pClay: A Precise Parallel Algorithm for Comparing Molecular Surfaces

Georgi D. Georgiev*
Department of Computer Science and Engineering, Lehigh University, Bethlehem, PA, USA

Kevin F. Dodd*
Department of Computer Science and Engineering, Lehigh University, Bethlehem, PA, USA

Brian Y. Chen[1]
Department of Computer Science and Engineering, Lehigh University, Bethlehem, PA, USA
chen@cse.lehigh.edu

Abstract

Comparing binding sites as geometric solids can reveal conserved features of protein structure that bind similar molecular fragments and varying features that select different partners. Due to the subtlety of these features, algorithmic efficiency and geometric precision are essential for comparison accuracy. For these reasons, this paper presents pClay, the first structure comparison algorithm to employ fine-grained parallelism to enhance both throughput and efficiency. We evaluated the parallel performance of pClay on both multicore workstation CPUs and a 61-core Xeon Phi, observing scaleable speedup in many thread configurations. Parallelism unlocked levels of precision that were not practical with existing methods. This precision has important applications, which we demonstrate: A statistical model of steric variations in binding cavities, trained with data at the level of precision typical of existing work, can overlook 46% of authentic steric influences on specificity ($p \leq .02$). The same model, trained with more precise data from pClay, overlooked 0% using the same standard of statistical significance. These results demonstrate how enhanced efficiency and precision can advance the detection of binding mechanisms that influence specificity.

2012 ACM Subject Classification Applied computing → Molecular structural biology; Computing methodologies → Volumetric models; Computing methodologies → Parallel algorithms

Keywords and phrases Specificity Annotation, Structure Comparison, Cavity Analysis

Digital Object Identifier 10.4230/LIPIcs.WABI.2019.6

1 Introduction

Molecular shape and electric fields have a strong influence on binding specificity. At binding interfaces, complementary molecular shapes can accommodate some ligands and hinder those that fit poorly. Electric fields attract molecules with complementing charges and repel others. This connection, between molecular recognition and the geometric complementarity of surfaces and fields, is evidence by which human investigators infer the roles of individual mechanisms in function. Comparison software can detect this kind of evidence and use it to make similar inferences. Some methods detect proteins with geometrically conserved binding sites, supporting the inference that they bind similar partners [12, 7, 4, 27, 33, 11, 15, 18]. Other methods find variations in the electric fields near binding sites, suggesting that they accommodate differently charged ligands [19, 5, 29, 34]. These techniques, and their potential for large scale and accurate applications, depend on rapid and precise digital representations of molecular shape, which are the focus of this paper.

Rapid and precise algorithms can integrate many observations to support inferences that are impossible with single comparisons. For example, a single comparison does not provide a

[1] Corresponding Author * Equal contribution

frame of reference that would be needed to assess whether or not two binding sites are different enough that they have different binding preferences. After all, conformational variations and single mutations can occur in many ways that change nothing about binding. This kind of inference is traditionally reserved for experts with a wealth of biochemical experience. However, statistical models can be trained on the steric differences between closely related ligand binding sites that prefer the same ligands. In such cases, structures of close homologs or even single mutants could provide the primary data, but the subtle variations needed to train the model would have to be found with many individual comparisons. Once trained, the statistical model provides a frame of reference that reveals steric variations that are too large to be typical of binding sites with the same binding preferences. The large variations found would therefore be indicators of binding sites that have different binding preferences [6]. To support and advance statistical models like these, this paper presents pClay, the first structure comparison algorithm that maximizes precision and computational throughput using arbitrary precision representations and parallel algorithms.

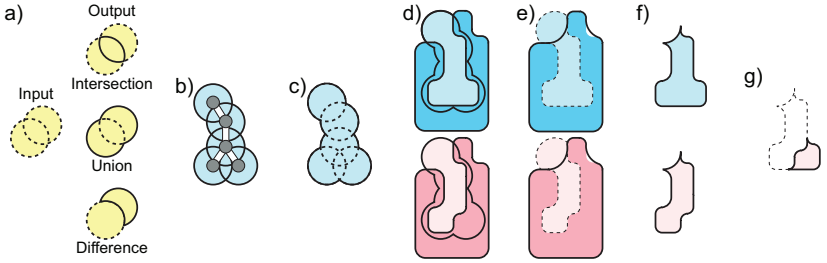

Figure 1 CSG operations on Molecular Surfaces. a) Basic CSG operations. Input solids are yellow with dotted outlines. Outputs have solid outlines. b) Ligand with grey atoms and white bonds, with spheres centered on each atom (light blue). c) The union of atom-centered spheres. d) Two molecular surfaces (blue, red) in complex with two ligands shown as sphere unions (black lines). e) CSG difference of the sphere unions minus molecular surfaces (dotted lines), shown with molecular surfaces (blue and red, no outline) and envelope surfaces (black outline). f) Intersection of differences with envelope surfaces (light blue and red). g) The CSG difference between binding cavities reveals a variation in steric hindrance that causes differences in binding preferences.

pClay performs geometric comparisons using Constructive Solid Geometry (CSG) operations (Fig. 1a) on analytically represented three dimensional solids. These operations, which include unions, intersections and differences, can be combined like arithmetic operators to sculpt a geometric solid. This sculptural nature of CSG inspires both the name pClay, a portmanteau of "protein" and "clay," and also the solid geometric approach to the analysis of protein shape that pClay makes possible. For example, the union of large spheres centered at ligand atoms can represent the neighborhood of a ligand (Fig. 1b,c). The difference between the spheres and the molecular surface of a receptor can describe the solvent-accessible binding cavity in the receptor (Fig. 1d,e). The CSG difference between one binding cavity and another is the cavity region that is solvent accessible in one protein and inaccessible in the other (Fig. 1g). This difference, the variation between the two cavities, could be small, when binding preferences are similar, or large, when steric hindrance creates differences in specificity.

