Eurographics Symposium on Parallel Graphics and Visualization (2011)
T. Kuhlen, R. Pajarola, and K. Zhou (Editors)

Data-Parallel Mesh Connected Components Labeling and Analysis

C. Harrison[1], H. Childs[2], and K.P. Gaither[3]

[1]Lawrence Livermore National Laboratory
[2]Lawrence Berkeley National Laboratory / University of California, Davis
[3] The University of Texas at Austin (Texas Advanced Computing Center)

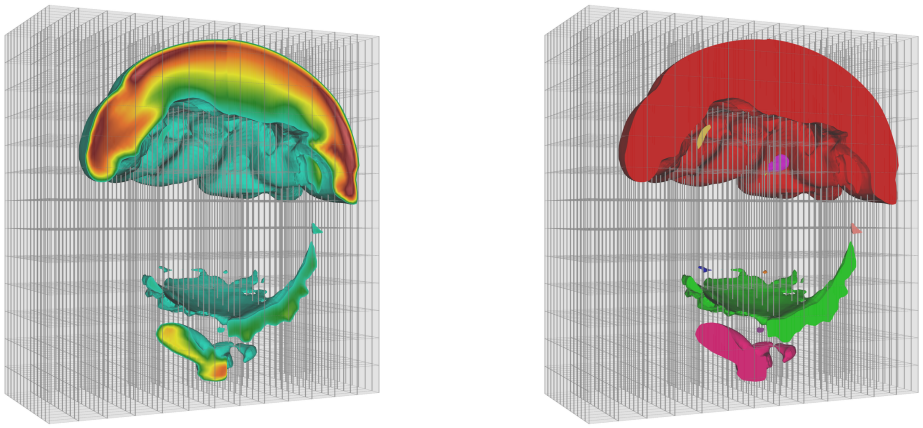


Figure 1: *(Left) Sub-volume mesh extracted from a 21 billion cell structured grid decomposed across 2197 processors. (Right) Sub-volume mesh colored by result from our connected components labeling algorithm.*

Abstract

We present a data-parallel algorithm for identifying and labeling the connected sub-meshes within a domain-decomposed 3D mesh. The identification task is challenging in a distributed-memory parallel setting because connectivity is transitive and the cells composing each sub-mesh may span many or all processors. Our algorithm employs a multi-stage application of the Union-find algorithm and a spatial partitioning scheme to efficiently merge information across processors and produce a global labeling of connected sub-meshes. Marking each vertex with its corresponding sub-mesh label allows us to isolate mesh features based on topology, enabling new analysis capabilities. We briefly discuss two specific applications of the algorithm and present results from a weak scaling study. We demonstrate the algorithm at concurrency levels up to 2197 cores and analyze meshes containing up to 68 billion cells.

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computational Geometry and Object Modeling]: Geometric algorithms, languages, and systems—

1 Introduction

Parallel scientific simulations running on today's state of the art petascale and terascale computing platforms generate massive quantities of high resolution mesh-based data. Scientists can analyze this data by eliminating portions of the data and visualizing what remains, through operations such

as isosurfacing, selecting certain materials and discarding the others, isolating hot spots, etc. In the context of massive data, these approaches can generate complex derived geometry with intricate structures that require further techniques to effectively analyze.

In these instances, representations of the topological structure of a mesh can be helpful. For example, a labeling of the connected components in a mesh provides a simple and intuitive topological characterization of which parts of the mesh are connected to each other. These unique sub-meshes contain a subset of cells that are directly or indirectly connected via series of cell abutments.

The global nature of connectivity poses a challenge for distributed memory computers, which are the most common resource for analyzing massive data. In this setting, pieces of the mesh are distributed across processors since the entire data set is too large to fit into the memory of a single processor. Cells comprising connected sub-meshes may span any of the processors, but the information on which cells abut across processors may not be available. These factors constrain the approaches available to resolve connectivity.

Data-parallel algorithms have been developed for graphs, but these algorithms fail to take advantage of optimizations inherent to mesh-based scientific data. Further, data-parallel algorithms specifically for mesh-based scientific data have also been developed. However, these algorithms focus on regular grids and their performance has only been studied at low levels of concurrency with modest data sets. This study presents a new mesh-based algorithm, one that operates on both structured and unstructured meshes and scales well even for very large data. Our multi-stage algorithm constructs a unique label for each connected component and marks each vertex with its corresponding connected component label. The final labeling enables analysis such as:

- Calculation of aggregate quantities for each connected component.
- Feature based filtering of connected components.
- Calculation of statistics on connected components.

In short, the algorithm provides a useful tool for domain scientists with applications where physical structures, such as individual fragments of a specific material, correspond to the connected components contained in a simulation data set. This paper presents an outline of the algorithm (Section 4), an overview of two specific applications of the algorithm (Section 5), and results from a weak scaling performance study (Section 6).

2 Related Work

Research into connected components algorithms has focused primarily on applications in computer vision and graph theory. Several efficient serial algorithms have been developed. We review these serial algorithms, survey existing parallel algorithms, and then discuss implications of these approaches in a distributed-memory parallel setting.

2.1 Applications of connected components

2.1.1 Computer Vision

Labeling connected components in binary images is a common image segmentation technique used in computer vision [RW96]. To gain efficiency, labeling algorithms use sweeps that exploit the structured nature of image data. The most common sweep techniques are tailored to provide results for four- or eight-connected image neighborhoods. [ST88] outlines an approach to efficiently label connected components in three- and higher dimensional images using linear bintrees. Although these techniques are quite effective in the area of computer vision, the approach does not easily generalize to the problem of resolving connectivity in unstructured meshes.

