High-Performance Computation of Distributed-Memory Parallel 3D Voronoi and Delaunay Tessellation

Tom Peterka
Argonne National Laboratory
9700 S. Cass Ave.
Argonne IL 60439 USA
tpeterka@mcs.anl.gov

Dmitriy Morozov
Lawrence Berkeley National Laboratory
1 Cyclotron Rd.
Berkeley CA 94720 USA
dmitriy@mrzv.org

Carolyn Phillips
Argonne National Laboratory
9700 S. Cass Ave.
Argonne, IL 60439 USA
cphillips@anl.gov

***Abstract*—Computing a Voronoi or Delaunay tessellation from a set of points is a core part of the analysis of many simulated and measured datasets: N-body simulations, molecular dynamics codes, and LIDAR point clouds are just a few examples. Such computational geometry methods are common in data analysis and visualization; but as the scale of simulations and observations surpasses billions of particles, the existing serial and shared-memory algorithms no longer suffice. A distributed-memory scalable parallel algorithm is the only feasible approach. The primary contribution of this paper is a new parallel Delaunay and Voronoi tessellation algorithm that automatically determines which neighbor points need to be exchanged among the subdomains of a spatial decomposition. Other contributions include periodic and wall boundary conditions, comparison of our method using two popular serial libraries, and application to numerous science datasets.**

***Keywords*—computational geometry; Voronoi, Delaunay tessellation**

I. Introduction

Space-filling tessellations are fundamental geometric constructs for converting particle data into a continuous manifold that can be interpolated, differentiated, and integrated. Applications of Voronoi and Delaunay tessellations abound: point-cloud surface reconstructions, geographical information systems, finite element methods, and particle-based simulations all require converting discrete point data into a continuous field during some stage of their computation, analysis, or visualization. The explosion in data rates in all these applications requires distributed-memory parallel tessellations; yet, few scalable algorithms exist.

In this paper, we demonstrate several marked improvements over the prior state of the art. Our new parallel algorithm is scalable in time and space. It features both parallel Voronoi and Delaunay tessellations, automatic neighbor particle determination, periodic boundary conditions, truncated wall boundary conditions, a comparatively lightweight data model, and parallel netCDF output format. We implement our parallel solution on top of two popular serial computational geometry libraries and compare performance.

Our parallel algorithm exploits the symmetric influence of neighbor points across subdomain boundaries. That is, an input point in a neighboring subdomain (called a block) affects the tessellation in a local block if and only if there is an input point in the local block that affects the tessellation in the neighboring block. Thus, neighbor point exchanges can be determined locally, enabling fast communication. Our algorithm determines these exchanges automatically in several tessellation stages interspersed with neighborhood communication. Each stage communicates fewer points than the previous stage; the first exchange features a heuristic neighbor determination in order to reduce communication, while the second stage guarantees that all required points have been sent to all neighbors.

We evaluate our algorithm with synthetic points, dark matter tracer particles in two cosmology simulations, polymers in two molecular dynamics simulations, and subatomic particles in a plasma fusion simulation. In addition to testing the relative impact of two underlying serial libraries on the performance and memory footprint of our algorithm, we evaluate strong and weak scaling of our parallel algorithm with tests of up to 2048^3 particles and 128K processes on IBM Blue Gene/Q and Cray XC30 supercomputers.

II. Background and Related Work

Computational geometry terms used throughout the paper are defined below followed by a review of serial and parallel tessellation algorithms and other software on which we depend for this work.

A. Voronoi and Delaunay Definitions

Figure 1 defines the geometric terms used throughout this paper. Raw data (for example, particles in an N-body simulation) are represented as x, y, z *input points*; several such input points are drawn as black dots in the diagram. A Voronoi tessellation consisting of polygons (one such polygon highlighted in red) is drawn in the figure along with its dual Delaunay tessellation represented by triangles (one such triangle highlighted in blue). The diagram is drawn in 2D for simplicity, but all our work is in 3D: polygons in the figure correspond to polyhedra and triangles to tetrahedra.

SC14, November 16-21, 2014, New Orleans
978-1-4799-5500-8/14/$31.00 ©2014 IEEE

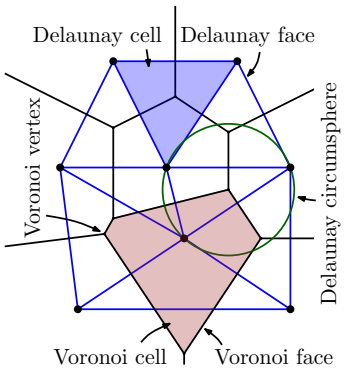

Fig. 1. Voronoi and Delaunay tessellation nomenclature. Input points are black dots. Voronoi tessellation is illustrated with black edges. Delaunay tessellation is in blue. A Voronoi cell is highlighted with a red shaded polygon. A Delaunay cell is highlighted with a blue shaded triangle. An empty Delaunay circumsphere is shown in green.

The Voronoi tessellation of an input point set is a decomposition of the ambient space into a set of convex polyhedra called *Voronoi cells*. Each Voronoi cell V_i is associated with one input point s_i, called the *site* of the cell. Given a set of input points, each Voronoi cell V_i is the convex polyhedron containing the region of the ambient space no farther from the site of the cell s_i than from any other site s_j. Formally,

$$V_i = \left\{ x \in \mathbb{R}^3 \mid d(x,s_i) \leq d(x,s_j) \ \forall \ j \right\}, \tag{1}$$

where $d(x,s_i)$ is the distance between a location in the ambient space x and the site s_i. Voronoi cells can be *finite* or *infinite*. Two Voronoi cells in the center of Figure 1 are finite; the remaining cells around the periphery are infinite. A Voronoi cell is infinite if and only if its site lies on the convex hull of the input points. Two neighboring Voronoi cells V_i and V_j intersect along a common two-dimensional *Voronoi face*, which is a subset of the plane of points equidistant from the two sites, s_i and s_j, $\{x \in \mathbb{R}^3 \mid d(x,s_i) = d(x,s_j)\}$. In 3D, Voronoi faces are polygons.

The Delaunay tessellation is the dual of the Voronoi tessellation: its vertices are the input points. In 3D, its faces (also called facets) are triangles. If two Voronoi cells intersect in a common face, then there is a dual *Delaunay edge*. Equivalently, a Delaunay edge connects sites s_i and s_j if and only if there exists a point $x \in \mathbb{R}^3$ that is closer to s_i and s_j than to any other site and equidistant from both. Similarly, the intersections of three and four Voronoi cells form a Delaunay triangle and tetrahedron, respectively. It follows immediately from the definition of the Voronoi cells that the *circumsphere* of a Delaunay tetrahedron (the unique sphere that passes through the four vertices of the tetrahedron) is centered on a Voronoi vertex (the dual of the tetrahedron) and contains no input points in its interior. One such circumsphere is shown in Figure 1.

Throughout the paper we assume that the input points lie in *general position*, meaning no five of them (in 3D) lie on the same sphere. In practice, this assumption is often violated (consider, for example, points on the integer lattice). It is nonetheless justified because the serial software we rely on has sophisticated ways of simulating a symbolic perturbation of the input. Surveying such work is outside the scope of this paper, but we refer the interested reader to an article by Seidel [1].

B. Computational Geometry Algorithms

Voronoi and Delaunay tessellations are constructed by partitioning the space into cells according to the positions of the input points. The serial algorithms for such a computation are classical topics in computational geometry. Any book on the subject [2], [3] covers them extensively. We recommend the latter reference for a particularly succinct treatment of all the topics relevant to this paper. In 2D, Lawson's classical algorithm [4] identifies edges that violate the Delaunay condition and flips them until no such edges remain. In the worst case it performs $O(n^2)$ flips, and it does not extend to higher dimensions.

