary contribution of this paper is to illustrate our approach of providing portable, data-parallel algorithms for large-scale high-performance data analytics on multi-core and GPU architectures. We provide detailed descriptions and results for the algorithms we developed for halo analysis in a cosmology simulation. We believe that utilizing such an approach should facilitate information extraction and knowledge discovery from big data across a broad range of scientific applications.

In this paper, we present data-parallel algorithms for halo and center finding operators. These algorithms have been implemented using PISTON, which is a framework for constructing portable data-parallel functions. PISTON is a component of the VTK-m project within the SDAV SciDAC Institute. Making use of Nvidia's Thrust library, code written using PISTON/VTK-m can be compiled to different backends, including CUDA, OpenMP, Intel TBB. It has previously been demonstrated that PISTON algorithms can make efficient use of available parallelism on a variety of multi-core and many-core architectures, including multi-core CPUs, Nvidia and AMD GPUs, and Intel Xeon Phis, by simply changing a compile flag [16, 20]. This portability is critical to the scientists, because they often do not have the resources to rewrite and reoptimize their algorithms for each architecture separately.

The performance improvements using our code have enabled halo analysis to be performed on a very large data set (8192^3 particles across 16,384 Titan nodes) for which analysis using only the existing CPU algorithms was not feasible, while the portability of our data-parallel approach allowed the same analysis code to be compiled and run on several architectures, including Nvidia GPU and Intel Xeon Phi. The science results enabled in part by this analysis for the "Q Continuum" run are described in detail in [11].

2 COSMOLOGY APPLICATION

A very simplified picture describes the process of galaxy formation in the following way. The dark matter halo is a massive, gravitationally bound object. Hot gas falls into this potential well, heats up, and star formation is triggered. These stars eventually form galaxies. Halos merge over time and accrete mass from the surrounding area and some halos grow large enough to host many galaxies, resulting in so-called clusters of galaxies. The age of a halo and its formation history also carries information about the type of galaxies that will reside inside it: if a halo formed very early, the star formation in the galaxy slowed down over time and at the current epoch, such a halo will host a red (in astronomy terms, "dead")

*e-mail: csewell@lanl.gov
†e-mail:ollie@lanl.gov
‡e-mail:heitmann@anl.gov
§e-mail:habib@anl.gov
¶e-mail:ahrens@lanl.gov

large galaxy. If the halo is still very young, the galaxies inside it will still be star forming and therefore blue.

Halos also carry information about the content of the universe. In a simulation, we specify the amount of dark matter and dark energy and the amplitude of the primordial power spectrum at the start of the simulation. The number of halos as a function of their mass (the so-called mass function) will change depending on the cosmological parameters input into the simulation. The mass function can be measured from a diverse set of observations (X-ray, optical or cosmic microwave background observations are recent examples) and can be compared to the theoretical predictions to extract cosmological parameters. In particular, clusters of galaxies (and therefore halos at the high mass end) are very sensitive to cosmological parameters, in particular to dark energy. They are therefore one of the major dark energy probes used in ongoing and upcoming surveys.

One caveat about the dark matter halos is that we do not observe the dark matter directly, but we observe it rather through tracers, like hot gas in the X-ray or the distribution of galaxies in the optical. This leads to a difficulty in the definition of the halo. Depending on the observation of interest, different halo definitions are therefore used. For X-ray observations, the temperature of the gas is connected to the mass of the dark matter halo. Here, the most convenient mass definition is given by the spherical overdensity (SO) mass within a small radius. Often an overdensity of 500 or 1000 with respect to the critical density of the universe is used. From, for example, weak lensing mass maps, it is very obvious though that halos are not spherical. Therefore, another commonly used definition for halos is the friends-of-friends (FOF) definition. The FOF halo finding algorithm traces the iso-density contours of halos, therefore allowing it to capture the nontrivial boundaries of a halo more accurately. If the simulation has enough resolution, halos within halos (so-called subhalos) can also be identified and galaxies will live within those structures. Therefore, for optical observations, the use of the FOF halo definition is more useful than the overdensity definition. The halo mass function for the FOF halo definition at one specific linking length (b=0.2) has been found to be very close to universal. This universality allows us to make predictions for the mass function using only ingredients from linear theory, once the universal form itself was calibrated carefully against one cosmological model.

Finally, a very accurate determination of the halo center is crucial for all the above. If the halo center is not at the potential minimum, the SO mass will be biased low. The correct placement of the central galaxy in the halo is also very important for comparison with observations. Halo properties like the concentration again rely heavily on accurate center determination. Due to the high computational cost of computing the potential minimum particle, other definitions for halo center are sometimes used instead.

3 Related Work

The friend-of-friends (FOF) halo finding algorithm is very commonly employed to efficiently identify groupings of particles [14]. It can function as a stand-alone isodensity-based halo-finder, as a base algorithm for sub-halo finding using hierarchical FOF, or FOF in phase space. Each particle is connected to its "friends"; that is, to all other particles within a specified "linking length" distance to it. Two particles will end up in the same halo if there exists any chain of friends between them.

Distributed-memory parallel algorithms have been developed in order to divide the work of halo finding and center finding across processors. For example, the parallel halo finder in HACC distributes the particles among all processes, each of which runs an efficient serial halo finder algorithm which constructs and traverses a KD-tree [13, 26]. Particles in "overload regions" are distributed to two processes, with the size of the overload regions related to the maximum feasible halo length, in order to ensure that each halo is found in its entirety by at least one process. After each process has computed its halos, the parallel halo finder identifies redundant halos found by neighboring processes and assigns them to one or the other. Similarly, the Parallel HOP code utilizes MPI and domain decomposition to parallelize halo finding [22].

In contrast, our halo and center finder algorithms take advantage of shared-memory parallelism within a single MPI process, as found on many-core accelerators. As such, they can serve as a replacement for the serial algorithms used within each process while still making use of these same distributed memory domain decomposition algorithms to parallelize across nodes.