The utility of these computations can be seen in multiple applications: When applying this approach to the S1 subsites of trypsins and elastases, we observed that it could identify threonine 226 which, in elastases, sterically hinders the longer substrates prefered by trypsins that might otherwise bind [8]. To illustrate the importance of precision, that region of hindrance is only 50 percent larger than a carbon atom (31 $Å^3$). A similar approach identified

"gatekeeper" residue 338 in the tyrosine kinases [14], which creates steric clash with larger drugs [22]. We have also observed that a CSG-based comparison of electrostatic isopotentials can reveal single amino acids crucial for selecting ligands in the in the cysteine proteases [5] and for stabilizing the three interfaces of the SMAD trimer [29]. Experimental validation has demonstrated the correctness of our prediction that arginine 235 forms critical electrostatic interactions for the activity of the ricin toxin [34]. By making CSG analysis possible on geometric solids that are exact, up to machine precision, pClay ensures that subtle but influential details cannot be overlooked.

The precision that pClay achieves derives from solids that have analytical representations, like spheres and tetrahedra. pClay can assemble these primitives into solvent excluded regions, which we call *molecular solids*. The boundary of a molecular solid is the classic molecular surface, also known as the solvent excluded surface or Connolly surface, which was originally developed by Richardson and others [21, 9]. While we can construct molecular solids with CSG operations on many individual primitives, pClay exploits molecular properties to sidestep those operations and achieve greater efficiency. The resulting molecular solids avoid the "photocopier effect", where multiple CSG operations can accumulate geometric errors. They can also be exported as triangle meshes, generated at an arbitrary degree of precision, for compatibility with other software.

pClay enhances computational efficiency through parallelism. We achieve parallelism in pClay in a number of ways, most notably by recasting Marching Cubes, a traditional method for implementing CSG operations [23, 17], into a series of parallel breadth first searches (BFS). In pClay, we use BFS to traverse cubic lattices and identify contiguous regions of cubes within defined boundary regions. These breadth first traversals can be distributed evenly across arbitrary numbers of threads. By dividing the computation in this way, parallelism can make comparisons faster and also enable more detail to be considered. This advancement stands in qualitative contrast with existing efforts to parallelize structure comparisons (e.g. [7]), where throughput was increased without enhancing precision. To demonstrate the parallel scalability of our method, pClay was tested on both modern multicore processors as well as on a Xeon Phi, a manycore processor with 61 cores.

Relative to existing methods, pClay is the first algorithm to use arbitrarily precise representations of molecular surfaces for protein structure comparison. It is also the first structure comparison method to use fine grained parallelization, enhancing both precision and computational throughput. Several methods do employ arbitrarily precise representations of the molecular surface, using NURBs [2], alpha shapes [32] or spherical coordinates [26, 28], but they are used for visualization and have not been integrated into comparison algorithms. Other methods parallelize structure comparison to refine representations of binding sites [7], to accelerate database searches [20], or create cloud-based search services [16], but use parallelism to enhance throughput and not also precision. To our knowledge, pClay is the first integration of arbitrary precision and parallelism into a structure comparison method.

2 Methods

As input, pClay accepts a collection of geometric solids and an expression of CSG operations. We convert the CSG expression into a binary tree, a CSG tree, where the nodes of the tree are geometric solids. The input solids, which include spheres, spindles, tetrahedra or molecular surfaces, are leaves on the CSG tree, while the result of CSG operations are the nonleaf nodes. The final result of all operations, the root node, is the output. pClay can also generate a closed triangular mesh at user-defined resolutions to approximate the boundary

of the output.

To perform CSG operations, pClay implements a parallel version of Marching Cubes [23] (Section 2.1), which we summarize below. Our method requires three basic functions to be performed by every node in the CSG tree. These functions are *containsPoint()*, *intersectSegment()*, and *findSurfaceCubes()*. Given any point p in three dimensions, containsPoint(p) determines exactly if p is inside or outside the solid. A point exactly on the surface is said to be inside the solid. Second, given a line segment s, intersectSegment(s) determines all points of intersection between the surface of the operand and s, as well as the interior or exterior state of each interval on the segment. Finally, given a cubic lattice l that surrounds the primitive, findStartingCubes(l) finds a few cubes of the lattice that are *surface cubes*, having at least one corner inside and one corner outside the solid. These cubes are used to initiate a parallel breadth first search for all surface cubes, called *findAllSurfaceCubes()*, which is implemented once for all primitives (Section 2.2). To implement leaf nodes it is thus sufficient to describe how these basic functions are implemented for that solid. Nonleaf nodes implement the basic functions as logical operations, as we will explain in Section 2.3.

Below, we first describe how the output approximations are generated using a parallelization of Marching Cubes and how we find all surface cubes beginning the output from the starting cubes generated by the basic function. We next explain how the three basic functions are implemented for every primitive. Finally, we detail how the basic functions are implemented in nonleaf nodes.

2.1 Parallel Marching Cubes

As input, Marching Cubes accepts a set of geometric solids (Fig. 2a), which we will refer to as operands, and a CSG expression tree to be performed on the operands. It also accepts a resolution parameter in angstrom units that specifies the degree to which the result of the CSG expression should be approximated in the output.

We begin by defining an axis aligned cubic lattice surrounding the input operands, where each cube has sides equal to the user-specified resolution parameter (Fig. 2b). This step is performed by examining the sizes of all operands and the related CSG operations.