Instead, the higher-dimensional algorithms for constructing Delaunay tessellations work incrementally. They insert new points one by one in random order and update the tessellation, either by flipping Delaunay faces or identifying the full set of Delaunay cells that needs to be replaced. Notable results on randomized incremental construction include those of Clarkson and Shor [5] and Guibas et al. [6]. The latter algorithm is extended to higher dimensions by Edelsbrunner and Shah [7], who give a randomized algorithm with expected running time of $O\left(n \log n + n^{\lceil d/2 \rceil}\right)$, where n is the number of input points, and d is the dimension. All these algorithms identify the standard reduction of the problem of computing Delaunay tessellation in dimension d to the computation of the convex hull in dimension $d+1$, by lifting the input points onto a paraboloid.

The Quickhull algorithm [8], which underpins one of the serial libraries we use in this paper, processes the input points by picking the farthest point from the already processed set. Other than this change it performs the same incremental operations as the other algorithms. The authors report empirical speedups, although not for the kinds of point sets that arise from the reduction of the Delaunay problem.

Previous distributed-memory parallel computational geometry algorithms date back to the 1990s and are either 2D or for smaller problem and system sizes than encountered today in high-performance computing (HPC). Miller and Stout [9] examined parallel algorithms for a convex hull of 2D points. Jeong [10] derived a parallel 2D Voronoi diagram algorithm, and Dehne et al. [11] presented a parallel 3D convex hull algorithm. The work of Jeong and Dehne et al. was theoretical; neither paper included performance measurements on an actual machine. Blelloch et al. [12] generated a 2D Delaunay triangulation for various data distributions on 8 processes. Cignoni et al. [13] presented two parallel algorithms for 3D Delaunay tessellation for 64 processes.

The largest example to date of parallel tessellations is that of Peterka et al. [14]. They generated a Voronoi tessellation of 1024^3 dark matter tracer particles in situ with a cosmology simulation running on 16K MPI processes. Their parallelization, built on the Qhull serial implementation of the Quickhull algorithm, calculated only the Voronoi and not the Delaunay tessellation. Moreover, the single-stage parallel algorithm required the user to provide the width of the ghost region over which to exchange particles. This requirement limited its usability to specific applications where the width of the ghost region was known in advance; hence, the authors restricted their evaluation to a cosmology simulation where past experi-

ence dictated the appropriate ghost size. For general-purpose usage, estimating an accurate ghost region size is difficult, and errors in this estimate have direct consequences: overestimating causes unnecessary communication and memory usage, while underestimating leads to an erroneous tessellation.

C. Dependencies

DIY [15], [16] is our data-parallel programming library. Built atop MPI [17], [18], DIY provides configurable data partitioning and scalable data exchange in a distributed-memory HPC environment. DIY is initialized with information from an input data model about its block decomposition and neighborhood connectivity. In this paper, we adopt DIY's terminology and define a *block* as the fundamental unit of domain decomposition and local work. It is a hexahedral region of space containing a subset of the input points. An MPI process may own more than one block. A *neighborhood* is the union of a block with its immediate spatially adjacent blocks that share a face, edge, or corner. Therefore, in 3D, a (1-)neighborhood is a convex set of up to 27 blocks. In this project, we rely on DIY's neighborhood exchange algorithm; in particular on its ability to incrementally enqueue input points to neighbors within a distance given by the tetrahedron circumsphere radius (Section III-A) and then to exchange the enqueued points with those neighbors. DIY can optionally include neighbors on opposite sides of the domain when periodic boundary conditions are selected.

For the serial computational geometry engine, we build with either the Qhull or CGAL library. Qhull[1] is an open-source implementation of the Quickhull algorithm. It handles high-dimensional input points and is robust to floating-point roundoff errors. Matlab, R, Octave, and Mathematica all use Qhull for their computational geometry functions. CGAL (Computational Geometry Algorithms Library) [19] is an alternative implementation that calculates the Delaunay tessellation, which the user can convert to the dual in order to produce the Voronoi tessellation. Qhull is self-contained, whereas CGAL requires Boost[2] and GNU numerical libraries.[3]

From a programmability standpoint, CGAL[4] [19]–[21] is easier to work with than Qhull. Qhull is an older library with a C interface and no documentation for the library API; hence, determining how to access different geometric entities required reading its source code. In comparison, CGAL has a modern C++ interface with a documented API. For either library, rather than building our code atop the native data of the geometry engine, we choose to copy the geometry data into our own structures and then delete the geometry engine's copy. This allows us to experiment with different geometry libraries and more easily interface with other libraries such as the Parallel NetCDF (PnetCDF) I/O library.

PnetCDF [5] [22] is a high-level I/O library for reading and writing a self-describing portable file in Unidata's NetCDF format.[6] A PnetCDF file contains one or more multidimensional arrays. Through a C interface, one first declares variables, dimensions, and attributes in "define" mode; individual arrays are then written in "data access" mode. Both PnetCDF and HDF5 [23], another popular high-level scientific data library, derive their parallel performance from MPI-I/O [24].

[1] www.qhull.org
[2] www.boost.org
[3] gmplib.org, www.mpfr.org
[4] www.cgal.org
[5] www.mcs.anl.gov/parallel-netcdf
[6] www.unidata.ucar.edu/software/netcdf/

III. Parallel Algorithm

Our approach is to develop a correct and efficient distributed-memory parallel algorithm and implement it on top of mature, high-quality serial computational geometry libraries that already exist. Our parallel algorithm is modular and independent of the particular serial library used; essentially, the serial tessellation is a black box. The choice of black box will have different performance implications on the parallel algorithm; we compare the performance when using two popular serial geometry libraries in Section V-A. The focus of this section, however, is on the parallel algorithm irrespective of the underlying serial library.

Our parallel algorithm is initialized with information from an input data model that includes a spatial decomposition of the input points. Thus, each block contains a defined region of space and all the points inside that space. Each block calculates a local tessellation based on its subset of points and owns that part of the global tessellation. In order to generate the correct global tessellation, points are exchanged between neighboring blocks, and the local tessellations are adjusted. The overall steps are listed in Algorithm 1.

Algorithm 1 Parallel Tessellation

1: Compute initial local tessellation
2: Exchange initial neighbor points
3: Compute augmented local tessellation
4: Exchange remaining neighbor points
5: Compute final local tessellation
6: Write tessellation to storage

The local tessellation is computed three separate times in steps 1, 3, and 5 of Algorithm 1, each time with more points as a result of the interspersed neighbor point exchanges in steps 2 and 4. Only the Delaunay tessellation is computed; the dual Voronoi tessellation is generated as needed from the Delaunay as explained in Section IV. Even though both Qhull and CGAL use an underlying incremental algorithm, to the best of our knowledge, Qhull does not expose the ability to add input points to an existing tessellation. When using Qhull, therefore, we delete the previous tessellation and compute it anew each time. If CGAL is used, the later stages are less expensive than the earlier ones because we take advantage of the incremental addition of input points.

The primary contribution of our parallel algorithm is the correct handling of the neighborhood communication in order to generate the correct global tessellation. In the remainder of this section, we first describe the algorithm that determines which points are exchanged between a block and its neighbors. Then we prove that this algorithm results in the correct global tessellation. Next, we survey how this code handles different boundary conditions that affect the global tessellation. We also discuss the memory and storage requirements of our parallel algorithm.

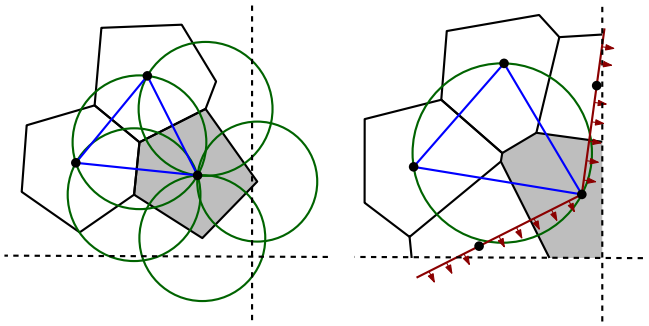

Fig. 2. Left: Circumspheres at vertices of a finite Voronoi cell. Right: An infinite Voronoi cell and the half-spaces supported by the facets of the convex hull. The edges of the blocks are shown by dotted lines.