Our friend-of-friends halo finder algorithm is based on a standard parallel connected component algorithm [21, 1, 15], with an edge assumed to exist between two particles if and only if the distance between them is less than the specified "linking length" parameter. GPU implementations for a variety of connected components algorithms have been presented [17, 25, 23, 12, 18]. However, one significant difference between the standard connected component problem and the friend-of-friends halo finding problem is that most connected component algorithms assume that either an adjacency matrix or an edge list are available as input. Since the memory limitations of accelerators preclude in-memory storage of all edges in this application, we extend this algorithm to efficiently find the halos without ever explicitly storing an edge list or adjacency matrix.

One common definition for the center of a halo is the "most bound particle" (MBP): the particle within a halo with the lowest potential, where the potential for a given particle is computed as the sum over all other particles of the negative of mass divided by distance. The most straightforward way to find this is to simply compute the potential for each particle and then take the minimum, which requires $O(n^2)$ operations. HACC optimizes this using an A* search algorithm, which uses an optimistic heuristic to estimate the potential for each particle, and locates the particle with minimum potential through the search without having to explicitly compute the potentials for all particles. It has been reported that this algorithm reduces the time compared to the brute force approach by a constant factor of approximately eight [6].

Due to the time it takes to compute the true MBP center, various other methods have been devised to estimate the center. One of the most common is to identify the "most connected particle" (MCP), which is the particle within the halo with the most friends. In most halos, such as those with a roughly spherical shape, one would expect to find the MCP in a region of high density that is close to the MBP. However, there are cases (such as a bar-bell shaped halo) where the MCP center may be far from the MBP center.

Our MBP center finding algorithm exploits the highly parallel nature of the problem of computing pairwise potentials to make use of the large core counts on many-core accelerators such as GPUs. Finding the MCP center is a simple extension to the traversal of virtual edge lists in our FOF halo finder using particle binning.

The DBSCAN algorithm [19] is a generalization of the friend-of-friends halo finding algorithm. It will only include particles with at least f friends as part of any halo. (For the standard FOF algorithm, $f = 1$.) Thus, it can eliminate spurious combinations of distinct halos via a long, narrow string of connected particles. A version of this algorithm is implemented as a simple extension to our FOF halo algorithm by excluding from consideration any particle with too few friends while traversing the virtual edge lists.

A spherically overdense (SO) halo finder finds a center and radius of a sphere such that the density ratio within it matches a specified value [24]. In HACC, the SO halo finder starts with the center of a FOF halo, and then searches for a radius that meets the criterion. In this paper, we do not present a data-parallel algorithm for computing SO halos themseleves. However, in practice, the most time-consuming step in finding SO halos is computing the FOF halo centers used to seed the SO halos. Since our FOF halo finding and

center finding algorithms output their results to the standard HACC data structures on the CPU, they can be used interchangably with existing CPU algorithms, such as SO halo finding. Thus, we can greatly accelerate the overall process of finding SO halos by using the data-parallel algorithms to compute MBP centers on the GPU.

GPU implementations of a Poisson solver have previously been used for accelerating scientific computations. For example, it has been used to solve the two-dimensional barotropic vorticity equation in ocean dynamics [3], and for the Particle Mesh Ewald method in molecular dynamics [5]. However, to our knowledge, it has not been used for MBP center finding in cosmology simulations.

4 Data Parallelism

Our halo and center finding algorithms have been built using data-parallel primitives, which are portable across multi-core/many-core architectures. Data parallelism is a programming model in which independent processors perform the same operation on different pieces of data. In 1990, Guy Blelloch described a scan vector model, consisting of a set of data-parallel primitives very similar to those now available in Thrust. He outlined a variety of higher-level algorithms constructed using this scan vector model in the fields of data structures, computational geometry, graphs, and numerical analysis [4]. Since each of these data-parallel primitives can be implemented efficiently on a wide range of parallel architectures, an algorithm that uses only these primitives will then likely be very efficient and portable. Thrust provides implementations for such primitives on GPU and multi-core CPU architectures.

Examples of several of these data-parallel primitives, which are used in our halo and center finding algorithms, are given in Figure 1. Line 1 performs a unary `transform`, in which a user-defined functor (`add_constant`) is applied to each element of vector **B**, with the results stored in **C**. The `for_each` primitive, shown in Line 2, is similar, but allows for side effects. It can generate more or fewer elements in its output than in the input. Here, `for_each` takes a range of indexes as input, using a virtual vector of consecutive integers known as a counting iterator, and applies a user-defined functor `double_it`, which, for each index, outputs into **C** both the value of **B** at that index, and twice that value. An `inclusive_scan` is shown in Line 3. Using the plus binary operator, it computes a running sum (prefix sum) of all elements up to and including the current element in the input. Line 4 shows an `exclusive_scan`. In contrast to `inclusive_scan`, `exclusive_scan` includes all elements of the input up to but not including the current element. An `inclusive_scan` that uses the maximum binary operator rather than addition, and that scans the vector in reverse (using the rbegin and rend iterators) is shown in Line 5. An example of a segmented operation, `inclusive_scan_by_key`, is shown in Line 6. This takes an additional input vector (**A**) containing the segment descriptors. The scan is computed independently in each segment, where a segment is defined as a consecutive range of identical values in **A**. Line 7 illustrates a `sort_by_key`, in which vector **B** is sorted, and the elements of **A** are moved along with their corresponding **B** value. The last two lines show the vectorized binary search operators. The `lower_bound` (or `upper_bound`) operator finds the first (or last) index at which each element of **B** could be inserted into the sorted vector **A** without violating the ordering.

```
        A      0      0      4      6      6      6
        B      4      5      2      1      3      0

 1) transform(B.begin(), B.end(), C.begin(), add_constant(1))
        C      5      6      3      2      4      1
 2) for_each(cnt_itr(0), cnt_itr(0)+2, double_it(B, C))
        C      4      8      5     10
 3) inclusive_scan(B.begin(), B.end(), C.begin(), thrust::plus<int>())
        C      4      9     11     12     15     15
 4) exclusive_scan(B.begin(), B.end(), C.begin(), 0, thrust::plus<int>())
        C      0      4      9     11     12     15
 5) inclusive_scan(B.rbegin(), B.rend(), C.rbegin(), thrust::max<int>())
        C      5      5      3      3      3      0
 6) inclusive_scan_by_key(A.begin(), A.end(), B.begin(), C.begin())
        C      4      9      2      1      4      4
 7) sort_by_key(B.begin(), B.end(), A.begin())
        B      0      1      2      3      4      5
        A      6      6      4      6      0      0
 8) lower_bound(A.begin(), A.end(), B.begin(), B.end(), C.begin())
        C      2      3      2      2      2      0
 9) upper_bound(A.begin(), A.end(), B.begin(), B.end(), C.begin())
        C      3      3      2      2      2      2
```