Once the lattice is defined, we invoke findStartingCubes(l) on each input solid (Fig. 2c,f). The surface cubes identified are provided as input to findAllSurfaceCubes(), which identifies all remaining surface cubes of all inputs solids in parallel (Fig. 2h). The process of identifying surface cubes for all input solids also necessarily determines the interior/exterior state of the points on these cubes in relation to specific solids. We then compute the interior/exterior state of these points in relation to all other solids in an embarrassingly parallel manner. Once this assessment is made for any point, we can access whether that point is inside or outside the output region (Fig. 2i). In this way, we find the subset of cubes that contain a corner inside and a corner outside the output region.

Next, on each cube of the output surface, we identify edges that connect one corner that is inside the output region to one that is outside (Fig. 2j). Since these edges must pass through the output surface, we call segIntersect() on the root node to find the point of intersection between the edge and the output surface (Fig. 2k). This process is parallelized across the list of edges, ensuring that the calculation is never duplicated when dealing with adjacent cubes.

Finally, once intersections for every edge on every surface cube are determined, triangles are generated in each cube following a lookup table (Fig. 2l). The collection of all resulting triangles form a closed triangular mesh that approximates the output region (Fig. 2m,n).

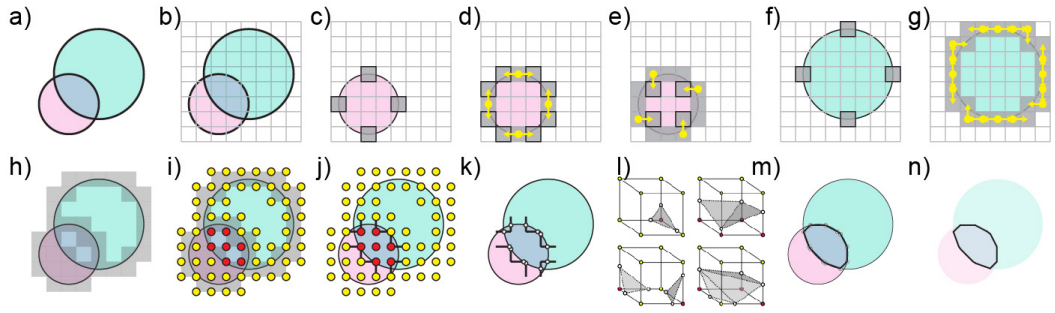

Figure 2 a) Input operands (red, green). b) Cubic lattice around operands (gray). c,f) surface cubes (gray boxes). d,e,g) several steps of floodfill propagation (starting at yellow circle, following yellow arrow). i) Corner points of each surface cube with exterior (yellow) or interior (red) state. j) Segments that cross the boundary of the output surfaces (Black lines). k) Intersection points (white circles) segments intersect the output surface. l) Lookup table of 3D surface constructions with different edge intersection patterns. m,n) Triangles (black lines) approximating output region (gray).

2.2 Finding All Surface Cubes

findAllSurfaceCubes() accepts a cubic lattice (Fig. 2b), a list of starting cubes (e.g. Fig. 2c,f), and a CSG tree node for which to find all remaining surface cubes. We perform a parallel floodfill algorithm to find all remaining surface cubes: Each available thread is assigned a cube from the queue. Each thread tests cubes adjacent to the assigned cube to find any that are also on the surface of the input solid (Fig. 2d). This test is performed by calling containsPoint() on the corners of the adjacent cube. If at least one corner is inside the input solid and another corner is outside, the adjacent cube is stored on a queue of upcoming cubes. Once all cubes adjacent to the initial surface cubes have been either added to the queue or discarded, all threads are then directed to find cubes adjacent to those still on the queue (e.g. Fig. 2e), and so on, until the queue is empty, and all cubes on the surface of the input solid have been identified. Duplicate entries onto the queue are avoided by recording previously-examined cubes on a parallel hash table.

2.3 Nodes of the CSG Tree

pClay supports several kinds of simple and complex solids for CSG operations. These are spheres, tetrahedra, spindles, and molecular surfaces. Our implementation of each type supports three basic functions: containsPoint(), intersectSegment(), and findSurfaceCubes(). To describe the implementation of these solids, we describe how each method is implemented for the solid. Spheres and tetrahedra are excluded because their implementations are trivial.

Spindles. Spindles (Fig. 3a) define the solvent excluded region between two atoms that are too close to permit a sphere representing a solvent molecule to pass between them (Fig. 3b). "Broken" spindles (Fig. 3c) can occur when the edge of the solvent sphere can pass beyond the centerline of the two atoms. Conceptually, spindles are the volume within a cylinder minus the volume within a coaxial torus. We define spindles by center point, perpendicular vector, major radius, and minor radius taken from the torus (Fig. 3d), and end cap positions along the perpendicular vector (Fig. 3e). The center point is the perpendicular projection of the center of the solvent sphere onto the segment between atom centers (Fig. 3b). The perpendicular vector points from the center point towards the center of one atom. The major radius is the radius of the circle defined by the center of the solvent sphere as it rotates

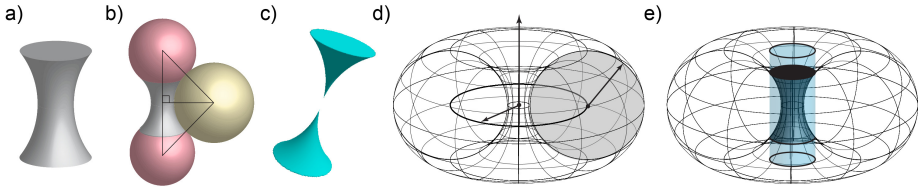

Figure 3 a) Spindle. b) Formation of a spindle (gray) from two atoms (red) and a solvent sphere (yellow). c) "Broken" spindle. d) Torus defining the characteristics of a spindle, including center point (black dot), perpendicular vector (vertical arrow), major radius (arrow from center point to ellipse), minor radius (arrow from ellipse to torus surface). e) Cylinder (light blue).

about the two atoms. The minor radius is the radius of the solvent sphere. The endcaps are circles perpendicular to the perpendicular vector that are defined by the point of tangency between the solvent sphere and the atoms, as the solvent sphere rotates about the atoms. The boundary surface of a spindle is defined by the end caps and elsewhere by the interior curve of the torus (Fig. 3d).