A. Neighbor Point Exchange

In this section, we describe how, using only local information, each input point of a block can be checked whether it must be sent to a neighboring block. Input points are classified into two categories: (1) those points whose Voronoi cells are finite and (2) those points whose Voronoi cells are infinite. (Recall from Section II-A that the site of an infinite Voronoi cell lies on the convex hull of the local input points.) Whether and to which neighboring blocks an input point must be sent differ for the two cases.

1) Points with Finite Voronoi Cells: For an input point with a finite Voronoi cell, to test whether the point must be exchanged, we construct circumspheres at the Voronoi cell vertices, as shown in the left side of Figure 2. These spheres are the circumspheres of the tetrahedra of the dual Delaunay tessellation. The center of each sphere is a vertex of the Voronoi cell, and the radius is the distance from the vertex to the Voronoi cell site (input point associated with the cell). The left side of Figure 2 highlights one such Voronoi cell in gray and its set of circumspheres in green. If the Voronoi cell is correct in the global tessellation, then by definition, no other input point may be inside its circumspheres. Thus, if the circumspheres do not intersect a neighboring block, then no remote point can change the cell: we say it is *finalized*. If, however, a circumsphere intersects one or more neighbor blocks, then it is possible that a remote point contained in a neighboring block could change the Voronoi cell. Thus, the input point (the site of the Voronoi cell) is sent to all the neighbors intersected by any circumsphere of its Voronoi cell. In the left side of Figure 2, the site of the gray cell will be sent to its right, lower, and lower-right neighbors. The green and red circumspheres in Figure 3 highlight finalized and nonfinalized cells, respectively.

2) Points with Infinite Voronoi Cells: For an input point with an infinite Voronoi cell, determining where the point must be sent is more complicated. The right side of Figure 2 shows an infinite Voronoi cell highlighted in gray. The input point of an infinite Voronoi cell is on the convex hull of the input points in the block. Each facet of the convex hull that has the input point as a vertex defines an outer half-space, illustrated in the figure with dark red lines with arrows. A point in such a half-space may be a Delaunay neighbor of the input point under consideration. An input point on the convex hull supports multiple half-spaces, which in turn intersect many neighboring blocks. Conservatively, the input point would be sent to all the intersected neighbors.

In practice, it is unlikely that all intersected neighboring blocks contain points that will create edges with every point on the local convex hull. We would like to reduce communication and memory footprint by limiting unnecessary point exchanges whenever possible. Hence, we use the following two-pass heuristic. In the first pass, the site of each infinite cell is sent to its (single) nearest neighbor block. (In the future, this heuristic might be tuned by adding additional neighbors to the first pass.) This is the most likely block to contain a remote point; and after the subsequent local tessellation, the Voronoi cell may no longer be infinite. In the second pass, any local input points that still have infinite Voronoi cells are sent to all 26 neighbors, excluding the neighbor already covered in the first pass. Local input points that now have finite cells are handled as in Section III-A1. The two-pass heuristic point exchange corresponds to steps 2-4 of Algorithm 1; the details of each pass of the neighbor exchange are in Algorithm 2.

Algorithm 2 Neighbor Point Exchange

Pass 1:
1: Enqueue finite cell sites that are near to block boundaries to any neighbors within circumsphere radii
2: Enqueue infinite cell sites to single closest neighbor
3: Exchange enqueued neighbor points
4: Compute new tessellation

Pass 2:
5: Enqueue local finite cell sites that were infinite in Pass 1 to any neighbors within circumsphere radii
6: Enqueue remaining local infinite cell sites to all neighbors
7: Exchange enqueued neighbor points

The following table illustrates the advantages of our heuristic on synthetic data partitioned into eight blocks. The first column lists the total number of input points; the second, the largest number of points on the convex hull of any block after the initial local computation; the third, the largest number of points sent to *every neighbor* by any block after the second pass. Approximately one-third of the points on the convex hull remain to be sent to all neighbors in the final pass; but for the majority of convex hull points, it sufficed to send them only to the single nearest neighbor.

Number of points	Infinite cells	Sent to all
512	35	12
4,096	65	25
32,768	103	38
125,000	151	49
421,875	187	62
1,000,000	217	76

B. Proof of Correctness of the Global Tessellation

We now prove that Algorithm 2 will result in the correct global tessellation, provided that the blocks used are sufficiently large in spatial extent. Table I summarizes the notation needed in order to prove the correctness of our algorithm. The global space is partitioned into a collection of blocks, $\{B_i\}$. We denote by N_i the 1-neighborhood of the block B_i. We assume in

TABLE I. NOTATION

Symbol	Meaning
P	the set of all input points
$\mathrm{Del}(P)$	Delaunay tessellation of P
$\sigma \in \mathrm{Del}(P)$	a tetrahedron in $\mathrm{Del}(P)$
$\mathrm{o}(\sigma)$	the circumsphere of σ
B	a block of the input domain decomposition
$\mathrm{Del}_B(P)$	the set of tetrahedra in $\mathrm{Del}(P)$ with at least one vertex in B
U_B	the set of vertices of $\mathrm{Del}_B(P)$
N	the 1-neighborhood of B, i.e., the union of B with its immediate neighbor blocks
P_N	the subset of points in P in the neighborhood N

our algorithm and in the following proof that the input point set P is such that all Delaunay edges $\langle p,q\rangle$, incident on a point $p \in B_i$, are contained in N_i; that is, $p \in B_i \rightarrow q \in N_i$. Consequently, Delaunay tetrahedra cannot be larger than the 1-neighborhood of blocks. This restriction could be relaxed by employing neighborhoods larger than 1-neighborhoods or, alternatively, by employing multiple rounds of communication between overlapping 1-neighborhoods. We did not, however, do this in our algorithm.

The following proof may be summarized as follows. Under the assumption that no Delaunay edges extend beyond a neighborhood, it suffices to compute the Delaunay tessellation of the input points in the neighborhood, that is, $\mathrm{Del}_B(P) = \mathrm{Del}_B(P_N)$ (Theorem 2). Moreover, if U_B are the vertices of $\mathrm{Del}_B(P)$, then for any superset $W \supseteq U_B$, $\mathrm{Del}_B(P) = \mathrm{Del}_B(W)$ (Lemma 3). The union of the input points sent to B and the input points originally in B are a superset of U_B, and, therefore, suffice (Theorem 5).

An important consequence of the logic is that we can rely on symmetry to simplify the communication protocol: when constructing a Delaunay tessellation, if point p in block B_i needs to construct an edge to point q in a neighbor B_j, then point q also needs to construct an edge to point p. This allows communication to be determined locally without the need to poll neighbors as to whether they have any points contained in the local circumspheres. Instead, we only need to determine which points to send to neighbors, thus avoiding a more complicated handshake protocol. (See the right side of Figure 3.)

Details of the proof follow.

Lemma 1. *If a point set $Q \not\subseteq P$ falls inside the circumsphere of a Delaunay tetrahedron σ, $Q \subseteq \mathrm{o}(\sigma), \sigma \in \mathrm{Del}(P)$, then each vertex of σ has an edge to some point $q \in Q$ in $\mathrm{Del}(P \cup Q)$.*

Proof: We first prove that if a single point $q \notin P$ falls into the circumsphere of a Delaunay tetrahedron σ, $q \in \mathrm{o}(\sigma), \sigma \in \mathrm{Del}(P)$, then q has edges to every vertex of σ in $\mathrm{Del}(P \cup \{q\})$. Let p be a vertex of the tetrahedron σ. Let O be the circumcenter of σ. The perpendicular bisector between p and q intersects the line segment pO at O': this intersection exists because, by assumption that q falls inside $\mathrm{o}(\sigma)$, we have $|qO| < |pO|$. Since O' lies on pO and, therefore, inside $\mathrm{o}(\sigma)$, O' is closer to p and q than to any point in P. By definition of Delaunay triangulation, there is a Delaunay edge between p and q. Extending the claim from a single point q to a set of points Q follows by induction. ■

The following theorem establishes that it suffices to compute the Delaunay tessellation of the point set restricted to the neighborhood of a block, $\mathrm{Del}_B(P_N)$, in order to correctly compute the Delaunay tetrahedra that have a vertex in the block, $\mathrm{Del}_B(P)$.