Figure 1: Example data-parallel primitives used by our algorithms, shown with Thrust pseudocode

5 Algorithms

5.1 FOF Halo Finding

5.1.1 Parallel Sparse Connected Components

The pseudocode for a standard connected components algorithm for sparse graphs based on [21, 1, 15] is shown in Listing 1. The basic strategy is to create a pseudoforest defined by a function D which maps each vertex to its parent. Initially, each vertex is its own parent. We then iteratively attempt to graft trees onto smaller vertices of other trees, and then perform one level of pointer jumping on each vertex to compress the depth. Once all vertices are in rooted stars (i.e., trees with depth one or less), the algorithm terminates, with D now defining a pseudoforest in which each connected component corresponds to an independent tree. Assuming all edges or vertices can be processed in parallel, each iteration takes constant time. The standard algorithm includes an additional step each iteration in which rooted stars are grafted onto other trees (even if the vertex is not smaller), which guarantees that the algorithm terminates in $O(\log(n))$ iterations, with n the number of vertices. In practice, we generally omit this step, since we do not encounter the worst-case input, and thus this step tends to result in a net slowdown. This algorithm, which is essentially a parallel union-find, has wide application beyond just halo analysis.

```
// Initialize each vertex to its own tree
for all vertices i
  D(i):=i

while (true) {
  // Process edges in parallel, grafting trees
  // onto smaller vertices of other trees
  for all (i,j) ∈ E pardo
    if (D(i)=D(D(i)) and D(j)<D(i))
      set D(D(i)):=D(j)

  // Exit if all the vertices are in rooted stars
  for all vertices i pardo
    set star(i):=true
  for all vertices i pardo
    if (D(i) ≠ D(D(i)))
      set star(i),star(D(i)),
          star(D(D(i))):=false
  for all vertices i pardo
    set star(i):=star(D(i))
  if (star(i) for all i) break

  // Pointer jumping on each vertex
  for all i pardo set D(i):=D(D(i))
}
```

Listing 1: Pseudocode for a standard sparse parallel connected components algorithm based on [21, 1, 15]

If we define an edge to exist between two particles if and only if the distance between them is less than the linking length, the connected components are the FOF halos. Unfortunately, it could take $O(n^2)$ time and $O(n^2)$ memory to directly compute and store all edges. In practice, the graph would not be fully connected for these N-body cosmology simulations, but particles may still have a large number of "friends" (particles within a distance less than the linking length), and an algorithm requiring storage proportional to n times a large constant factor would not be practical on accelerators such

as GPUs, which have limited memory compared to the CPU. Our algorithm makes use of space partitioning to efficiently compute the edge list "on the fly" without ever storing it in memory.

5.1.2 Space Partitioning

If the domain is partitioned into bins with edge length equal to the linking length, direct "friends" of a particle (i.e., particles with which it may share an edge) can only exist in its own bin or one of its 26 neighbor bins. Given a compact representation for each particle of all the particles in its neighbor bins, we can compute the edge list on the fly, parallel processing the particles (vertices) rather than edges. For each particle, we compute the distance to every particle in its neighbor bins, to determine whether the distance is actually less than the linking length (in which case an edge exists).

The goal of the binning algorithm is to use data-parallel primitives to compute, for each particle, a compact representation of which other particles are in its neighbor bins (including its own bin). This approach has the advantage of requiring an amount of memory that is proportional to the number of particles rather than to the number of bins. Since the linking length is generally very small relative to the domain size (the dimensions of the universe), the number of bins (most of which are empty) is much larger than the number of particles.

In order to find the particles in neighbor bins for a given particle, we need for all particles to be ordered by bin, and to have a way to quickly find the range of indexes corresponding to the particles in a given bin. Therefore, we store the indexes of the first and last particles in each neighbor bin for each particle. For n particles, this would require $27 \times 2 \times n$ storage, which is much less than the total number of bins. However, the memory requirement is further reduced by taking advantage of the fact that, in this uniform 3D binning with standard consecutive numbering of the bins, the 27 neighbor bins (including its own bin) will be arranged in nine sets of three contiguous bins. Thus, we only store indexes for nine ranges within the vector of particles (which has been sorted by bin). This is accomplished by computing a vector of the $9 \times n$ neighbor bin ids, and then performing a `lower_bound` vectorized binary search into the sorted vector of each particle's bin id in order to find the start of the ranges of neighbor particles, and a `upper_bound` vectorized binary search in order to find the end of the ranges of neighbor particles. Since all the particles in a bin will need indexes for the same ranges, memory could be further reduced by saving neighbor pointers once for each non-empty bin rather than for each particle, although we do not currently use this potential optimization. Nevertheless, we do eliminate ranges for which no friends of the particle are found so that they are not checked again in subsequent iterations, using a Boolean flag vector.

This binning algorithm is illustrated for a simple 2D example in Figure 2. The input is **I**, containing the particle ids, and their corresponding **X** and **Y** coordinates. Line 1 computes into **B** the bin id in which each particle is located. For example, particle 0 is located in Bin 13. Line 2 sorts **B** and **I** by bin id. Thus, after the sort, vector **I** contains all the particle ids, grouped by the id of the bin in which they are located, in order of increasing bin id. Particle 2, which is in bin 7, is first, while Particle 1, in bin 18, is last. Corresponding bin ids are in **B**. Due to how the bins are numbered (consecutively across rows), the nine neighbor bins correspond to three ranges of three consecutive bins. In lines 3 and 4, for each particle id in **I**, we compute into **R1** and **R2** the ids of the first and of the last bin, respectively, for each of these three ranges. For example, particle 2 is in bin 7, so possible friends are located in bins 1-3, 6-8, and 11-13. Finally, in lines 5 and 6, we then convert these *ranges of bin ids* to *ranges of indexes of particles in* ***I***, storing these indexes in **N1** and **N2**. For example, particles in bins 11-13 are located at indexes [1,4) in **I**: particles 3, 0, and 4. This mapping of a bin id to an index where the particles in that bin start or end is accomplished using vectorized binary searches (`lower_bound` and `upper_bound`). For each value in **R1** (or **R2**), it finds the first (or last) index at which it could be inserted into the sorted **B** vector without violating the ordering (see Section 4).