To implement containsPoint(p), note that the spindle is rotationally symmetric about the perpendicular vector. Thus, a plane K can be defined coplanar to p and the perpendicular vector of the torus. In K, p is inside the spindle only if it is inside the rectangle that defines the rotational cross section of the cylinder and also outside the circle that defines the rotational cross section of the torus.

intersectSegment(s) is computed by first setting up the calculation by translating the center of the spindle to the origin and rotating its axis to align it with the x axis. s is translated and rotated with it. We can describe the torus aligned to the x axis as

$$(x^2 + y^2 + z^2 + R^2 - r^2)^2 - 4R^2(y^2 + z^2) = 0$$

where R is the major radius, and r is the minor radius of the torus. In the torus equation, we substitute x, y and z with the line expressions $x_0 + td_x$, $y_0 + td_y$, and $z_0 + td_z$, where x_0, y_0, z_0 are segment starting points, and t parameterizes the line containing the line segment. The result of this substitution is a quartic equation on t, and roots of the equation will be parameters on the segment at points of intersection between the segment and the torus. We converted this equation into a monic quartic using Maxima, a computer algebra system [25].

To find the roots of this equation, we produce the Frobenius companion matrix of this quartic polynomial. The roots are the eigenvalues of this matrix. Here, complex eigenvalues will correspond to nonexistent points of intersection between the segment and the torus while real eigenvalues correspond to intersection points on the torus. We find these intersection points and eliminate any intersections that are outside of the cylinder. Separately, we also find intersections with the end caps of the spindle, treating them first as infinite planes and then determining if the intersection point is within the circle on the plane. Intersections between the segment and the endcaps or between the segment and the torus are returned as intervals where the segment is inside the spindle.

findStartingCubes() is implemented by first generating the segment between the endcap centers. The lattice cube containing one center is identified, and if it is not a surface cube, the adjoining cube, through whose face which the segment passes, is identified as the next cube to examine. This process is repeated along the segment until the segment ends at the other encap center or a surface cube has been found. In the case where the spindle is broken (Fig. 3d), two segments are generated, with the same process performed on each segment starting at the center of each endcap.

Molecular Solids. pClay generates molecular solids from unions and differences of solids, such as spheres, spindles, and tetrahedra, that are positioned with the power diagram [1, 10]. This approach follows the classic methods for generating molecular surfaces, such as CASTp [32], MSMS [31], GRASP2 [30], which also use power diagrams or similar constructs like alpha shapes. For this reason, we paraphrase our approach here, expanding on points that differ from classic methods. As in the earlier methods, our approach represents water molecules as solvent spheres, which can be of any given radius. By calling basic functions from simpler primitives, pClay achieves an efficient implementation of the basic functions for the entire molecular solid without describing it as a CSG operation of many individual primitives.

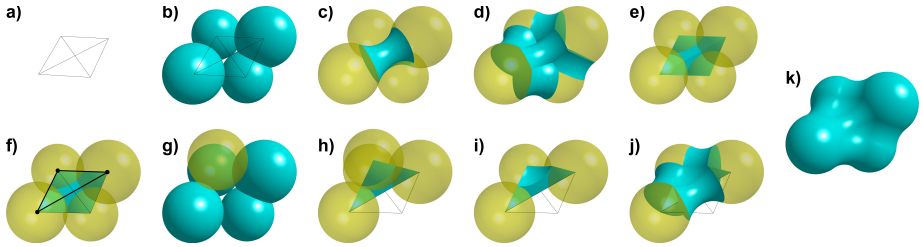

Figure 4 Molecular Surface Construction. a) Dual graph of a power diagram on four atoms (black lines, points). b) Sphere primitives from atoms (teal). c) Atoms (yellow) with one spindle (teal). d) Atoms with all spindles from edges of the dual graph. e) Tetrahedron primitive (teal). f) One triangle of the dual graph (black lines, dots). g) Solvent sphere (yellow) tangent to three atoms. h) New tetrahedron (teal) with corners in the center of the three atoms of the triangle and the solvent sphere. i) Cup region inside the new tetrahedron (teal). j) Cup, shown with three adjacent spindles (teal) and three atoms of the triangle (yellow). k) Finished molecular solid.

We begin with an input file from the Protein Data Bank (PDB). Using atomic coordinates and Van der Waals radii for each atom, we first compute a power diagram with REGTET [3]. The power diagram divides three dimensional space into cells corresponding to each atom of the input. The size of a cell relates to the Van der Waals radius of the atom, through the power function. Using the power diagram, we construct a topologically dual geometric graph (Fig. 4a), which has a vertex at the center of each atom and an edge between any vertices that correspond to adjacent cells. This *dual graph* defines the location of the primitives that will comprise the molecular solid. In sequential stages, we generate all primitives of the same type in parallel, starting with sphere primitives, then spindles, tetrahedra, and so on.

