MORSE DECOMPOSITIONS OF THREE-DIMENSIONAL PIECEWISE CONSTANT VECTOR FIELDS: CONCEPTS AND EFFICIENT IMPLEMENTATIONS

by

Nicholas Brunhart-Lupo

A thesis submitted to the Faculty and the Board of Trustees of the Colorado School of Mines in partial fulfillment of the requirements for the degree of Doctor of Philosophy (Mathematical & Computer Science).

Golden, Colorado

Date _____

Signed: _____
Nicholas Brunhart-Lupo

Signed: _____
Dr. Andrzej Szymczak
Thesis Advisor

Golden, Colorado

Date _____

Signed: _____
Dr. Randy Haupt
Professor and Head
Department of Electrical Engineering & Computer Science

ABSTRACT

Discovering and analyzing vector field features, such as stationary points, periodic trajectories or separatrices, is a challenging task. Common approaches to solving this problem use numerical integration to approximate trajectories. The associated inaccuracy brings consistency issues. Accumulated error may permit approximate trajectories to cross, which breaks the assumptions of many algorithms, leading to potentially inconsistent output. Moreover, some vector fields, for example chaotic ones, have infinitely many periodic trajectories and therefore cannot be analyzed using the classical approaches that would attempt to compute all of them.

A more robust approach to vector field analysis is based on *Morse decomposition*, using regions of flow circulation as features. These regions include stationary points and periodic orbits, as well as more complex features, like chaotic dynamics. Even though some Morse decomposition algorithms use numerical integration, they are more robust than the classical methods. One such approach uses Piecewise Constant (PC) vector fields to compute regions of circulation in 2D and 2.5D. This method builds upon mathematical results related to multivalued flows to overcome discontinuities in a PC vector field. Instead of numerical integration, it uses only simple geometric operations to construct trajectories, which facilitates robust handling of numerical error. Features of interest are represented by the strongly connected components of the *transition graph*, a finite directed graph representing all possible trajectories in a PC vector field. Transition graphs have a natural multi-resolution structure that can be exploited to obtain efficient adaptive refinement algorithms for approximating the circulating regions.

In this thesis, we present three new contributions to this approach. First, the PC vector field analysis algorithm is extended to 3D to support a wider range of datasets. We describe related work leading to this modified system, implementation, and results of this

approach for a variety of analytical, and simulated data sets. Secondly, we describe a parallel implementation of the PC framework that make it applicable to larger datasets and reduce computation time. The third contribution is a distributed implementation for large scale 3D vector fields and a new representation of the transition graph to reduce memory usage.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

There are a number of people whose support has helped bring this thesis to completion.

I would like to thank my advisor for his constant encouragement throughout the years, and his unyielding optimism even during the darkest days of broken code.

I would like to thank my graduate committee for taking the time to help bring this work to fruition, and I would like to single out Dr. Gruchalla in particular for his writing advisement and computing resources that made this thesis possible. The staff and resources of NREL were extremely valuable to my work and greatly appreciated.

Last, but certainly not least, I would like to thank my parents for literally providing the foundations for any success I might have.

CHAPTER 1

INTRODUCTION

This thesis describes the theory and implementation of a robust Piecewise Constant (PC) 3D vector field feature extraction algorithm. We also present two modifications to the algorithm; one to maximize parallelism on a shared memory computer and a second to permit analysis using distributed machines. In this chapter, we define the problem domain, and the layout of this thesis.

Vector fields are of great interest to researchers in a variety of disciplines. Fluid simulation, computer vision, and biology are just a few domains where data may be represented by a vector field. Features that have traditionally attracted the greatest interest of researchers and domain experts represent stagnant or circulating flow. The simplest examples of such features are stationary points and periodic trajectories. Other features of interest include boundaries between coherent flow regions which represent natural barriers with relatively small amount of mixing of fluid across them. In the time-independent case considered in this proposal, these structures are known as separatrices.

A representative case is the cooling jacket of an engine, detailed in Laramee et al. [1]. In this problem, the goal is to create an efficient cooling system to ferry waste thermal energy from an internal combustion engine. An image of such a device can be seen in Figure 1.1. A number of constraints are present: the jacket must conform to the shape of the engine, it must weigh as little as possible and it must permit the engine to reach an optimal operating temperature. Further, the flow of the coolant is an important issue. Flow must reach each cylinder, should not stagnate, must not reach a very high velocity, and should minimize pressure loss. Some of these constraints can be mapped to 3D vector field features. In particular, stationary points or periodic trajectories, that could point one to areas of stagnant of circulating flow in the vector field would be indicative of stagnating

coolant, since thermal energy would not be properly transported away.

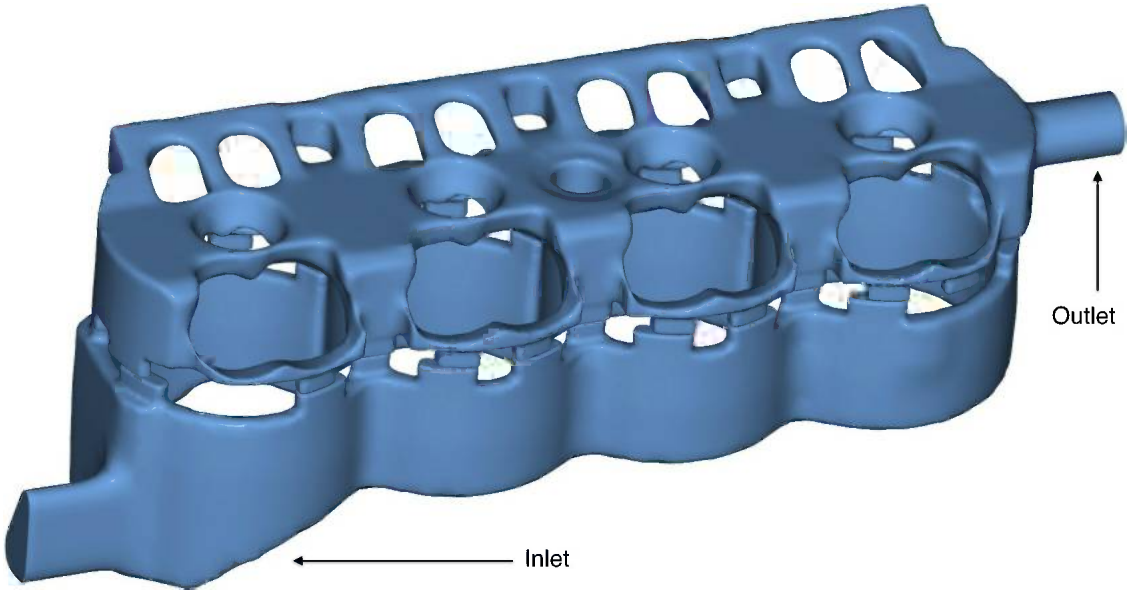


Figure 1.1: A cooling jacket for an engine. Colors indicate temperature. Note complex jacket geometry.

We can also consider the problem of optimal combustion within the cylinders themselves. Garth et al. [2] note that there is a preferred mixing pattern for various engine types, consisting of tumbles and swirls. These patterns generally map to 3D vector field features that can be extracted. Moving away from engines, there are applications to wind velocities in hurricanes, or airflow around a turbine. Even chemical reaction studies may make use of 3D vector field analysis.

There are nontrivial difficulties in 3D vector field analysis; for example, there can be a large number of features residing in the datasets, making analysis of the flow and extracting interesting features demanding. Further, some methods are not fully automatic, in that they require some manual tuning of parameters in order to find features (such as step size for numerical integration, etc). Also, most solutions make use of numerical integration, which can introduce numerical problems in the analysis, potentially leading to inconsistent results as well as being time consuming.

Prior work focuses primarily on the extraction of traditional features (fixed points, periodic orbits, connecting trajectories) from 2D or 2.5D fields (i.e. planar and surface cases, respectively). As we discuss in the following chapter, most of these methods are prone to numerical problems. For some methods, the analysis of vector fields in these domains is taxing both in problem formulation and execution. Extending them to 3D will either increase the difficulty of the formulation, or make analysis infeasible.

This has led to the use of combinatorial depiction of the flow, based on graph representation [3, 4]. Such approaches are enticing, as they can be highly efficient and do not suffer from the same consistency issues that one would find in an algorithm that relies on numerical integration. Furthermore, approaches focused on finding classical features cannot deal with vector fields containing chaotic dynamics (such as the Lorenz equations [5]), since they typically contain infinitely many periodic orbits. On the other hand, chaotic systems have been rigorously analyzed using set-theoretic approaches similar to the one proposed by Mrozek and Pilarczyk [6].

Turning to the data sets themselves, most simulations are 3D, including the engine cooling jacket data set and the combustion chamber data set. As stated before, most approaches have been developed for 2D or 2.5D data. Therefore, they can only be used for slices through a 3D dataset, or, more generally, data restricted or extrapolated to a 2D surface. While this can yield interesting results, there is a certain loss of information when tackling the data sets in this manner.

In response, this thesis describes the theory and implementation of a robust Piecewise Constant (PC) 3D vector field feature extraction algorithm, extending the PC framework presented by Szymczak and Zhang [7] to 3D.

In Chapter 2, we examine previous efforts in this field of study, as well as describing numerical integration based techniques and their relative disadvantages. Later, we discuss combinatorial vector fields and Morse decompositions. This examination includes the work that we directly build from, i.e. the work of Szymczak and Zhang [7].

Chapter 4 discusses our 3D approach in detail, with a corresponding implementation. We also present results for selected data sets, demonstrating the ability to extract meaningful features, and our conclusions. The initial work on this 3D variant has been published in the Computer Graphics Forum journal, and presented at *EuroVis 2012* [8].

Chapter 5 describes a parallel version of the 3D algorithm designed for a shared memory machine, based on a novel dual-graph approach. We detail the algorithm, and its differences with the original, discuss the implementation, and present results derived from a subset of the datasets in Chapter 4. This work has been presented at *Topoinvis '13* and published as a book chapter[9].

In Chapter 6, we present a parallel implementation of the algorithm suitable for distributed machines based on a message passing interface (MPI). We also introduce a new, compact representation of the transition graph that allows us to reduce the memory usage by approximately 50%. This version of the algorithm has allowed us to analyze datasets of sizes well beyond reach of the current graphics hardware.

Chapter 7 summarizes the contributions of this thesis and discusses the direction of future work.

