Parallel DTFE Surface Density Field Reconstruction

Esteban Rangel*†, Nan Li†, Salman Habib†, Tom Peterka†, Ankit Agrawal*, Wei-keng Liao*, Alok Choudhary*
*Electrical Engineering and Computer Science Department
Northwestern University
Evanston, IL USA
†Argonne National Laboratory
Lemont, IL USA

***Abstract*—We improve the interpolation accuracy and efficiency of the Delaunay tessellation field estimator (DTFE) for surface density field reconstruction by proposing an algorithm that takes advantage of the adaptive triangular mesh for line-of-sight integration. The costly computation of an intermediate 3D grid is completely avoided by our method and only optimally chosen interpolation points are computed, thus, the overall computational cost is significantly reduced.**

The algorithm is implemented as a parallel shared-memory kernel for large-scale grid rendered field reconstructions in our distributed-memory framework designed for N-body gravitational lensing simulations in large volumes. We also introduce a load balancing scheme to optimize the efficiency of processing a large number of field reconstructions. Our results show our kernel outperforms existing software packages for volume weighted density field reconstruction, achieving ∼10× speedup, and our load balancing algorithm gains an additional ∼3.6× speedup at scales with ∼16k processes.

***Keywords*-parallel surface density; Delaunay tessellation field estimator;**

I. Introduction

Reconstructing a continuous field from a discrete point set is a fundamental operation in scientific data analysis. From a given set of irregularly distributed points in 3D, the task is to create a field defined on a regular grid from some property defined on the points. The property can be any locally describing quantity [1], e.g., mass density, with the usual assumption being that the points are acting as tracers of an underlying continuous field. The gridded field representation of irregularly distributed points is often preferred for some tasks such as visualization or applying certain mathematical operations, e.g., the Fourier transform. For many applications in astrophysics and cosmology [2] [3], only the 2D projected field is required, however, practically all methods for creating a 2D grid compute a costly 3D grid as an intermediate representation [4].

Our work is motivated by a gravitational lensing [5] simulation where accurate surface density[1] estimation is a critical and costly step [6], but the techniques developed generalize to any application requiring high-performance grid interpolation with low noise properties. For many types of analysis that involve gathering meaningful statistics [7], it is common for multiple gridded fields of sub-volumes to be needed from a much larger domain volume. The configuration of these sub-volumes may represent regions of interest, such as high particle concentrations, or an arrangement in some particular geometry, such as co-location along the line of sight to an observer [8]; depending on the application, sub-volumes may be completely disjoint, or have significant overlap. The number of particles within each sub-volume may also vary considerably. Together, the typically non-uniform spatial distribution of interesting sub-volumes and their varying number of particles often cause a significant load imbalance on the computing resources needed by data-parallel approaches to large-scale N-body simulations. While there are a few freely available parallel density estimation software packages [9] [4], they tend to focus on generating a single continuous gridded field from the input.

We propose a hybrid (shared/distributed-memory) algorithm that uses the DTFE method [10] to create low-noise gridded field representations of specified sub-volumes. Our distributed memory parallelization strategy uses a uniform

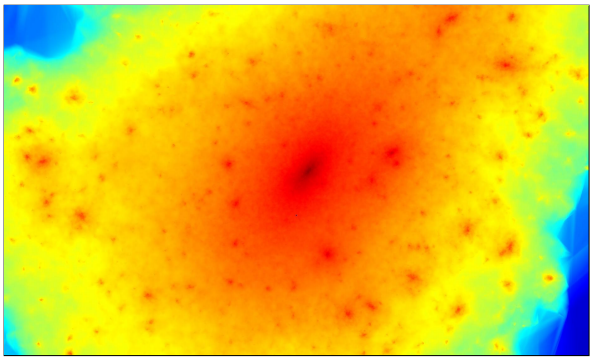


Figure 1: An example of a typical surface density field computed during a strong lensing study from an N-body particle simulation. The DTFE method was used to generate this 2048×2048 grid representing ∼1.5 million particles within a sub-volume of $(4Mpc\ h^{-1})^3$. This is the largest structural object in the final snapshot of a 1 billion particle N-body simulation with volume of $(256Mpc\ h^{-1})^3$.

[1]Surface density is volumetric density integrated along a path, i.e., it is the mass per unit area.

2168-9253/16 $31.00 © 2016 IEEE
DOI 10.1109/CLUSTER.2016.40

volume decomposition of the data with ghost zones that replicate particles in adjacent volumes. The ghost zones are large enough such that any region of interest can be computed independently by the process owning the volume where the region is located, using our shared memory kernel. The density kernel was algorithmically optimized for computing surface density by identifying the mathematically optimal points for interpolation when using the DTFE method. In cases where there is an expected computational load imbalance across processes, we propose an algorithm for determining an *a priori* communication pattern for work sharing that is based on modeling the total computation.

Our load balancing results show experiments on two large science quality N-body simulation datasets. The first is a high mass resolution simulation with 1024^3 particles and different distributions for the (1024×1024) grids that need to be computed. We show that with a high mass resolution simulation where work imbalance becomes problematic, we can regain much of the performance loss with our load balancing algorithm, showing a speedup of $\sim$2.8 with 240 MPI processes (6 OpenMP threads/process) on an InfiniBand analysis/visualization cluster. The second larger volume simulation with less mass resolution has 3200^3 particles. We demonstrate scaling up to 16,384 MPI processes (4 OpenMP threads/process) on an IBM BG/Q supercomputer and show a load balancing speedup of $\sim$3.6. We also evaluated our method against existing parallel software without enabling our load balancing functionality. As mentioned, since these software packages construct a single large gridded field, the datasets are restricted to a much smaller size on the order of 10^6 particles. We ran our implementation in *single-process-multiple-thread* mode and show that our grid rendering kernel is more efficient for computing surface density by $\sim10\times$ when compared to available shared-memory parallel software. To compare against distributed-memory parallel software, we ran in *multiple-process-single-thread* mode and decomposed the single large gridded field into multiple subgrids, showing $\sim8\times$ improvement in execution time.

The rest of the paper is organized as follows. In Section II we describe existing density estimation software and the parallelization strategy used. In Section III we provide essential background on the DTFE method and point location, performing a brief analysis of point location strategies. We then discuss surface density and the typical approach for rendering gridded fields. In Section IV, we describe our kernel algorithm for gridded surface density using the DTFE method and our load balancing algorithm. Section V describes our experiments and the performance improvements we achieved. Finally, in Section VI we discuss future work and provide concluding remarks.