2.1.2 Graph Theory

The cell abutment relationships in an unstructured mesh can be encoded into a sparse undirected graph representation, so methods for finding the connected components of graphs are applicable, at least in spirit, to mesh-based data. Connected components algorithms are used in various graph theory applications to identify partitions. There are two common approaches. The first employs a series of Depth-first or Breadth-first searches [HT71]. Initially all vertices are unmarked. Each search starts at an unmarked vertex walking the graph edges and marking each reached vertex. New searches are executed until all vertices are marked. Each search yields a tree which corresponds to a single connected component. This approach is analogous to a region growing scheme.

The second approach uses the Union-find algorithm [CSRL01] for disjoint-sets. This is typically used for tracking how connected components evolve as edges are added to a graph. This incremental approach requires only local connectivity information and efficiently handles merging disjoint-sets as each new edge is added. The Union-find algorithm and data structures are also used to efficiently construct topological representations such as a Contour Tree [CSA00] or a Reeb Graph [TGSP09] of a data set. We use the serial Union-find algorithm as a key building block for our approach. Union-find is discussed in detail in Section 3.1.

2.2 Parallelism

2.2.1 Parallelism in computer vision and graph algorithms

[AP92, CT92] provide overviews of several parallel computer vision algorithms for connected components labeling, including a few approaches for distributed-memory machines. Like the serial computer vision algorithms, these approaches are limited to regular grids.

A naïve distributed-memory implementation of the graph search approach (i.e. DFS or BFS) would have a run-time proportional to the number of cells in the largest sub-mesh and would require complex communication to track visited cells as mesh regions grow across processors. The run-time of this approach would be prohibitive for data sets with sub-meshes on the order of the size of the entire mesh. A naïve distributed-memory implementation of the Union-find algorithm is difficult due to the indirect memory access patterns used by the disjoint-set data structures to gain efficiency. Both of these approaches map conceptually well to a PRAM (Parallel Random Access Memory) [JaJ92] model of computation. However, they are difficult to implement efficiently in a distributed-memory parallel context.

Much of the research into parallel algorithms for graph connected components has focused on shared-memory architectures [HW90, AW91]. [CAP88] provides a parallel algorithm using the Union-find algorithm that is structurally similar to a connected components algorithm. They obtained speedups on a shared-memory machine, but observed poor performance when mapping their algorithm to a PRAM model on a distributed-memory architecture.

[KLCY94, BT01, MP10] use a hybrid local-global approach on distributed-memory machines. [KLCY94] uses a breadth-first search to resolve local connectivity, followed by a global PRAM step to incorporate edges that cross processors. [BT01] extends [KLCY94] using a diffrent a PRAM scheme implemented over MPI and provides results for 2D and 3D structured grids with maximum size of 500,000 cells. [MP10] presents a distributed-memory Union-find algorithm for identifying the spanning forest of a graph. Their global step distributes the Union-find data structure and operations across all processors using a complex message passing scheme, which limits scalability.

Our algorithm also uses a multi-stage approach with local and global steps. Because our algorithm is designed for mesh based simulation data, we take advantage of the sparse nature of mesh connectivity. A novel contribution of our approach is a compression from a label set the size of the total number of cells to a much smaller intermediate labeling. This allows us to use the serial Union-find algorithm for the global resolve and avoid the drawbacks of distributing the Union-find. Our approach also handles the constraint that connectivity information across processors is not known a priori, a problem that does not occur in general graph applications. Finally, this study presents performance characteristics at considerably higher concurrency levels and larger data sizes (including, for the first time, unstructured meshes) than have been studied in previous work. Further, properties of mesh-based data create opportunities for optimizations not possible with graph data, making the performance characteristics substantially different. For example, mesh-based data can use ghost data to optimize across boundaries (described in Section 4.2).

2.2.2 Parallelism strategy for end user tools

Our algorithm is intended for data sets so large that they cannot fit into the memory of a single node. Popular end user visualization tools for large data, such as EnSight [Com09], ParaView [AGL05], and VisIt [CBB*05], follow a distributed-memory parallelization strategy. Each of these tools instantiate identical visualization modules on every MPI task, and the MPI tasks are only differentiated by the sub-portion of the larger data set they operate on. The tools rely on the data set being decomposed into pieces (often referred to as domains), and they partition the pieces over their MPI tasks. This approach has been shown to work well to date, with the most recent example demonstrating VisIt to perform well on meshes with trillions of cells using tens of thousands of processors [CPA*10]. Our algorithm follows the strategy of partitioning data over the processors and has been implemented as a module inside VisIt.

3 Algorithm building blocks

This section describes four fundamental building blocks used by our algorithm. The first is the serial Union-find algorithm which allows us to efficiently identify and merge connected components. The second is a parallel binary space partitioning scheme which allows us to efficiently compute mesh intersections across processors. The third is the practice of generating ghost data, which, if available, allows us to use an optimized variant of our algorithm. The fourth is the data structures for storing mesh-based data.

3.1 Union-find

The Union-find algorithm enables efficient management of partitions. It provides two basic operations: UNION and FIND. The UNION operation creates a new partition by merging two subsets from the current partition. The FIND operation determines which subset of a partition contains a given element.

To efficiently implement these operations, relationships between sets are tracked using a disjoint-set forest data structure. In this representation, each set in a partition points to a root node containing a single representative set used to identify the partition. The UNION operation uses a union-by-rank heuristic to update the root node of both partitions to the representative set from the larger of the two partitions. The FIND operation uses a path-compression heuristic which updates the root node of any traversed set to point to the current partition root. With these optimizations each UNION or FIND operation has an amortized run-time of $O(\alpha(N))$ where N is the number of sets and $\alpha(N)$ is the inverse Ackermann function [Tar75]. $\alpha(N)$ grows so slowly that it is effectively less than four for all practical input sizes. The disjoint-set forest data structure requires $O(N)$ space to hold partition information and the values used to implement the heuristics. The heuristics used to gain efficiency rely heavily

on indirect memory addressing and do not lend themselves to direct a distributed-memory parallel implementation.