At every vertex of the dual graph, we create sphere primitives with the appropriate Van der Waals radius of each atom (Fig. 4b). Next, we examine every edge on the dual graph and generate a spindle between the atoms on at the endpoint of each edge, except for overlong edges that are longer than the sum of Van set Waals radius of the endpoint atoms and the diameter of the solvent sphere (Fig. 4c,d). Once all spindles are completed, we identify all tetrahedra in the dual graph that lack an overlong edge and we generate a tetrahedron primitive for each one (Fig. 4e).

Next, we identify triangles on the dual graph that are not between two tetrahedra (Fig. 4f). These triangles define triplets of atoms that may be on the molecular surface. To determine whether the atoms are on the surface, we place a solvent sphere tangent to all three atoms (Fig. 4g). If the solvent sphere does not collide with any other atoms, we create a *negsphere*: a sphere primitive in the tangent location in the same size as the solvent that describes a region of the solvent outside the molecular surface. We also generate a tetrahedron with corners on the triangle and at the center of the negsphere (Fig. 4h). The region inside this tetrahedron and outside the negsphere is both inside the solvent excluded

region and not occupied by spindles or atoms or other tetrahedra. We call this concave subset of a tetrahedron a *cup* (Fig. 4i), and describe cups as a negsphere-tetrahedron pair. The concave surface of the cup is continuous with the three adjacent spindles and atoms (Fig. 4j). Once all triangles that are not between two tetrahedra have been examined for the presence of a cup, the combination of spheres, spindles, tetrahedra and negspheres form a molecular solid (Fig. 4k).

To support the three basic functions, we store all of these primitives in a data structure for rapid range-based lookup. First, we generate a bounding box for each primitive. Next, we generate a lattice of cubes, where each cube is 2 angstroms on a side. Finally, we associate each primitive with all lattice cubes that intersect its bounding box. These associations act as a hashing function that enables us to rapidly identify any primitives nearby a given cube in the lattice. Since real molecules have finite atomic density, and since primitives are constructed from atoms and between atoms, the number of primitives associated with any cube is finite. As a result, a hashing function based on the lattice achieves algorithmically constant time lookup of nearby primitives.

For containsPoint(p), given a point p, if p is outside the coarse lattice, then we immediately return false, because p must be outside the molecular surface. If not, we determine which cube c of the coarse lattice contains p. Next, we identify all primitives associated with c. We use the containsPoint() function of each associated primitive to determine if p is inside the primitive. If p is inside a negSphere, then p is outside the molecular surface. if p is inside any other primitives, then p is considered inside the molecular surface. If p is not inside any primitives, it is outside.

For intersectSegment(s), given a segment s, we generate a list of cubes C that contain some interval of s. Next, we generate a list of primitives P associated with the cubes in C. We then query each primitive p in the list P for an interval of intersection between p and s using the intersectSegment() method of each primitive. The output intervals generated are the union of the intervals in tetrahedra, spindles and spheres minus the union of intervals inside negspheres.

For findStartingCubes(l), during the construction of the molecular solid, we record the points of tangency between all negspheres and atom spheres. For each of these points, we identify the lattice cubes of l that contain them. We also generate starting cubes from all spindles and isolated spheres in the protein structure, calling findStartingCubes() on each of these primitives. From these cubes, we return only cubes that exhibit at least one corner inside and at least one corner outside the molecular solid.

CSG Operations. CSG operation nodes are non-leaf nodes that represent the outcome of a CSG operation on its operand nodes. They fulfill the three basic functions by calling on its operand nodes, which we refer to as A and B in the text below.

Given a point p, the CSG union returns true only when containsPoint(p) returns true on at least one operand. The CSG intersection returns true only when containsPoint(p) returns true on both operands. The CSG difference between A and B returns true only when A.containsPoint(p) is true and B.containsPoint(p) is false.

For a given segment s, intersectSegment(s) on any CSG operation calls intersectSegment(s) on operands A and B, generating intervals a and b. When run on a CSG Union, Intersection or Difference, the output is, respectively, the union, intersection, or difference of a and b.

Given a cubic lattice l, calling getSurfaceCubes(l) on a CSG union, intersection, or difference returns the setwise union of cubes returned by calling A.getSurfaceCubes() and B.getSurfaceCubes(). We always return a union of cubes because examining the union

of cubes can avoid circumstances where a disconnected region in the final solid is lost. Performance profiling revealed that considering the union of all cubes is a minor cost in overall performance, except in artificially constructed cases that create many irrelevant cubes.

Implementation Details. pClay is implemented in C and C++. Interprocess communication was built with Intel's Threading Building Blocks library. A C wrapper connects REGTET [3] to pClay. Benchmarks were run on a dual Xeon E5-2609 system with 8 cores at 2.5 Ghz and 32GB of ram and on a Xeon Phi 7120P with 61 cores at 1.2 Ghz and 16GB ram.

Datasets Used. *Dataset A* is 100 nonredundant pdb structures from VAST [24], with BLAST p-value cutoff 10e-7. *Dataset B* is 30 spheres, spindles and tetrahedra randomly generated in a cube with 10Å sides. *Dataset C* is 14 binding cavities from a nonredundant subset of the trypsins (1a0j,1aks,1ane,1aq7,1bzx,1fn8,1hrw,1trn,2eek,2f91), chymotrypsins (1eq9,8gch) and elastases (1b0e,1elt). More info: `http://www.cse.lehigh.edu/~chen/papers/WABI2019/appendix.pdf`

3 Experimental Results

3.1 Accuracy of molecular solid generation

We compared the surfaces of molecular solids from pClay to surfaces made with the trollbase library, an established tool for molecular surface generation used in GRASP2 [30], VASP [8], VASP-E [5], and MarkUs [13]. pClay surfaces were created at 0.25Å resolution to yield a similar number of triangles and thus a fairer comparison. Using proteins from dataset A, surfaces generated by pClay had an average of 197,718.54 points. From these points, we measured the distance to the closest point on the surface generated by trollbase for the same protein. That distance averaged 0.00383 Å (sd 0.0004 Å, max .22024 Å). This tiny deviation, on surfaces built from thousands of atoms, demonstrates the accuracy of pClay surfaces.