II. Related Work

The DTFE public software [9] is a freely available C++ code for rendering fields defined on a grid from point sets. While the software can not directly produce a surface density field, it can compute the 3D density field, which can be converted into surface density as described in Section III. The code was designed to implement the DTFE method and uses the CGAL [11] computational geometry library for constructing the Delaunay triangulation. Grid rendering is done using the walking algorithm described in Section III. The DTFE public software uses OpenMP to parallelize computation in shared memory architectures implementing an equal volume decomposition with ghost particle regions. Computation on the sub-volumes is performed by individual threads and the resulting density fields are recombined to produce the final result. No attempt is made to balance workloads because the intention of the software is for large volume analyses in cosmology where the distribution of particles becomes mostly homogeneous at the scales of interest. Significant thread imbalance is noticed when small volumes are parallelized, as is needed by high mass resolution N-body simulations, since parallelization is limited to the memory of a single computing node.

The TESS Density Estimator [4] is a freely available C++ MPI software for computing surface density. Like the DTFE public software, the TESS Density Estimator runs in two stages: constructing a Voronoi tessellation with the Qhull library [12], followed by estimating density at grid points covered by Voronoi cells. In both codes, the two stages are executed in parallel, however, the primary difference is that TESS constructs a global tessellation [13], rather than overlapping triangulations, to interpolate the gridded density field. The advantage to this method is that ghost zones do not need to be specified by a user, but, additional interprocess communication is required. Another fundamental difference is the interpolation method; TESS uses a zero-order interpolation rather than a first-order linear interpolation. The software allows for the data partitioning to be specified, and we will use this feature to make a comparison with our implementation.

A. Complexity Analysis

Throughout the paper we use the following notation when discussing complexity: N, the number of particles (points in 3D), and N_g, the number of grid cells per grid dimension. We describe the computational complexity of both software packages together here since the two stages they both implement are very similar. The CGAL library uses a *flip* algorithm for constructing a Delaunay triangulation with overall complexity of $\mathcal{O}(N^2)$, and Qhull computes the Delaunay triangulation by computing the convex hull of the points projected onto a higher dimensional paraboloid, also with overall complexity of $\mathcal{O}(N^2)$. Both software packages either incrementally or fully construct the 3D density grid and require $\mathcal{O}({N_g}^3)$ operations. Thus, the total computational complexity for both methods is $\mathcal{O}(N^2 + {N_g}^3)$.

III. Background

A. DTFE Method

DTFE is a first order multi-dimensional interpolation method that relies on a Delaunay triangulation of the input to obtain the local gradients of a function. First proposed by Bernardeau and van de Weygaert [1] for producing volume-weighted velocity fields, the gradient within a tetrahedron is assumed constant, and discontinuous at its boundaries. The field value for a d-dimensional point $\boldsymbol{x}$ is interpolated using the Delaunay vertices $\boldsymbol{x_0}, \boldsymbol{x_1}, ...\boldsymbol{x_d}$ of the containing tetrahedron by

$$\widehat{f(\boldsymbol{x})} = f(\boldsymbol{x_0}) + \widehat{\nabla f}|_{Del} \cdot (\boldsymbol{x} - \boldsymbol{x_0}) \tag{1}$$

where $\widehat{\nabla f}|_{Del}$ is the estimated constant field gradient using the known field values $f(\boldsymbol{x_0}), f(\boldsymbol{x_1}), ...f(\boldsymbol{x_d})$. For density field reconstruction, the on-site density values are estimated by the inverse volume of the contiguous Voronoi cell, thereby ensuring mass conservation. The estimated density for each input, $\boldsymbol{x_i}$, is given by

$$\widehat{\rho(\boldsymbol{x_i})} = \frac{(d+1)m}{\sum_{j=1}^{N_{\mathcal{T},i}} V(\mathcal{T}_{j,i})} \tag{2}$$

where $N_{\mathcal{T},i}$ denotes the number of tetrahedra $\mathcal{T}_{j,i}$ having $\boldsymbol{x_i}$ as a vertex, m is the mass, and $(d+1)$ is the tetrahedral volume normalization factor.

B. Surface Mass Density

The projected 3D density, known as the surface mass density, is used by the *thin lens approximation* in gravitational lensing. The equation for the surface density is given as

$$\Sigma(\boldsymbol{\xi}) = \int \rho(\boldsymbol{\xi}, z)\ dz \tag{3}$$

C. Rendering Gridded Fields

It follows that the projection of a field defined on a regular uniform 2D grid can be computed from a field defined on a regular uniform 3D grid in a straightforward manner. The density value for a grid cell having dimensions $(\Delta x, \Delta y)$ with representative point $\boldsymbol{\xi}$ in the 2D gridded density field is approximated by

$$\widehat{\Sigma_{i,j}(\boldsymbol{\xi})} = \sum_{k=1}^{N_z} \rho_{i,j,k}\widehat{(\boldsymbol{\xi}, z_k)}\Delta z \tag{4}$$

where N_z is the number of 3D grid cells in the z dimension, and $(\boldsymbol{\xi}, z_k)$ is the representative point(s) for the 3D cell with dimensions $\Delta x, \Delta y, \Delta z$. Using this method, it is necessary to determine the containing tetrahedra for all the representative points of the 3D grid in order to interpolate their field values. This is usually incrementally done using the walking method (equation 6) by locating representative points for adjacent grid cells to minimize the walk length. The computational complexity is therefore $\mathcal{O}(N_{cell})$, where N_{cell} is the number of 3D grid cells. Note that when $\Delta x, \Delta y, \Delta z$ are fixed, as in the case for a uniform grid, there is a tendency to under-sample interpolation points in regions where the particle spacing is smaller than the grid spacing. In practice, care must be taken to avoid this issue, with a common approach to estimate the density of the 3D cell volume using a Monte Carlo method:

$$\widehat{\Sigma_{i,j}(\boldsymbol{\xi})} = \sum_{k=1}^{N_z} \left\langle \rho_{i,j,k}\widehat{(\boldsymbol{\xi}, z_k)} \right\rangle \Delta z \tag{5}$$

While this helps with under-sampling, it does not alleviate over-sampling in low density regions.