3.2 Generating a BSP-Tree

To determine if a component on one processor abuts a component on another processor (meaning they are both actually part of a single, larger component), we will need to relocate the cells in a way that guarantees spatially abutting cells will reside on the same processor. We do this by creating a binary space partitioning (BSP) [FKN80] which partitions two- or three-dimensional space into $N_{processors}$ pieces. Each split in the BSP is designed to give each processor approximately the same number of elements (and thus memory), in the hope of avoiding load imbalance and/or potentially exhausting memory.

It is difficult to efficiently sub-divide spatial elements into tree-based data structures in a shared memory parallel setting (for example see [BSGE98, BS99]). In our case, the tree generation must be performed in a distributed memory setting, because there are many more cells than can fit on a single node. Our algorithm is recursive. We start by creating a region that spans the entire data set. On each iteration and for each region that represents more than 1/Nth of the data (measured in number of elements covered), we select "pivots", which are possible locations to split a region along a given axis. This axis changes on each iteration. All processors traverse all of their elements, and their positions with respect to the pivots are categorized. If a pivot exists that allows for a good split, then the region is split into two sub-regions and recursive processing continues. Otherwise we choose a new set of pivots, whose choice incorporates the closest matching pivots from the previous iteration as extrema. If a good pivot is not found after some number of iterations, we use the best encountered pivot and accept the potential for load imbalance.

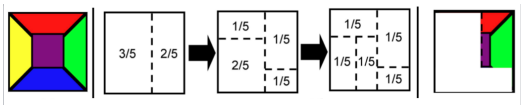


Figure 2: *The process for constructing a BSP-tree in a distributed memory setting. On the left, the cells from the original mesh. Assume the red portions are on processor 1, blue on 2, and so on. The iterative strategy starts by dividing in X, then in Y, and continues until every region contains approximately $1/N_{processors}$ of the data. Each processor is then assigned one region from the partition and we communicate the data so that every processor contains all data for its region. The data for processor 3 is shown on the far right.*

After constructing a BSP-tree, we assign one portion of the partition to each MPI task and then re-distribute the cells such that each MPI task contains every cell from its partition. Cells that span multiple partitions are duplicated.

3.3 Ghost Cells

When a large data set is decomposed into domains, interpolation artifacts can occur along domain boundaries. The typical solution for this problem is to create "ghost cells," a redundant layer of cells along the boundary of each domain. Ghost cells are either pre-computed by the simulation code and stored in files or calculated at run-time by the post-processing tool. More discussion of ghost cells can be found in [ILC10, CBB*05].

Ghost cells can provide benefits beyond interpolation. They also can be used to identify the location of the boundary of a domain and provide information about the state of abutting cells in a neighboring domain. It is in this way that we incorporate ghost cell information into our algorithm. Note that this paper uses ghost cells that are generated at run-time and uses the collective pattern described in [CBB*05], not the streaming pattern described in [ILC10].

3.4 Data Structures

The Visualization ToolKit library (VTK) [SML96] is used to represent mesh-based data. The data model has a mesh (structured or unstructured) with multiple fields stored on either the cells or the vertices. VTK provides routines for identifying cell abutment and we use these routines within a domain.

4 Algorithm description

Our algorithm identifies the global connected components in a mesh using four phases. We first identify the connected components local to each processor (Phase 1) and then create a global labeling across all processors (Phase 2). We next determine which components span multiple processors (Phase 3). Finally, we merge the global labels to produce a consistent labeling across all processors (Phase 4). This final labeling is applied to the mesh to create per-cell labels which map each cell to the corresponding label of the connected component it belongs to. In terms of parallelization, Phase 1 is embarrassing parallel, Phase 2 is a trivial communication, Phase 3 has a large all-to-all communication, followed by embarrassingly parallel work, and Phase 4 has trivial communication following by more embarrassingly parallel work.

We present two variants of the algorithm. This first provides a general solution, applicable to any domain-decomposed 3D mesh. The second algorithm is an optimized variant of the first which can be used if ghost data is available for the mesh. The optimizations in the second variant greatly reduce the communication and processing required to resolve global connectivity.

4.1 General Algorithm

Phase 1: Identify components within a processor

The purpose of this phase is for each processor to label the connected components for its portion of the data. As

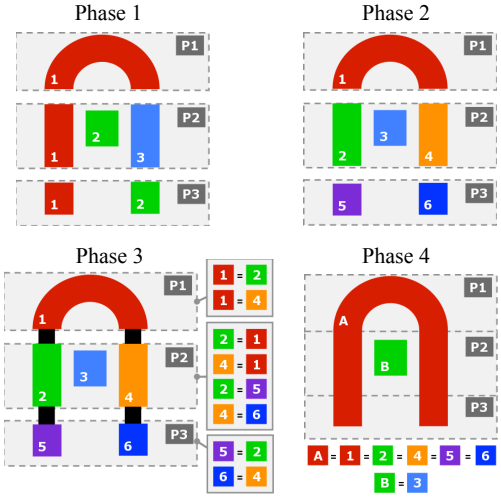


Figure 3: *Example illustrating the four phases of our algorithm on a simple data set decomposed onto three processors.*

mentioned in Section 2, the Union-find algorithm efficiently constructs a partition through an incremental process. A partition with one subset for each point in the mesh is used to initialize the Union-find data structure. We then traverse the cells in the mesh. For each cell, we identify the points incident to that cell. Those points are then merged ("unioned") in the Union-find data structure.

