Chapter 1
A Distributed-Memory Algorithm for Connected Components Labeling of Simulation Data

C. Harrison[1], J. Weiler[2], R. Bleile[2], K.P. Gaither[3], and H. Childs[2,4]
[1]**Lawrence Livermore National Laboratory**
[2]**The University of Oregon**
[3]**The University of Texas at Austin (Texas Advanced Computing Center)**
[4]**Lawrence Berkeley National Laboratory**

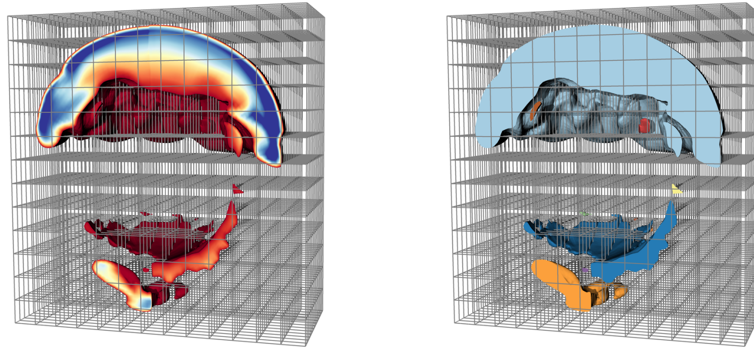

Fig. 1.1 (Left) Sub-volume mesh extracted from a 21 billion cell structured grid decomposed across 2197 processors. (Right) Sub-volume mesh colored by the results from the connected components labeling algorithm described in this chapter.

Abstract This chapter describes a data-parallel, distributed-memory algorithm for identifying and labeling the connected sub-meshes within a three-dimensional mesh. The identification task is challenging in a distributed-memory setting because connectivity is transitive and the cells composing a sub-mesh may span many processors. The algorithm employs a multi-stage application of the Union-find algorithm and a spatial partitioning scheme to efficiently merge information across processors and to produce a global labeling of connected sub-meshes. Marking each vertex with its corresponding sub-mesh label allows mesh features to be isolated based on topology, enabling important analysis capabilities. The algorithm performs well in parallel; results are presented from a weak scaling study with concurrency levels up to 2197 cores and meshes containing over two billion cells. This chapter is an extension of previous work by Harrison et al. [8]. It contains significant algorithmic improvements over the previous version, improved exploration of key bottlenecks in the algorithm, and improved clarity of presentation.

1.1 Introduction

Parallel scientific simulations running on today's state of the art petascale computing platforms generate massive quantities of high resolution mesh-based data. Scientists often analyze this data by eliminating portions and visualizing what remains, through operations such as isosurfacing, selecting certain materials and discarding the others, isolating hot spots, etc. These approaches can generate complex derived geometry with intricate structures that require further techniques for effective analysis, especially in the context of massive data.

In these instances, representations of the topological structure of a mesh is often helpful. Specifically, 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 in distributed-memory parallel environments, which are the most common setting for analyzing massive data. This is because massive data sets are typically too large to fit into the memory of a single processor, so pieces of the mesh are distributed across processors. Cells comprising connected sub-meshes may span any of the processors, but the relationships of how cells abut across processors frequently has to be derived. To deal with this problem, sophisticated techniques to resolve connectivity are necessary.

This chapter explores an algorithm that operates on both structured and unstructured meshes and scales well even with very large data, as well as its underlying performance characteristics. The algorithm executes in multiple stages, ultimately constructing a unique label for each connected component and marking each vertex with its corresponding connected component label. This final labeling enables analyses such as: calculation of aggregate quantities for each connected component, feature based filtering of connected components, and 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 chapter presents the algorithm (§1.4), results from a weak scaling performance study (§1.5), and further analysis of the slowest phase of the algorithm (§1.6).