Theorem 2. *If no edge in $\mathrm{Del}(P)$, incident on a vertex $p \in B$, leaves N, then $\mathrm{Del}_B(P) = \mathrm{Del}_B(P_N)$.*

Proof: $\mathrm{Del}_B(P) \subseteq \mathrm{Del}_B(P_N)$ follows immediately from the definition of the Delaunay tessellation and the assumption of the theorem. To show $\mathrm{Del}_B(P) \supseteq \mathrm{Del}_B(P_N)$, let $\sigma \in \mathrm{Del}_B(P_N)$. Then, by definition, $\mathrm{o}(\sigma)$ is empty in N, and σ contains a point $p \in B$. Suppose σ is not in $\mathrm{Del}_B(P)$. Then $\mathrm{o}(\sigma)$ contains a set of points Q outside P_N, that is, $Q \subseteq P, Q \cap P_N = \emptyset$. Then, by Lemma 1, each vertex of σ, and specifically p, has an edge to a point in Q, which contradicts the assumption that no edge leaves N. ■

Lemma 3. *If $P \supseteq W \supseteq U_B$, then $\mathrm{Del}_B(W) = \mathrm{Del}_B(P)$.*

Proof: $\mathrm{Del}_B(P) \subseteq \mathrm{Del}_B(W)$ follows immediately from the definition of the Delaunay tessellation. To show $\mathrm{Del}_B(P) \supseteq \mathrm{Del}_B(W)$, let $\sigma \in \mathrm{Del}_B(W)$. Suppose $\sigma \ni p \in B$. Suppose σ is not in $\mathrm{Del}_B(P)$. Then $\mathrm{o}(\sigma)$ contains a set of points $Q \subseteq P$. By definition of a Delaunay tetrahedron, $Q \cap W = \emptyset$. Then, p has an edge to some $q \in Q$ in $\mathrm{Del}(P)$, which contradicts the assumption that $U_B \subseteq W$. ■

Lemma 4. *Let B' be a neighbor of block B; $B' \subseteq N$ with point $q \in B'$. If for some set of points $P' \subseteq P$, with $q \in P'$, the circumspheres of all the tetrahedra in $\mathrm{Del}(P')$ that contain q do not intersect B, then $q \notin U_B$.*

Proof: Suppose the lemma is false, and there is some edge pq in $\mathrm{Del}(P)$, with $p \in B$. Then, the perpendicular bisector of pq must intersect the cell of q in the Voronoi tessellation of P'. Since the bisector of pq is a plane and the Voronoi cell is a convex polyhedron, the fact that they intersect implies that there is a vertex O of the polyhedron that lies closer to p than to q (because we assumed that the points lie in general position, the bisector of pq cannot pass through a Voronoi vertex). But that means that the circumsphere centered at O that passes through q contains point p, which contradicts the assumption of the lemma. ■

The following theorem implies that the only points that need to be sent to a neighboring block are those whose surrounding circumspheres intersect the neighbor, thus proving the correctness of our neighbor point communication algorithm.

Theorem 5. *The set of vertices in B after the two passes of the algorithm contains U_B as a subset.*

Proof: By assumption, $P_N \supseteq U_B$. A point q in a neighbor block B' whose surrounding circumspheres (circumspheres of the tetrahedra containing q) do not intersect B cannot belong to U_B (Lemma 4). By the end of Algorithm 2, we have not sent a point $q \in B'$ to B only if we have found a set of circumspheres that pass through q, none of which intersects B. Since adding a point to the Delaunay tessellation can only shrink the union of circumspheres around any other point, we do not need to retest the Voronoi cells that were already finite in the first pass of the algorithm. ■

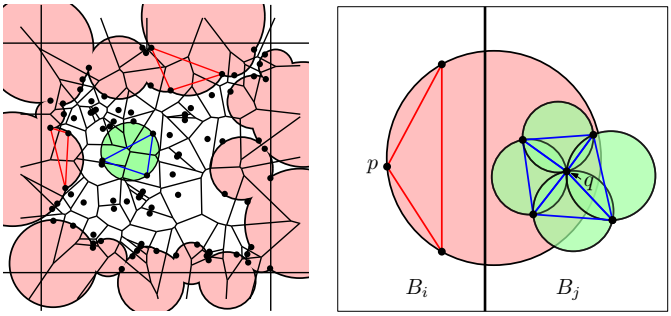

Fig. 3. Left: The green Delaunay circumsphere is contained entirely in the local block, and therefore the blue triangle remains Delaunay in the final triangulation. The red circumspheres intersect other blocks; the triangles they circumscribe are not yet finalized. Right: Despite the fact that point q falls into an (unfinished) circumsphere supported by point p, we don't need to send it to block B_i since its surrounding circumspheres are finalized (empty of any points).

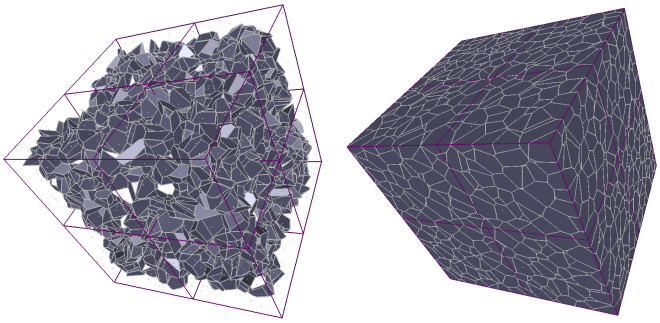

Fig. 4. Left: No boundary conditions. Right: Wall boundary conditions on the outermost six sides of overall domain of 16^3 particles and 8 blocks.

C. Boundary Conditions

Our tessellation code supports three types of boundary conditions: (1) no boundary conditions, where the points are embedded in infinite space, (2) periodic boundary conditions, where the points are embedded in a three-dimensional torus, and (3) wall boundary conditions, where the points are inside a box. The "no boundary condition" case requires no additional effort. Blocks on the boundary of the global domain do not have neighbors in specified directions; thus, no points are enqueued for exchange in these directions.

1) Periodic Boundary Conditions: Periodic boundary conditions are common in cosmology and molecular dynamics. For periodic boundary conditions, a block at one side of the global domain is a neighbor of a block on the opposite side of the domain; in other words, the domains wraps around a torus. Our method supports optional periodic boundary conditions: when enabled, our code tracks where the domain wraps around and correctly handles data transfers between neighbors across these boundaries. When using our Voronoi or Delaunay tessellation in subsequent applications (Section IV), all information, such as the positions of vertices, is stored locally in the block in its final transformed geometry. That is, users of a finished tessellation can safely ignore the details of any underlying wraparound transformations that may have been performed during the construction of the tessellation.

2) Wall Boundary Conditions: Wall boundary conditions can be found in simulations of particles packed in a container or sedimenting against a surface. Under such conditions, if a Voronoi cell intersects a wall, the Voronoi cell should have a face that conforms to the wall. As shown in Figure 4, our code supports an optional planar wall boundary condition where the walls form the boundaries of the global domain. Details of implementing this wall boundary condition follow.

To generate the planar wall cut of a Voronoi cell intersecting a wall, we use the Voronoi definition that every planar face of a Voronoi cell is the midplane between the site and another input point. Therefore, to apply a planar wall boundary condition, we can add a "virtual" input point on the opposite side of the wall to force a Voronoi face to coincide with the wall, in essence cutting the cell at the wall. Specifically, for each input point in the block, if the associated Voronoi cell is finite, then a new point is generated for each wall that intersects the Voronoi cell. If the Voronoi cell is infinite, then a new point is generated for each of the walls. [7] These new points are the reflection of the site point across the planar boundary walls. The virtual points are then added to the list of points in the block, as if they were received from a neighbor. Thus, in the final tessellation, following the final neighbor exchange, each Voronoi cell that intersects a planar wall at the boundary has a face that conforms to the wall. These virtual points have no other impact on the tessellation and are not actually communicated to any neighboring blocks.