In pseudocode:

```
UnionFind uf;
For each point p:
   uf.SetLabel(p, GetUniqueLabel())
For each cell c:
   pointlist = GetPointsIncidentToCell(c)
   p0 = pointlist[0]
   For each point p in pointlist:
      if (uf.Find(p0) != uf.Find(p))
         uf.Union(p0, p)
```

The execution time of this phase is dependent on the number of union operations, the number of find operations, and the complexity of performing a given union or find. The number of finds is equal to the sum over all cells of how many points are incident to that cell. Practically speaking, the number of points per cell will be small, for example eight for a hexahedron. Thus the number of finds is proportional to the number of cells. Further, the number of unions will be less than the number of finds. Finally, although the run-time complexity of the Union-find algorithm is nuanced, each individual union or find is essentially a constant time operation, asymptotically-speaking. Thus the overall execution time of this phase for a given processor is proportional to the number of cells contained on that processor.

Phase 2: Component re-label for cross-processor comparison

At the end of Phase 1, on each processor, the components within that processor's data have been identified. Each of these components has a unique local label and the purpose of Phase 2 is to transform these identifiers into unique global labels. This will allow us to perform parallel merging in subsequent phases. Phase 2 actually has two separate re-labelings. First, since the Union-find may create non-contiguous identifiers, we transform the local labels such that the numbering ranges from 0 to N_P, where N_P is the total number of labels on processor P. For later reference, we denote $N = \sum N_P$ as the total number of labels over all processors. Second, we construct a unique labeling across the processors by adding an offset to each range. We do this by using the MPI rank and determining how many total components exist on lower MPI ranks. This number is then added to component labels. At the end of this process, MPI rank 0 will have labels from 0 to $N_0 - 1$, MPI rank 1 will have labels from N_0 to $N_0 + N_1 - 1$ and so on. Finally, a new scalar field is placed on the mesh, associating the global component label with each cell.

Phase 3: Merging of labels across processors

At this point, when a component spans multiple processors, each processor's sub-portion has a different label. The goal of Phase 3 is to identify that these sub-portions are actually part of a single component and merge their labels. We do this by re-distributing the data using a BSP-tree (see Section 3.2) and employing a Union-find strategy to locate abutting cells that have different labels. The Union-find strategy in this phase has four key distinctions from the strategy described in Phase 1.

- The labeling is now over cells (not points), which is made possible by the scalar field added in Phase 2.
- We merge based on cell abutment, as opposed to Phase 1, where we merged when two points were incident to the same cell.
- Each cell is initialized with the unique global identifier from the scalar field added in Phase 2, as opposed to the arbitrary unique labeling imposed in Phase 1.
- Whenever a union operation is performed, we record the details of that union for later use in establishing the final labeling.

In pseudocode:

```
CreateBSPTree()
UnionFind uf;
For each cell c:
  uf.SetLabel(c, label[c])
For each cell c:
  For each neighbor n of c:
       if (uf.Find(c) != uf.Find(n))
            uf.Union(n, c)
            RecordMerge(n, c)
```

After the union list is created, we discard the re-distributed data and each processor returns to operating on its original data.

Phase 4: Final assignment of labels

Phase 4 incorporates the merge information from Phase 3 with the labeling from Phase 2. Recall that in Phase 2 we constructed a globally unique labeling of per-processor components and denoted N as the total number of labels over all processors. The final labeling of components is constructed as follows:

- After Phase 3, each processor is aware of the unions it performed, but not aware of unions on other processors. However, to assign the final labels, each processor must have the complete list of unions. So we begin Phase 4 by broadcasting ("all-to-all") each processor's unions to construct a global list.
- Create a Union-find data structure with N entries, each entry having the trivial label.

```
UnionFind uf
For i in 0 to N-1:
   uf.SetLabel(i, i)
```

- Replay all unions from the global union list.

```
For union in GlobalUnionList:
   uf.Union(union.label1, union.label2)
```

  The Union-find data structure can now be treated as a map. Its "Find" method transforms the labeling we constructed in Phase 2 to a unique label for each connected component.
- Use the "Find" method to transform the labeling from the scalar array created in Phase 2 to create a final labeling of which connected component each cell belongs to.

```
For each cell c:
  val[c] = uf.Find(val[c])
```

- Optionally transform the final labeling so that the labels range from 0 to $N_C - 1$, where N_C is the total number of connected components.

Note that the key to this construction is that every processor is able to construct the same global list by following the same set of instructions. They essentially "replay" the merges from the global union list in identical order, creating an identical state in their Union-find data structure.

4.2 Ghost cell optimized algorithm

One of the strengths of the general algorithm is that local connectivity, which encapsulates the majority of cell abutments, is resolved concurrently on each processor. After Phase 2 completes, the only cells that can contribute and merge labels across processors are those cells that could potentially abut cells residing other processors. If we can identify which cells intersect the spatial boundary of each processor, we can limit the re-distribution in Phase 3 to this subset of cells. Processing a reduced set of cells in Phase 3 can lead to significant performance gains.

We created an optimized variant of the general algorithm using this strategy. To do so, we add a new preprocessessing step ("Phase 0"), and modify Phase 3 to reduce the amount of re-distributed cells. Phases 1, 2, and 4 are reused from the general algorithm.

Phase 0: Identify cells at processor boundaries

The goal of this preprocessing phase is to identify cells that abut the spatial boundary of the data contained on each processor. When ghost cells are present this is trivial: the boundary cells are those that are adjacent to ghost cells. Note that we cannot directly use the ghost cells to represent processor boundaries since they themselves lack ghost data. For example, an isosurface operation for a ghost cell will generate incorrect geometry since that ghost cell is lacking the requisite additional ghost data to perform interpolation. Further, since ghost data can have incorrect geometry, we remove all ghost cells after the boundary is identified.