1.2 Related Work

The majority of research to date in connected components algorithms has been focused on computer vision and graph theory applications. This previous research is useful for contributing high-level ideas, but ultimately the algorithms themselves are not directly applicable to the problem considered here. Computer vision algorithms typically depend on the structured nature of image data, and so cannot be easily applied to unstructured scientific data. Graph theory algorithms are more appropriate, since the cell abutment relationships in an unstructured mesh can be encoded

as an undirected graph representation. But this encoding results in a very sparse graph, with the edges having special properties — neighboring cells typically reside on the same processing elements, although not always — that graph theory algorithms are not optimized for. For more discussion of these algorithms, we refer the reader to [8]. That said, previous graph theory research on connected components has used the Union-find algorithm [6], which is also used for the algorithm described in this chapter. Further, the Union-find algorithm and data structures have been used in topology before, for the efficient construction of Contour Trees [3] and Reeb Graphs [13]. Union-find is discussed further in §1.3.1.

The algorithm described in this chapter is intended for distributed-memory parallelism. With this technique, Processing Elements (PEs) — instances of a program — are run on each node, or on each core of a node. By using multiple nodes, the memory available to the program is larger, allowing for processing of 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 ParaView [1] and VisIt [4], follow this distributed-memory parallelization strategy. Both of these tools instantiate identical visualization modules on each PE, and the PEs 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 these pieces over their PEs. This approach has been shown to be effective; VisIt performed well on meshes with trillions of cells using tens of thousands of PEs [5]. The algorithm described in this chapter follows the strategy of partitioning data over the PEs and has been implemented as a module inside VisIt. It uses the Visualization ToolKit (VTK) library [11] to represent mesh-based data, as well as its routines for identifying cell abutment within a piece.

1.3 Algorithm Building Blocks

This section describes three fundamental building blocks used by the algorithm. The first is the serial Union-find algorithm, which efficiently identifies and merges connected components. The second is binary space partitioning trees, which enable efficient computation of mesh intersections across PEs. The third is the concepts of exteriors cells and ghost data, which significantly accelerate the algorithm.

1.3.1 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 par-

tition 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 [12]. $\alpha(N)$ 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 a direct distributed-memory parallel implementation.

1.3.2 Binary Space Partitioning (BSP)

A binary space partitioning (BSP) [7] divides two- or three-dimensional space into a fixed number of pieces. BSPs are used in the connected components labeling algorithm described in this chapter to determine if a component on one PE abuts a component on another PE (meaning they are both actually part of a single, larger component). The BSP is constructed so that there is a one-to-one correspondence between the PEs and the pieces of the BSP tree. Explicitly, if there are N PEs, then the BSP will partition space into N pieces and each PE will be responsible for one piece. The PEs then relocate their cells according to the BSP; each cell is assigned a piece from the BSP based on its partition, and then that cell is sent to the corresponding PE.

It is important that the BSP is balanced, meaning that each piece has approximately the same number of cells. If disproportionately many cells fall within one piece, then its PE may run out of memory when the cells are relocated. As a result, the PEs must examine the cells and coordinate when creating the BSP.

The BSP construction and cell relocation can be very time consuming. More discussion of their complexity can be found at the end of this chapter (§1.6).

1.3.3 Exterior Cells and Ghost Cells

Exterior cells and ghost cells are used by the algorithm to reduce the amount of data needed to coordinate between PEs. Both techniques identify cells that are on the boundary of a PE's piece. Ghost cells identify exactly the cells on the boundary, while exterior cells identify a superset of the boundary cells.

Exterior cells are the cells that lie along the exterior of a volume, which does not necessarily strictly correspond to the exterior of the PE's piece. Consider the example of removing a material: the exterior cells of the remainder will likely have

a portion along the PE piece boundary, but it will also likely have a portion along the interior of the piece, where the material interface lies.

"Ghost cells" are the result of placing 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 analysis tool. They are typically created to prevent interpolation artifacts at piece boundaries. More discussion of ghost cells can be found in [4] and [9].

Ghost cells are also useful for connected components labeling. They identify the location of the boundary of a piece and provide information about the state of abutting cells in a neighboring piece. Note that the results discussed in this chapter uses ghost cells that are generated at run-time, using the collective pattern described in [4], not the streaming pattern described in [9].