CHAPTER 2
LITERATURE REVIEW

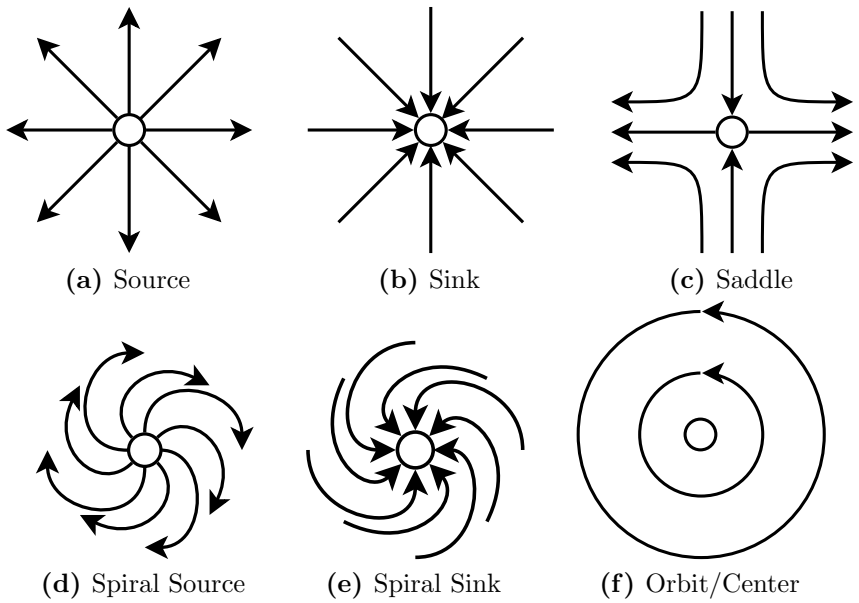
The goal of vector field topology has traditionally been to discover topological features of vector fields, such as stationary points, periodic orbits, and separatrices. In this chapter, we will define these features and outline common methods to extract them. We will also discuss more recent approaches (i.e. discrete vector fields and Morse decompositions) whose goal is to improve the robustness of vector field topology.

2.1 Features

In vector field topology, one is typically interested in features that are stable under small perturbation of the vector field (structurally stable features). Features that do not have this property may be a result of noise or numerical inaccuracy and therefore they should not be relied on in vector field analysis. Below, we discuss prior work on finding structurally stable stationary points, periodic orbits, and connecting trajectories for visualization applications. For this discussion, we will be focusing on non-degenerate scenarios to aid in clarity.

2.1.1 Stationary Points

We can rope sources, sinks, and saddle points into a category of *stationary points*. Generally, stationary points are points where the vector field vanishes. Sources are repelling points from which trajectories radiate: a trajectory followed from a point close to a source backward in time will converge to it. Conversely, sinks are points to which trajectories are attracted. Following trajectories from a point close to a sink forward in time will converge to that sink. A saddle point is a stationary point as well; following a trajectory close to a saddle forward in time will either converge or diverge depending on the trajectory chosen. Examples of these stationary points are shown in the top row of Figure 2.1.



(a) Source **(b)** Sink **(c)** Saddle

(d) Spiral Source **(e)** Spiral Sink **(f)** Orbit/Center

Figure 2.1: Types of stationary points, as identified by Helman and Hesselink [10].

We can compute these points directly from the vector field. As noted by Chen et al. [3], significant work has been done in regards to stationary points. One such example is from Helman and Hesselink [10]; they demonstrate how stationary points in a vector field can be identified and classified based on the behavior of the vector field around that point.

A standard method to distinguish sources, sinks, and saddles is based on linearization of the vector field, or more precisely, on the Jacobian of the vector field at the stationary point. The real portions (R_1 and R_2) of the eigenvalues describe the attraction and/or repulsion of the field. The imaginary components, I_1 and I_2, quantify circulation (see Table 2.1).

Table 2.1: Stationary point classification, by Helman and Hesselink [10]. See Figure 2.1 for reference.

Classification	Real Part	Imaginary Part
Sink	$R_1, R_2 < 0$	$I_1 = I_2 = 0$
Source	$R_1, R_2 > 0$	$I_1 = I_2 = 0$
Spiral Sink	$R_1, R_2 < 0$	$I_1 = -I_2 \neq 0$
Spiral Source	$R_1, R_2 > 0$	$I_1 = -I_2 \neq 0$
Saddle	$R_1 < 0, R_2 > 0$	$I_1 = I_2 = 0$
Center	$R_1 = R_2 = 0$	$I_1 = -I_2 \neq 0$

For example, a sink-type critical point would have negative R_1 and R_2. In the source case, the real portions are positive. Sinks or sources with non-zero I_1 and I_2 are spiral sinks or sources, i.e. trajectories spiral around those points infinitely many times as they converge (when followed forward or backward, respectively). For a saddle stationary point, both eigenvalues have no imaginary component; the real portions consist of one positive and one negative value.

With tools like these, stationary points can be classified. We now turn to more difficult features to extract, namely periodic orbits.

2.1.2 Periodic Orbits

Periodic orbits are trajectories that keep returning to the same point as they are followed forward/backward in time. A *stable*, or attracting, orbit is much like a sink in that trajectories starting from points close to the orbit will converge to that orbit, if we follow the periodic trajectory forward in time. If trajectories of all points near a periodic orbit converge to that orbit as they are followed backward in time, the periodic orbit is called *unstable*.

Periodic orbits are generally much harder to compute than stationary points. Periodic orbits are often found as stationary points of a *Poincaré map*, described as follows. First define a 2D plane P, called a *Poincaré section* in the 3D vector field. We now construct a point to point mapping for that plane. For any point p on P, follow its trajectory until it returns to P at a point o. The function that assigns o to p is the Poincare map for P. The fixed points of this map correspond to periodic orbits in the vector field. Further, the classification of orbits (in the sense of an attracting or repelling orbit) can be inferred from the derivative of the return map at the fixed points.

Mrozek and Pilarczyk show an example of this in action; Poincaré maps are used to rigorously prove the existence of a periodic orbit [6].

Wischgoll and Scheuermann [11] use a similar technique to discover periodic orbits in piecewise linear planar vector fields. In this case, a trajectory is followed by numerical integration until it returns to the same mesh element a number of times. This is sufficient to

establish a probable periodic orbit, and provides a triangle strip that represents the orbit. By examining areas where trajectories could enter and exit the strip and by following trajectories intersecting those areas forwards and backwards in time, the existence of a periodic orbit can be established with more confidence.

Moving from this point, Theisel et al. [12] introduced a novel way of grid-independent streamline connection. This approach applies to 2D fields, but the problem is converted to 3D by lifting the lower dimensional vector field, in this case by providing each vector with a zero for the z coordinate. This scheme requires a *seed line* to be placed in the field, with time being the seed line's third dimension. Then, using numerical integration, a surface is created from this seed line by 'pushing' it along the flow, forward and backward in time. Periodic trajectories intersecting the seed line are found by intersecting the forward and backward stream surfaces. By choosing a proper set of seed lines, one can detect all periodic orbits.

2.1.3 Connecting Trajectories & Separatrices

Connecting trajectories are trajectories that converge to a feature when followed forwards and backwards in time. There are several types of such trajectories, classified by the features at which they start and end. In the 2D case, connecting trajectories of interest connect a saddle point to a sink or attracting periodic orbit, or a source or a repelling periodic orbit to a saddle point. In 3D, isolated connections between saddles of different types and connections between stationary points and hyperbolic periodic orbits are also of interest [13].

For planar and surface cases, connecting trajectories described above are also referred to as separatrices, since they segment the flow domain into regions with the same asymptotic behavior; all trajectories followed in one region will converge to the same features as they are followed forward and backward in time.

To discover this class of features, Helman and Hesselink [10] introduced a representative algorithm. Starting at the saddle points, four trajectories are followed (two in the unstable directions, and two in the stable directions) forward and backward in time until they appear to converge to another fixed point. The authors note that its possible for two saddle points to

be connected by this approach. Such connections are structurally unstable and, in most cases, they are due to numerical integration issues and the use of arbitrarily chosen thresholds.

2.2 Algorithms & Consistency Guarantees

While each of these features can be extracted and displayed, there is a question of consistency in each case; an integrated trajectory will not be the same as the 'true' continuous trajectory. For example, the most popular Runge-Kutta methods have an associated error of $\epsilon \approx \frac{\eta}{h} + h^n$, where η is the machine epsilon, h is the step size, and n is the order of the method. One must use small step sizes to ensure that important features are not missed in the vector field. However, smaller step sizes demand additional processing time. Complicating matters, we note that $\lim_{h \to +0} \frac{\eta}{h} + h^n = \infty$. Therefore, the benefit of decreasing h diminishes or disappears if it is too small.

In general, numerically integrated trajectories can intersect (Figure 2.2). This poses a fundamental problem since many algorithms for vector field topology are based on the assumption that they *cannot* intersect. Thus, numerical integration can induce consistency problems in those algorithms, or as Bhatia et al. [14] note that this can be especially dangerous during visualization, as these issues can provide a false impression of the vector field. For example, Helman and Hesselink [10] had to add a number of special cases to their algorithm for displaying vector field topology to account for numerical integration issues.

Figure 2.2: Demonstration of numerical integration upon trajectories. The dotted lines indicate true trajectory paths, while the integrated curves are solid. Note the intersection.

Thus, other algorithms and approaches have been devised to either mitigate the effects, or circumvent the use, of numerical integration. These methods are discussed below.

2.2.1 Edge Maps

In this method, Bhatia et al. [15] introduced the notion of *edge maps*. Edge maps are a mapping of line segments obtained by subdividing the mesh edges based on vector field dynamics. The goal of this approach is to combat the problem of crossing trajectories, and enable the approximation of trajectories with arbitrary accuracy in a way that does not lead to consistency issues.

For example, in a triangulated 2D mesh we can map portions of edges of triangles that are connected by flow inside that triangle, creating an origin-destination pair. The exit point of a trajectory can be approximated as a linear function mapping the origin edge portion into the destination edge portion. If higher precision is needed, edges can be subdivided multiple times and reattached to build a finer set of origin-destination pairs, resulting in a better approximation of the trajectories of the piecewise linear (PL) vector field. Thus, following a trajectory through a triangle requires one linear interpolation operation based on an edge map, instead of using numerical integration. Note that this approach bears some resemblance to [7], with the absence of a connectivity graph.