D. Memory and Storage Requirements

1) Data Size: Upon completion of our algorithm, each block contains the following tessellation data: (1) the number and positions of its original input points; (2) additional neighbor points received during the communication phases of the algorithm, and (3) the number of Delaunay tetrahedra, the four vertex indices, and four neighbor indices for each tetrahedron. In the datasets that we tested, each input point is shared by 7 Delaunay tetrahedra on average. The third (center right) plot of Figure 6 shows the total memory usage per particle of our algorithm during the course of tessellating 262,144 synthetic particles generated by randomly displacing particles lying on a 64^3 grid by a maximum distance of 2 grid cells. Both implementations of our algorithm, one using Qhull and the other with CGAL, are shown, together with annotations of the significant stages of the algorithm.

Our memory measurements confirm an earlier comparison [25], which says that the CGAL library requires approximately four times less memory than Qhull. The ratio in Figure 6 is closer to 2.5 times smaller for CGAL because we copy the tessellation into our own data structure and because our measurements reflect the resident size of the entire executable. The fact that Qhull does not allow incremental updates to its tessellation is also evident in Figure 6 by the three spikes in memory usage in contrast to the CGAL usage that remains fairly constant.

2) Storage Format: We write the tessellation in parallel with PnetCDF [22] into a single file in order to take advantage

[7] This is performed in lieu of determining precisely which set of walls cut the infinite cells.

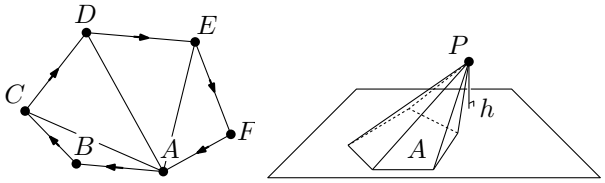

Fig. 5. Left: decomposition of a convex polygon into triangles to compute its area. Right: decomposition of a convex polyhedron into pyramids to compute its volume.

of parallel I/O performance on HPC storage systems while at the same time having a self-describing portable file format. Two related issues need to be addressed when converting our parallel unstructured mesh to an array-based file model: the fact that our model is composed of independent parallel blocks, and each block is a different size. We write one array in the file for each primary component of the data model; for example, all the input point positions from all blocks are in one array in the file. To do so, processes first exchange data sizes for each local contribution to the global arrays for each component of the data model and compute prefix sums of starting offsets for their blocks; then processes write their blocks' data collectively for each global array.

Consumers of the tessellation file may also be block-based parallel programs; hence, the block structure must be recoverable from the PnetCDF file. Therefore, the block offsets for each global array are stored along with additional information about the blocks (extents, global ID, and list of neighboring blocks). Functions for parallel reading and writing are included in our library. Including metadata about the blocks (extents, neighbors) and neighbor points received during the algorithm introduces overhead in the file size that increases with the number of blocks. For example, the file size varies from 300 B per particle to 1.3 KB per particle for 16 to 16K processes, respectively (1 process = 1 block in this case). A large fraction of the output is integer data, allowing the file to compress to approximately 50% of its original size. As another example, the tessellation from a synthetic dataset of 32K particles and 8 blocks was 10 MB raw and 4.5 MB compressed.

IV. Using the Resulting Voronoi and Delaunay Tessellations in Subsequent Applications

We describe below how to use a parallel tessellation in applications. In order to compute some property over the geometric entities, for example Voronoi cell volume, or simply for visualization, the tessellation needs to be traversed by visiting each of its geometric entities (cells, faces, edges, vertices) once. Applications using the tessellation may also be parallel programs; hence, our construction of the tessellation must support both serial and parallel Delaunay and Voronoi traversal. The Delaunay tessellation is available directly, and its dual Voronoi tessellation is constructed on the fly by our code. Such traversals are part of standard machinery in computational geometry, but for the benefit of readers who may not be familiar with this topic, we review below how we generate a Voronoi traversal from our Delaunay data and also how we compute the volume of a Voronoi cell.

As explained in Section III-D, we compute and store in memory and on disk only the input points and the Delaunay tetrahedra. Each tetrahedron is represented by the four indices into the input points and the four indices to neighboring tetrahedra that intersect it in a common triangular face. The two arrays are coordinated such that the ith tetrahedron is opposite the ith vertex for i = 0,1,2,3. If one of the faces of the tetrahedron lies on the convex hull of the input points, the tetrahedron is missing a neighbor across that face. We record this as a -1 entry for its neighbor.

Delaunay vertices in the same tetrahedron may span neighboring blocks. To facilitate local traversal of Delaunay tetrahedra and their conversion into Voronoi faces, we replicate input points corresponding to remote Delaunay vertices in the neighboring blocks such that each tetrahedron can be visited locally. Similarly, all the neighboring Delaunay tetrahedra corresponding to the vertices for one Voronoi cell are contained in the same block such that a Voronoi cell can be visited locally. In such cases, entire Delaunay tetrahedra are duplicated across neighboring blocks.

Algorithm 3 Delaunay to Voronoi Tessellation

```
1: for all input points do
2:     get neighbor edges of Delaunay tetrahedra
3:     for all neighbor edges do
4:         generate edge star
5:         for all tetrahedra in edge star do
6:             compute the tetrahedron circumcenter, c
7:             add c to the vertices of the Voronoi face
```

Since the Voronoi and Delaunay tessellations are dual, the Voronoi tessellation can be derived from the Delaunay any time. Each Voronoi face is dual to a Delaunay edge: specifically, the vertices of the Voronoi face are the circumcenters of the tetrahedra that share its dual Delaunay edge. We refer to all edges of the Delaunay tessellation that include an input point as the *neighbor edges* of the point and to the set of tetrahedra that share an edge, listed in the order that they encircle the edge, as an *edge star*.

Algorithm 3 iterates over Voronoi geometric entities derived from the Delaunay tessellation. Looping over input points in line 1 is equivalent to iterating over all Voronoi cells. Similarly, looping over the neighbor edges in line 3 is equivalent to iterating over all the Voronoi faces of the current Voronoi cell, and looping over the Delaunay tetrahedra of the edge star in line 5 is equivalent to visiting the vertices of the current Voronoi face. The key step in Algorithm 3 is generating the edge star data structure. The algorithm for computing the edge star, given the edge and one initial tetrahedron that contains the edge, is described in Algorithm 4.

Algorithm 4 Edge Star

```
1: o ← a tetrahedron containing the input edge e = (x, y)
2: add o to the edge star
3: u, v ← the two vertices of o that are not in e
4: t ← the neighbor of o opposite u
5: while t ≠ o do
6:     add t to the edge star
7:     u ← v
8:     v ← the fourth vertex of t, i.e., the vertex not in {x, y, u}
9:     t ← neighbor of t opposite u
```

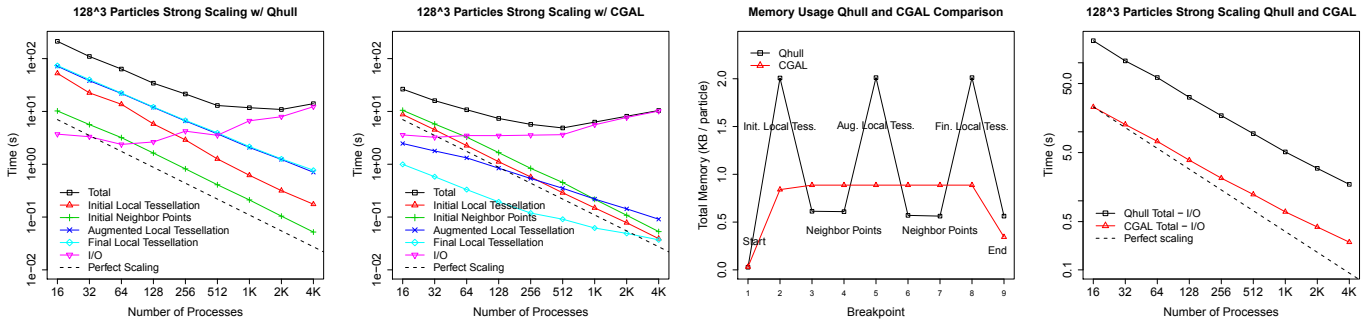

Fig. 6. Strong scaling for 128^3 particles tessellated in parallel using Qhull (far left) and CGAL (center left). Total time is further broken down into main component times. Center right: total memory usage during the execution of the algorithm normalized by the number of particles. Far right: strong scaling (excluding I/O time) comparison between Qhull and CGAL for 128^3 particles.