1.4 Algorithm

The algorithm identifies the global connected components in a mesh using five phases. It first identifies which pieces are at the boundary (Phase 1). It then identifies the connected components local to each PE (Phase 2) and then creates a global labeling across all PEs (Phase 3). It next determines which components span multiple PEs (Phase 4). Finally, it merges the global labels to produce a consistent labeling across all PEs (Phase 5). 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 parallel considerations, Phases 1 and 2 are embarrassing parallel, Phase 3 is a trivial communication, Phase 4 has a large all-to-all communication, followed by embarrassingly parallel work, and Phase 5 has trivial communication following by more embarrassingly parallel work.

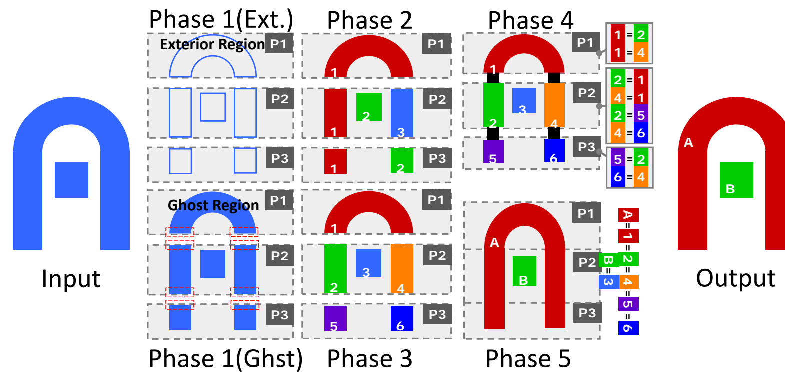

Fig. 1.2 Example illustrating the five phases of the algorithm on a simple data set decomposed onto three PEs. Phase 1 has two variants, and both variants are shown — "exterior cells" on the top and "ghost cells" on the bottom.

Phase 1: Identify cells at PE boundaries: The goal of this phase is to identify cells that abut the spatial boundary of the data contained on each PE. We consider two methods for doing this: ghost data and exterior cells. The ghost data option is superior, since ghost data always lies along the PE boundary, and so only the minimum number of cells need to be considered. The exterior cells option is inferior, since some cells are external to a component, but interior to the PE boundary and these cells cannot be distinguished and thus must be processed unnecessarily. Unfortunately, ghost cells are not present in all data sets; the algorithm uses the ghost data option when ghost data is present, and falls back to the exterior cells option when it is not.

Phase 1, ghost cells option: Ghost cells are useful because they are always adjacent to boundary cells; finding the cells adjacent to ghost cells is equivalent to finding the list of cells on the boundary. Note that ghost cells cannot be used directly to represent PE 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, all ghost cells are removed 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 1, exterior cells option: Again, exterior cells are the cells that are on the exterior of the components. Only the cells on the boundary need to be considered in Phase 4, and these cells are a superset of the cells on the boundary. However, they are a subset of all cells and discarding the cells in the interior of the components substantially improves Phase 4 performance.

The benefit of this approach varies based on the data set. If a component has a high surface area to volume ratio, then proportionally less cells will be in the interior and the number of cells discarded is less. Further, the proportion of exterior cells that are not on the boundary compared to those that are on the boundary is data dependent. That said, a factor of 4X to 10X reduction is typical in the number of cells processed in Phase 4 by focusing on exterior cells.

The exterior cells can be calculated by using a standard “external faces” algorithm. For each face, look at the number of cells incident to that face (or each edge in two dimensions). The faces that have one cell incident to it are exterior, and so those cells are marked as exterior.

Phase 2: Identify components within a PE: The purpose of this phase is for each PE to label the connected components for its portion of the data. As mentioned in §1.3.1, 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. It then traverses the cells in the mesh. For

each cell, it identifies 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 PE is proportional to the number of cells contained on that PE.