There are issues of complexity in terms of building the edge pairs. For some triangles, the PL flow can be quite complex (the work notes twenty-three equivalence classes of edge maps).

2.2.2 Combinatorial Vector Fields

Recent work on reliable extraction of vector field topology is built off discrete Morse Theory [16]. For example, Reininghaus and Hotz [17] present a purely combinatorial approach to vector field analysis.

The first step of this approach is to build a graph representation of the flow. The graph representation is a matching between 2D simplices and incident mesh edges. The matching is computed using a maximum weight matching algorithm. Roughly speaking, the objective is to make the edges of the matching follow the direction of the flow throughout the

domain. Discrete trajectories, approximating the trajectories of the underlying vector field, are sequences of vertices in which every second pair is connected by a matching edge and all other pairs are not. They can be analyzed using simple graph search algorithms. This leads to a robust algorithm for vector field topology which in this case consists of discrete stationary points, periodic trajectories, and separatrices. Initially slow, a faster algorithm based on these concepts was put forward by Reininghaus et al. [18].

This approach carries a few drawbacks, notably in accuracy. In this case the computed trajectories deviate substantially from the exact continuous vector field seperatrices. This is a result of the formulation; trajectories are strongly restricted by the underlying mesh, and errors between this chosen path and the underlying trajectory accumulates. The trajectories will not converge to the continuous trajectories even as the mesh size (i.e. maximum diameter of a triangle) trends to zero. Gyulassy et al. [19] remedy this by increasing the accuracy (on average) for computing the Morse-Smale complex of a scalar field by employing a randomized algorithm and subdivision.

2.3 Morse Decompositions

Morse decompositions are based on the fundamental theorem of dynamical systems [20, 21]. This theorem states that any flow can be decomposed into a chain-recurrent component and a gradient-like component.

A point in the flow is chain-recurrent if there exists an ϵ-pseudo-orbit from that point to itself, where $\epsilon > 0$. A pseudo-orbit is a sequence of points that would be considered to be an orbit, under some accuracy ϵ [21].

Thus, the chain-recurrent components of the flow represent all stagnant or circulating flow regions in the system. Morse sets (computed from a finite graph representations of all trajectories) provide a set-theoretic upper bound on the chain-recurrent components, as any chain-recurrent component is contained in a Morse set.

Computation of a Morse Decomposition is detailed by Chen et al. [3], using a geometric approach for 2D and 2.5D vector fields. A piecewise linear input vector field (on a triangle

mesh) is converted into a directed graph by inserting a node for every triangle. The nodes are connected based on flow across the edges of their representative triangle. If a particle can flow from one triangle to another, they will be connected in the graph (see Figure 2.3). Once all the proper arcs have been inserted, Morse sets can be extracted by using a strongly connected component algorithm.

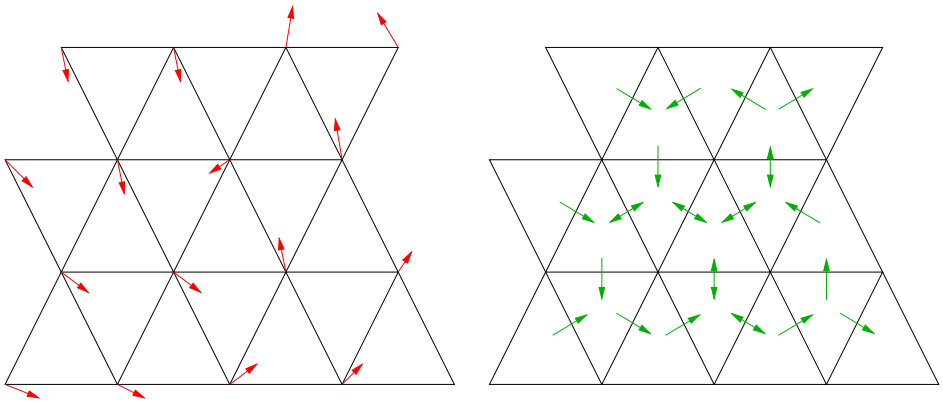


Figure 2.3: 2D triangle mesh under piecewise linear regime (Chen et al. [3]). Red arrows indicate vector field, green arrows indicate graph arc connectivity.

The advantage of Morse Decomposition is clear in this geometric method. Without a reliance on numerical integration for determining actual trajectories, and using common graph algorithms, we can efficiently determine topological features in the vector field. However, in this specific algorithm, there is a limitation in that the Morse sets are large, and may be too coarse for practical use.

Chen et al. [4] present a different method to compute a Morse Decomposition. It is based on a different graph representation of the flow, called a τ-map. In this case, a triangle T can be (using numerical integration) pushed along the flow for time τ. The resulting deformed triangle will cover a portion of the mesh. In essence, this deformed triangle represents the endpoints for all trajectories that start in the triangle and are followed for τ. All triangles this deformed triangle intersects are endpoints of arcs originating from T in the graph. As in the algorithm discussed previously, the Morse sets are computed as strongly connected components of the τ-map graph. The resulting Morse decompositions are typically much

finer than the ones obtained using the original approach.

2.3.1 Piecewise Constant Vector Field Framework

A new approach presented by Szymczak and Zhang [7] uses the previous work of Chen et al. as a starting point. Chief in distinction from its predecessors, this algorithm focuses around the use of piecewise constant vector fields (PC vector fields), and builds a finer graph representation of the flow that naturally supports results of sub-triangle accuracy. This work was originally developed for 2.5D vector fields.

A piecewise constant 2D vector field provides a non-zero vector $f(D)$ for every 2D cell D (triangle) of a triangular mesh M. In order to build a graph representation of the flow, f is extended to lower dimensional cells (edge pieces and vertices). In general, for an arbitrary lower dimensional cell C, $f(C)$ is defined as the intersection of the convex hull of vectors assigned to 2D cells containing C, and the linear space of C.

Vector fields tend to be represented in a piecewise linear form, with a vector assigned to the vertices of a mesh. Thus, the field must be converted, usually by assigning f for a 2D cell by averaging the vectors at the vertices of that cell. There is a loss of vector field accuracy when converting a continuous (piecewise linear) field to a PC vector field. This can be mitigated by building a high resolution PC mesh. As we increase the resolution (increasing the number of mesh elements), we more closely approximate the continuous vector field. It should be noted that it is commonplace for input vector fields, especially from real world datasets, to suffer measurement inaccuracy or numerical error, meaning that even features that are correct for the exact input vector field may not represent correct features of the physical system it approximates. Note that the convex-valued PC framework by Szymczak [22] can be used to obtain Morse decompositions guaranteed to be correct for continuous vector fields at the cost of substantially higher running time and memory use.

In a PC vector field, trajectories are greatly simplified; they appear as polygonal lines with knots on the mesh edges. Therefore, they can be constructed without standard numerical integration. We can see an example of such trajectories in Figure 2.4.

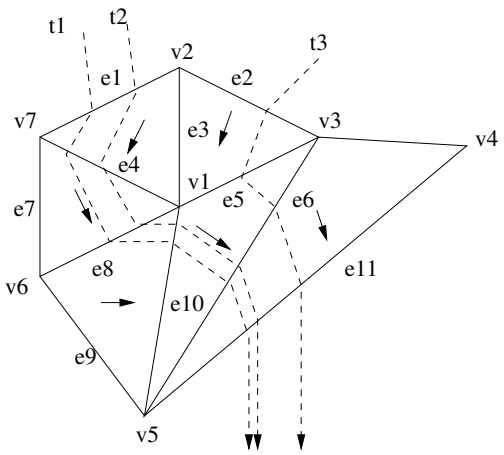


Figure 2.4: 2D triangle mesh with three trajectories (dashed lines). Vector fields for 2D mesh cells are shown by center bold arrows in each cell. Each 1D and 0D mesh cell has been labeled.

The second major contribution of this work is the *transition graph*, a finite encoding of all trajectories. Like the graph structure presented by Chen et al. [4], it connects mesh entities based on the vector field dynamics. Nodes of this graph represent edge pieces and vertices of the mesh. Edges indicate that a trajectory in the vector field are determined by the constant velocity trajectory segments within 2D cells of the mesh. An example of a transition graph is shown in Figure 2.5. Again, as in previous algorithms, features are directly discovered from this transition graph by finding the strongly connected components.

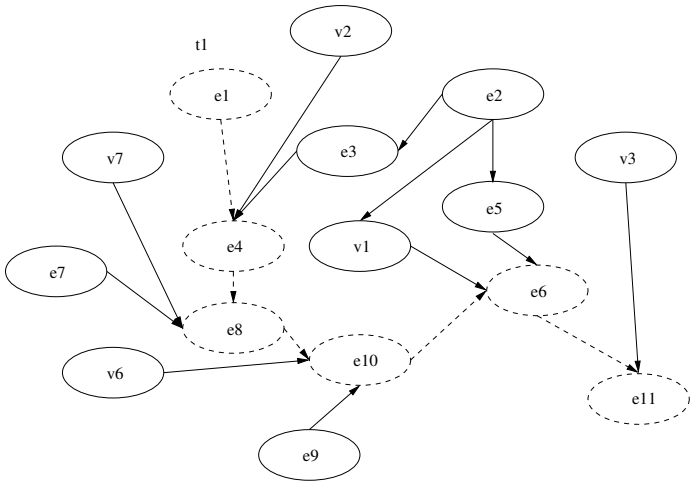
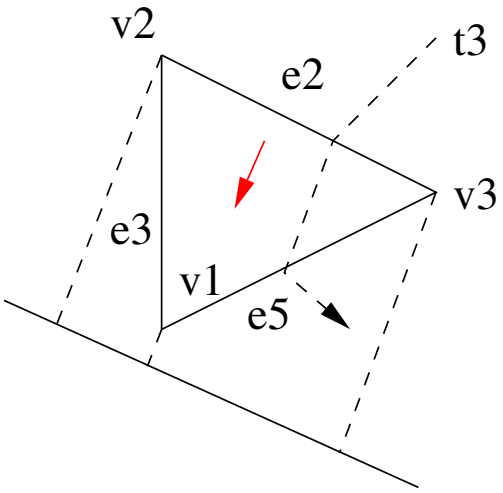


Figure 2.5: Transition graph of Figure 2.4 with trajectory record for trajectory t_1.