3.2 Performance Comparison and Parallel Scaling

To our knowledge, VASP [5] is the only existing algorithm for comparing molecular solids using CSG. It lacks exact primitives or parallelism, but it can identify steric elements of protein structure that control specificity [8, 6, 14]. We compared the performance of pClay and VASP on the same CSG operations using Xeon (CPU) and Xeon Phi (PHI) processors.

Random Primitives. First, we compared CSG performance on the union of primitives in dataset B, generating mesh outputs at resolutions 1.0Å, .5Å, .25Å and .125Å. Since VASP does not use primitives, it was provided triangle meshes of identical primitives. All CSG trees were balanced, but imbalanced trees had nearly identical runtimes (not shown for brevity).

On one CPU core, pClay required .113 seconds to compute the union at 1.0Å resolution. 9.492 seconds were required to compute the same union at .125Å resolution (Fig. 5a). Increasing to 8 threads, runtime dropped to .03 seconds for unions at 1.0Å resolution, and 1.465 seconds to at .125Å. In contrast, single-threaded VASP required 3 seconds to compute the same union at 1.0Å , and 64 at .125Å. pClay far outperformed VASP on one thread.

On 8, 16, 32, and 60 PHI cores, which are slower than CPU cores, runtimes exhibited sublinear improvement (Fig. 5b). Runtimes on coarser resolutions improved less than for finer resolutions. The difference in parallel speedup (Fig. 5c) arises from small problem

sizes at coarse resolutions, where communications and setup outweigh the advantages of parallelism.

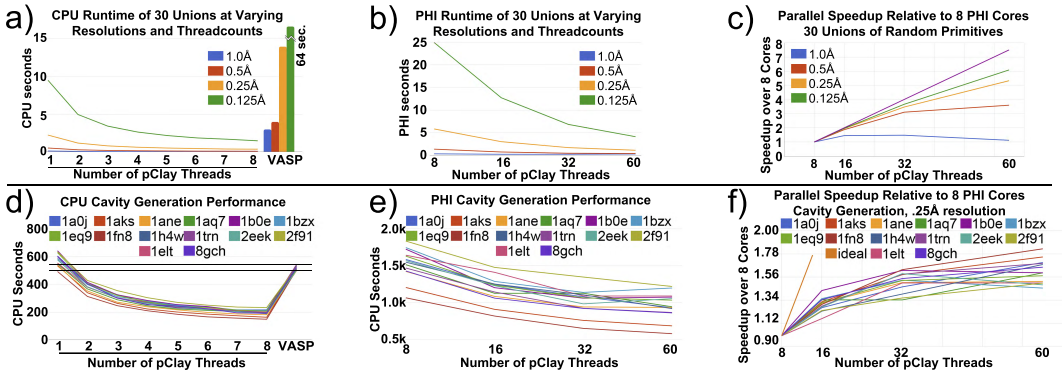

Figure 5 a) Time to compute the union of 30 random primitives at varying resolutions and CPU cores. VASP performance (single threaded) is shown in vertical bars. b) Time spent to compute the unions on PHI cores. c) Parallel speedup on PHI cores. d) Time spent for pClay to produce several binding cavities on CPU cores, compared to single-core VASP. e) Time to produce the same cavities on PHI cores. e) Parallel speedup of pClay in cavity production on varying PHI cores.

Binding Cavities. We also tested pClay by generating binding cavities, using the method from Fig. 1, on dataset C. While pClay is capable of much finer resolutions, all cavities were generated at .25Å, the practical resolution limit for VASP. Figure 5d plots cavity generation times for these cavities. pClay required between 493 and 643 seconds on one CPU core, and between 149 and 233 seconds on 8 cores. Single threaded VASP required between 499 and 538 seconds to perform the same work. Single threaded, pClay was slightly slower than VASP, but much faster when adding a second core, and faster still when adding more.

Cavity generation was also run on 8, 16, 32, and 60 Xeon Phi cores. Runtimes on PHI cores are slower than on CPU cores because PHI cores have slower clock speeds. Runtimes fell slowly as threads increased (Fig. 5e). Substantial increases in the number of PHI cores resulted in only modest improvements in runtimes (Fig. 5f). This result contrasted from those performed on CPU cores, where performance improved substantially with increases in parallelism. These results point to bottlenecks in the PHI architecture affected by cavity generation, which is more data intensive than unions of random primitives.

3.3 Evaluating pClay on Existing Applications

The added precision of pClay creates several new applications. We demonstrate one such application by producing training data for VASP-S, a statistical model for detecting differences in ligand binding specificity with steric causes [6]. VASP-S is trained on the volumes of individual CSG differences computed from cavities with the same binding preference. This training enables VASP-S to estimate the probability (the p-value) that two given cavities have similar binding preferences. If p is lower than a threshold α, VASP-S rejects the hypothesis that two cavities have similar binding preferences, and predicts that they are different.