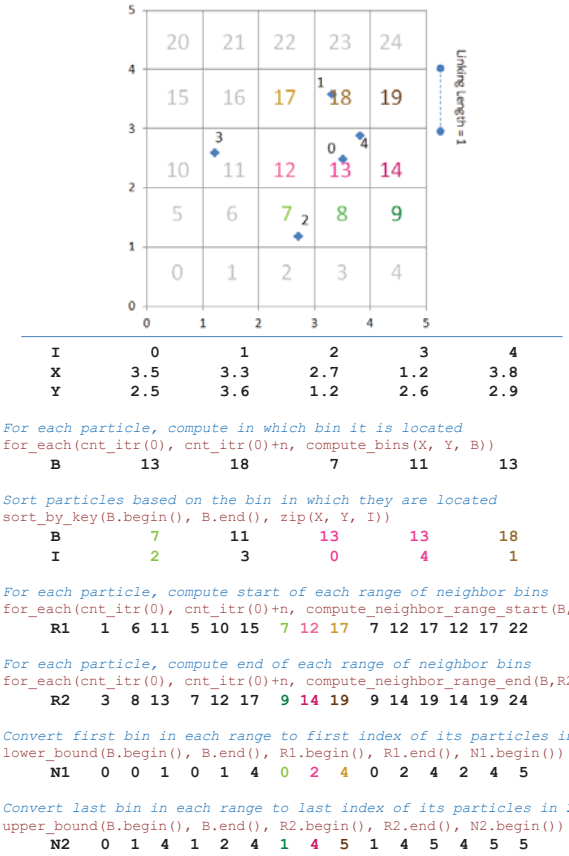

Figure 2: Illustration of the binning algorithm on simple example 2D input. The input is **I**, containing the particle ids, and their corresponding **X** and **Y** coordinates. The output are vectors **N1** and **N2**. For each element in the input, there are three corresponding elements in each output vector, which give the ranges of indexes in **I** for particles in its neighbor bins. (**N1** contains the first index of each range, and **N2** contains the last index of each range). The green, purple, and brown shading of elements corresponding to selected bins go along with the example explained in the text. See text for details.

The output are vectors **N1** and **N2**, which contain the beginning and ending of three ranges (nine in 3D) in the sorted particle vectors for which each particle will need to search for its potential friends. For example, Particle 0 is at index i=2 in the sorted **I** vector (Line 2). We need to search for friends of Particle 0 only within the neighbor bins, which are given by the ranges **R1**$[3 \cdot i]$ to **R2**$[3 \cdot i]$, **R1**$[3 \cdot i + 1]$ to **R2**$[3 \cdot i + 1]$, and **R1**$[3 \cdot i + 2]$ to **R2**$[3 \cdot i + 2]$. In this case, with i=2, this gives bins 7-9, 12-14, and 17-19. Elements corresponding to these bins are shown in shades of green, purple, and brown, respectively, in the figure. Note these bin ranges encompass the nine bins surrounding Particle 0 in the figure. The values of **N1** and **N2** at these same indexes give the indexes of particles in the sorted vector **I** that correspond to these bin ranges. Note the end of the range is exclusive. Thus, the particles in bins 7-9 can be found at **I**[**N1**[6]] through **I**[**N2**[6]]: **I**[0] up to (but not including) **I**[1]. The particle at **I**[0] is 2. Similarly, the particles in bins 12-14 are found at **I**[**N1**[7]] through **I**[**N2**[7]]: Particles 0 and 4. The particles in bins 17-19 are found at **I**[**N1**[8]] through **I**[**N2**[8]]: Particle 1. We must then

explicitly check the distance between Particle 0 and each of these four particles (three not including itself). In this simple example, we have only avoided the need to check Particle 0 with one other particle (Particle 3), but in a large real problem, only a very small proportion of the total particles are likely to be within the neighbor bins, given how small the bins are relative to the full domain.

5.1.3 Implementation and Analysis

The halo finding algorithm is implemented using data-parallel Thrust primitives (such as `for_each`, `reduce`, and `sort`) along with custom functors. Binning allows us to compute the edges used by the sparse connected components algorithm in $O(f)$ time per particle ($O(n \cdot f)$ total work), where f is the number of potential friends (i.e., particles in neighboring bins). In the worst theoretical case (all particles in the same bin), $f = O(n)$. However, in this case, there actually are $m = O(n^2)$ edges, so there is no way to compute them all with less than $O(n^2)$ total work.

While this algorithm parallelizes well, and avoids the duplication of particles in overload zones required when parallelizing the serial algorithm by using multiple MPI ranks, it is not work optimal. With n vertices and m edges, the parallel connected components algorithm on which it is based performs $\log(n)$ iterations, each of which requires $O(n+m)$ operations, for a total work of $O((n+m)\log(n))$. A serial algorithm based on a depth-first or breadth-first search requires only $O(n+m)$ work. Work-optimal randomized parallel connected components algorithms exist [7, 8], but involve explicit storage and modification of the edge list. Since our parallel FOF halo finder requires a factor of $O(\log(n))$ extra work, it needs to run on enough cores to offset this difference before yielding an advantage.

5.2 Center Finding

We have also implemented algorithms using PISTON/VTK-m to find MBP and MCP centers for each halo. If both halos and their centers are to be computed using the data-parallel algorithm, they are most efficiently computed together, allowing them to share some pre-processing (such as binning the particles, which is needed for FOF halo finding and for MCP center finding, and transferring data to and from the accelerator). However, the code is structured such that our halo and center finders can be mixed and matched with other algorithms, such as the original HACC halo and center finders. Therefore, for example, one could compute halos using the original code on the CPU but the centers on the GPU.

The potentials for all particles in a halo can be easily computed in parallel using a `for_each` which applies a functor that loops over all particles in the halo to each element of the vector of particle ids. The index of the particle with the minimum potential can then be found using a `min_element` operator. This particle with minimum potential is the MBP center.