Phase 3: Component re-label for cross-PE comparison: At the end of Phase 2, on each PE, the components within that PE's data have been identified. Each of these components has a unique local label and the purpose of Phase 3 is to transform these identifiers into unique global labels. This will allow the algorithm to perform parallel merging in subsequent phases. Phase 3 actually has two separate re-labelings. First, since the Union-find may create non-contiguous identifiers, it transforms the local labels such that the numbering ranges from 0 to N_P, where N_P is the total number of labels on Processing Element P. For later reference, we denote $N = \sum N_P$ as the total number of labels over all PEs. Second, the algorithm constructs a unique labeling across the PEs by adding an offset to each range. It does this by using the PE rank and determining how many total components exist on lower PE ranks. This number is then added to component labels. At the end of this process, PE 0 will have labels from 0 to $N_0 - 1$, PE 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 4: Merging of labels across PEs:

At this point, when a component spans multiple PEs, each PE's sub-portion has a different label. The goal of Phase 4 is to identify that these sub-portions are actually part of a single component and merge their labels. The algorithm does this by re-distributing the data using a BSP (see §1.3.2) and employing a Union-find strategy to locate abutting cells that have different labels. Only the cells that lie on the boundary are needed to do this location. The cells identified in Phase 1 are used in the search process, but the cells known not to be on the boundary are excluded, saving about an order of magnitude in the number of cells considered.

The Union-find strategy in Phase 4 has four key distinctions from the strategy described in Phase 2:

- The labeling is now over cells (not points), which is made possible by the scalar field added in Phase 3.
- The algorithm now merges based on cell abutment, as opposed to Phase 2, where cells were merged if it had two points incident.
- Each cell is initialized with the unique global identifier from the scalar field added in Phase 3, as opposed to the arbitrary unique labeling imposed in Phase 2.
- Whenever a union operation is performed, it records the details of that union for later use in establishing the final labeling.