We hypothesized that training the VASP-S model with data generated at finer resolutions will produce more accurate predictions than a VASP-S model trained with coarser data. To test this hypothesis, we used cavities from the trypsins, which prefer to bind positively charged amino acids. These cavities contrast from those of the chymotrypsins and the elastases, which prefer large aromatics or small hydrophobics, respectively. Three training

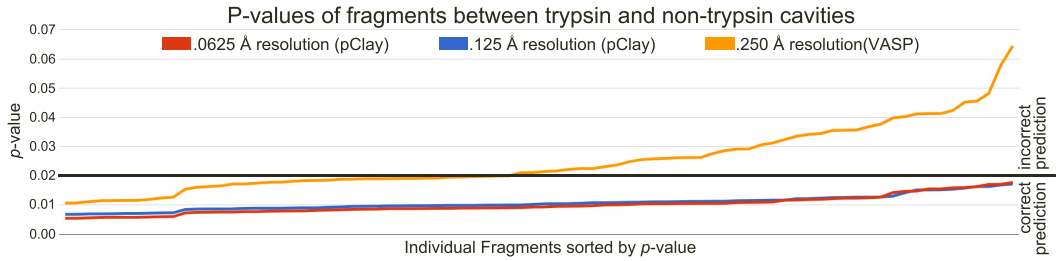

Figure 6 The p-value of the largest fragment between every trypsin-elastase and trypsin-chymotrypsin pair in Dataset C, estimated with training data generated at several resolutions (red, blue, orange lines). Fragments are sorted in ascending p-values along the horizontal axis. The black line indicates the α threshold of 0.02, below which we predict that the fragment is representative of proteins with different binding preferences. The finer-resolution training data, made possible with pClay, yielded more accurate predictions.

sets were constructed from these cavities by generating all possible CSG differences between all pairs of trypsins. VASP was used to produce a copy of the training set at 0.25 Å resolution and pClay was used to produce the same set at resolutions of 0.125 Å and .0625 Å. We then computed CSG differences between every trypsin and every nontrypsin at these resolutions. Finally, we estimate the p-value of the largest CSG difference between every trypsin and every non-trypsin in Dataset C, at all three resolutions (Fig. 6). We expect VASP-S to produce a low p-value on these CSG differences.

Using the conservative α threshold of 2%, when trained at 0.25 Å resolution, VASP-S predicts that 43 of the 81 CSG differences between trypsin and non-trypsin cavities indicate binding preferences. This discrepancy indicates 38 false negative predictions, where VASP-S incorrectly overlooked cavities with different binding preferences. However, when trained at 0.125 Å or 0.0625 Å resolution, VASP-S correctly predicts that all CSG differences were from cavities with different binding preferences, a 0% false negative rate. These results demonstrate that pClay can provide superior precision, ensuring that existing aggregate methods do not lose accuracy by overlooking useful predictions.

4 Discussion

We have presented pClay, the first parallel algorithm for performing CSG analysis of protein structures at arbitrarily high resolutions, up to machine precision. It leverages mathematically exact primitives that can be assembled into molecular solids and parallel depth first search to compute CSG operations with multiple threads.

Molecular solids from pClay are nearly identical to molecular surfaces generated by existing, widely used software. At hundreds of thousands of positions, pClay surfaces differed from surfaces generated by existing methods by only thousandths of an angstrom. While existing surface methods have generally been validated by visual examination, this exhaustive comparison sets a new standard for validation. Surface generation stresses the algorithms that underpin CSG operations, illustrating that pClay is making accurate comparisons.

We showed that pClay performs CSG operations efficiently on both artificial and realistic data. Evaluating the method on both Xeon CPU and Xeon Phi architectures, pClay exhibited scaleable multithreaded performance on all tests, though scaling was modest on the Xeon Phi for cavity generation. These results show that parallelism can drive both efficiency and precision for the comparison of protein structures.

Finally, we showed how the precision of pClay can advance existing methods by training a statistical classifier to distinguish elements of protein structures that have a steric influence on binding specificity. pClay provided training data to the classifier that was more precise than what could have been provided by existing methods, enabling more accurate estimates of statistical significance, and ultimately a total elimination of false negative predictions.

These capabilities enable applications in the detection and explanation of structural features that influence binding preferences. For example, the statistical model tested here finds elements of protein structures that could have a steric influence on specificity, thereby generating an explanation based on a steric mechanism that typically requires human expertise. As high throughput technologies increasingly reveal the ways in which disease proteins can vary, pClay is a glimpse into a new space of techniques that can use protein variants to supplement human experience in deciphering the structural mechanisms of molecular recognition.

References

1 F Aurenhammer. Power diagrams: properties, algorithms and applications. *SIAM Journal on Computing*, 16(1):78–96, 1987.

2 CL Bajaj, V Pascucci, A Shamir, RJ Holt, and AN Netravali. Dynamic maintenance and visualization of molecular surfaces. *Discrete Applied Mathematics*, 127(1):23–51, 2003.

3 J Bernal. REGTET: A program for computing regular tetrahedralizations. In *International Conference on Computational Science*, pages 629–632. Springer, 2001.

4 M Brylinski and J Skolnick. A threading-based method (FINDSITE) for ligand-binding site prediction and functional annotation. *P Natl Acad Sci USA*, 105(1):129–134, 2008.

5 BY Chen. VASP-E: Specificity annotation with a volumetric analysis of electrostatic isopotentials. *PLoS computational biology*, 10(8):e1003792, 2014.

6 BY Chen and S Bandyopadhyay. VASP-S: A volumetric analysis and statistical model for predicting steric influences on protein-ligand binding specificity. In *IEEE Int Conf Bioinformatics Biomed 2011*, pages 22–29. IEEE, 2011.