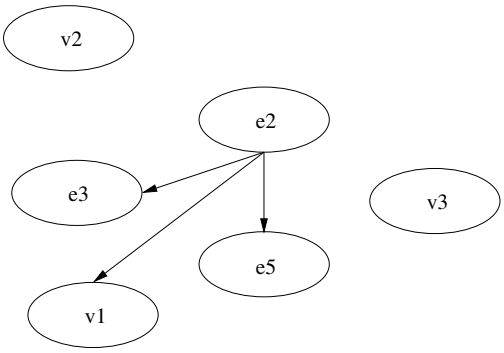
Figure 2.5 further shows that two trajectories (t_1, t_2) share the same path in the graph. This is accounted for in the algorithm, which has an additional feature, that of local refinement. Refinement is achieved by subdividing mesh element nodes and selectively reconnecting arcs, which, in the case of our example in Figure 2.4, would permit us to discriminate the paths of trajectories t_1 and t_2. In a more goal oriented sense, this has the effect of more closely approximating the vector field features, providing greater detail. There is no upper bound to the number of refinement operations, but memory requirements and processing time for the graph grow as the refinement level increases.

Figure 2.6 provides a closer look at refinement. In Figure 2.6(a), we can see a single triangle of the mesh from Figure 2.4. Here, we can make simple determinations of which mesh cells a trajectory will enter the 2D cell; in this case t_3 may enter only by e_2, v_2 and v_3, and leave only by e_3, e_5 and v_1. Lower dimensional mesh cells (in this case edges or 1D cells) can be subdivided as in Figure 2.6(c). Subdivided cells let us tighten entry and exit information: if t_3 enters by e_{2b} it cannot leave by e_3.

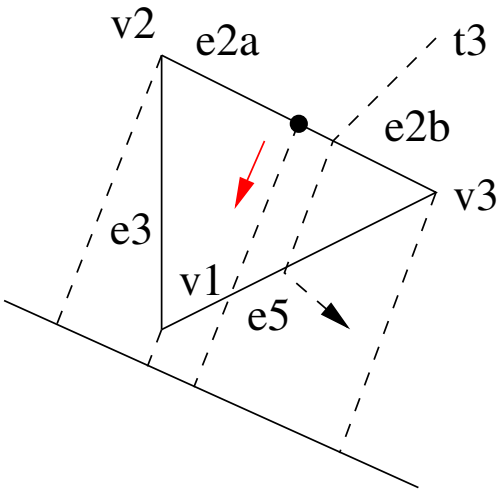
While the primary focus of this work was to analyze flow on surfaces, the tenants described are viable for extension to 3D.



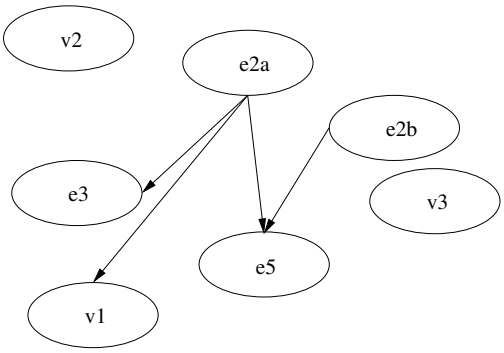
(a) Detail view of a PC vector field with a single 2D mesh cell. Lower dimensional mesh cells are projected to a space perpendicular to the vector field (red arrow).



(b) Corresponding transition graph for Figure 2.6(a).



(c) Detail view of a PC vector field with a single 2D mesh cell with a subdivided edge. Projection is same as previous, with subdivided edges shown offset in projected space.



(d) Corresponding transition graph for Figure 2.6(c). Note that the graph now has two additional nodes to match the edge pieces added by subdivision.

Figure 2.6: 2D triangle refinement detail.

CHAPTER 3
CONTRIBUTION

The contribution to vector field topology analysis is threefold; we first describe the concepts required to move piecewise constant vector fields to 3D, along with an initial implementation. We then present parallel versions of the algorithm for shared and distributed memory machines, respectively.

3.1 3D Formulation

We put forth an extension of the work done by Szymczak and Zhang [7] for use in the analysis of 3D vector fields, focusing on regular grids. We retain the transition graph, adding new rules to handle higher dimensional data sets. Also presented are new rules to handle the refinement of the graph.

This formulation permits the fast extraction of features that represent regions of circulating flow without the drawbacks of numerical integration. The graph also naturally expresses connecting trajectories, as well as simple classification of the circulating flow regions. A simple naive parallelization of the algorithm is used to speed refinement.

The use of Morse Decomposition is relatively new, thus we have validated the output of this approach with simple analytical vector fields (such as the Lorenz equations [5]) that have a known outcome. To examine its usefulness in the context of real world problems for visualization, we have also investigated the case of simulated data of real phenomenon, such as the Hurricane Isabel dataset [23].

3.2 Parallel Formulation

We update the algorithm (for both 2D and 3D) to make use of a multicore shared memory architecture, and test a 3D implementation against the same datasets as in the basic 3D formulation.

The simple naive parallelism used previously is removed and the basic tenants of the algorithm reworked to fully exploit parallel hardware, including the use of double-buffered transition graphs.

3.3 Distributed Formulation

We revise the representation of the transition graph to reduce memory usage by a significant fraction. Importantly, we modify the algorithm to support distributed computing environments, and analyze larger-scale datasets that were impossible to study on desktop hardware.

Depending on the input vector field, a high number of refinement iterations may be required to extract or distinguish complex features. The transition graph is a rich representation of trajectories that increases in size exponentially with the number of refinement passes. For 3D, this may make the transition graph unwieldy and consume vast amounts of memory. In 2D, computing connecting trajectories or stable Morse sets can cause the same issue. For example, we take the case of the Ocean dataset (2D) [24], which is sufficiently large and complex that previous implementations cannot process it in full.

This part of the thesis uses a distributed transition graph representation suitable for computing clusters, and a more compact representation of the graph to reduce memory requirements and simplify inter-processor communication. The new implementation allows us to analyze datasets of sizes that were beyond reach before. It can also be used to obtain finer results as it allows for more refinement iterations than the earlier implementations.

CHAPTER 4

PIECEWISE CONSTANT VECTOR FIELDS ON 3D REGULAR GRIDS

In this chapter, we introduce a 3D formulation of the 2D piecewise constant vector field algorithm from Szymczak and Zhang [7], with extensions to handle a cube mesh. Our goal is to bring the capabilities of this algorithm to the abundance of 3D vector fields, such as those found in the analysis of wind farm vortex meandering. We discuss the theoretical framework along with a preliminary implementation. Results from this published work are shown.

4.1 3D Piecewise Constant Vector Fields

In this section, we describe trajectories of a PC vector field on a regular 3D grid. Vertices are located at integral points in the x, y, and z dimensions. Quadrilateral faces, aligned with the coordinate planes, connect these vertices, and split the space into a number of cubes, forming mesh $\mathcal{M}$. A cell of $\mathcal{M}$ can be a cube, face, edge, or vertex.

We assume that the data is non-degenerate, i.e. it meets the following conditions (with c being a 3D cell and $F(c)$ being the vector assigned to that cell):

- $F(c)$ is not parallel to any of the coordinate planes; all components of $F(c)$ are non-zero.
- If $F(c_1)$ and $F(c_2)$ are parallel for two adjacent 3D cells c_1 and c_2, then $F(c_1)$ and $F(c_2)$ point in the same direction, i.e. one is a positive multiple of the other.

The standard simulation of simplicity technique [25] can be used to simulate a small perturbation of a potentially degenerate input vector field into one without degeneracy.

Since F assigns a vector to a cube in $\mathcal{M}$, trajectories move with constant velocity through interiors of 3D cells (cubes). However, we must also explore how this trajectory behaves when it leaves the confines of a cube, and encounters a lower dimensional cell. Here F is not uniquely defined, since lower dimensional cells are contained in more than one cube.

Thus, a trajectory followed by a standard ordinary differential equation (ODE) $\dot{x} = f(x)$ approach would be discontinuous in lower dimensional cells (see Figure 4.1). The solution to this problem is handled in the same manner as done by Szymczak and Zhang [7]. In this formulation, trajectories are solutions of the differential inclusion $\dot{x} \in F(x)$ [26], which behaves in a continuous manner. Here, $F(x)$ is a set of vectors, determined by extending F to lower dimensional cells in a convex-valued manner.

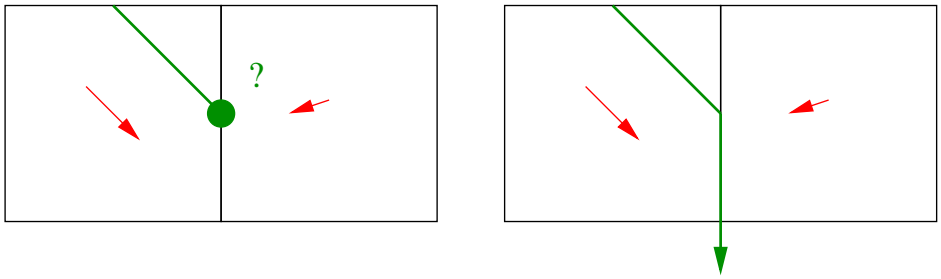


Figure 4.1: Discontinuity in a PC vector field. The left portion is a common problem with a ODE formulation to follow trajectories. It is not readily known what vector to assign to the point in lower dimensional cells. The right portion shows by intuition how this trajectory should behave.

To extend F to these cells to follow trajectories, we use the approach by Szymczak and Zhang [7] of convex hulls and define it in 3D. First let m be a cell in $\mathcal{M}$ that is of dimension less than three (faces, edges, vertices). Let $I(m) = \{c_0, \ldots, c_n\}$ be the set of cubes in $\mathcal{M}$ that are incident to m. Next, let $H(m) = Hull\left(\{F(c_0), \ldots, F(c_n)\}\right)$. This convex hull represents possible directions and magnitudes of flow for particles in m. Since any particle leaving the confines of the cell comes under the jurisdiction of a different cell, we want to assign only the vectors of the hull that provide a simple segment that is within m. Thus, we intersect $H(m)$ with the linear space of m. The linear space of m is defined by a linear space of equal dimension to m and parallel to m. This resultant set of vectors is denoted by $F(m)$.