1) Point Location by Walking: In order to interpolate the field value for a query point, $\boldsymbol{q}$, the tetrahedra containing $\boldsymbol{q}$ must be identified. The location of an arbitrary point inside a triangulation is a classic problem in computational geometry. It has been very well studied, with several algorithms providing optimal solutions in the literature [14]. In practice, however, algorithms that 'walk through' the triangulation have been widely adopted for their simplicity and a lack of need for preprocessing and additional storage. Walking algorithms only require an additional adjacency matrix for storing neighbor simplices, i.e., simplices that share a common facet. The general approach begins with choosing an initial simplex in the triangulation and moving towards $\boldsymbol{q}$ by moving to a neighbor simplex in the direction of $\boldsymbol{q}$ until the simplex containing $\boldsymbol{q}$ is identified. A robust technique for walking in three dimensions, Sambridge et al. [15], tests if $\boldsymbol{q}$ is on the interior side of a tetrahedral face by the following test

$$\begin{aligned}[(\boldsymbol{x_i} - \boldsymbol{x_j}) \cdot [(\boldsymbol{x_k} - \boldsymbol{x_j}) \times (\boldsymbol{x_l} - \boldsymbol{x_j})]] \cdot \\ [(\boldsymbol{q} - \boldsymbol{x_j}) \cdot [(\boldsymbol{x_k} - \boldsymbol{x_j}) \times (\boldsymbol{x_l} - \boldsymbol{x_j})]] \geq 0,\end{aligned} \tag{6}$$

moving to the neighbor tetrahedron of a face that fails, and locating the tetrahedron if all faces pass. The walking method is guaranteed to converge for a Delaunay triangulation, but may go into an infinite loop with an arbitrary triangulation; this is because walking moves in the general direction of $\boldsymbol{q}$, which is not always a straight line. The performance of walking can be greatly improved by choosing an initial tetrahedra that is close to $\boldsymbol{q}$, usually done by randomly sampling tetrahedra vertices and selecting the tetrahedron with the vertex that is nearest. For a regular grid, the tetrahedron containing the query point of an adjacent grid cell is clearly a good initial choice, however, the orientation tests from equation 6 must still be computed.

2) Ray-Tetrahedron Intersection: Similar to walking, Mücke et al. [16] proposed the idea of 'marching', where a line, ℓ, is used to traverse the triangulation by identifying tetrahedra that intersect ℓ. The points of ℓ are the query point, $\boldsymbol{q}$, and some initially located vertex $\boldsymbol{p}$, $\ell = \overline{\boldsymbol{pq}}$. Algorithms such as the Möller and Trumbore ray-triangle algorithm [17] are fast and efficient, but usually do not perform well in practice because of floating point round-off error. An algorithm proposed by Platis et al. [18]

using Plücker coordinates has better results, and we use it for the ray-tetrahedra intersection testing in our algorithm. The algorithm tests each face individually by checking the relative orientation of two rays, namely $\overrightarrow{\boldsymbol{pq}}$ and a face edge. For some three dimensional ray, r, determined by a point, $\boldsymbol{x}$, with direction, $\boldsymbol{l}$, the Plücker coordinates representation is a six-dimensional vector of the form

$$\boldsymbol{\pi_r} = \{\boldsymbol{l} : \boldsymbol{l} \times \boldsymbol{x}\} = \{\boldsymbol{u_r} : \boldsymbol{v_r}\}. \tag{7}$$

The relative orientation of two rays, r and s is determined by computing the sign of the permuted inner product:

$$\boldsymbol{\pi_r} \odot \boldsymbol{\pi_s} = \boldsymbol{u_r} \cdot \boldsymbol{v_s} + \boldsymbol{u_s} \cdot \boldsymbol{v_r} \tag{8}$$

The ray orientation test is performed for each edge of a tetrahedra face, but shared edge calculations can be reused. When all the permuted inner products for a face are greater than zero, r enters the face, if all are less than zero, r leaves the face, otherwise one of the degeneracy cases [2] has been encountered. For any intersecting face, the barycentric coordinates of the intersection is computed by

$$w_i = \boldsymbol{\pi_r} \odot \boldsymbol{\pi_{e_i}}, \qquad h_i = \frac{w_i}{\sum_{i=1}^{3} w_i} \tag{9}$$

where $\boldsymbol{\pi_{e_i}}$ is the Plücker vector for an edge. The Cartesian coordinates follow as

$$\boldsymbol{x}_{intersect} = h_0\boldsymbol{x_0} + h_1\boldsymbol{x_1} + h_2\boldsymbol{x_2} \tag{10}$$

where $\boldsymbol{x_0}$, $\boldsymbol{x_1}$, and $\boldsymbol{x_2}$ are the vertices of the face.

IV. Design and Implementation

Our approach follows the principles of data locality and replication to minimize communication and synchronization by creating particle ghost zones. Our load balancing scheme models the computational phases of a small set of test problems at runtime – drawn randomly from the problem domain – to estimate the remaining work and determine an *a priori* work sharing schedule. The distributed-memory algorithm consists of four phases: (1) data partitioning and redistribution, (2) workload modeling, (3) work sharing scheduling, and (4) execution and communication. These phases are described in detail after presenting our surface density kernel.

A. Shared Memory DTFE Kernel

In this subsection, we present the algorithm we designed for computing a surface mass density field defined on a 2D grid using the DTFE method. The algorithm computes each 2D grid value by marching through the triangulation (ray-tetrahedron intersection) along the line-of-sight path of integration, interpolating values from tetrahedra intersecting the 3D projection, ℓ, of the 2D grid cell point, $\boldsymbol{\xi}$.

[2]Degeneracy cases occur when the ray intersects a vertex, an edge, or is co-planar to a face.