In pseudocode:

```
CreateBSP()
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, the re-distributed data is discarded and each PE returns to operating on its original data.

Phase 5: Final assignment of labels: Phase 5 incorporates the merge information from Phase 4 with the labeling from Phase 3. Recall that in Phase 3 the algorithm constructed a globally unique labeling of per-PE components and denoted N as the total number of labels over all PEs. The final labeling of components is constructed as follows:

- After Phase 4, each PE is aware of the unions it performed, but not aware of unions on other PEs. However, to assign the final labels, each PE must have the complete list of unions. So Phase 5 begins by broadcasting ("all-to-all") each PE'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 3 to a unique label for each connected component.
- Use the "Find" method to transform the labeling from the scalar array created in Phase 3 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 PE 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.

1.5 Performance Study

The efficiency of the algorithm was studied with a performance study that used weak scaling on concurrency levels up to 2197 cores (and 2197 PEs) with data set sizes up to 21 billion cells. The study used 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.

1.5.1 Problem Setup

The data input came from a core-collapse supernova simulation produced by the Chimera code [2]. This data set was selected because it contains a scalar entropy field with large components that span many PEs. A data set was generated for each concurrency, using upsampling to ensure each PE would operate on a fixed number of cells. Interval volumes — the volume that lies between two isosurfaces, one with the "minimum" isovalue and one with the "maximum" isovalue — were extracted from the upsampled structured grid to create an unstructured mesh as input to the connected components algorithm.

The following factors were varied:

- Concurrency (12 options): Levels varied from 8 cores (2^3) to 2197 cores (13^3).
- Data sets (2 options): Data sizes with one million cells per PE and 10 million cells per PE were run. Table 1.1 outlines the data sizes for the latter case.
- Phase 1 Variant (3 options): Both the ghost cell and exterior cells variants of the algorithm were tested, as well as a variant with no identification of cells at PE boundaries (i.e., no Phase 1), since this variant was presented in previous work.

The cross product of tests were run, meaning $12 \times 2 \times 3 = 72$ tests.

Figure 1.1 shows rendered views of the largest interval volume data set used in the scaling study and its corresponding labeling result.

Table 1.1 Scaling study data set sizes for the runs with 10 million cells per PE. The study targeted PE counts equal to powers of three to maintain an even spatial distribution after upsampling. The highest power of three PE count available on the test system was $13^3 = 2197$ PEs, so PE counts from 8 to 2197 and initial mesh sizes from 80 million to 21 billion cells were studied. The interval volume 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.

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

1.5.2 Results

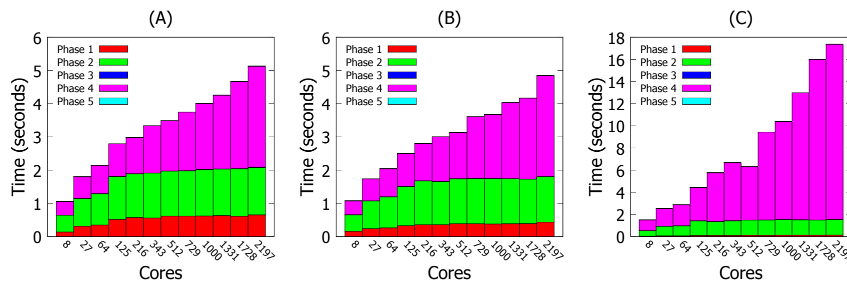

Fig. 1.3 Scaling study using one million cells per PE. Each figure corresponds to a variant for running Phase 1 and plots the timings for the five phases for each of the twelve concurrency levels for that variant. (A) shows the ghost cells variant. (B) shows the exterior cells variant. (C) shows the variant with no reduction of cells exchanged, which was presented in previous work and is included for comparative purposes. Figures A and B are very similar in performance and are on similarly scaled axes. Figure C performs significantly slower and is on a different scale. The time spent in Phase 1 for the ghost cell and exterior cell variants — which is not present in the third variant — leads to substantial savings in Phase 4. Phases 3 and 5 are negligible across all versions of the algorithm.

Figures 1.3 and 1.4 present the timing results from the cross product of tests. As expected, the timings for Phases 2, 3, and 5 are consistent between all variants of the algorithm. At 125 PEs and beyond, the largest subset of the interval volume on a single PE approaches the maximum size, either one million or 10 million cells depending on the study. For this reason, weak scaling for Phase 2 is expected. This is confirmed by flat timings for Phase 2 beyond 125 PEs. The ghost cell variant and exterior cell variant perform comparably in Phase 4, and both significantly outperform