With $F(m)$ defined, we introduce the notion of a *simple segment*, to aid in the discussion. A simple segment with carrier m is a segment of a trajectory over a non-zero time interval, contained in m, and has velocity in $F(m)$ for almost all times. With this definition we note that concatenated simple segments form trajectories, thus our discussion will focus around these simple segments.

The rest of this chapter is devoted to each of the several dimensions of carriers for simple segments, and how F is determined in these cases.

4.1.1 3D Cells

In Figure 4.2, we show a trajectory as it passes through a cube c in $\mathcal{M}$. The knots define the bounds of the simple segment. A simple segment with carrier c will enter the cube from a lower dimensional cell, transit for a nonzero time, and leave at another lower dimensional cell. In the figure, these cells are 2D faces, f_1, f_2 respectively.

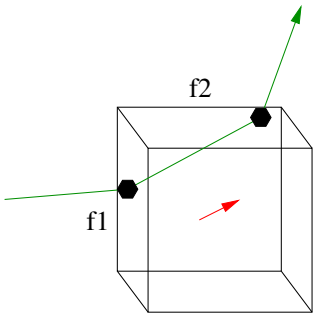


Figure 4.2: Trajectory of a PC vector field in a mesh cube.

4.1.2 2D Cells

Simple segments in a 2D cell (quadrilateral) f appear in Figure 4.3, and share similarities with the higher dimensional case. 2D cell f has two incident cubes, and so $I(f) = \{c_i, c_j\}$.

Figure 4.3: Simple segment examples for a 2D cell f. Here, the red vector indicates $F(f)$. On the left is a simple segment with edges for endpoints, while on the right we have the case of a simple segment with a vertex and an edge as an endpoint.

Hence $H(f) = Hull\left(\{F(c_i), F(c_j)\}\right)$ consists of all weighted averages of $F(c_i)$ and $F(c_j)$, i.e. is a line segment connecting the two vectors (Figure 4.4). By the general definition, $F(f)$

consists of all vectors in $H(f)$ parallel to f. From our non-degeneracy assumptions, $F(f)$ is either empty or consists of exactly one nonzero vector resulting from the intersection of $H(f)$ and the linear space of f. In the former case, there are no simple segments with carrier f; an example is shown in Figure 4.5. In the latter case, simple segments with carrier f move along f with velocity equal to the sole vector in $F(f)$ as shown in Figure 4.3.

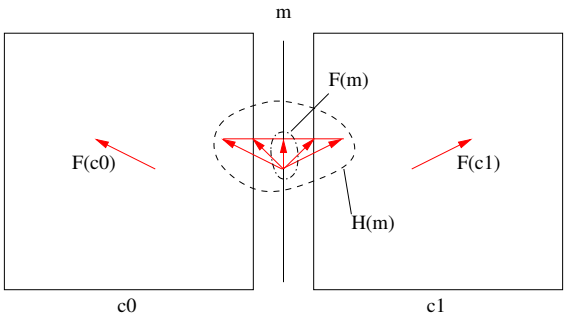


Figure 4.4: Interval of vectors initially considered for a face. View is perpendicular to the face.

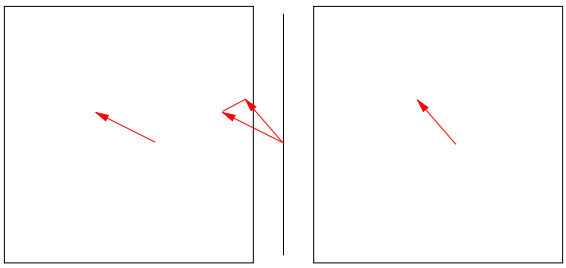


Figure 4.5: Case where $F(f) = \emptyset$.

4.1.3 1D Cells

For a mesh edge e, $I(e)$ typically (i.e. unless e is on the boundary of the domain) consists of four vectors. Thus, $H(e)$ can be visualized as a tetrahedron in the 3D vector space. $F(e)$ includes all vectors in that tetrahedron that are parallel to e (Figure 4.6). In general, $F(e)$ can be empty (see Figure 4.6(c), right), consist of a single vector, or be a convex set of vectors parallel to e (an 'interval' of vectors, as in Figure 4.6(a) and Figure 4.6(b)). If it is empty, there are no simple segments with carrier e. Otherwise, simple segments are paths that move along e, such that their average velocity over any time interval is in $F(e)$. Note

that even though this includes paths moving with non-constant velocity, ultimately only the directions that the vectors in $F(e)$ are pointing in will influence the construction of the transition graph.

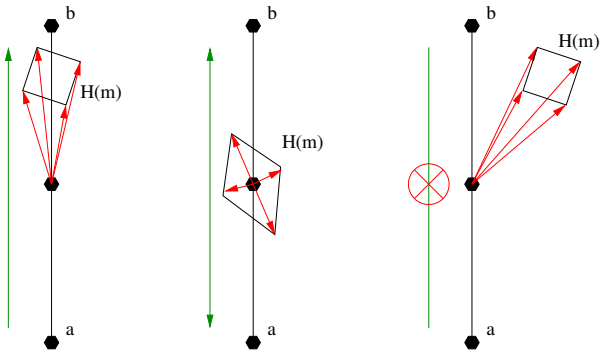


Figure 4.6: Examples of possible flow for an edge. Figure 4.6(a) shows flow in a single direction. The center figure shows flow that is bidirectional, as the interval saddles the center. The right image shows a pass-through case.

Thus, for a 1D cell e, if $F(e)$ is not empty, there will be a simple segment that has non-constant velocity. Although the prospect of simple segments with non-constant velocity might appear daunting, again it is important in our case only to know the direction(s) of the flow to define the simple segments. This is because trajectories may move with different velocities, but that will not change the start and end points of the simple segment.

4.1.4 0D Cells

Figure 4.7 describes the possible scenarios that we will encounter for vertex carriers. Note that simple segments will only be able to exist in the case of Figure 4.7(a). Such segments will remain at the vertex for some nonzero time; this we call a *stationary* vertex (Figure 4.7(b)).

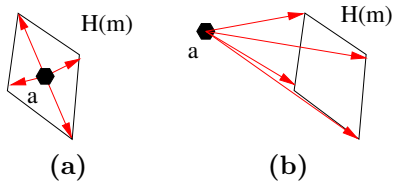


Figure 4.7: Vertex classification. The left subfigure indicates a probable stationary point, as the simple segment both starts and ends at this vertex. The right subfigure shows a pass-through case.

4.2 Transition Graph

With our understanding of simple segments for each mesh element, we now build a transition graph $\mathcal{G}$ around all simple segments in our input mesh. Nodes in $\mathcal{G}$ represent cells in $\mathcal{M}$ or pieces of those cells. Arcs represent the presence of flow connectivity between two nodes. Any two cells in $\mathcal{M}$ that are connected by a simple segment are connected by an arc in $\mathcal{G}$. Arcs in the graph have a *carrier* or mesh cell that contains them. Since any trajectory can be obtained by concatenating simple segments, any trajectory is represented by a path in the transition graph.

The following terminology is used. For a lower dimensional cell X of C with co-dimension of 1, the term X *attracts the flow in* C implies that $F(C) \cdot (X_{center} - C_{center}) \geq 0$, i.e F(C) points towards X. X_{center} and C_{center} refer to the center points of each cell. Otherwise X *repels the flow in* C. Lower dimensional cells of C (which are also lower dimensional cells of X) are labelled using a consensus of incident higher dimensional cells. If cells X_1 and X_2, both attracting the flow in C, are incident to lower dimensional cell Y, then Y shall also attract the flow. If cell X_1 attracts the flow in C, but X_2 repels the flow in C, then the cell is labeled as indeterminate.

4.2.1 Coarse Graph

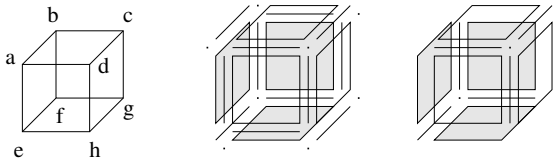
The first step in the construction of this graph is to initialize the data structure with arcs by using simple segments. This produces an initial transition graph that is at its coarsest level.

In a coarse graph, nodes are mesh cells of dimensions 0, 1, and 2. Arcs connect pairs of nodes connected by simple segments. Each mesh cell that is a carrier of a simple segment contributes a number of arcs to the graph, as described below.

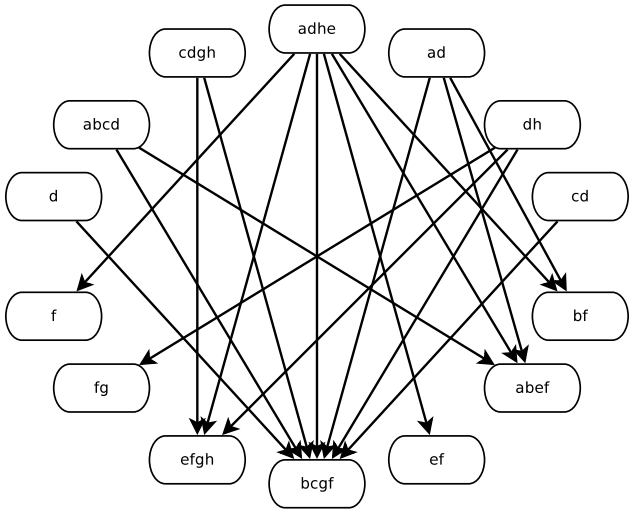
4.2.2 3D Cells

In Figure 4.8, we can see how a 3D cell c in $\mathcal{M}$ will contribute arcs to the transition graph. For simplicity, we assume that the PC flow vector is pointing into the paper. Shaded

cells are back facing. This projection along the vector provides a natural 2D representation of trajectories inside the cube, as simple segments in this view are points. This also lets us classify mesh cells based on their flow properties, namely if they are attracting the flow or repelling the flow in c. Back facing 2D cells attract the flow in c, while the front facing 2D cells repel the flow in c. Figure 4.9 illustrates a cube with its PC vector field as a red arrow. We view c from the top, so that lines are the sides of 2D cells. Red cells repel the flow in c, and blue attract the flow in c.



(a) Example cube, with labeled vertices. The flow vector for this cube is perpendicular to and pointing into the plane of the paper.



(b) Resulting transition graph for the cube.

Figure 4.8: Cube contribution to graph. In the upper subfigure, the leftmost portion shows a cube cell as defined by vertices a-h. The middle portion shows how the cube is composed of numerous constituent cells. Note that cells on the silhouette are safely ignored, as they will not issue simple segments, and thus contribute no arcs in this graph.