If all particles are within a single vector (as would be the case if FOF halos were computed using the algorithm in the previous section), the MBP centers for all halos can be computed in parallel (rather than one halo at a time) using segmented vectors, with the segments defined by the halo ids (and the vector sorted according to halo id). A vector containing the index of the first particle in each particle's halo can be computed using a min `inclusive_scan_by_key`, with a counting iterator as the input vector. Similarly, a vector containing the index of the last particle in each particle's halo can be computed using a reverse max `inclusive_scan_by_key`. For each particle, the functor for computing potentials then loops over the range defined by these starting and ending indexes. The minimum potential for each halo can be associated with each particle in the halo by a "segmented min-distribute", implemented in Thrust with a min `inclusive_scan_by_key` with the vector of potentials as input, followed by a reverse min `inclusive_scan_by_key`. The index of the MBP center can then be associated with each

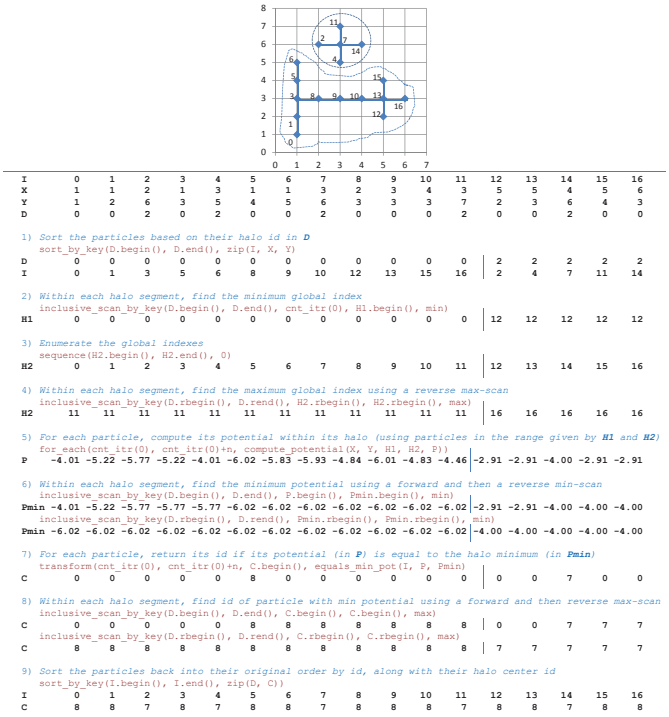

Figure 3: Illustration of finding MBP centers for all halos in parallel using segmented vectors, with simple example input. The input are the particle ids (**I**), coordinates (**X**, **Y**), and halo id (**D**, found using the halo finding algorithm). The output is a vector **C** containing for each particle the id of the MBP center for its halo. See text for more details.

particle in the halo using a "segmented max-distribute", implemented using a max `inclusive_scan _by_key` with a functor that returns its index for a particle with a potential equaling the minimum value and zero otherwise, followed by a reverse max `inclusive_scan_by_key`. The final result is a vector which contains, for each particle, the index of the center particle of its halo. The core of this algorithm, implemented using data-parallel Thrust primitives, is illustrated with example input in Figure 3.

The number of "friends" for each particle can be easily counted during any iteration through the virtual edge list in the FOF halo algorithm, followed by a max `reduce` and a max `transform _reduce`, or segmented `inclusive_scan_by_key`s, as with MBP centers, to find the index of the particle with the most friends in each halo (i.e., the MCP center). If only MCP centers, and not the halos themselves, are to be found, the binning process must still be completed as a pre-processing step in order to efficiently loop through the potential friends of each particle.

5.3 Extensions to Halo and Center Finding Algorithms

The FOF halo finding algorithm described above may be generalized to a version of the DBSCAN algorithm. Just as when finding MCP centers, the number of "friends" for each particle may be counted during the first iteration through the virtual edge list, and any particles with fewer than the specified number of friends can be excluded in subsequent searches for edges, ensuring that they are not included in any halo. While not strictly equivalent to the algorithm described in [19], it also helps to avoid the problem of two halos being combined into one through a thin line of particles.

Although it parallelizes very well, the MBP center finding algorithm described above is still an $O(n^2)$ algorithm. However, we have also implemented an alternative algorithm which avoids com-

puting potentials for every particle. In this algorithm, the potential field is computed on a grid. The grid is superimposed over the extents of the halo. The density is estimated on the grid by computing the cell in which each particle lies (as in the binning procedure used for FOF halo finding), sorting the particles by cell id, and using `upper_bound` to compute how many particles are in each cell. Once we have the density, we can calculate the potential field on the grid by solving the Poisson equation $\nabla^2\phi = \rho$. The boundary conditions for this problem are open, but we approximate the solution using the boundary condition $\phi = 0$ at $\partial(\Omega)$. Because of the boundary condition, we use the Discrete Sine Transformation (DST) instead of the standard Fast Fourier Transform (FFT). Our CUDA implementation uses Nvidia's CUFFT library, which does not support DST directly, so we implemented this by pre and post processing the input as described in [2].

First, we extended the input N^3 3D array to $(2N+2)^3$ with odd symmetry in parallel, using a `for_each` on a counting iterator for each of the $(2N+2)^3$ elements of the output. Each real element was converted into a complex number, and the complex 3D array was passed to CUFFT. The imaginary part of the output are the coefficients of the sine basis of the FFT which correspond to the coefficients of the DST. We then extracted the "upper-left" 1/8 of this 3D array (since the rest of the array is just high frequency alias of this). We can optionally choose to scale the result so the property $DST^{-1}(DST(x)) = x$ holds. Once we obtained the DST, we solved the Poisson equation by performing a forward DST on the density field to obtain the frequency domain, and then divided each coefficient by the eigenvalues of the tensor. Finally, we performed an inverse DST to get the potential of the equation.