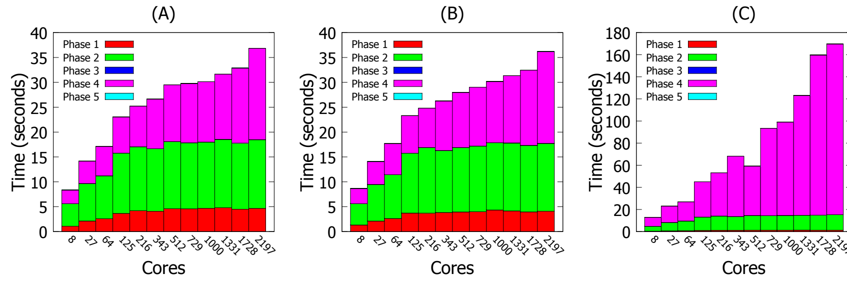

Fig. 1.4 Scaling study similar to that described in Figure 1.3, except using 10 million cells per PE. As expected, the performance is proportional for the unoptimized variant of Phase 1, and nearly proportional for the two optimized variants.

Table 1.2 Cells exchanged in Phase 4 for each variant for the 10 million cell per PE test variant. The total listed for each variant is the percentage of the total number of cells for that concurrency level.

Number of Cores	Total cells	Ghost cell variant	Exterior Cell variant	No Optimization
$2^3 = 8$	10.8M	1.2%	11.1%	100%
$3^3 = 27$	34.9M	2.3%	9.1%	100%
$4^3 = 64$	80.8M	2.4%	7.7%	100%
$5^3 = 125$	155M	2.6%	6.9%	100%
$6^3 = 216$	266M	2.6%	6.2%	100%
$7^3 = 343$	419M	2.7%	5.7%	100%
$8^3 = 512$	622M	2.7%	5.4%	100%
$9^3 = 729$	881M	2.7%	5.1%	100%
$10^3 = 1000$	1.2B	2.7%	4.9%	100%
$11^3 = 1331$	1.6B	2.7%	4.6%	100%
$12^3 = 1728$	2.1B	2.7%	4.5%	100%
$13^3 = 2197$	2.6B	2.7%	4.3%	100%

the variant with no boundary selection. These timings demonstrate the benefit of identifying per-PE spatial boundaries. The small amount of additional preprocessing time required for Phase 1 creates significant reduction in the number of cells transmitted and processed in Phase 4, as shown in Table 1.2.

Although the amount of data per PE is fixed, the number of connectivity boundaries in the interval volume increases as the number of PEs increases. This is reflected by the linear growth in both the number of union pairs transmitted in Phase 5 and the number of cores spanned by the largest connected component (See Table 1.3).

Table 1.3 Largest component information and number of global union pairs transmitted. There is a linear correlation (0.994322) between the number of cores spanned by the largest connected component of the interval volume and the number of union pairs transmitted in Phase 5. The percentage of cores spanned by the largest component converges to slightly less than 25%.

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

1.6 BSP Generation

BSP generation is described in this section, and its performance is analyzed. §1.6.1 describes Recursive Coordinate Bisection (RCB), a technique for generation BSPs. RCB requires a data structure for doing spatial searches; §1.6.2 explores the relative advantages of octrees and interval trees.

1.6.1 Recursive Coordinate Bisection (RCB)

RCB [10] is an algorithm that takes a list of points and a target number of regions and generates a BSP that partitions space such that every region in the BSP contains approximately the same number of points. The list of points is distributed across the PEs, so the RCB algorithm must operate in parallel. Again, in this context, the target number of partitions is the number of PEs, so that each PE can own one region. It is important that each region contains approximately the same number of points, otherwise a PE might receive so many points that it will run out of memory.

RCB starts by choosing a "pivot" to divide space. In the first iteration, the pivot is a plane along the x-axis (e.g., "X=2"). The pivot should divide space such that half of the point list is on either side of the plane. The algorithm then recurses. It embarks to find a plane in the y-axis for each of the two regions created by the initial split. These planes may be at different y locations. The algorithm continues iterating over the regions, splitting over X, then Y, then Z, then X again, and so on. At each

step, it tries to split the region so that half of the points are on each side of the plane (with modifications for non-powers of two). This process is illustrated in Figure 1.5.

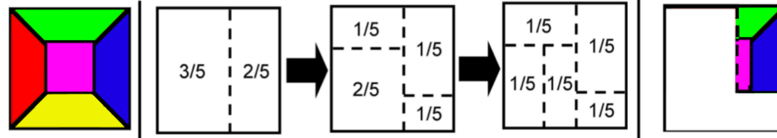

Fig. 1.5 RCB construction of a BSP-tree in a distributed memory setting. On the left, the decomposition of the original mesh. Assume the red portions are on PE 1, blue on 2, and so on. The iterative strategy starts by dividing in X, then in Y, and continues until every region contains approximately $1/N_{PEs}$ of the data. Each PE is then assigned one region from the partition and the data is communicated so that every PE contains all data for its region. The data for PE 3 is shown on the far right.

A key issue for RCB is pivot selection. The pivot selection requires iteration, with each proposed pivot requiring examination of how many points lie on either side. Previous RCB constructions [10] have used randomized algorithms. These algorithms choose a pivot at random, and then restrict the points considered to be those on the majority side of the pivot.