In pseudocode:

```
For each cell c:
  boundary[c] = false
  if (not IsGhostCell(c))
    For each neighbor n of c:
      if IsGhostCell(n):
        boundary[c] = true
RemoveGhostCells()
```

Phase 3': Merging of labels across processors

In Phase 3', we create the spatial partition using only the set of boundary cells identified in Phase 0, in contrast to the general algorithm which re-distributes all of cells. We identify abutment and construct each processor's union list using the same Union-find strategy from Phase 3 in the general algorithm. By using the boundary information identified in Phase 0, we reduce the number of cells redistributed by an order of magnitude. This minimizes the amount of communication and the complexity of the intersection tests used to identify cell abutment.

5 Applications

The labeling produced by a connected components algorithm is key to certain types of analyses. Having a topological description of the connected components of a mesh allows us to isolate features in ways fundamentally different from standard visualization tools. To demonstrate this, we present two applications on real data sets that use a connected components algorithm, also reporting the performance characteristics.

5.1 Turbulent flow

Turbulence is the most common state of fluid motion in nature and engineering and a complex subject in the physi-

	Num cells wo/ ghosts	Num cells w/ ghosts	Num comps
Input data set	68.7 billion	77.2 billion	1
After isosurface	1.08 billion	1.21 billion	2.02 million
After isovolume	1.81 billion	2.04 billion	2.01 million

Table 1: *Number of cells processed. The input mesh was a* 4096^3 *rectilinear grid of hexahedral cells (voxels). For the first test, applying an isosurface, the cell types are triangles. For the second test, applying an isovolume, the cells types are hexahedrons, tetrahedrons, wedges, and pyramids. The number of isovolume components is less than the number of isosurface components, since they contribute a different number at the boundary of the problem.*

cal sciences. Advances in understanding how to model turbulence is critical to advancing the state of turbulent theory and more practically, to the areas of aerospace vehicle design, combustion processes and environmental quality just to name a few. Turbulent flow is characterized by nonlinear stochastic fluctuations in time and three-dimensional space over a wide range of scales. Two important descriptors of these small-scale motions are energy dissipation rate and vorticity. Specifically, we wish to understand how energy dissipation rate and vorticity relate to each other and the characteristics of the vortical structures over time.

One easy way to visualize vorticity is by creating isosurfaces of thresholded values. These values in turbulent flow, however, are very small, and it is not clear that selecting a single threshold will provide the desired information. For this reason, we choose to visualize isovolumes (also known as “interval volumes" [FMS95]) of vorticity rather than isosurfaces of vorticity. Visualizing all isovolumes is both time-consuming and unnecessary. Many of these structures are miniscule and not relevant. We can cull out isovolumes by thresholding based on volume, but that only partially alleviates a potential clutter problem.

Creating connected components of the thresholded isovolumes presents us with a number of strong vortical structures that we can study over time. We can investigate questions such as: how are they born, do they die off, do they marry, and do they divorce. Understanding and tracking this behavior over time allows for in-depth view of what is happening in turbulent flow at the small scale, rather than being limited exclusively to global views.

Figure 4 shows an example of worm-like structures of vorticity extracted from a turbulent flow simulation. This view shows us the general clustering behavior at a given time-step and allows us to see that their shape is somewhat random in nature. The coloring allows us to see how the different isovolumes are clustered throughout the data set. The details of the simulation data set are given in the text for Table 1.

We ran on 256 processors of Longhorn, a Dell/Linux machine for remote, interactive visualization that contains a total of 256 nodes, 2048 cores (8 Intel Nehalem 2.3 GHz cores per node), 512 gpus, and 14.5 terabytes of aggregate memory. Our actual analysis used isovolumes and data with no ghost cells. For this paper, we added the option to calculate ghost data and also repeated the analysis with isosurfaces, giving a total of four tests. The overall performance on this $4K^3$ rectilinear mesh is described in Table 2 and the per-phase performance is described in Table 3.

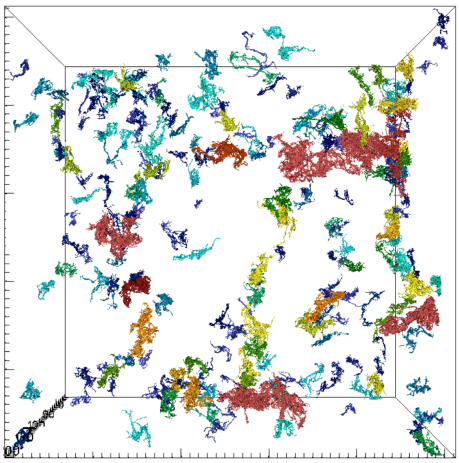


Figure 4: *The 224 “worms" that have volume larger than a certain threshold. Each worm is colored by its volume.*

Algorithm	Ghost	Iso	CC
Isosurf. wo/ Ghost	-	16.3s	30.2s
Isosurf. w/ Ghost	18.5s	17.1s	16.9s
Isovol. wo/ Ghost	-	24.4s	108.5
Isovol. w/ Ghost	18.3s	26.3s	69.6s

Table 2: *Performance of connected components identification algorithm in the context of overall performance, including time to calculate a layer of ghost cells (“Ghost"), apply either an isosurface or isovolume algorithm (“Iso"), and apply the connected components identification algorithm (“CC"). Note that read times regularly exceed one minute and vary greatly due to disk contention, caching by the operating system, and other factors. (All tests read the same data.)*