Many applications require the volume or surface area of Voronoi cells. Once we have traversed our data structure in order to generate Voronoi cell faces and vertices, it is straightforward to compute the volume and surface area as well. The area of a Voronoi face, a convex polygon, can be computed by triangulating the polygon from the first vertex and summing the area of the triangles. An illustration of this is shown on the left side of Figure 5. The area of each triangle ABC is calculated by computing the cross product, $area(ABC) = |AB \times AC|/2$. The total surface area of the Voronoi cell is the sum of the areas of its faces.

The volume of the Voronoi cell, a convex polyhedron, is the sum of the volumes of the pyramids formed between the site of the cell and each face. If A is the area of a Voronoi face and d is the length of its dual Delaunay edge, then the volume V of the pyramid that has the site of the Voronoi cell as its apex and the Voronoi face as its base polygon is $V = A(d/2)/3 = Ad/6$, (right side of Figure 5). Note that since a Voronoi face is the equidistant plane between the two points of its dual Delaunay edge, $h = d/2$ is the height of the pyramid.

V. Performance Evaluation and Application to Scientific Data

We first evaluate the performance of our parallel algorithm and then demonstrate its use in three scientific domains. Performance tests were run on the IBM Blue Gene/Q *Mira* and Cray XC30 *Edison* machines at the Argonne Leadership Computing Facility (ALCF) at Argonne National Laboratory and at the National Energy Research Scientific Computing Center (NERSC) at Lawrence Berkeley National Laboratory, respectively. Mira is a 10-petaflop system consisting of 48K nodes, each node with 16 cores (PowerPC A2 1.6 GHz) and 16 GB RAM. Edison is a 2.57-petaflop machine with 5576 nodes, each node with 24 cores (Intel Ivy Bridge 2.4 GHz) and 64 GB RAM. GCC (version 4.4.6 on Mira, version 4.8.1 on Edison) with -O3 optimization was used to compile the test code.

A. Synthetic Points

To test the scaling and memory usage of the parallel algorithm, we generated synthetic particles by randomly displacing points from regular grid positions in various size grids.

Figure 6 shows the full-scale performance results conducted on Mira by using 16 MPI processes per compute node. In the left two panels, we show the strong scaling of our parallel algorithm implemented with Qhull and CGAL, respectively. The total time is further subdivided into five of the six main components of Algorithm 1. Step 4, exchanging remaining neighbor points, is omitted because this step required less than 0.1 seconds. Comparing the far left panel to the center left panel, we observe that CGAL's local tessellation time is approximately 5 times faster than Qhull's, and its final tessellation is at least 10 times faster. Since CGAL's tessellation can be incrementally updated, the cost of the final tessellation is a small fraction of the initial tessellation. Overall, excluding the I/O time, using the CGAL library is 5-9 times faster than using Qhull. The file I/O does not scale well in this test, presumably because this file is too small to be written by more than 512 processes. Our focus in this paper is not I/O performance, and we do not investigate it further here. In the third pane of Figure 6, we show the total memory usage per particle during the execution of the algorithm (discussed in Section III-D). A comparison of strong scaling of total time minus I/O time for 128^3 particles with Qhull and CGAL is shown in the far right side of Figure 6.

Figure 7 shows strong and weak scaling of total time minus I/O time with CGAL for particle counts ranging from 128^3 to 2048^3 and up to 128K processors. Excluding I/O, the strong scaling efficiency for 128^3 particles is approximately 47%; for 2048 particles it is 90%. The horizontal dotted lines in Figure 7 give a visual reference for perfect weak scaling; each subsequent data point along the same dotted horizontal line represents eight times the number of input particles and eight times the number of processes. The data points near the ends of the middle dotted line (256^3 particles at 64 processes and 2048^3 particles at 32K processes) have a weak scaling efficiency of 98%.

B. N-Body Cosmological Simulation

We tested two sources of N-body cosmological simulation data.

1) HACC: HACC (Hardware/Hybrid Accelerated Cosmology Code) [26], [27] is an N-body cosmology code that exceeded 10 petaflops on ALCF's *Mira* IBM Blue Gene/Q and earned two Gordon Bell Finalist awards [28], [29]. The

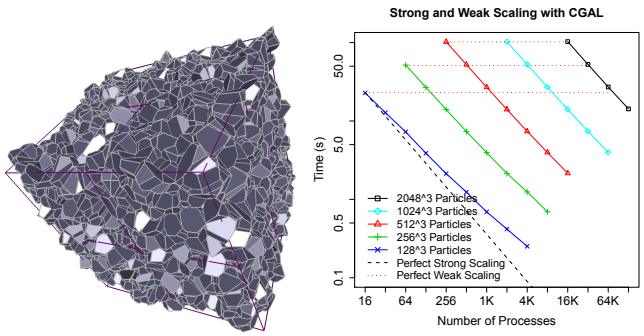

Fig. 7. Left: Voronoi tessellation of synthetic data using 16^3 particles and 8 blocks. Right: strong and weak scaling (excluding I/O time) using CGAL for a particle counts ranging from 128^3 up to 2048^3. Processor counts range from 16 to 128K.

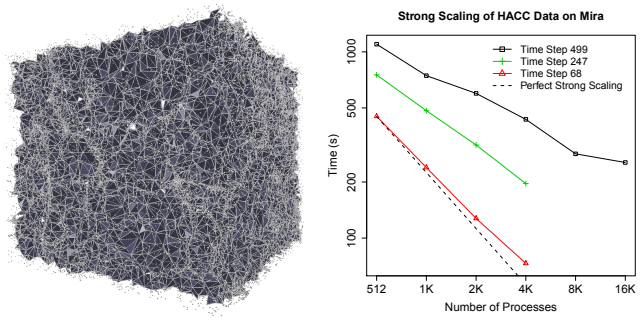

Fig. 8. Left: Delaunay tessellation of 128^3 dark matter tracer particles. Right: strong scaling (excluding I/O time) using CGAL for three time steps of HACC data of 1024^3 particles.

code solves the six-dimensional Vlasov-Poisson gravitational equations by using N-body particle methods.

Figure 8 shows strong scaling results of tessellating particle data (excluding I/O time) at three time steps from early, middle, and late in a recent simulation run of 1024^3 particles. The original simulation ran by using 512 blocks (MPI processes). To perform a strong scaling study, we further decomposed each block by factors of 2 in x, y, z directions until we arrived at the desired number of processes.

The primary difference between HACC particles and synthetic data is that as the cosmological simulation evolves, particles cluster into extremely dense and sparse regions. At the early, middle, and late time step, the ratio of maximum/minimum number of particles per process is 1.5, 4.3, and 10.4, respectively. The diminishing efficiency of the algorithm for unbalanced data is evident in the strong scaling efficiency: 77%, 48%, and 14% for the early, middle, and late time steps, respectively.

2) Nyx: Nyx is a newly developed N-body and gas dynamics code [30] written in C++ and Fortran90. It is based on the BoxLib[8] framework for structured-grid adaptive mesh methods. Using a hybrid programming model based on MPI and OpenMP, Nyx follows the evolution of dark matter particles gravitationally coupled to a gas using a combination of multilevel particle-mesh and shock-capturing Eulerian methods. High dynamic range is achieved by applying adaptive mesh refinement to both gas dynamics and gravity calculations. The multigrid method is used to solve the Poisson equation for self-gravity. The mesh structure used to update fluid quantities also evolves the particles via the particle-mesh method.