1) Optimal Line-of-Sight Interpolation: To determine the optimal points for interpolation along a line of sight, ℓ, we compute the points of intersection using equations 9 and 10 for each tetrahedron intersecting ℓ. Substituting the 3D density $\rho(\boldsymbol{\xi}, z)$ with the interpolation function in equation 1, the integral in equation 3 becomes

$$\widehat{\Sigma_{\mathcal{T}_i}(\boldsymbol{\xi})} = \int_a^b \widehat{\rho(\boldsymbol{x_0})} + \widehat{\nabla\rho}|_{Del} \cdot \left(\begin{bmatrix} \xi_x \\ \xi_y \\ z \end{bmatrix} - \boldsymbol{x_0} \right) dz \tag{11}$$

where $\boldsymbol{\xi}$ is the 2D projection of ℓ, $\mathcal{T}_i$ is a tetrahedron intersecting ℓ, $\boldsymbol{x_0}$ is an arbitrarily chosen vertex of $\mathcal{T}_i$ taken to compute the gradient, and a, b, are the z components to the points of intersection of ℓ and $\mathcal{T}_i$. Solving the integral in equation 11 gives

$$\widehat{\Sigma_{\mathcal{T}_i}(\boldsymbol{\xi})} = \left[\widehat{\rho(\boldsymbol{x_0})} + \widehat{\nabla\rho}|_{Del} \cdot \left(\begin{bmatrix} \xi_x \\ \xi_y \\ a + \frac{b-a}{2} \end{bmatrix} - \boldsymbol{x_0} \right) \right] (b-a) \tag{12}$$

showing that for any tetrahedra intersecting ℓ, equation 3 is solved exactly when using linear interpolation by interpolating at the midpoint of the intersection interval. Equations 11 and 12 are defined on individual tetrahedra, thus, the surface density for any 2D grid cell is then the sum of all surface densities obtained from tetrahedra intersecting ℓ, given by

$$\widehat{\Sigma_{i,j}(\boldsymbol{\xi})} = \sum_{k=1}^{N_{\mathcal{T}}} \widehat{\Sigma_{\mathcal{T}_k}(\boldsymbol{\xi})} \tag{13}$$

where $N_{\mathcal{T}}$ is the number of tetrahedra intersecting ℓ. There is potential for under-sampling in the x and y dimensions, as we described in the Section III, however, a Monte Carlo approximation can also be computed to obtain the mean surface density for the 2D grid cell, with the advantage over 3D grid rendering methods being one fewer degree of freedom in the error of the estimator.

2) Algorithm: Our convention is to integrate along the z dimension to make calculations simpler, however, in principle any arbitrary direction can be chosen by a simple rotation of the triangulation. In order to initiate the march, the algorithm must determine the first tetrahedron intersecting ℓ. This is done by constructing a 2D triangulation from the 3D Delaunay triangulation's convex hull – projecting down hull facets facing the opposite direction of integration,

$$\boldsymbol{n_{hull}} \cdot \hat{\boldsymbol{z}} < 0 \tag{14}$$

where $\boldsymbol{n_{hull}}$ is the surface normal of a hull face, and $\hat{\boldsymbol{z}}$ is the 3D z unit vector. Determining the first tetrahedron intersecting ℓ (Figure 3 line 5) is done by locating its projection, $\boldsymbol{\xi}$, in the 2D triangulation, where any point location method can be used.

With high resolution grids and a large number of particles per volume, it is inevitable that degeneracy cases will be encountered. The $RayTetra$ subroutine computes equation

8 and returns a degeneracy status, Err, along with points of intersection, a, b, and the neighbor of the exit face, $Neighbor$. To resolve degeneracy cases, we perturb ℓ with the subroutine $Perturb$.

Input: Line ℓ, Tetrahedron $\mathcal{T}$, ϵ
Output: Line ℓ
1: $\boldsymbol{\xi} \leftarrow xyProjection(\ell)$
2: $\boldsymbol{v} \leftarrow getRandomVertex(\mathcal{T})$
3: $\boldsymbol{\delta} \leftarrow xyProjection(\boldsymbol{v}) - \boldsymbol{\xi}$
4: **if** $\|\boldsymbol{\delta}\| > \epsilon$ **then**
5: $\quad \boldsymbol{\delta} \leftarrow \boldsymbol{\delta} \div \|\boldsymbol{\delta}\|$
6: $\quad \boldsymbol{\delta} \leftarrow \boldsymbol{\delta}\epsilon$
7: **end if**
8: $\ell \leftarrow \ell + \boldsymbol{\delta}$

Figure 2: $Perturb$

Input: Discrete point set X
Output: 2D grid $\mathbf{F}$
1: $D \leftarrow Delaunay(X)$
2: **for all** i such that $0 \leq i < N_{cell}$ **do**
3: $\quad \mathbf{F}_i \leftarrow 0$
4: $\quad \ell \leftarrow GridLineOfSight(i)$
5: $\quad \mathcal{T} \leftarrow HullTetrahedron(D, \ell)$
6: $\quad$ **while** $\mathcal{T}$ **do**
7: $\quad\quad (Err, a, b, Neighbor) \leftarrow RayTetra(\mathcal{T}, \ell)$
8: $\quad\quad$ **if** Err **then**
9: $\quad\quad\quad Perturb(\ell, \mathcal{T}, \epsilon)$
10: $\quad\quad$ **else**
11: $\quad\quad\quad \mathbf{F}_i = \mathbf{F}_i + Interpolate(\mathcal{T}, \ell, a, b)$
12: $\quad\quad\quad \mathcal{T} = Neighbor$
13: $\quad\quad$ **end if**
14: $\quad$ **end while**
15: **end for**

Figure 3: $Kernel$

3) Complexity Analysis: Constructing a Delaunay triangulation takes $\mathcal{O}(N^2)$ operations and contains $\mathcal{O}(N^2)$ tetrahedra in 3D. For each ${N_g}^2$ 2D grid cell, a march through the triangulation is needed, making the worst case complexity $\mathcal{O}(N^2 + {N_g}^2 N^2)$ and average case complexity $\mathcal{O}(N^2 + {N_g}^2 (N^2)^{1/3})$, assuming a cube sub-volume.

B. Data Partitioning and Redistribution

Data partitioning is done using a spatial volume decomposition where each sub-volume is of equal size and not guaranteed to have an equal number of particles. The particle imbalance is strongly dependent on the size of the sub-volumes – with more imbalance in highly clustered situations (typically occurring at later simulation times) and smaller sub-volume scales. Abiding by the principles of data locality and replication, we create particle ghost zones large enough such that any surface density field subproblem within the active region of a process can be completed without communication to any other process. More formally, if a surface density field, $\mathbf{F}$, has physical length $l_{\mathbf{F}}$, ghost zones will contain duplicate particles beyond the sub-volume boundaries within a distance of $\frac{l_{\mathbf{F}}}{2}$.

The input particle datasets we used were generated from N-body simulations run on a supercomputer using a greater number of processes. Data was written to several files containing offsets within each file for an individual process's particles. The simulation implements an equal size sub-volume spatial decomposition, thus, on disk the data block written by a process represents a contiguous sub-volume. We use MPI-IO to perform a parallel read of the data using an arbitrary block assignment and spatially redistribute the particles. Finally, all processes perform a neighbor-to-neighbor exchange to fill the ghost zones. The set of locations for computing each surface density field is generally much smaller, which can be read by a single process to perform a broadcast; each process discards locations outside of its local sub-volume.