5.2 Turbulent flow in a nuclear reactor

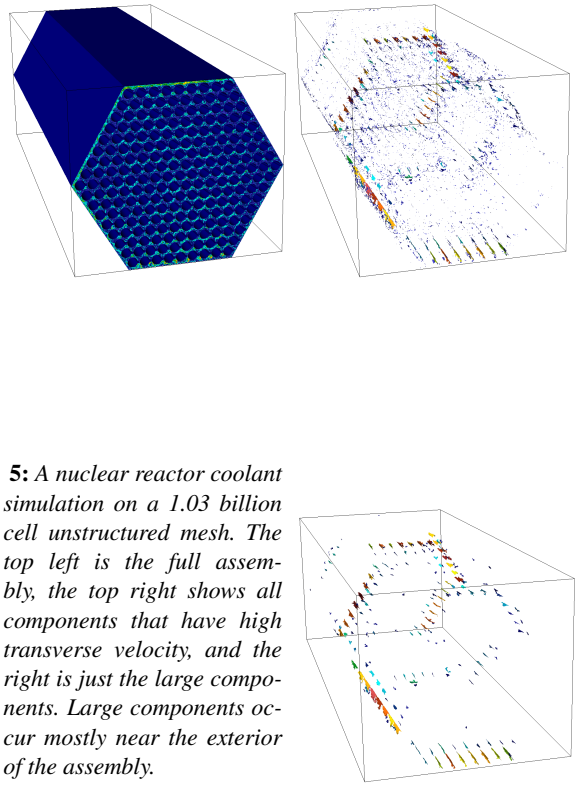
In [FLPS08], Fischer et al use the Nek5000 code to simulate the flow of coolant around a 217-fuel rod nuclear reactor. In this simulation, coolant flows through the assembly with a strong bias along a fixed axis (the “z-axis"), with each

Algorithm / Phase	0	1	2	3	4
Isosurf. wo/ Ghost	-	4.4s	0.01s	24.2s	1.5s
Isosurf. w/ Ghost	4.2s	4.3s	0.01s	5.0s	3.1s
Isovol. wo/ Ghost	-	12.5s	0.03s	89.2s	6.7s
Isovol. w/ Ghost	12.9s	13.2s	0.03s	37.9s	5.4s

Table 3: *Performance of connected components algorithm.*

rod also being aligned to this axis. It is not desirable for the coolant to travel directly down this fixed axis. If one of the rods is "hot," the ideal scenario is for coolant to absorb heat and then move away, letting other material come in to continue the cooling process. One important question is where "pockets" of coolant are transferring through the assembly most quickly. Since each rod is aligned with the z-axis, this is equivalent to locating regions with significant x,y-velocity, which can be accomplished via an isosurface operation on this derived field. Of course, only regions above a certain size criteria represent significant trends, so we once again only study components above a size threshold.

This simulation takes place on a 1.012 billion cell unstructured mesh of hexahedrons. There was no ghost data available and we could not calculate it for comparison's sake as we did in Section 5.1. We used 30 nodes of Argonne National Laboratory's "Eureka" machine, with each node containing two 2.0 GHz quad-core Xeons (a total of 240 MPI tasks). The resulting isosurface had 3.04 million cells spread over 25,189 components. By discarding components below a size threshold, we arrived at 214 "large" components. These components almost all resided at the exterior of the assembly, meaning that coolant is communicating better in the exterior than in the interior. The resulting visualizations can be seen in Figure 5 and specific performance measures in Table 4.



5: *A nuclear reactor coolant simulation on a 1.03 billion cell unstructured mesh. The top left is the full assembly, the top right shows all components that have high transverse velocity, and the right is just the large components. Large components occur mostly near the exterior of the assembly.*

Stage	Read	Isosurface	Phase 1	2	3	4
Time	14.7	1.1s	0.1s	0.01s	0.5s	0.1s

Table 4: *Performance of connected components algorithm on reactor coolant simulation.*

Num cores	Input mesh size	Isovol. mesh size
$2^3 = 8$	80 million	10.8 million
$3^3 = 27$	270 million	34.9 million
$4^3 = 64$	640 million	80.7 million
$5^3 = 125$	1.25 billion	155.3 million
$6^3 = 216$	2.16 billion	265.7 million
$7^3 = 343$	3.43 billion	418.7 million
$8^3 = 512$	5.12 billion	621.5 million
$9^3 = 729$	7.29 billion	881.0 million
$10^3 = 1000$	10 billion	1.20 billion
$11^3 = 1331$	13.3 billion	1.59 billion
$12^3 = 1728$	17.2 billion	2.06 billion
$13^3 = 2197$	21.9 billion	2.62 billion

Table 5: *Scaling study data set sizes. We targeted processor counts equal to powers of three to maintain an even spatial distribution after upsampling. The highest power of three processor count available on our test system was* $13^3 = 2197$ *processors. This allowed us to study processor counts from 8 to 2197 and initial mesh sizes from 80 million to 21 billion cells. The isovolume operation creates a new unstructured mesh consisting of portions of approximately 1/8th of the cells from the initial mesh, meaning that each core has, on average, 1.2 million cells.*

6 Performance study

To further explore the performance characteristics of our algorithm, we conducted a weak scaling study that looked at concurrency levels up to 2197 cores with data set sizes up to 21 billion cells. We ran this study on Lawrence Livermore National Laboratory's "Edge" machine, a 216 node Linux cluster with each node containing two 2.8GHz six-core Intel Westmere processors. The system has 96GB of memory per node (8GB per core) and 20TB of aggregate memory.

6.1 Problem setup

We used synthetic data as input, upsampling structured grid data from a core-collapse supernova simulation produced by the Chimera code [BMH*08]. This data set was selected because it contains a scalar entropy field with large isovolume components that span many processors. To test weak scaling we upsampled the input data set, creating new data sets with 10 million cells for every processor. We extracted isovolumes from the upsampled structured grid to create an unstructured mesh for input to the connected components algorithm. Table 5 outlines the number of cores and the corresponding data sets used in our scaling study. Figure 1 shows rendered views of the largest isovolume data set used in the scaling study and its corresponding labeling result. We tested both the general and ghost cell optimized variants of the algorithm.