Other lower dimensional elements (edges and vertices) of the cube are labelled using this information (demonstrated in Figure 4.9). For a given cube c, an edge e is attracting if all incident 2D cells to e in c are attracting. If all incident 2D cells to e in c are repelling, e is

repelling. A third possible state for e arises from this method; it might be neither attracting nor repelling, and this state can occur if one incident 2D cell is repelling and another attracting. Such a situation implies that the edge is on the silhouette of the projected cube. The other lower dimensional element, vertices, also use incident cell checks for labeling.

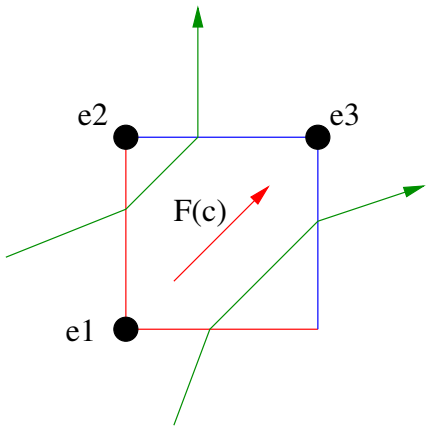


Figure 4.9: Classification of edges in a cube c; edges here are shown as a dot since we are viewing the 1D face from the top. The view is from the top of a cube c. Red faces are repelling, blue are attracting. Edge e_1 repels the flow, since the two faces that are adjacent to it also repel the flow. The opposite situation occurs for e_3, where the incident faces attract the flow. The third case, where a mesh element is neither repels or attracts the flow is shown in the case of e_2.

For the cube shown in Figure 4.8, trajectories entering from 2D cell s_{adhe} (the cell defined by vertices a,d,h, and e) can exit through 2D cells s_{abef}, s_{bcgf}, s_{efgh} as a simple segment can reach each of those cells from the first. Correspondingly, this is represented in 2D by the fact that these three cells overlap the cell s_{adhe} when projected to the paper's surface along the vector $F(c)$. This simple geometric principle of checking intersecting mesh cells in 2D permits us to form the arcs in the transition graph as shown in the figure.

Therefore, we connect all nodes that repel the flow to nodes that attract the flow with an arc in $\mathcal{G}$, if and only if they intersect in the 2D orthogonal projected space defined by the flow vector in c.

4.2.3 Face Contribution

Now we construct the 2D cell contribution to the graph. For a 2D cell, s, with $F(s) \neq \emptyset$, trajectories enter either by a lower dimensional cell (0D, 1D or 2D) of s, or by entering/leaving

the middle via one of the two adjacent 3D cells. Figure 4.10 shows an example 2D cell in the left subfigure. The figure shows how a simple segment, for example, leaving edge ac will always pass through edge ab. Simple segments starting at edge cd can leave the cell through two other edges, ab and bd, as well as the vertex b. As in the 3D cell strategy above, we determine which of the lower dimensional cells are attracting or repelling, according to $F(s)$.

Figure 4.11 shows how a trajectory might enter or leave through one of the adjacent 3D cells. To accommodate these trajectories we add the 2D cell itself in the contribution to the transition graph.

Thus we insert arcs into $\mathcal{G}$ to connect lower dimensional cells of 2D cell s that repel the flow to s itself, and then connect s to cells that attract the flow in s. The right subfigure of Figure 4.10 shows the resultant contribution to the transition graph.

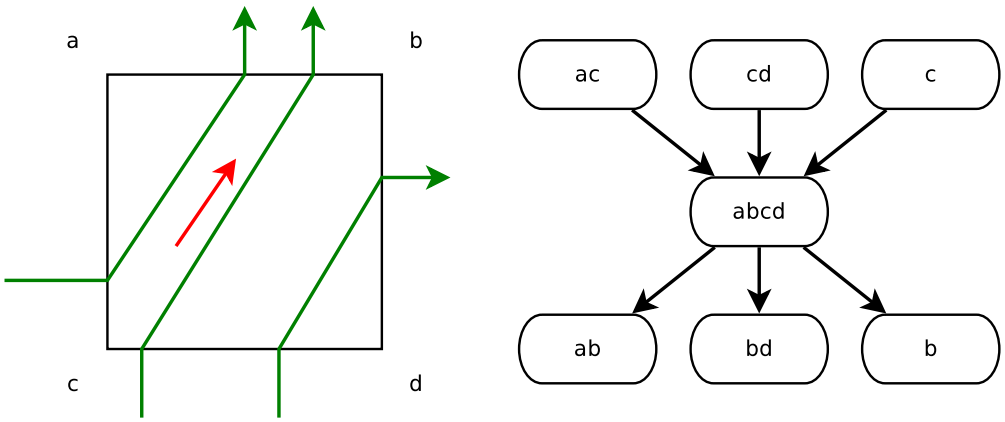
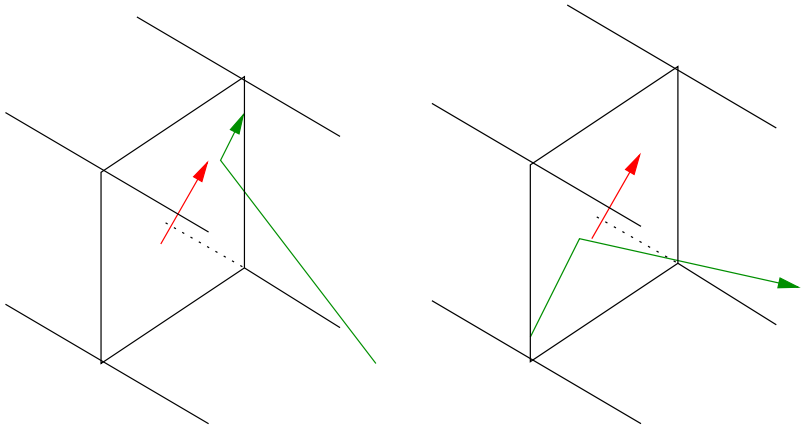


Figure 4.10: 2D cell contribution to graph. The left subfigure shows a 2d cell s_{abcd}, defined by vertices a to d, with $F(s)$ in red.

4.2.4 Edge Contribution

Since we have previously determined the flow for edge e_{ab}, we build the graph like so: If the flow is from a to b, we create arcs $\mathbf{a} \rightarrow \mathbf{e}$ and $\mathbf{e} \rightarrow \mathbf{b}$ in $\mathcal{G}$. Consequently, if the flow is from b to a, we create arcs $\mathbf{b} \rightarrow \mathbf{e}$ and $\mathbf{e} \rightarrow \mathbf{a}$ in $\mathcal{G}$. If the flow is in both directions, we must create both sets of arcs (Figure 4.12).



(a) **(b)**

Figure 4.11: Trajectories that enter/leave (respectively) a 2D cell by incident 3D cells.

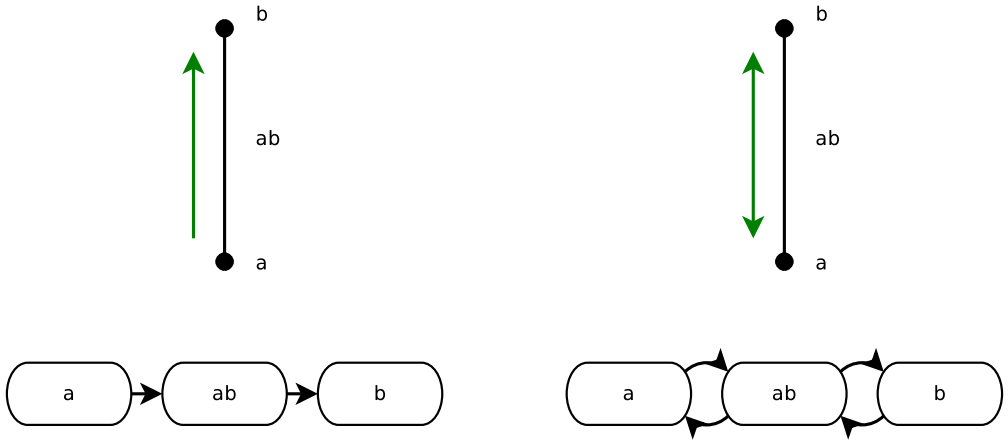


Figure 4.12: Edge contributions to graph, based on flow. Left shows a transition graph representing trajectories that move in a single direction in an edge. If it is possible that trajectories might travel in both directions along this edge, we account for it with multiple arcs, as seen in the right portion.

4.2.5 Vertex Contribution

The contribution of this element is simpler than the others, as shown in Figure 4.13. If the vertex v is stationary, we include a loop arc from v to itself in $\mathcal{G}$.

4.3 Refinement

As built in the previous section, $\mathcal{G}$ is at its coarsest level. With all possible trajectories recorded in $\mathcal{G}$, and the strongly connected components computed, we move to refine the graph. This is a local graph operation and can be done iteratively. The objective of refinement is to

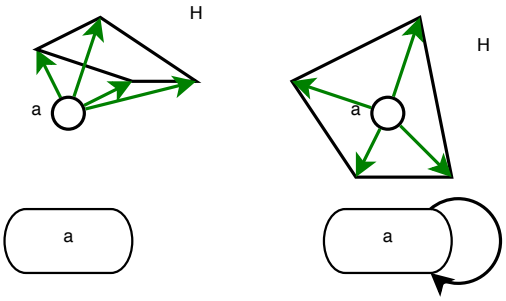


Figure 4.13: Vertex carrier contributions to the transition graph.

improve the upper bound on circulation expressed in the transition graph.

In general, the refinement process is done per node $\mathbf{n}$ in the transition graph, assuming $\mathbf{n}$ is in a strongly connected component. Since each node represents a mesh cell (or pieces thereof), $\mathbf{n}$ is split into nodes $\mathbf{n}_1, \ldots, \mathbf{n}_i$. Arcs of $\mathbf{n}$ are then updated to ensure that all nodes connected by a simple segment are connected in the graph. Once this property is satisfied, we remove $\mathbf{n}$ from the transition graph.

After all nodes in all strongly connected components have been subdivided, the SCC detection algorithm is run again to extract the refined components. It is by this iterative process of refining nodes and computing the strongly connected components that we contract the upper bound on features.