C. Modeling Workload

We assume all surface density fields to be of the same size and resolution, which is not unreasonable for many types of analysis. This allows for the input that defines where to compute the fields to be specified as a set of points, with additional parameters for the physical length and grid resolution. Input points for computing density fields within the sub-volume of a process are considered its local work items. All processes estimate the total amount of time needed to complete their local work by performing the following steps:

1) Count the number of particles, n_i, needed to complete each local work item.
2) Time the execution of a random local work item, t_r.
3) Exchange results, (n_r, t_r), with all processes.
4) Fit the global timing data to a predictive model, f_{pre}.
5) Estimate the remaining work by $f_{pre}(n_i)$

1) Counting particles: The modeling phase begins with each process counting the number of particles, n_i, needed for computing each of its surface density fields, $\mathbf{F}_i$. This is done by centering a cube, with dimensions specified by an input parameter, on the input point representing the field location. The number of particles in the cube is n_i.

2) Timing: A randomly chosen field is computed by each process, recording the triangulation time, t_{del}, and interpolation time, t_{interp}.

3) Data Exchange: The result of each process's test problem timing, $(n_r, t_{del,r}, t_{interp,r})$, is shared with all processes. This is implemented with MPI using the `MPI_Allgather` function providing implicit synchronization.

4) Fitting: Each process independently fits two models for computing the Delaunay triangulation and grid interpolation, both as a function of n_i. The Delaunay model is simply taken from the *quickhull* algorithm average case complexity analysis,

$$f_{tri}(n) = c\ n\ log_2(n) \tag{15}$$

where c is the parameter fit for capturing environmental characteristics. Fitting is performed using ordinary least squares.

$$c = (\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{t_{del}} \tag{16}$$

The interpolation model is a power law function,

$$f_{interp}(n) = \alpha\ n^{\beta} \tag{17}$$

To fit the function we use the non-linear least squares Gauss-Newton method. The initial guess is taken from linearly fitting the log of the data and the log of the function.

D. Work Sharing

Once the timing estimates for each work item have been completed, the total amount of local work time, t_i, can be computed and shared with all processes, implemented with `MPI_Allgather`. Each process computes $\langle \mathbf{t} \rangle$ to determine the senders and receivers in the work sharing schedule. We construct a list of pairs to hold the process ID and total time for all processes. $P_i = (id_i, t_i)$

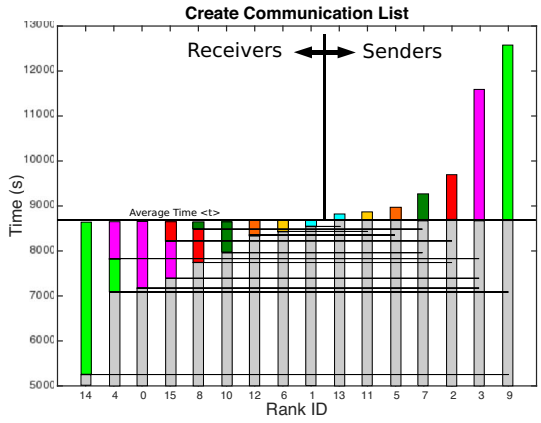


Figure 4: Each process independently computes its communication list; determining if it is a sender or receiver based on the average total time of all processes.

After executing the $CreateCommunicationList$ routine, the $RecvList$ contains information on which processes are going to send work, and in what order messages will be received. The $SendList$ contains information on which process, when, and how much work a sender process can send to a receiver processes. Senders then need to determine at what point in the computation of their local work they should pause and send to a free receiver in their list. To

Input: Process data P
Output: SendList, RecvList

```
1:  myProcID = getMyProcID();
2:  lr = −1
3:  Ps ← SortByTimeDescending(P)
4:  for all t in Ps.t do
5:    if t > ⟨t⟩ then
6:      lr = lr + 1
7:      break
8:    end if
9:  end for
10: cr = size(Ps) − 1
11: for all i such that 0 ≤ i < lr do
12:   while cr ≥ lr and Ps[i].t > ⟨t⟩ do
13:     if (Ps[i].t − ⟨t⟩) > (⟨t⟩ − Ps[cr].t) then
14:       if myProcID = Ps[i].id then
15:         SendList.add(Ps[cr], ⟨t⟩ − Ps[cr].t)
16:       else if myProcID = Ps[cr].id then
17:         RecvList.add(Ps[i].id)
18:       end if
19:       Ps[i].t = Ps[i].t − (⟨t⟩ − Ps[cr].t)
20:       Ps[cr].t = ⟨t⟩
21:       cr = cr − 1
22:     else
23:       if myProcID = Ps[i].id then
24:         SendList.add(Ps[cr], Ps[i] − ⟨t⟩
25:       else if myProcID = Ps[cr].id then
26:         RecvList.add(Ps[i].id)
27:       end if
28:       Ps[cr].t = Ps[cr].t + Ps[i].t − ⟨t⟩
29:       Ps[i].t = ⟨t⟩
30:     end if
31:   end while
32: end for
```

Figure 5: $CreateCommunicationList$

do this, senders sort their $SendList$ by send time ($Ps.t$) in ascending order. Taking the time difference between subsequent list items, senders determine which local work items can be computed to fill time between sending work sharing messages. We view the time to fill between messages as work bins, where the objective is to fill them with local work items. Taking this a step further, the amount of work to send to a receiver can also be considered a work bin. We combine the two work bin lists to simultaneously solve both problems using a greedy first-fit approximation [19] to the variable bin packing problem, where we sort the work items in descending order and sort the bins in ascending order. The result is a permutation of the local work items that minimize delays in communication. Note that this work sharing scheme is greedy in minimizing the overall amount of communication, that is, the senders with the most work to share send to receivers with the largest ability to receive.

E. Execution and Communication

Receivers simply execute all their local work and listen for a message from the next sender in their list. This is implemented by calling `MPI_Recv`. When a work sharing message is received, the process receives a copy of the sender's particle set and density field positions. The receiver process then executes the new local work items. These steps are repeated until there are no more receive messages in the list. Senders execute their local work items and call `MPI_Send` after iterations determined by the optimization algorithm.