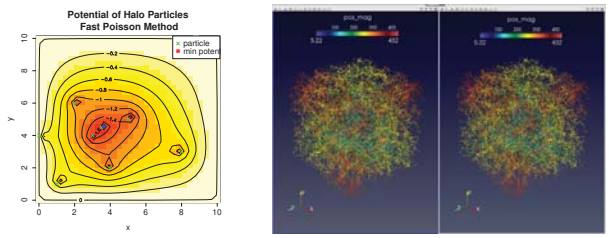

Figure 4: (A, left) Illustration of Fast Poisson Method for a simple 2D example. (B, right) Visualization of halos found by the original HACC algorithms (left) and our data-parallel algorithms (right), shown here for a 1024^3 particle data set using 128 nodes on the Moonlight supercomputer. Each halo is represented by a point at its center, colored based on its distance from the center of the domain.

The grid cell with minimum potential can then be found using `min_element`. The potential is then computed in parallel for all particles located in a cell that is within a specified radius of the minimum grid potential cell. All steps of this algorithm have been implemented using data-parallelism. The total work for creating the input density grid is $O(n\log(n))$, where n is the number of particles. The total work for solving the Poisson equation on the grid using DST is $O(g\log(g))$, where g is the grid resolution. In the final step, potentials are computed for r particles within the search radius, for a total work of $O(r \cdot n)$. Assuming g and r are much less than n, this is much less total work than the $O(n^2)$ algorithm.

While our tests have succesfully used this method to find the exact MBP center within a small radius of the minimum potential grid cell in the vast majority of cases, we have not yet proven bounds on the grid resolution and search radius necessary to guarantee the optimal solution. Some halos, such as those with a small number of particles or that are very un-relaxed (containing a lot of substructure or two or more large major components), can be more difficult for this algorithm to find precisely. The method is illustrated for a simple 2D example in Figure 4A.

6 Results

We evaluated these algorithms in several scenarios on three different machines. First, we ran a moderate-sized test problem (1024^3 particles) using the GPUs on the Moonlight supercomputer at Los Alamos National Laboratory, using both one MPI rank per node and 16 MPI ranks per node. To demonstrate the portability of the data-parallel algorithms, we ran a small test problem (256^3 particles) on an Intel Xeon Phi accelerator on a single node of the Stampede system at the Texas Advanced Computing Center. We ran both a moderate-sized test problem (1024^3 particles) and a very large problem (8192^3 particles) on the GPUs of the Titan supercomputer at Oak Ridge National Laboratory. Due to memory constraints, the large simulation runs on Titan can only use one MPI rank per node, making the speed-up by utilizing the GPU especially valuable. Finally, we evaluated the Poisson center-finding algorithm on a single node of Moonlight with various sized halos.

6.1 Moonlight

The original CPU code and our new data-parallel code for halo and center finding were compared on a 1024^3 particle data set, using 128 nodes on the Moonlight supercomputer at Los Alamos National Laboratory, with 16 MPI processes per node. The original CPU code was also tested with one MPI process per node. Each node has a 16-core 2.6 GHz Intel Xeon E5-2670 CPU, 64 GB of RAM, and two Nvidia Tesla M2090 GPUs. Due to memory constraints, access to the GPUs in our version was serialized on each node between two groups of eight processes each.

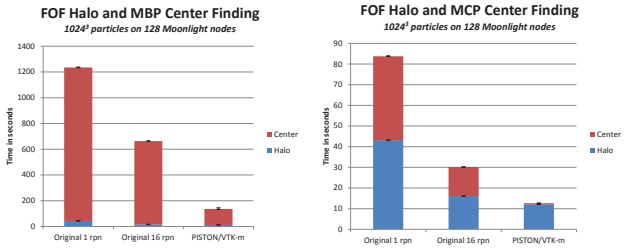

Figure 5: (A, left) Timing results for FOF halo finding and MBP center finding on Moonlight. (B, right) Timing results for FOF halo finding and MCP center finding on Moonlight. Error bars show minimum and maximum over three runs (but are often too small to be visible).

As shown in Figure 5A, the original CPU code using 16 MPI ranks per node took 16 seconds to find halos and 647 seconds to find MBP centers, for a total of 663 seconds. With only one MPI rank per node, it took 43 and 1192 seconds, respectively, for a total of 1235 seconds. Our code took a total of 135 seconds to find both the halos and their MBP centers, a factor of about 4.9 faster. When finding MCP centers, as shown in Figure 5B, the original CPU code with 16 ranks per node took 14 seconds, for a total of 30 seconds (41 seconds, for a total of 84 seconds, with one rank per node), while our code took 12.2 seconds for halo finding and 0.5 seconds for MCP center finding, for a total of 12.7 seconds, about 2.5 times faster. The accuracy of the results was verified by performing a diff on the FOF halo properties files produced by the two codes, which outputs the id, total mass, number of particles, and center for each halo. For this data set, 52,738 halos containing a total of 325,065,528 particles were found by both codes. The output halo ids, mass, and number of particles were identical for both codes. For 37 of the 52,738 halos, the reported MBP centers were very slightly different, but spot-checking a few of these indicated that the two particles had essentially the same potential within the resolution of the computation. For the MCP centers, about 10% reported different centers, but it was verified that the two reported centers each had the exact same number of "friends" and thus were

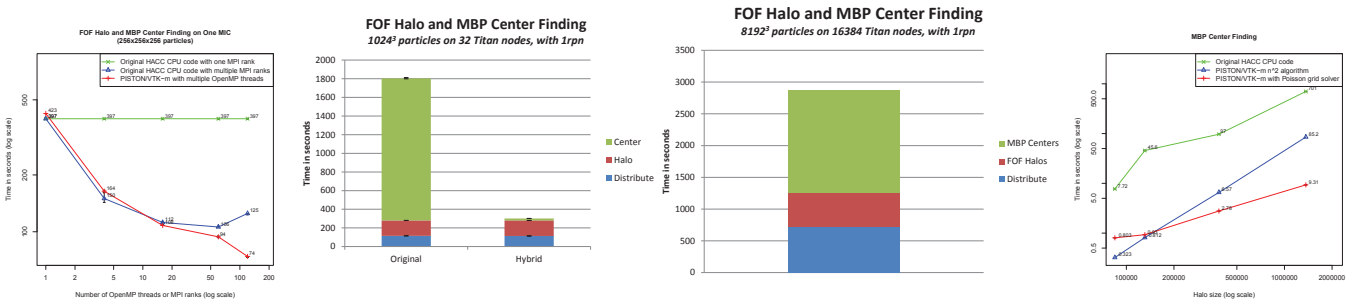

Figure 6: (A, left) Timing results for FOF halo finding and MBP center finding on a single MIC node on Stampede. (B, middle left) Timing results for FOF halo finding and MBP center finding with 1024^3 data set on Titan, with one process per node. (C, middle right) Timing results for FOF halo finding and MBP center finding with 8192^3 data set on Titan, with one process per node. (D, right) Timing results for center finding on a single node on Moonlight. Error bars show minimum and maximum over three runs (but are often too small to be visible).

equally correct. A simple ParaView visualization of halo centers found by each method is shown in Figure 4B.