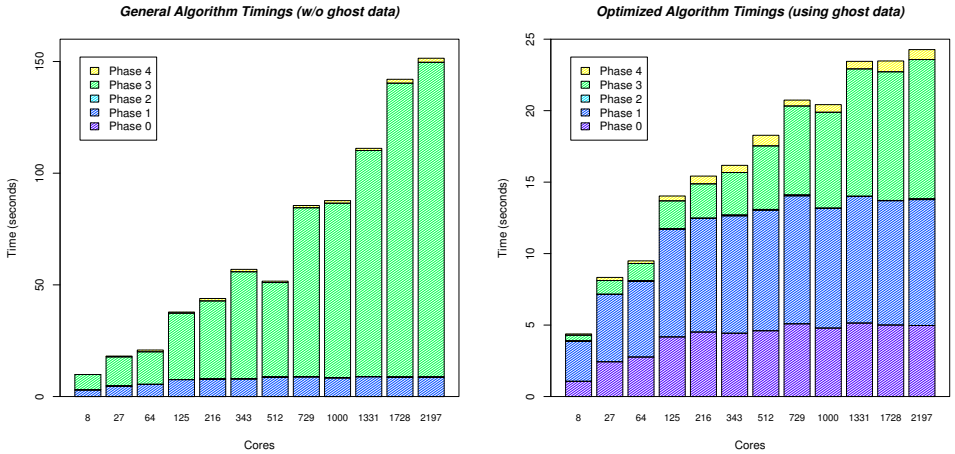


Figure 6: *(Left) Scaling study timings for the general algorithm. (Right) Scaling study timings for the ghost data optimized algorithm. Phase 3 timings are significantly reduced by using ghost cells. Note that the two graphs have different scales, going up to 150 seconds for the general algorithm, but only up to 25 seconds for the ghost data optimized algorithm.*

Num cores	Num cells in largest comp.	Num cores spanned	Num global union pairs
$2^3 = 8$	10.1 million	4	16
$3^3 = 27$	32.7 million	17	96
$4^3 = 64$	76.7 million	29	185
$5^3 = 125$	146.6 million	58	390
$6^3 = 216$	251.2 million	73	666
$7^3 = 343$	396.4 million	109	1031
$8^3 = 512$	588.9 million	157	1455
$9^3 = 729$	835.5 million	198	2086
$10^3 = 1000$	1.14 billion	254	2838
$11^3 = 1331$	1.51 billion	315	3948
$12^3 = 1728$	1.96 billion	389	5209
$13^3 = 2197$	2.49 billion	476	6428

Table 6: *Largest component information and number of global union pairs transmitted. As the size of the data set increases we see a linear correlation (0.994322) between the number of cores spanned by the largest connected component of the isovolume and the number of union pairs transmitted in Phase 4. The percentage of cores spanned by the largest component converges to slightly less than 25%.*

6.2 Performance

Figure 6 presents the timing results from our scaling study. As expected, the timings for phases 1,2, and 4 are consistent between both variants of the algorithm. At 125 processors and beyond the largest subset of the isovolume on a single processor approaches the maximum size, 10 million cells. At this point we expect weak scaling for Phase 1. This is confirmed by flat timings for Phase 1 beyond 125 processors. The ghost cell optimized variant significantly outperformed the general algorithm in Phase 3. These timings demonstrate the benefit of having ghost data available to identify per-processor spatial boundaries. The small amount of additional preprocessing time required for Phase 0 allows us to reduce the number of cells transmitted and processed in Phase 3 by an order of magnitude.

By upsampling, we maintain a fixed amount of data per processor, however the number of connectivity boundaries in the isovolume increases as the number of processors used in decomposition increases. This is reflected by the linear growth in both the number of union pairs transmitted in Phase 4 and the number of cores spanned by the largest connected component (See Table 6).

7 Conclusion and Future Work

We have presented a data-parallel algorithm that identifies and labels the connected components in a domain-decomposed mesh. Our algorithm is designed to fit well into currently deployed distributed-memory visualization tools. The labeling produced by our algorithm provides a topological characterization of a data set that enables new types of analysis. We presented two applications which demonstrate our approach is suitable for analyzing the massive data sets created by today's parallel scientific simulations. Our scaling study demonstrated a significant speed up in execution time when ghost data is available.

As far as future work, an improved understanding of the construction of the BSP-tree and overall communication patterns would reveal more about the performance of this algorithm. Further, although we demonstrated our algorithm works well on large data sets at high levels of concurrency, previous approaches have not been studied in this context. Studying their performance would allow for better comparison to our own algorithm.

Acknowledgments

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. This work was also supported by the Director, Office of Advanced

Scientific Computing Research, Office of Science, of the U.S. Department of Energy under Contrct No. DE-AC02-05CH11231 through the Scientific Discovery through Advanced Computing (SciDAC) program's Visualization and Analytics Center for Enabling Technologies (VACET). This work was also supported by the Texas Advanced Computing Center through the use of Longhorn. Additionally, the work was supported in part by the National Science Foundation, grants OCI-0906379 and OCI-0751397.

References

[AGL05] AHRENS J., GEVECI B., LAW C.: Visualization in the paraview framework. In *The Visualization Handbook* (2005), Hansen C., Johnson C., (Eds.), pp. 162–170.

[AP92] ALNUWEITI H., PRASANNA V.: Parallel architectures and algorithms for image component labeling. *Pattern Analysis and Machine Intelligence, IEEE Transactions on 14*, 10 (Oct. 1992), 1014 –1034.

[AW91] ANDERSON R. J., WOLL H.: Wait-free parallel algorithms for the union-find problem. In *Proceedings of the twenty-third annual ACM symposium on Theory of computing* (New York, NY, USA, 1991), STOC '91, ACM, pp. 370–380.