Naturally, the subdivision process for a node $\mathbf{n}$ depends on the dimensionality of the mesh element that $\mathbf{n}$ represents. Since 0D cells cannot be subdivided, we do not consider them for a refinement operation. The rest of this section deals with the two possible cell cases that we encounter.

4.3.1 2D Nodes

A 2D node in $\mathcal{G}$ is a quadrilateral (as this is a cube mesh). Such a cell can be easily separated into four smaller cells, as shown in Figure 4.14.

Figure 4.14 also demonstrates how transition graph will change in regards to outgoing arcs that have a 3D carrier. The cube that f belongs to has a flow that is perpendicular to the paper, and we are projecting this cube along this vector. The shaded cell k is back

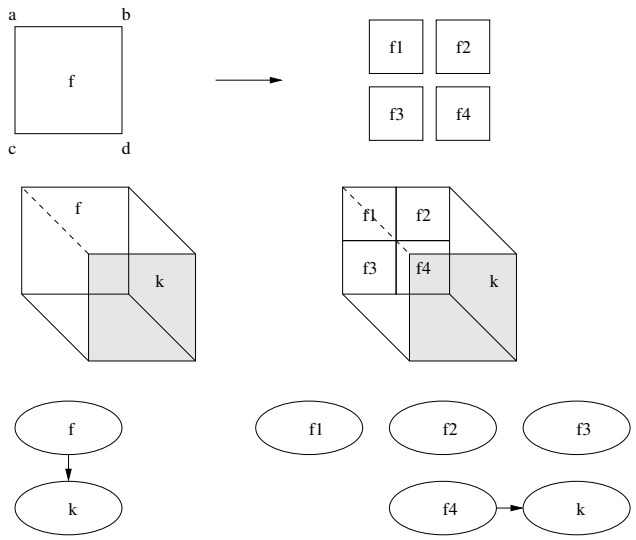


Figure 4.14: Face refinement process for an arc with 3D carrier. The left side of the figure shows the state of the mesh and the transition graph before subdivision. The right side shows how the graph and mesh changes after f is refined.

facing. With f subdivided, we look at the simple segments which begin at $f_{1,\ldots,4}$ and insert arcs into the transition graph to represent trajectory portions. In the figure, we can see how, after subdivision, cell f has only one of its portions connected to cell k in the graph. Though this example shows refinement in an outgoing-arc context, this procedure functions with incoming arcs as well.

We also consider the simple segments within the 2D cell itself that must be reconnected (detailed example shown in Figure 4.15); edge pieces must be reconnected to the proper $f_{1,\ldots,4}$, etc. This is done using the projection technique, in which the $f_{1,\ldots,4}$ cells are projected to the line perpendicular to $F(f)$. Edge pieces or vertices are then also projected to this line. If the two projections intersect, an arc is inserted to connect the cells.

4.3.2 1D Nodes

A 1D node is a mesh edge. These nodes can be easily subdivided by bisection into edge pieces. Again, if we construct a vector from the edge, we can use a pair of scalar values to represent pieces of this edge. Some care must be taken when reconnecting arcs that are within the edge cell node itself; this is shown in Figure 4.16. As 1D cells are incident to 2D

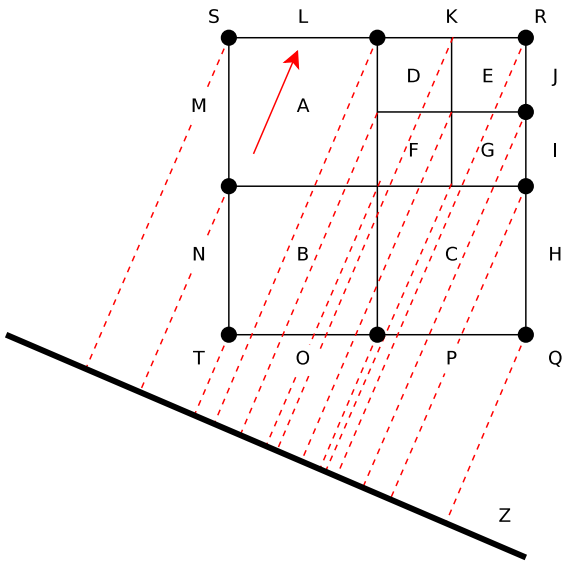


Figure 4.15: Detail of 2D cell under multiple rounds of refinement. Here, Z is the plane perpendicular to the vector field as shown by the red arrow. $A, \ldots, G$ are 2D cell pieces, $H, \ldots, Q$ are 1D cell pieces, and $R, \ldots, U$ are 0D cells. Note how the projections of cells form an interval on Z. Intersections of intervals here determine transition graph arc insertions. For example, F is the endpoint of arcs $O \to F$ and $P \to F$, and the start point for arc $F \to K$. P is connected by arcs $P \to \{C, E, F, G\}$.

and 3D cells we use the projection technique to determine connectivity for arcs that have a carrier of dimension greater than 1.

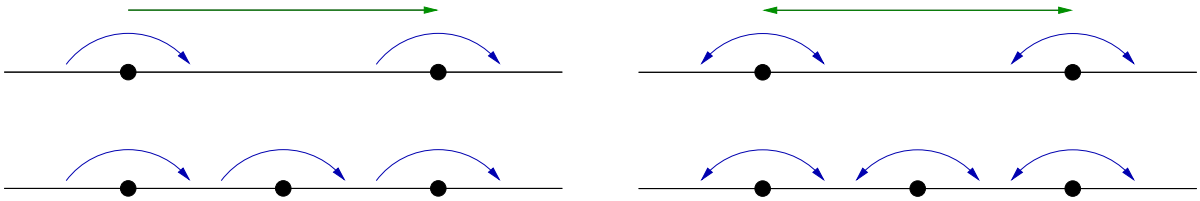


Figure 4.16: Edge refinement details. Trajectory direction defines how an edge piece should be connected to its neighbors in the transition graph.

It is not always the case that we are provided with a grid input mesh. As an alternative, we can function with a tetrahedral input mesh, where the 3D mesh cells are tetrahedra. In fact, we could even represent a grid mesh with a tetrahedral mesh.

To analyze a field with such a mesh, no major changes need to be made to the algorithm. As long as the highest dimensional mesh element is convex, and proper subdivision procedures are used.

4.4 Algorithm

The theoretical concepts presented in previous chapters are tied together in an iterative algorithm. This section focuses on the procedure, organized in the order of execution, and is shown in Figure 4.17. There are three distinct phases; input and transition graph initialization, refinement and component detection, and output.

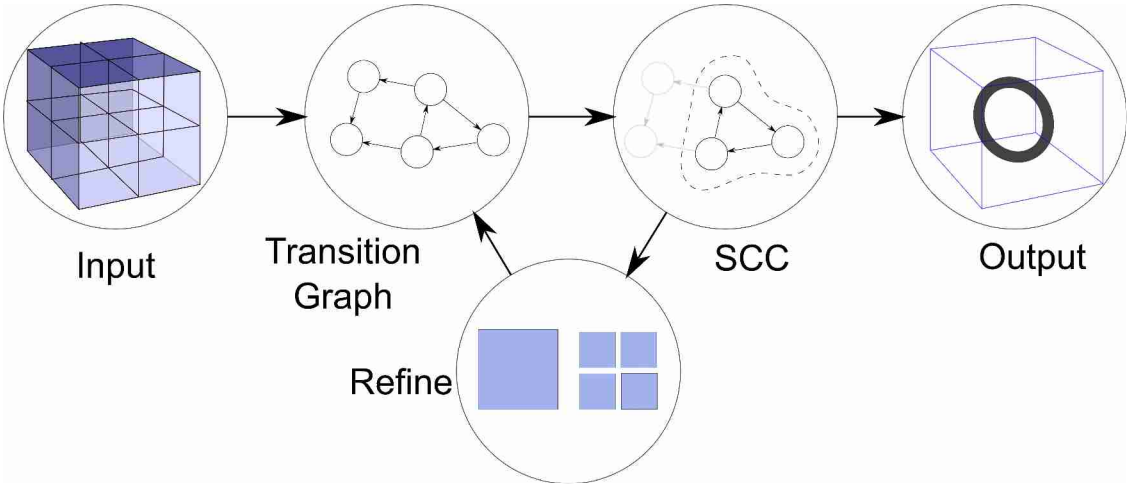


Figure 4.17: Algorithm flow.

4.4.1 Input

Our algorithm takes as an input a 3D vector field defined in a grid, which provides $F(c)$ for each 3D mesh cell c. Common third party vector field data sources define a vector for each vertex of a grid mesh instead of a 3D cell, as they are targeted for piecewise linear algorithms. A piecewise constant vector field can be derived from this source by assigning a vector from a consistent corner of a 3D cell to $F(c)$, or by a combination of all eight vectors to compute $F(c)$.

We translate this grid into a mesh graph for adjacency queries, etc. Nodes of this graph represent mesh cells; i.e. vertex, edge, face, or cube cells. Arcs in the graph connect higher dimensional mesh elements to their constituent lower dimensional elements (edge nodes have arcs to two vertex nodes, and so on). This representation of the mesh is largely agnostic to its input form; a tetrahedral mesh can be represented in this manner as well.

It is not always the case that we are provided with a grid input mesh; for example, some datasets may use a tetrahedral 3D cell. Processing these sets does not require a large change to the algorithm. As long as the highest dimensional mesh element is convex, and proper subdivision procedures are used, the algorithm should function under these different representations.

4.4.2 Coarse Transition Graph Construction

Using the mesh graph, we next build a coarse transition graph as discussed in Section 4.2. Graph nodes are stored in lists associated with each mesh cell. With the graph constructed, we now must find the features of the vector field.

4.4.3 Component Detection

Morse sets are represented by *strongly connected components* (SCCs) of $\mathcal{G}$ ([7]). In a strongly connected graph, for every pair of nodes (a, b) in that graph, there exists a path from a to b (i.e. each node in the graph can be reached from any other) [27]. The strongly connected components of a graph are its maximal strongly connected subgraphs. We provide an example of two SCCs in a graph in Figure 4.18.

In general, features of the vector field will appear as strongly connected components of $\mathcal{G}$; the upper bounds of periodic orbits (and those that are close to periodic) are represented by loops in $\mathcal{G}$. Likewise, for a stationary point, the mere addition of an arc from the vertex to itself will permit it to be found as a SCC of $\mathcal{G}$.