V. Experimental Results

The DTFE surface density grid rendering kernel described in this paper is implemented in C using OpenMP to parallelize the loop over the individual grid cell computations. Computing multiple fields is implemented in C++ and parallelized using MPI for all interprocess communication. The kernel comparison with the DTFE software was performed on a workstation with two Intel Xeon Processors (E5-2620; 15M Cache, 2.00 GHz), 12 physical cores supporting 2 hardware threads per core. The distributed-memory experiment was carried out on Cooley, an InfiniBand data analysis/visualization cluster at Argonne National Laboratory with two six-core 2.4GHz Intel Haswell processors and 384GB memory/node. The distributed-memory comparison with the TESS density estimator [4] was performed on Cooley without OpenMP support. Cooley was also used for our first shared/distributed-memory experiments on a high resolution N-body simulation using 6 OpenMP threads/process. The second shared/distributed-memory experiment was carried out on Mira, an IBM Blue Gene/Q supercomputer with a PowerPC A2 1600 MHz processor containing 16 cores and 16 GB memory/node using 4 OpenMP threads/process. For creating Delaunay triangulations we use the Qhull software library [12]. In all our experiments, triangulations are constructed without any threading as this was not the focus of this work. Our code makes no special use of triangulation library data structures so substituting the triangulation library can be performed with little refactoring.

1) Shared-Memory Comparison: For an empirical comparison with walking based software, we compared our implementation with the publicly available DTFE software [9] using the provided demo dataset from a publicly available N-body simulation software called Gadget [20]. The results are displayed in Figure 6 for a 1024^3 grid. The dataset has 650,466 particles in a volume of $(100\ \mathrm{Mpc}/h)^3$. The timing comparison is restricted only to the time it takes to interpolate the grid and not for generating the triangulation. The software task is to compute a field defined on a regular grid using a single point for computing the density at each grid cell. For clarity, our algorithm did not run using dynamic grid spacing, but rather an equally spaced grid in all three dimensions. In this way, both approaches are locating and interpolating exactly the same number of grid cells. Overall, as shown in Figure 6, our method is approximately an order of magnitude faster and threads have a much more evenly distributed execution time. Both codes were run using 24 threads. The difference in overall performance is attributed to the pre-factor in time complexity of the algorithms, which in this case is $\mathcal{O}(N^2+{N_g}^3)$. The difference comes from how the two methods perform point location for determining the tetrahedra for interpolation; the DTFE public software uses the 'walking' method as opposed to our 'marching' method.

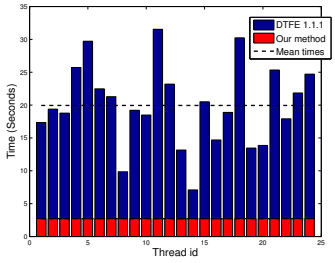
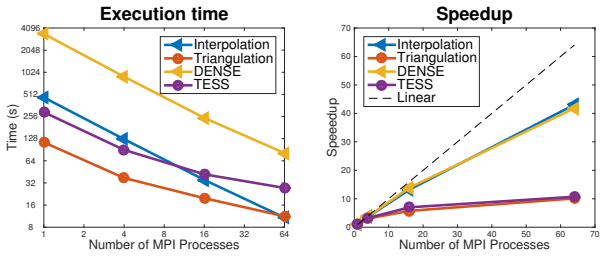


Figure 6: Comparison of the thread timing from the publicly available DTFE software [9] and using the kernel algorithm described here for a single 1024^3 grid.

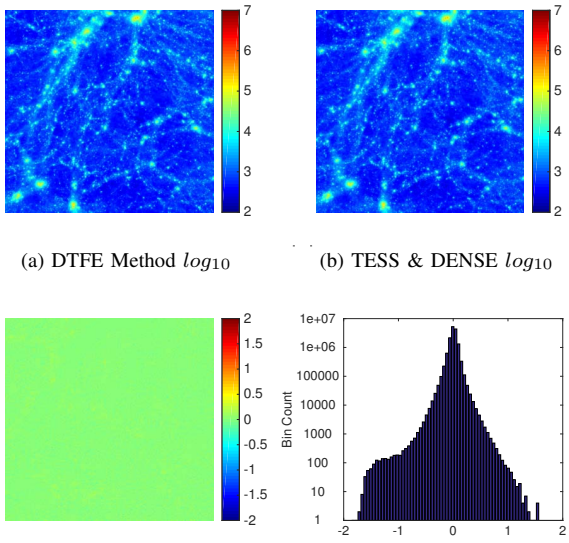


(a) (b)

Figure 7: Execution and speedups for TESS, DENS, and the corresponding steps of our algorithm.

2) Distributed-Memory Comparison: The next experiment is another comparison with the distributed-memory TESS Density Estimator [4], a freely available C++ MPI software for computing surface density. The dataset for this experiment is a 32 $\mathrm{Mpc}\ h^{-1}$ sub-volume with 1,711,563 particles taken from the large 3200^3 particle simulation we use in this paper. The TESS Density Estimator, runs in two stages: constructing a Voronoi tessellation, TESS, followed by estimating density at grid points covered by Voronoi cells, DENSE. Figure 7 presents the execution and speedup of the corresponding stages of TESS and DENSE with our algorithm. To compare we ran our implementation in *multiple-process-single-thread* mode and decomposed the single large grid into multiple sub-grids. The results show $\sim 8\times$ improvement in execution time. In this experiment

we generated a 4096×4096 surface density grid from the entire dataset. Shown in Figure 8 are the outputs from single process runs of the codes.



(a) DTFE Method log_{10} (b) TESS & DENSE log_{10}

(c) Ratio Map (d) Ratio Histogram

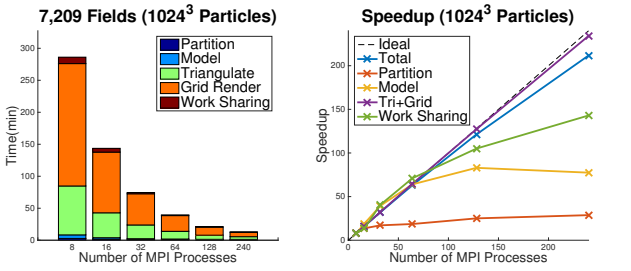
Figure 8: The 4096×4096 grids generated by our DTFE implementation and TESS/DENSE. We show a ratio map of the fields and the corresponding histogram to understand any bias between the estimators. The ratio map is calculated as $log_{10}(DTFE/DENSE)$.