[BMH*08] BRUENN S. W., MEZZACAPPA A., HIX W. R., BLONDIN J. M., MARRONETTI P., MESSER O. E. B., DIRK C. J., YOSHIDA S.: Mechanisms of core-collapse supernovae and simulation results from the chimera code. In *CEFALU 2008, Proceedings of the International Conference. AIP Conference Proceedings* (2008), pp. 593–601.

[BS99] BARTZ D., STRABER W.: Asynchronous parallel construction of recursive tree hierarchies. In *Parallel Computation*, vol. 1557. 1999, pp. 427–436.

[BSGE98] BARTZ D., STRABER W., GROSSO R., ERTL T.: Parallel construction and isosurface extraction of recursive tree structures. In *Proceedings of WSCG* (1998), vol. III.

[BT01] BUS L., TVRDÍK P.: A parallel algorithm for connected components on distributed memory machines. In *Proceedings of the 8th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface* (London, UK, 2001), Springer-Verlag, pp. 280–287.

[CAP88] CYBENKO G., ALLEN T. G., POLITO J. E.: Practical parallel union-find algorithms for transitive closure and clustering. *International Journal of Parallel Programming 17* (1988), 403–423. 10.1007/BF01383882.

[CBB*05] CHILDS H., BRUGGER E., BONNELL K., MEREDITH J., MILLER M., WHITLOCK B., MAX N.: A contract based system for large data visualization. In *VIS '05: Proceedings of the conference on Visualization '05* (2005).

[Com09] COMPUTATIONAL ENGINEERING INTERNATIONAL, INC.: *EnSight User Manual*, December 2009.

[CPA*10] CHILDS H., PUGMIRE D., AHERN S., WHITLOCK B., HOWISON M., PRABHAT, WEBER G., BETHEL E. W.: Extreme Scaling of Production Visualization Software on Diverse Architectures. *IEEE Computer Graphics and Applications 30*, 3 (May/June 2010), 22–31. LBNL-3403E.

[CSA00] CARR H., SNOEYINK J., AXEN U.: Computing contour trees in all dimensions. In *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms* (Philadelphia, PA, USA, 2000), SODA '00, Society for Industrial and Applied Mathematics, pp. 918–926.

[CSRL01] CORMEN T. H., STEIN C., RIVEST R. L., LEISERSON C. E.: *Introduction to Algorithms*, 2nd ed. McGraw-Hill Higher Education, 2001.

[CT92] CHOUDHARY A., THAKUR R.: Evaluation of connected component labeling algorithms on shared and distributed memory multiprocessors. In *Parallel Processing Symposium, 1992. Proceedings., Sixth International* (Mar. 1992), pp. 362 –365.

[FKN80] FUCHS H., KEDEM Z. M., NAYLOR B. F.: On visible surface generation by a priori tree structures. In *Proceedings of the 7th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1980), SIGGRAPH '80, ACM, pp. 124–133.

[FLPS08] FISCHER P., LOTTES J., POINTER D., SIEGEL A.: Petascale algorithms for reactor hydrodynamics. In *J. Phys.: Conf. Ser.* (2008), vol. 125, pp. 1–8.

[FMS95] FUJISHIRO I., MAEDA Y., SATO H.: Interval volume: a solid fitting technique for volumetric data display and analysis. In *Proceedings of the 6th conference on Visualization '95* (Washington, DC, USA, 1995), VIS '95, IEEE Computer Society, pp. 151–.

[HT71] HOPCROFT J. E., TARJAN R. E.: *Efficient algorithms for graph manipulation*. Tech. rep., Stanford, CA, USA, 1971.

[HW90] HAN Y., WAGNER R. A.: An efficient and fast parallel-connected component algorithm. *J. ACM 37* (July 1990), 626–642.

[ILC10] ISENBURG M., LINDSTROM P., CHILDS H.: Parallel and Streaming Generation of Ghost Data for Structured Grids. *IEEE Computer Graphics and Applications 30*, 3 (May/June 2010), 32–44.

[JaJ92] JAJA J.: *Introduction to Parallel Algorithms*. Addison-Wesley Professional, Apr. 1992.

[KLCY94] KRISHNAMURTHY A., LUMETTA S. S., CULLER D. E., YELICK K.: Connected components on distributed memory machines. In *Parallel Algorithms: 3rd DIMACS Implementation Challenge October 17-19, 1994, volume 30 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science* (1994), American Mathematical Society, pp. 1–21.

[MP10] MANNE F., PATWARY M.: A scalable parallel union-find algorithm for distributed memory computers. In *Parallel Processing and Applied Mathematics*, Wyrzykowski R., Dongarra J., Karczewski K., Wasniewski J., (Eds.), vol. 6067 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2010, pp. 186–195.

[RW96] RITTER G. X., WILSON J. N.: Handbook of computer vision algorithms in image algebra, 1996.

[SML96] SCHROEDER W. J., MARTIN K. M., LORENSEN W. E.: The design and implementation of an object-oriented toolkit for 3d graphics and visualization. In *VIS '96: Proceedings of the 7th conference on Visualization '96* (1996), IEEE Computer Society Press, pp. 93–ff.

[ST88] SAMET H., TAMMINEN M.: Efficient component labeling of images of arbitrary dimension represented by linear bintrees. *Pattern Analysis and Machine Intelligence,IEEE Transactions on 10*, 4 (July 1988), 579–586.

[Tar75] TARJAN R. E.: Efficiency of a good but not linear set union algorithm. *J. ACM 22* (April 1975), 215–225.

[TGSP09] TIERNY J., GYULASSY A., SIMON E., PASCUCCI V.: Loop surgery for volumetric meshes: Reeb graphs reduced to contour trees. *IEEE Transactions on Visualization and Computer Graphics 15* (2009), 1177–1184.