6.2 Stampede

For the tests on Moonlight and Titan, the data-parallel algorithms were compiled to the Thrust CUDA backend. In order to demonstrate the portability of the data-parallel algorithms, the same code was compiled to the Thrust OpenMP backend (including our own OpenMP implementation of scan) and run on a 256^3 particle data set on an Intel Xeon Phi SE10P (MIC) Coprocessor on a single node of the Stampede system at the Texas Advanced Computing Center. The Xeon Phi has 61 cores, capable of running up to four threads each. The code was compiled using the Intel compiler, but no attempt was made to manually optimize the OpenMP backend specifically for the MIC architecture. Figure 6A shows the scaling of the original algorithms with the number of MPI processes and of the data-parallel algorithms with the number of OpenMP threads. Since threads are more lightweight than processes, and because using threads rather than processes avoids the need for additional replication of points in the overload regions between processes, the data-parallel algorithms can scale to more cores.

When only considering the FOF halo finding algorithm, however, our algorithm (46 seconds) performs better than the original algorithm using one process (65 seconds), but does not surpass the original algorithm with 60 processes (6.3 seconds). While our halo finder using GPUs was able to outperform the original algorithm using all 16 CPUS cores per node on Moonlight (12.1 vs. 16.1 seconds), the disadvantage that the parallel algorithm is not work-optimal ($O((n+m)\log(n))$ work compared to $O(n+m)$ for the serial algorithm) outweighs the advantage of avoiding extra overload regions and heavier-weight processes. Nevertheless, the superior performance of the parallel center finder yields an overall better performance on the Xeon Phi. Since center-finding times tend to dominate halo-finding times, the impact of the non-work optimal halo-finding is masked in the overall performance results shown in Figure 6A. It should also be noted that, the larger the halos, the more time is spent in center finding compared to halo finding, and thus the greater the advantage of the data-parallel algorithms for the overall analysis pipeline.

6.3 Titan

Our next goal was to accelerate the analysis for a very large 8192^3 particle simulation on the Titan supercomputer at Oak Ridge National Laboratory. Titan consists of 18,688 AMD 16-core Opteron processors, each paired with an Nvidia Kepler GPU. For the Q Continuum run, half a trillion particles were evolved, and analysis was to be performed at 100 time slices. Traditionally, with smaller simulations, analysis would take roughly the same amount of time as the simulation itself. However, with the very large halos that form in the Q Continuum simulation, the original CPU algorithms were expected to take many days to compute MBP centers, while the simulation itself required only a few hours. Because the memory requirements increase with the number of MPI processes due to overload regions, the HACC simulation with this data can only use one MPI process per node, along with the associated GPU, given the main memory available on Titan. (In other words, loading all 8192^3 particles plus all the extra copies in overload regions needed when running with more than one rank per node would exceed the total of about 260TB of memory available on Titan.) Furthermore, HACC's OpenCL physics kernels running on the GPUs could only support one rank per node. This creates a significant bottleneck for the original serial-per-process halo and center finding code, as it essentially leaves 15 of 16 cores on each node idle. Thus, there is an even greater potential for speed-up by utilizing the GPUs for analysis than in the 16-process per node scenario reported above on Moonlight. However, this configuration also results in a very large number of particles per process (about 90 million). Due to the memory requirements for binning the particles, the 6 GB GPU memories cannot hold all of the data structures necessary for efficient FOF halo finding at once. A long-term solution for this issue would be to implement the algorithms using a streaming interface that could compute results without ever having to hold all the data in memory at once. A somewhat simpler but less elegant and less efficient solution would be to re-partition the particles within a process into multiple virtual processes (just as the global parallel halo finder partitions points among real processes), and serialize access to the GPU among these virtual processes. A third option was to utilize a hybrid approach, in which the halos are computed on the CPU using the original HACC code, but the centers are computed on the GPU using PISTON. Our MBP center finding algorithm requires much less memory than the halo finding algorithm (especially if centers are found one halo at a time rather than all at once), but provides the large majority of the speed-up, since MBP center finding takes much longer than FOF halo finding with the original CPU code.

Using this hybrid implementation on a 1024^3 data set (output from a simulation run) using 32 processors with one process per node, the total time to compute FOF halos and find the MBP center for each was about 30 minutes using only the original CPU code, and about five minutes when using our code on the GPU to find the MBP centers (Figure 6B). Distributing particles and finding the FOF halos took essentially the same amount of time in both cases (since the original CPU code was used for these operations in both cases): about 115 seconds and 165 seconds, respectively. However, the original CPU code spent 1524 seconds for MBP center finding, while our code on the GPU spent only 21 seconds. This is about a 70x improvement for MBP center finding, and about a 6x improvement for the total analysis computation (including distributing particles, FOF halo finding, and MBP center finding).

Finally, we ran the data-parallel algorithms on one early time step and one relatively late time step from a large 8192^3 particle simulation run. A timing break-down for the late time step (when many halos have formed), using 16,384 nodes, each with a GPU, is shown in Figure 6C. Running the original algorithm for this data set was expected to be too expensive, so only the data-parallel algorithms were run. The 1024^3 data set described above was intended to be representative of the large problem, scaled down to 32 nodes, so the relative performance would be expected to be similar.

6.4 Poisson Solver

Our Poisson solver center finder was compared to our $O(N^2)$ MBP center finder and to the original HACC CPU MBP center finder on one node of Moonlight. Both of our implementations utilized the GPU. As shown in Figure 6D, the extra overhead involved in computing the potential field on the grid resulted in slower performance than our other MBP center finder for small halos. However, for larger halos, the reduction in the number of particles for which a potential has to be explicitly computed far outweighs this overhead, resulting in much faster performance. For each test case, a grid size of 63^3 and a search radius of two cells in each dimension were used, and the correct MBP center was found by all three algorithms.