7 BY Chen, VY Fofanov, DH Bryant, BD Dodson, DM Kristensen, AM Lisewski, M Kimmel, O Lichtarge, and LE Kavraki. The MASH pipeline for protein function prediction and an algorithm for the geometric refinement of 3D motifs. *J Comput Biol*, 14(6):791–816, 2007.

8 BY Chen and B Honig. VASP: a volumetric analysis of surface properties yields insights into protein-ligand binding specificity. *PLoS computational biology*, 6(8):e1000881, 2010.

9 ML Connolly. Analytical molecular surface calculation. *Journal of applied crystallography*, 16(5):548–558, 1983.

10 Herbert Edelsbrunner and Ernst P Mücke. Three-dimensional alpha shapes. *ACM Transactions on Graphics (TOG)*, 13(1):43–72, 1994.

11 L Ellingson and J Zhang. Protein surface matching by combining local and global geometric information. *PloS one*, 7(7):e40540, 2012.

12 F Ferre, G Ausiello, A Zanzoni, and M Helmer-Citterich. Functional annotation by identification of local surface similarities: a novel tool for structural genomics. *BMC Bioinf*, 6(1):194, 2005.

13 M Fischer, QC Zhang, F Dey, BY Chen, B Honig, and D Petrey. MarkUs: a server to navigate sequence–structure–function space. *Nucleic acids research*, 39(suppl_2):W357–W361, 2011.

14 BG Godshall and BY Chen. Improving accuracy in binding site comparison with homology modeling. In *IEEE Int Conf Bioinformatics Biomed 2012*, pages 662–669. IEEE, 2012.

15 L He, F Vandin, G Pandurangan, and C Bailey-Kellogg. Ballast: a ball-based algorithm for structural motifs. *Journal of Computational Biology*, 20(2):137–151, 2013.

16 C-L Hung and Y-L Lin. Implementation of a parallel protein structure alignment service on cloud. *International journal of genomics*, 2013, 2013.

17 T Ju, F Losasso, S Schaefer, and J Warren. Dual contouring of hermite data. In *ACM transactions on graphics (TOG)*, volume 21 (3), pages 339–346. ACM, 2002.

18 F Kaiser, A Eisold, and D Labudde. A novel algorithm for enhanced structural motif matching in proteins. *Journal of Computational Biology*, 22(7):698–713, 2015.

19 K Kinoshita, Y Murakami, and H Nakamura. eF-seek: prediction of the functional sites of proteins by searching for similar electrostatic potential and molecular surface shape. *Nucleic acids research*, 35(suppl_2):W398–W402, 2007.

20 J Konc, M Depolli, R Trobec, K Rozman, and D Janežič. Parallel-ProBiS: Fast parallel algorithm for local structural comparison of protein structures and binding sites. *Journal of computational chemistry*, 33(27):2199–2203, 2012.

21 B Lee and FM Richards. The interpretation of protein structures: estimation of static accessibility. *Journal of molecular biology*, 55(3):379–IN4, 1971.

22 Y Liu, K Shah, F Yang, L Witucki, and KM Shokat. A molecular gate which controls unnatural ATP analogue recognition by the tyrosine kinase v-Src. *Bioorg Med Chem*, 6(8):1219–1226, 1998.

23 WE Lorensen and HE Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. In *ACM siggraph computer graphics*, volume 21 (4), pages 163–169. ACM, 1987.

24 Thomas Madej, Christopher J Lanczycki, Dachuan Zhang, Paul A Thiessen, RC Geer, A Marchler-Bauer, and SH Bryant. MMDB and VAST+: tracking structural similarities between macromolecular complexes. *Nucleic acids research*, 42(D1):D297–D303, 2013.

25 WA Martin and RJ Fateman. The MACSYMA system. In *Proceedings of the second ACM symposium on Symbolic and algebraic manipulation*, pages 59–75. ACM, 1971.

26 NL Max and ED Getzoff. Spherical harmonic molecular surfaces. *IEEE Computer Graphics and Applications*, 8(4):42–50, 1988.

27 EC Meng, BJ Polacco, and PC Babbitt. 3D Motifs. In *From Protein Structure to Function with Bioinformatics*, pages 187–216. Springer, 2009.

28 RJ Morris, RJ Najmanovich, A Kahraman, and JM Thornton. Real spherical harmonic expansion coefficients as 3D shape descriptors for protein binding pocket and ligand comparisons. *Bioinformatics*, 21(10):2347–2355, 2005.

29 BE Nolan, E Levenson, and BY Chen. Influential Mutations in the SMAD4 Trimer Complex Can Be Detected from Disruptions of Electrostatic Complementarity. *Journal of Computational Biology*, 24(1):68–78, 2017.

30 D Petrey and B Honig. GRASP2: visualization, surface properties, and electrostatics of macromolecular structures and sequences. *Methods in enzymology*, 374:492–509, 2003.

31 MF Sanner, AJ Olson, and J-C Spehner. Reduced surface: an efficient way to compute molecular surfaces. *Biopolymers*, 38(3):305–320, 1996.

32 W Tian, C Chen, and J Liang. CASTp 3.0: Computed Atlas of Surface Topography of Proteins and Beyond. *Biophysical Journal*, 114(3):50a, 2018.

33 J Venkateswaran, B Song, T Kahveci, and C Jermaine. TRIAL: A Tool for Finding Distant Structural Similarities. *IEEE/ACM Trans Comput Biol Bioinform*, 8(3):819–831, 2011.

34 Y Zhou, X-P Li, BY Chen, and NE Tumer. Ricin uses arginine 235 as an anchor residue to bind to P-proteins of the ribosomal stalk. *Scientific reports*, 7:42912, 2017.