The dataset used for the experiments on Edison, in the left side of Figure 9, is from a middle time step of the Nyx simulation, with particle imbalance factor between the processes ranging from 1.5 to 7.1 for 1K to 32K processes. In addition to the load imbalance inherent in the original dataset, the imbalance factor increases with the number of blocks in our scalability study. The reason is that for our scaling experiment we subdivide blocks into regular-size smaller blocks, which compounds load imbalance when the data are highly clustered. The overall strong scaling efficiency between 1K and 32K processes is 46%.

[8] ccse.lbl.gov/BoxLib

C. Soft Matter Simulations

Simulations of microphase separated soft matter systems are further applications for a large-scale tessellation code. In such systems, populations of molecules, modeled as multiple types of beads (particles) bonded together, self-organize in order to form two or more domains. Each domain contains only one type of bead. These domains can form simple structures such as lamallae or more complicated and exotic geometries such as the double gyroid. By applying a Voronoi tessellation to such a system and grouping the cells of each component, properties of the individual domains, such as density and shape, can be studied as a function of system properties, such as temperature and concentration [31].

To demonstrate this, we apply a Voronoi tessellation to over three hundred time steps of a system of 2,765 chains of an A-B-A triblock copolymer (176,960 total beads) generated from the model of Pike et al. [32] and found in an alternating gyroid morphology. By summing the Voronoi cells associated with the A and B species, we establish that the A species composes $48.5 \pm 0.1\%$ of the simulation volume while the B species composes $51.5 \pm 0.1\%$. This result suggests that because the B part of the chain component has to stretch or fold, while the A part of the chain does not, the B domain dilates relative to the A domain. This type of information could not be obtained without the ability to easily measure the volume of each domain, which the Voronoi tessellation allows.

We also can use a Voronoi tessellation to construct the surface separating one domain from another. In Figure 10, we have computed the Voronoi tessellation of 1,000 A-B-C "telechelics" composed of two nanospheres (A and C) connected by a polymer tether beads (B) for a total of 8,000 beads in a double gyroid morphology [33]. Only the Voronoi cells associated with the A species are shown in Figure 10. The tubes and nodes of a gyroid phase are evident in the structure of the A domain, as is the sixfold wheel structure characteristic of the double gyroid phase. The facets of the tessellation define a surface of the A domain. Such surfaces are usually constructed by using isosurface methods, which require averaging over many time steps; whereas by using the tessellation, such surfaces can be constructed for every time step.

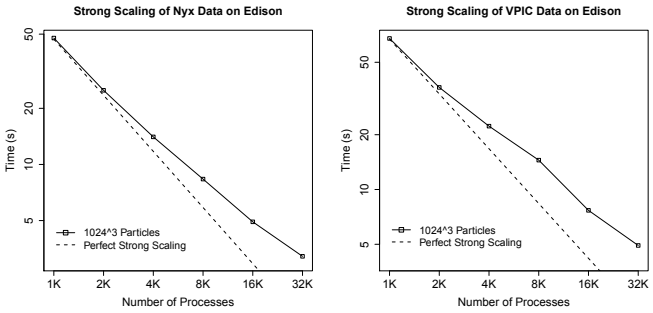

Fig. 9. Left: Strong scaling (excluding I/O time) using CGAL for a Nyx middle time step of 1024^3 particles. Right: Same for one time step of VPIC data.

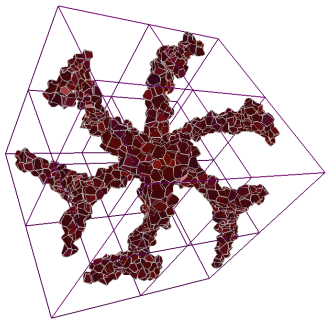

Fig. 10. Voronoi cells of the A species of the double gyroid. The sixfold "wheel" structure formed by the A domain is characteristic of the double gyroid phase.

D. Plasma Physics Simulations

VPIC [34] is a high-performance 3D electromagnetic relativistic particle-in-cell code designed to minimize data movement. The code integrates the relativistic Maxwell–Boltzmann equations in a linear background medium. Individual computational particles represent many physical particles (for example, electrons). The particular dataset we used in our experiments is a 1024^3 random subsample of the trillion-particle dataset used in a parallel I/O study by Byna et al. [35]. Delaunay and Voronoi tessellations can help describe the spatial distribution of highly energetic particles (including identifying their dense regions), although in this paper we use this dataset to further measure the scalability of our algorithm. The running times to compute the Delaunay tessellation using 1K to 32K processors on Edison are shown in the right side of Figure 9. The strong scaling efficiency between 1K and 32K processes is 43%.

VI. Conclusion

We presented a new parallel algorithm for computing Delaunay and Voronoi tessellations by spatially decomposing the tessellation into blocks and then by using a combination of three local (serial) tessellations interleaved with two neighbor point exchanges. The primary challenge of parallelizing the tessellation is automatically determining the points to exchange among neighboring blocks in order to form the correct global tessellation. We proved that our algorithm produces the correct result provided that all Delaunay edges fit within a 1-neighborhood of blocks. We also proved that if a tessellation cell in block A is affected by a point in a neighboring block B, then the point's associated cell in block B is also affected by the point in block A. Hence, all communication can be determined locally: a block can decide whether and to which neighbors a point must be sent without having to ask them first. The block will, in turn, receive the correct set of points from its neighbors.

We implemented our algorithm with two serial computational geometry libraries, and we tested strong and weak scaling up to 2048^3 synthetically generated particles and 128K processes. To further demonstrate the use of our algorithm for scientific data, we applied it to two datasets from cosmology simulations, two datasets from soft matter simulations, and one dataset from a plasma physics simulation.

Voronoi and Delaunay tessellations have many applications for large N-body datasets. Discrete particles can be converted into a continuous field that can be interpolated, differentiated, and integrated, operations not possible on the original particles without some type of sampling or estimation. The statistics of tessellated cells can further be used to identify and quantify structures such as cosmological halos and voids, to calculate macroscopic statistics or morphologies of molecular dynamics simulations, or to describe the spatial distribution of highly energetic particles in plasma simulations.

We envision three extensions of our work. First, we are considering improving memory performance by adapting the data structure to the native data model of the underlying serial library. Second, we are considering lifting the restriction that Delaunay cells must be smaller than the neighborhood size. This requires potentially exchanging points among larger neighborhoods (2-neighbors, 3-neighbors, and so forth) over multiple communication rounds. Third, we are exploring hybrid parallel formulations of the algorithm that combine shared and distributed memory.

Acknowledgments

We gratefully acknowledge the use of the resources of the Argonne Leadership Computing Facility (ALCF) and the National Energy Research Scientific Computing Center (NERSC). We are especially grateful to Salman Habib, Katrin Heitmann, Hal Finkel, and Adrian Pope of the HACC team at Argonne for the use of their data; George Zagaris of Kitware for use of his HACC GenericIO library; Zarija Lukić of LBNL for the Nyx data; Prabhat, Suren Byna, and Kuan-Wu Lin for assistance with VPIC data; Wei-keng Liao of Northwestern University for assistance converting the in-memory data model to pnetCDF; and Gurdamen Khaira, Jian Qin, Juan De Pablo, and Ryan Marson for providing data for molecular dynamics. This work was supported by Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contracts DE-AC02-06CH11357 and DE-AC02-05CH11231. Work is also supported by DOE with agreement No. DE-FC02-06ER25777.

References

[1] R. Seidel, "The Nature and Meaning of Perturbations in Geometric Computing," *Discrete and Computational Geometry*, vol. 19, pp. 1–17, 1998.

[2] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars, *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 2008.

[3] H. Edelsbrunner, *Topology and Geometry for Mesh Generation*. Cambridge University Press, 2001.

[4] C. L. Lawson, "Software for C^1 Surface Interpolation," in *Mathematical Software III*, J. R. Rice, Ed. New York: Academic Press, 1977, pp. 161–194.

[5] K. L. Clarkson and P. W. Shor, "Applications of Random Sampling in Computational Geometry," *Discrete and Computational Geometry*, vol. 4, pp. 387–421, 1989.