From Figure 8c, we see the density maps produced by the two different methods are mostly in agreement. The small differences are largely attributed to the different interpolation schemes. The bump from Figure 8d is a result of how the two methods differ in dealing with the asymmetric bias – due to particle noise [21] – inherent to inverse volume based density estimators.

3) Shared/Distributed-Memory with Load Balancing: Our shared/distributed-memory experiments were performed on N-body simulation datasets generated by HACC (Hardware/Hybrid Accelerated Cosmology Code) [22] [23]. We use two science quality datasets to demonstrate the overall performance and load balancing capability of our method.

The first dataset is a simulation called *Planck* and has 1024^3 particles in a 256 Mpc h^{-1} volume. We perform two experiments on this dataset where we computed thousands of density fields in different spatial configurations.

Galaxy-Galaxy Lensing Experiment: The next set of figures were generated by computing 7,209 density fields centered on the positions of simulated galaxies. Galaxy positions are assigned to the most dense regions in the simulation volume by a model for the galaxy distribution. This configuration of fields is particularly challenging computationally, not only because fields are required in the most highly concentrated particle regions, but also because the decomposition sub-volumes become more imbalanced the more we decompose to achieve higher performance.



(a) (b)

Figure 9: Speedup on up to 240 processes showing the overall performance scaling to be near linear until 64 processes, at which point the cost of partitioning and the overhead of modeling and work sharing become apparent. This experiment computes density fields in the most highly concentrated regions of particles.

Figure 9 presents the speedup and timing of our parallel algorithm. To illustrate reasons for the performance drop off at higher number of processes, we include the speedup of all phases. We can see that as the number of processes increases, the partitioning phase is nearly flat, showing we become bound by the IO operation. Similarly the overhead of modeling becomes apparent since it is nearly constant due to each process computing a random test problem calculation. The early improvements are due to estimating the time for fewer local work items, but this flattens as computing the test problem dominates the phase. Figure 10 demonstrates how timing becomes more imbalanced as sub-volumes become smaller due to particles becoming more imbalanced. The problem is two-fold since there are more work items in sub-volumes with higher particle concentrations, and the work items themselves are more costly.

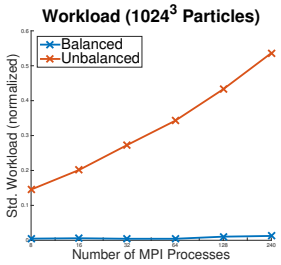
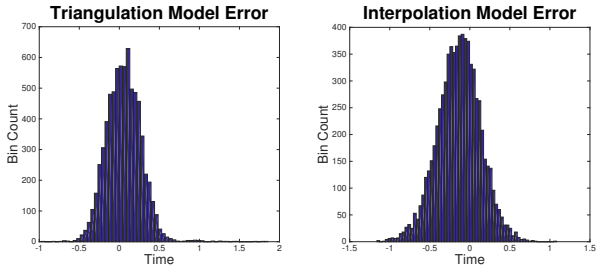


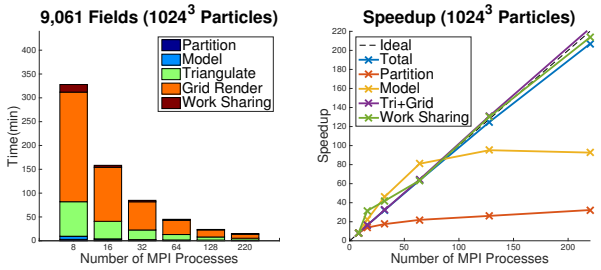
Figure 10: Workload imbalance measured by calculating the standard deviation in compute time (unbalanced estimated by model prediction) showing the trend for more imbalance as sub-volumes become smaller. Balanced compute time was obtained from execution wall time.

Model Prediction: We characterize the modeling predictive accuracy in Figure 11 to support our speedup and imbalance claims as shown in Figure 10. The error distributions are symmetric with mean centered near zero.



(a) (b)

Figure 11: Histograms of the error made by work prediction models fit with 7,209 test samples compared with actual wall timing from the galaxy-galaxy lensing experiment. Triangulation model $\mu = 5.0925$ (left) Interpolation model $\mu = 12.1209$ (right)

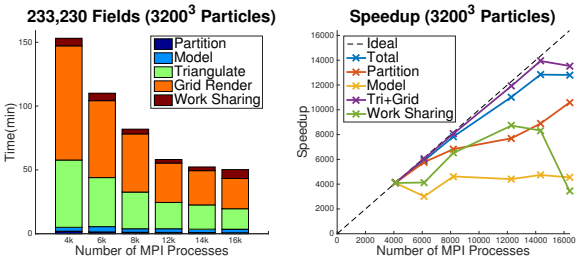


(a) (b)

Figure 12: Speedup on up to 220 processes showing the overall performance scaling to be near linear with only small deviation. This experiment computes density fields in both high and low regions of particle concentration.

Multiplane Lensing Experiment: We repeated the experiment on the same dataset using a different configuration for the density fields. Here we are creating density fields along an observer's entire line of sight in the complete volume. We construct 700 such line-of-sight field configurations for a total of 9,061 density fields. These fields are a mixture of high and low density sub-volumes. Interestingly, we see better overall scalability performance in this experiment. Looking at the work sharing speedup in Figure 12, we conclude that there are more small work items to complete and the variable bin size optimizer can be more efficient, resulting in less waiting time for blocking MPI messages to complete. The explanation is rooted in the observation that more work items were performed while work sharing message sizes remained the same.

Large Scale Experiment: The second dataset is a simulation called *MiraU*, a large volume $(1491Mpc\ h^{-1})^3$, 3200^2 ($\sim$32 billion) particle simulation where we computed 1024×1024 surface density grids centered on the 233,230 most massive objects found by a density based clustering algorithm. This is similar to the galaxy-galaxy lensing experiment we performed on the *Plank* dataset, but simply at a much larger scale.



(a) (b)

Figure 13: Speedup on up to 16,384 processes showing the overall performance scaling to be near linear until 16,384 processes, where a few degenerate point configurations make the model predicted execution time inaccurate. This experiment computes density fields in the most highly concentrated regions of particles.

From Figure 13 we can see that the speedup is near linear until 16,384 processes, then, the computing speedup drops significantly. A closer look indicated that a small number of degenerate point configurations on a few MPI processes made the model predicted execution time inaccurate and delayed sending work to idle processes. This is clearly supported by the corresponding drop in work sharing speedup and points out a drawback to our *a priori* model for work sharing.