7 Conclusion

We have presented data-parallel algorithms for FOF halo finding, including a version of DBSCAN; most connected particle center finding; and most bound particle center finding, including a grid-based approximation method which reduces the total number of operations. These algorithms have been implemented using the PISTON component of VTK-m, a library of portable data-parallel visualization and analysis operators, utilizing Nvidia's Thrust library. The exact same code has been compiled and run on different multi-core and many-core architectures, including GPUs using CUDA and Xeon Phis using OpenMP. These data-parallel algorithms have allowed the HACC analysis operators to exploit the parallelism available on shared-memory accelerators, resulting in large speed-ups, especially for MBP center finding, and in contexts in which only one MPI rank per node may be run. This has enabled halo analysis to be performed on a 8192^3 particle data set for which analysis using traditional CPU algorithms was not feasible.

There are several potential directions for future work based on the limitations of our current approach. First, the analysis results presented here were all computed in post-processing. *In-situ* analysis is a promising alternative, although it could present load-balancing problems, since some halos are much larger than others. Next, while the Poisson-based center finder is much faster than the MBP center definition for large halos, the latter is currently better vetted by the domain scientists (which is why it was used for the large runs presented in this paper). Thus, there is still a need to sufficiently vet a faster center finder (such as the Poisson-based method) as we move to even larger simulations. Finally, as discussed with regards to the Stampede results, a non-work optimal parallel algorithm such as our FOF halo finder will not necessarily be faster than running a serial algorithm with domain decomposition across multiple MPI ranks when the number of available cores is relatively small. Therefore, better, work-optimal algorithms (perhaps using randomization) could be beneficial.

Acknowledgements

This work was supported by the Scientific Discovery through Advanced Computing (SciDAC) program funded by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research under the Institute of Scalable Data Management, Analysis and Visualization (SDAV). An award of computer time was provided by the Innovative and Novel Computational Impact on Theory and Experiment program. This research used resources of the OLCF, which is supported by DOE/SC under contract DE-AC05-00OR22725. Xeon Phi results were run on resources of the Texas Advanced Computing Center. SH and KH were supported by the U.S. DOE, Basic Energy Sciences, Office of Science, under contract No.DE-AC02-06CH11357. We also thank Pat Fasel, George Zagaris, Robert Maynard, Nicolas Frontiere, and Hal Finkel for valuable contributions to this work.

References

[1] B. Awerbuch and Y. Shiloach. New connectivity and msf algorithms for ultra-computer and pram. *IEEE Tr. on Computers*, 36(10), 1987.

[2] M. Bader. Algorithms of scientific computing: Fast poisson solvers. Technical University of Munich, Summer 2013.

[3] F. Bleichrodt, R. Bisseling, and H. Dijkstra. Accelerating a barotropic ocean model using a gpu. *Ocean Modelling*, 41:16–21, 2012.

[4] G. Blelloch. *Vector models for data-parallel computing*. PhD thesis, MIT, 1990.

[5] P. Ewald. Die berechnung optischer und elektrostatischer gitterpotentiale. *Ann. Phys.*, 1921.

[6] P. Fasel. Cosmology analysis software. *Los Alamos National Laboratory Tech Report*, 2011.

[7] H. Gazit. An optimal randomized parallel alg. for finding connected components in a graph. *SIAM J. on Computing*, 20(6), 1991.

[8] J. Greiner. A comparison of data-parallel algorithms for connected components. Technical report, CMU, 1993.

[9] S. Habib, V. Morozov, N. Frontiere, H. Finkel, A. Pope, and K. Heitmann. Hacc: Extreme scaling and performance across diverse architectures. *Supercomputing*, 2013.

[10] K. Heitmann. Large-scale simulations of sky surveys. *IEEE Computing in Science and Engineering*, 2014.

[11] K. Heitmann, N. Frontiere, C. Sewell, S. Habib, A. Pope, H. Finkel, S. Rizzi, J. Insley, and B. S. The q continuum simulation: Harnessing the power of gpu accelerated supercomputers. *The Astrophysical Journal Supplement*, accepted for publication.

[12] S. Hiloach and U. Vishkin. An o(log n) parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57–67, 1982.

[13] C.-H. Hsu, J. Ahrens, and K. Heitmann. Verification of the time evolution of cosmological simulations via hypothesis-driven comparative and quantitative visualization. *IEEE Pacific Vis. Symposium*, 2010.

[14] J. Huchra and M. Geller. Groups of galaxies i. nearby groups. *ApJ*, 257:423, 1982.

[15] J. JaJa. *An Introduction to Parallel Algorithms*, chapter 5, pages 204–222. Addison-Wesley, 1992.

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

[17] W. mei W. Hwu. *GPU Computing Gems Emerald Edition*, chapter 35, page 572. Morgan Kaufmann, 2011.

[18] V. Oliveira and R. Lotufo. A study on connected components labeling algorithms using gpus. *SIBGRAPI*, 2010.

[19] M. A. Patwary and et al. A new scalable parallel dbscan algorithm using the disjoint-set data structure. *Supercomputing*, 2012.

[20] C. Sewell, L.-t. Lo, and J. Ahrens. Portable data-parallel visualization and analysis in distributed memory environments. *IEEE Symposium on Large-Scale Data Analysis and Visualization*, October 2013.

[21] Y. Shiloach and U. Vishkin. An o(logn) parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57–67, 1982.

[22] S. Skory, M. Turk, M. Norman, and A. Coil. Parallel hop: A scalable halo finder for massive cosmological data sets. *The Astrophysical Journal Supplement Series*, 191(1), 2010.

[23] J. Soman and et al. Some gpu algorithms for graph connected components and spanning tree. *Parallel Processing Letters*, 20(325), 2010.

[24] J. L. Tinker and et al. Toward a halo mass function for precision cosmology: the limits of universality. *Astrophysics J.*, (688), 2008.

[25] V. Vineet, S. Pawan Harish, and P. Narayanan. Fast min. spanning tree for large graphs on the gpu. *High Perf. Graphics*, 2009.

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