Although it is possible to iterate on pivot location until equal numbers of points lie on either side of the pivot, it is not practical. Each pivot candidate requires parallel coordination. And a pivot that includes the balance by a negligible amount will not substantially improve further processing in Phase 4. So, to make RCB execution go faster, the requirement of a balanced spatial partition was relaxed. As a result, the points in each region were unbalanced. The effective tradeoff is decreased RCB execution time versus increased time later in Phase 4 by the PEs that got more than the average number of points.

The algorithm presented in this chapter used a modified version of RCB (not a randomized version). It chooses five potential pivot points that are evenly spaced through the volume. It then identifies the pair of pivots that contain the ideal pivot and places another five evenly spaced pivots between them. The best choice of those five is the pivot. This method works well in practice, because it minimizes communication, as the five pivot selections can be communicated simultaneously. Table 1.4 analyzes the communication time, while Table 1.5 shows the effective balance of the modified "2-Pass" RCB scheme on a large problem.

1.6.2 Octrees and Interval Trees

For each pivot candidate, the algorithm must determine how many points lie on either side of its dividing plane. If there are N PEs, then $N-1$ pivots must be chosen (this can be thought of as the number of interior nodes in a binary tree). Each pivot likely involves some number of candidate pivots (10 in the case of this algorithm). If

Table 1.4 Comparison of communication patterns for randomized RCB and the modified "2-Pass" RCB scheme used in this study.

Factor	Randomized RCB	2-Pass RCB
Num. passes	N	2
Num. communications per pass	1	1
Num. searches per pass	1	5
Total	N communications, N searches	2 communications, 10 searches

Table 1.5 Breakdown of balance of points in BSP created by "2-Pass" RCB method for run with 2197 processors. The maximums are no more than 30% bigger than average and the minimums are no more than 25% smaller than average. These inequities were deemed desirable in the context of reduced parallel communication.

Problem	Maximum	Average	Minimum
2197 pieces with 2.2B cells	1.3M	1.0M	805K
2197 pieces with 10.9B cells	6.4M	4.9M	3.8M

done poorly, this would involve $(10N-10) \times O(numpoints)$ comparisons. Placing the points in a search structure can substantially reduce the cost for checking on the quality of a pivot candidate.

This section considers the tradeoffs between octrees and interval trees, comparing the speed of searches on two data sets meant to model the best and worst case scenario for an interval tree. Since the interval tree is derived by data points and not spatially, ordering of data has a profound effect on the efficiency of the model. (Note that sorting the data points spatially considerably increases overall initialization time.) The octree is spatially created, and will, therefore, always create the same outcome whether the data is sorted or unsorted. Both data sets contained one million points evenly spread across a rectilinear grid with one file being sorted and the other file optimally unsorted. The test involved running the octree and interval tree models against five different searches using both types of input data. The first search covered the entire bounding box. Each of the following four searches decreased the bounding size by half of the previous size. Again, this technique was designed to measure the best case search for the octree to the worst case search for the octree, given its early termination criteria. Results are shown in Figure 1.6.

Although the first tests were fairly conclusively, additional tests were performed on real world data, namely a data set operated on by one PE from the scaling study. Results are shown in Figure 1.7. The results for the tests show that the octree maintains a consistent and very quick search time with low variation, on the order of ten thousandths of a second. The interval tree however performed quite poorly in some cases, on the order of hundredths of a second, and excelled in others, reaching one ten thousandth of a second. The interval tree's quick regions were identified as those which did not contain any data. The interval tree was able to use an early termination scheme to quickly decide if there were no points in the search bounds. But when it