[6] L. J. Guibas, D. E. Knuth, and M. Sharir, "Randomized Incremental Construction of Delaunay and Voronoi Diagrams," *Algorithmica*, vol. 7, pp. 381–413, 1992.

[7] H. Edelsbrunner and N. R. Shah, "Incremental Topological Flipping Works for Regular Triangulations," *Algorithmica*, vol. 15, pp. 223–241, 1996.

[8] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa, "The Quickhull Algorithm for Convex Hulls," *ACM Trans. Math. Softw.*, vol. 22, pp. 469–483, Dec. 1996. [Online]. Available: http://doi.acm.org/10.1145/235815.235821

[9] R. Miller and Q. F. Stout, "Efficient Parallel Convex Hull Algorithms," *IEEE Trans. Comput.*, vol. 37, no. 12, pp. 1605–1618, Dec. 1988.

[10] C.-S. Jeong, "An Improved Parallel Algorithm for Constructing Voronoi Diagram on a Mesh-Connected Computer," *Parallel Computing*, vol. 17, no. 4, pp. 505–514, 1991.

[11] F. Dehne, X. Deng, P. Dymond, A. Fabri, and A. A. Khokhar, "A Randomized Parallel 3D Convex Hull Algorithm for Coarse Grained Multicomputers," in *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '95. New York, NY: ACM, 1995, pp. 27–33. [Online]. Available: http://doi.acm.org/10.1145/215399.215410

[12] G. Blelloch, J. C. Hardwick, G. L. Miller, and D. Talmor, "Design and Implementation of a Practical Parallel Delaunay Algorithm," *ALGORITHMICA Special Issue on Coarse Grained Parallel Algorithms*, vol. 24, pp. 243–269, August 1999.

[13] P. Cignoni, C. Montani, R. Perego, and R. Scopigno, "Parallel 3d Delaunay Triangulation," in *Computer Graphics Forum*, vol. 12, no. 3. Wiley Online Library, 1993, pp. 129–142.

[14] T. Peterka, J. Kwan, A. Pope, H. Finkel, K. Heitmann, S. Habib, J. Wang, and G. Zagaris, "Meshing the Universe: Integrating Analysis in Cosmological Simulations," in *Proceedings of the SC12 Ultrascale Visualization Workshop*, Salt Lake City, UT, 2012.

[15] T. Peterka, R. Ross, W. Kendall, A. Gyulassy, V. Pascucci, H.-W. Shen, T.-Y. Lee, and A. Chaudhuri, "Scalable Parallel Building Blocks for Custom Data Analysis," in *Proceedings of the 2011 IEEE Large Data Analysis and Visualization Symposium LDAV'11*, Providence, RI, 2011.

[16] T. Peterka and R. Ross, "Versatile Communication Algorithms for Data Analysis," in *EuroMPI Special Session on Improving MPI User and Developer Interaction IMUDI'12*, Vienna, AT, 2012.

[17] A. Geist, W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, W. Saphir, and T. Skjellum, "MPI-2: Extending the Message-Passing Interface," in *Proceedings of Euro-Par'96*, Lyon, France, 1996.

[18] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, S. Kumar, E. Lusk, R. Thakur, and J. L. Träff, "MPI on a Million Processors," in *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 20–30.

[19] A. Fabri and S. Pion, "CGAL: The Computational Geometry Algorithms Library," in *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, ser. GIS '09. New York, NY: ACM, 2009, pp. 538–539. [Online]. Available: http://doi.acm.org/10.1145/1653771.1653865

[20] M. H. Overmars, "Designing the Computational Geometry Algorithms Library CGAL," in *ACM Workshop on Applied Computational Geometry*, M. C. Lin and D. Manocha, Eds., Philadelphia, PA, May, 27–28 1996, Lecture Notes in Computer Science 1148.

[21] R. C. Veltkamp, "Generic Programming in CGAL, the Computational Geometry Algorithms Library," in *Proceedings of the 6th Eurographics Workshop on Programming Paradigms in Graphics, Budapest, Hungary, 8 September 1997*, 1997, pp. 127–138.

[22] J. Li, W.-k. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale, "Parallel netCDF: A High-Performance Scientific I/O Interface," in *Proceedings of Supercomputing 2003*, Phoenix, AZ, 2003.

[23] M. Folk, A. Cheng, and K. Yates, "HDF5: A File Format and I/O Library for High Performance Computing Applications," in *Proceedings of Supercomputing 1999*, Portland, OR, 1999.

[24] K. Coloma, A. Ching, A. Choudhary, R. Ross, R. Thakur, and L. Ward, "New Flexible MPI Collective I/O Implementation," in *Proceedings of Cluster 2006*, 2006.

[25] Y. Liu and J. Snoeyink, "A Comparison of Five Implementations of 3D Delaunay Tessellation," *Combinatorial and Computational Geometry*, vol. 52, pp. 439–458, 2005.

[26] S. Habib, A. Pope, Z. Lukić, D. Daniel, P. Fasel, N. Desai, K. Heitmann, C.-H. Hsu, L. Ankeny, G. Mark, S. Bhattacharya, and J. Ahrens, "Hybrid Petacomputing Meets Cosmology: The Roadrunner Universe Project," *Journal of Physics Conference Series*, vol. 180, no. 1, p. 012019, 2009.

[27] A. Pope, S. Habib, Z. Lukic, D. Daniel, P. Fasel, K. Heitmann, and N. Desai, "The Accelerated Universe," *Computing in Science Engineering*, vol. 12, no. 4, pp. 17 –25, July-Aug. 2010.

[28] S. Habib, V. Morozov, H. Finkel, A. Pope, K. Heitmann, K. Kumaran, T. Peterka, J. Insley, D. Daniel, P. Fasel, N. Frontiere, and Z. Lukic, "The Universe at Extreme Scale: Multi-Petaflop Sky Simulation on the BG/Q," in *Proceedings of SC12*, Salt Lake City, UT, 2012.

[29] S. Habib, V. Morozov, N. Frontiere, H. F. A. Pope, and K. Heitmann, "HACC: Extreme Scaling and Performance Across Diverse Architectures," in *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC13. New York, NY: ACM, 2013, pp. 6:1–6:10.

[30] A. S. Almgren, J. B. Bell, M. J. Lijewski, Z. Lukić, and E. Van Andel, "Nyx: A Massively Parallel AMR Code for Computational Cosmology," *The Astrophysical Journal*, vol. 765, pp. 39–52, 2013.

[31] C. L. Phillips, C. R. Iacovella, and S. C. Glotzer, "Stability of the Double Gyroid Phase to Nanoparticle Polydispersity in Polymer-Tethered Nanosphere Systems," *Soft Matter*, vol. 6, pp. 1693–1703, 2010.

[32] D. Q. Pike, F. A. Detcheverry, M. Mller, and J. J. de Pablo, "Theoretically Informed Coarse Grain Simulations of Polymeric Systems," *The Journal of Chemical Physics*, vol. 131, no. 8, pp. –, 2009. [Online]. Available: http://scitation.aip.org/content/aip/journal/jcp/131/8/10.1063/1.3187936

[33] R. L. Marson, C. L. Phillips, J. A. Anderson, and S. C. Glotzer, "Phase Behavior and Complex Crystal Structures of Self-Assembled Tethered Nanoparticle Telechelics," *Nano Letters*, 2014.

[34] K. Bowers, B. Albright, L. Yin, B. Bergen, and T. Kwan, "Ultrahigh Performance Three-Dimensional Electromagnetic Relativistic Kinetic Plasma Simulation," *Physics of Plasmas*, vol. 15, pp. 055 703–1–055 703–7, 2008.

[35] S. Byna, J. Chou, O. Rübel, Prabhat, H. Karimabadi, W. S. Daughton, V. Roytershteyn, E. W. Bethel, M. Howison, K.-J. Hsu, K.-W. Lin, A. Shoshani, A. Uselton, and K. Wu, "Parallel I/O, Analysis, and Visualization of a Trillion Particle Simulation," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA: IEEE Computer Society Press, 2012, pp. 59:1–59:12. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389077