VI. Conclusion

Performing high throughput analysis tasks that involve computing high quality surface density fields in state of the art N-body cosmological simulations poses computational challenges primarily related to load distribution. We address the issue by proposing a fast and accurate grid rendering kernel for the DTFE method that is readily parallelizable and computationally efficient, achieving well balanced multithread performance when compared to existing software packages. We leverage the computational predictability of the kernel to achieve well balanced workloads in our distributed memory framework by accurately modeling computation to determining an *a priori* work sharing schedule. Future work will involve kernel optimizations for emerging hardware such as many-core CPUs and extending the work sharing algorithm to a more general framework.

ACKNOWLEDGMENT

We thank Katrin Heitmann for running the HACC simulations to provide the N-body data needed for this work. Argonne National Laboratory's work was supported under U.S. DOE contract DE-AC02-06CH11357. Partial support was provided by the SciDAC program funded by the U.S. DOE, Office of Science, jointly by Advanced Scientific Computing Research and High Energy Physics and by an Argonne LDRD award. This research used resources of the ALCF, supported by DOE/SC under contract DE-AC02-06CH11357, NERSC, supported by DOE/SC under Contract No. DE-AC02-05CH11231, and of the OLCF, supported by DOE/SC under contract DE-AC05-00OR225. This work is supported in part by the following grants: NSF awards CCF-1029166, IIS-1343639, CCF-1409601; DOE awards DE-SC0007456, DE-SC0014330; AFOSR award FA9550-12-1-0458; NIST award 70NANB14H012; DARPA award N66001-15-C-4036.

REFERENCES

[1] F. Bernardeau and R. van de Weygaert, "A new method for accurate estimation of velocity field statistics," *Monthly Notices of the Royal Astronomical Society*, vol. 279, no. 2, pp. 693–711, 1996.

[2] M. Bradač, P. Schneider, M. Lombardi, M. Steinmetz, L. Koopmans, and J. F. Navarro, "The signature of substructure on gravitational lensing in the Λcdm cosmological model," *Astronomy & Astrophysics*, vol. 423, no. 3, pp. 797–809, 2004.

[3] R. S. Ellis, "Gravitational lensing: a unique probe of dark matter and dark energy," *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, vol. 368, no. 1914, pp. 967–987, 2010.

[4] T. Peterka, H. Croubois, N. Li, S. Rangel, and F. Cappello, "Self-adaptive density estimation of particle data," *Submitted to SIAM Journal on Scientific Computing SISC Special Section on CSE15: Software and Big Data*, 2015.

[5] P. Schneider, "Gravitational lensing statistics," in *Gravitational Lenses*. Springer, 1992, pp. 196–208.

[6] S. Seitz, P. Schneider, and J. Ehlers, "Light propagation in arbitrary spacetimes and the gravitational lens approximation," *Classical and Quantum Gravity*, vol. 11, no. 9, p. 2345, 1994.

[7] N. Li, M. D. Gladders, E. M. Rangel, M. K. Florian, L. E. Bleem, K. Heitmann, S. Habib, and P. Fasel, "Pics: Simulations of strong gravitational lensing in galaxy clusters," *arXiv preprint arXiv:1511.03673*, 2015.

[8] M. Petkova, R. B. Metcalf, and C. Giocoli, "glamer–ii. multiple-plane gravitational lensing," *Monthly Notices of the Royal Astronomical Society*, vol. 445, no. 2, pp. 1954–1966, 2014.

[9] M. C. Cautun and R. van de Weygaert, "The dtfe public software-the delaunay tessellation field estimator code," *arXiv preprint arXiv:1105.0370*, 2011.

[10] W. E. Schaap, "The delaunay tessellation field estimator," Ph.D. dissertation, Ph. D. thesis, Groningen University, 2007.

[11] "Cgal, computational geometry algorithms library, http://www.cgal.org."

[12] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa, "The quickhull algorithm for convex hulls," *ACM Transactions on Mathematical Software (TOMS)*, vol. 22, no. 4, pp. 469–483, 1996.

[13] T. Peterka, D. Morozov, and C. Phillips, "High-performance computation of distributed-memory parallel 3d voronoi and delaunay tessellation," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2014, pp. 997–1007.

[14] J. O'Rourke and J. E. Goodman, *Handbook of discrete and computational geometry*. CRC Press, 2004.

[15] M. Sambridge, J. Braun, and H. McQueen, "Geophysical parametrization and interpolation of irregular data using natural neighbours," *Geophysical Journal International*, vol. 122, no. 3, pp. 837–857, 1995.

[16] E. P. Mücke, I. Saias, and B. Zhu, "Fast randomized point location without preprocessing in two-and three-dimensional delaunay triangulations," in *Proceedings of the twelfth annual symposium on Computational geometry*. ACM, 1996, pp. 274–283.

[17] T. Möller and B. Trumbore, "Fast, minimum storage ray/triangle intersection," in *ACM SIGGRAPH 2005 Courses*. ACM, 2005, p. 7.

[18] N. Platis and T. Theoharis, "Fast ray-tetrahedron intersection using plucker coordinates," *Journal of graphics tools*, vol. 8, no. 4, pp. 37–48, 2003.

[19] J. Kang and S. Park, "Algorithms for the variable sized bin packing problem," *European Journal of Operational Research*, vol. 147, no. 2, pp. 365–372, 2003.

[20] V. Springel, "The cosmological simulation code gadget-2," *Monthly Notices of the Royal Astronomical Society*, vol. 364, no. 4, pp. 1105–1134, 2005.

[21] S. Rau, S. Vegetti, and S. D. White, "The effect of particle noise in n-body simulations of gravitational lensing," *Monthly Notices of the Royal Astronomical Society*, vol. 430, no. 3, pp. 2232–2248, 2013.

[22] S. Habib, V. Morozov, H. Finkel, A. Pope, K. Heitmann, K. Kumaran, T. Peterka, J. Insley, D. Daniel, P. Fasel *et al.*, "The universe at extreme scale: multi-petaflop sky simulation on the bg/q," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012, p. 4.

[23] S. Habib, A. Pope, H. Finkel, N. Frontiere, K. Heitmann, D. Daniel, P. Fasel, V. Morozov, G. Zagaris, T. Peterka *et al.*, "Hacc: Simulating sky surveys on state-of-the-art supercomputing architectures," *New Astronomy*, vol. 42, pp. 49–65, 2016.