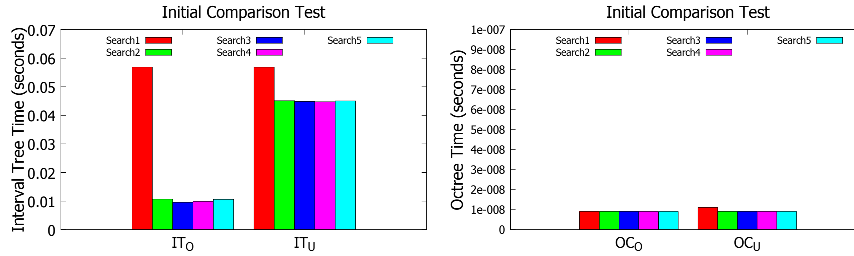

Fig. 1.6 Interval Tree (IT) tests are on the left, Octree (OC) tests are on the right, each with ordered (o) and unordered (u) variants. Searches #1-#5 are from largest bounding box to smallest, respectively. The run time for octree was significantly faster (note the axes have different scales), and was sometimes less than the precision of the timing instrumentation.

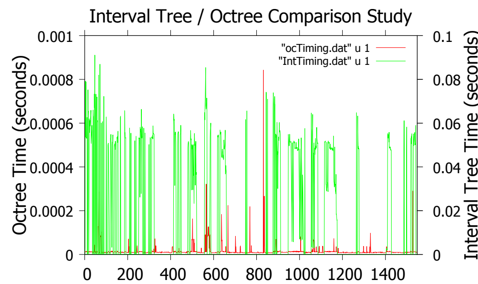

Fig. 1.7 The tree structures were built with a sample of data that existed on one of the 128 processors during a testing phase of the algorithm. Each of the 1548 searches that were done on that data were also recorded and represents the x axis. The y axes are time in seconds for the octree and interval tree respectively to complete the requested search. Two y axes were provided to visualize both data sets on the same plot at different scales since the octree worst case search time (approximately 0.0009 seconds) is on the order of the interval trees fastest search time (approximately 0.0001 seconds).

could not terminate early, it spent a good deal of time adding up the points that were within the range. The octree, on the other hand, was able to use early termination criteria to deal with these cases. These tests clearly solidified the octree as the better of the two data structures in question for this task, which had a run time similar to the interval tree's best case for every case.

1.7 Summary

This chapter described a distributed-memory parallel algorithm that identifies and labels the connected components in a domain-decomposed mesh. The labeling produced by the algorithm provides a topological characterization of a data set that enables important analyses. The algorithm is designed to fit well into currently deployed distributed-memory visualization tools, and this was demonstrated in a scaling study.

Acknowledgments

This work was supported by the Director, Office of Advanced Scientific Computing Research, Office of Science, of the U.S. Department of Energy under Contract DE-AC02-05CH11231, was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344, and was also supported in part by the National Science Foundation, grants OCI-0906379 and OCI-0751397.

References

1. Ahrens, J., Geveci, B., Law, C.: Visualization in the ParaView Framework. In: Hansen, C., Johnson, C. (eds.) The Visualization Handbook, pp. 162-170. (2005)
2. 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, pp. 593-601. (2008)
3. Carr, H., Snoeyink, J., Axen, U.: Computing contour trees in all dimensions. In: Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms, pp. 918-926. Society for Industrial and Applied Mathematics, Philadelphia (2000)
4. H. Childs, E. Brugger, B. Whitlock, J. Meredith, S. Ahern, K. Bonnell, M. Miller, G. H. Weber, C. Harrison, D. Pugmire, T. Fogal, C. Garth, A. Sanderson, E. W. Bethel, M. Durant, D. Camp, J. M. Favre, O. Rübel, P. Navrátil, M. Wheeler, P. Selby, F. Vivodtzev. VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data. In Proceedings of SciDAC 2011, July 2011.
5. Childs, H., Pugmire, D., Ahern, S., Whitlock, B., Howison, M., Prabhat, M., Weber, G., Bethel, E.W.: Extreme Scaling of Production Visualization Software on Diverse Architectures. IEEE Comput. Graph. Appl. **30**, 22–31 (2010)
6. Cormen, T.H., Stein, C., Rivest, R.L., Leiserson, C.E.: Introduction to Algorithms, 2nd ed. McGraw-Hill Higher Education (2001)
7. 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, pp. 124-133. ACM, New York (1980)
8. Harrison, C., Childs, H., and Gaither, K.P.: Data-Parallel Mesh Connected Components Labeling and Analysis. EuroGraphics Symposium on Parallel Graphics and Visualization (EGPGV), pages 131–140, Apr. 2011.
9. Isenburg, M., Lindstrom, P., Childs, H.: Parallel and Streaming Generation of Ghost Data for Structured Grids. IEEE Comput. Graph. Appl. **30**, 32–44 (2010)
10. Nakhimovski, I.: Bucket-Based Modification of the Parallel Recursive Coordinate Bisection Algorithm. In: Linköping Electronic Articles on Computer and Information Science, 2(15), Dec. 1997.
11. 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, pp. 93–ff. IEEE Computer Society Press, San Francisco (1996)
12. Tarjan, R.E.: Efficiency of a Good But Not Linear Set Union Algorithm. J. ACM. (1975) doi: 10.1145/321879.321884
13. Tierny, J., Gyulassy, A., Simon, E., Pascucci, V.: Loop surgery for volumetric meshes: Reeb graphs reduced to contour trees. IEEE Trans. Vis. Comput. Graphics. **15**, 1177–1184 (2009)