We compute the components using a non-recursive extension of the algorithm, introduced by Tarjan [28], and modified by Nuutila and Soisalon-Soinen [29].

4.4.4 Refinement

Components found by these first steps are coarse. We would like to reduce the upper bounds of features in the vector field to find a nominal representation of those features. Thus we begin the refinement process.

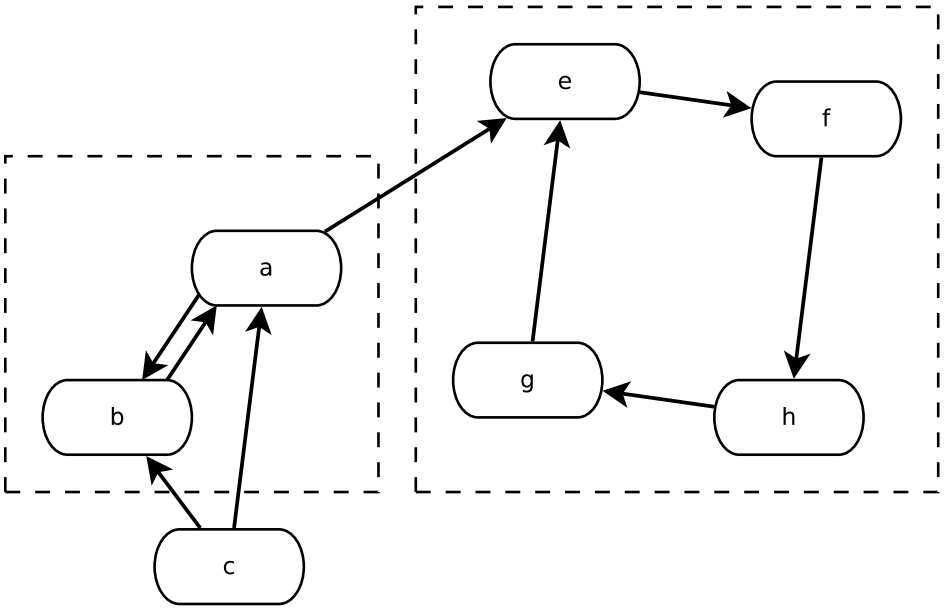


Figure 4.18: Strongly connected component example. The two portions enclosed in the dotted lines are each a strongly connected component of the graph.

This process subdivides each face or edge cell that is in a strongly connected component in the transition graph. As described in Section 4.2, the node to be refined (the *parent* node) is split into a number of *child* nodes. The subdivision process also reconnects arcs to the new nodes, while the parent node is unlinked. Unlinking the node effectively deletes it, but still retains its information for later use (see Section 4.5).

After all nodes have been subdivided, the strongly connected components of the graph are again detected. Any node not in a component is either deleted, or retained in a simple form to aid computing a Morse connection graph.

This constitutes one round of refinement, and is repeated a user-specified number of times.

4.4.5 Output

When the specified number of refinement operations has been completed, we will be in possession of a number of strongly connected components that represent the upper bounds of recurrence. To visualize these components, we construct a triangulation for rendering in OpenGL. For a component c, we iterate over all arcs $a_0, a_1, \ldots, a_n$ for all nodes in c. For an

arc a_i (assuming both endpoint nodes are in c), we compute a convex hull h_i of the union of endpoints, generate triangles to represent this hull, and write it to disk. To view c, we render $h_0, \ldots, h_n$, which presents a union of these hulls to the user.

4.5 Performance & Complexity

Memory and processing time consumed is directly related to the size of the transition graph; a graph of several million nodes is not uncommon for larger data sets. The number of nodes and their connectivity changes primarily during the refinement process, so we focus our investigation there.

Every time we refine the transition graph, nodes are subdivided and added to the graph. Only 2D and 1D nodes are subdivided into four new nodes and two new nodes respectively. In a worst case scenario, the graph will grow at a rate between 2 and 4 times per refinement iteration. This rate depends on the data set in question and how many nodes in the graph belong in a strongly connected component. For example, a component that is the size of the whole input volume is the worst case, and would cause all nodes in the graph to be subdivided. In summary, refinement operations will exponentially increase the size of the graph, leading to increased memory requirements and processing time.

To combat this growth, an optional step can be introduced at the user's discretion at the output phase of the algorithm, named *coarsening*. The refinement operation can be reversed for selected nodes; sibling nodes, under certain conditions, can be deleted and replaced with their parent node, which will assume their arcs and connectivity. This will reduce the number of nodes in the graph, while still maintaining the quality of the extracted features.

To coarsen a node, none of its siblings from a previous refinement may have been deleted. Further the siblings must all belong to the same component. If this is the case, the parent node of these child nodes is retrieved. All arcs (excluding duplicates) from the child nodes are reassigned to the parent node, and the child nodes are deleted.

4.6 Implementation & Results

To evaluate our approach we use a mix of analytical and simulated vector fields with known, or likely known features. Unless otherwise noted, these results were obtained on an Intel® i7-2630 based computer with 16 gigabytes of RAM.

4.6.1 Analytical Datasets

The following datasets are analytically generated; each follows a formula and has known features.

The first example presented is the Periodic Band dataset. In this test, the objective is to determine the ability of the algorithm to detect a simple orbit in an analytical vector field. The vector field is constructed as follows. Given a point $\vec{p} = [x, y, z]$ we can derive the vectors as in Table 4.1.

Table 4.1: Periodic band vectors.

$\vec{i} := [-x, -y, 0]$ $\quad \vec{u} := [-y, x, 0]$ $\quad \vec{v} := [0, 0, .25 - z]$
$\vec{w} := [0, 0, -x]$ $\quad d := |\vec{p}| - r$

Using r as radius, the vector field is defined as $f(\vec{p}) = \vec{u} + .05d\vec{i} + 2.0\vec{w} + 2.0\vec{v}$. The domain for this volume is $[-15, 15] \times [-15, 15] \times [-15, 15]$ with a resolution of $30 \times 30 \times 30$. Four rounds of refinement were used, taking 27 seconds and 750MB of RAM. The results of this test are shown in Figure 4.19. A clearly defined periodic orbit can be seen.

The Lorenz equations provide us with another, more interesting evaluation. These equations are given by:

$$\begin{aligned} \dot{x} &= \sigma(x - y) \\ \dot{y} &= Rx - y - xz \\ \dot{z} &= xy - bz \end{aligned}$$

As noted in [30], the Lorenz equations exhibit chaos for $R = 28$, $\sigma = 10$, and $b = \frac{8}{3}$. However, Mrozek and Pilarczyk [6] shows that for $R = 260$ or $R = 350$, periodic orbits become

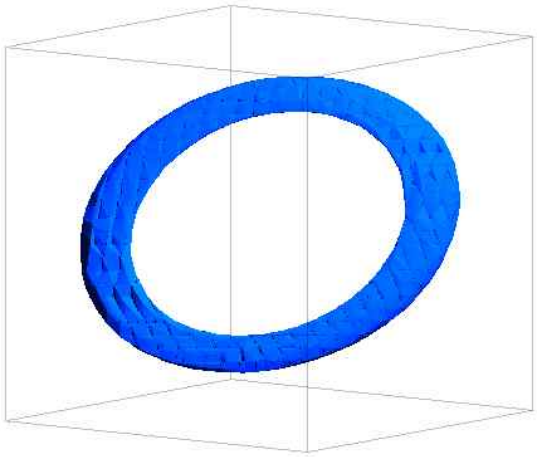


Figure 4.19: Periodic test and four rounds of refinement.

apparent. We test our algorithm using these three values for R.

For the first dataset in the Lorenz series, we set $R = 28$ and $\sigma = 10$, and $b = \frac{8}{3}$. The volume domain is $[-30, 30] \times [-30, 30] \times [-10, 50]$. The resolution of our mesh grid is $48 \times 48 \times 48$.

In Figure 4.20, a sequence of images is presented. From top left (Figure 4.20(a)) to bottom right (Figure 4.20(f)), each frame shows how the extracted Morse sets are displayed as the number of refinements increases from zero to five. From this figure, we can see how refinement improves the bounds of the feature. The feature is similar to the Lorenz attractor, with the exception of two holes that appear in later refinement steps. This is an artifact of PC approximation; the stationary points there are now separate recurrent components. Note that the output of the decomposition, even when coarsening operations are applied to reduce its size, approaches six million triangles, making further optimization of the display necessary for interactive viewing.

For five rounds of refinement, the algorithm took approximately one minute, and used 2 GB of memory.

We next generate a dataset using $R = 260$, $\sigma = 10$, and $b = \frac{8}{3}$. Following Mrozek and Pilarczyk [6], we shift the domain to $[-120, 120] \times [-120, 120] \times [120, 360]$ with a resolution of $24 \times 24 \times 24$.

In Figure 4.21 we present the results of eight rounds of refinement for this test. The periodic orbit appears in the output, and bears similarity to the results obtained by Mrozek and Pilarczyk [6]. Finding these features over eight refinement rounds took no more than 7.5 minutes, and consumed 10 GB of RAM.

In the last of the Lorenz series, the parameters of the dataset are as follows: $R = 350$, $\sigma = 10$, and $b = \frac{8}{3}$. We use the domain $[-150, 150] \times [-150, 150] \times [200, 500]$ with a resolution of $24 \times 24 \times 24$.

Figure 4.22 shows the results of seven rounds of refinement. The displayed components strongly resemble the results arrived at by Mrozek and Pilarczyk [6]. Processing statistics for this set were: 14 GB of RAM, 7 minutes of runtime.

4.6.2 Simulation Datasets

The following sets were constructed from simulations of physical phenomena.

Bénard-Rayleigh Convection The Bénard-Rayleigh convection simulation dataset was produced by Daniel Weiskopf [31]. Output of our algorithm is presented in Figure 4.23, and shows the cells created by the liquid as the warming fluid rises to the top of the domain and the cooling liquid is forced to the bottom.

Hurricane Isabel Simulation We used a subsampled version ($50 \times 50 \times 33$) of the Hurricane Isabel [23] dataset to extract features that are consistent with circulation patterns that are expected in this type of weather phenomena, as shown in Figure 4.24.

Square Cylinder In this next simulation, a square cylinder is placed into a fluid flow. It was prepared by Camarri et al. [32] and is available in full through the International CFD Database [33]. Our algorithm is able to consistently extract large separated

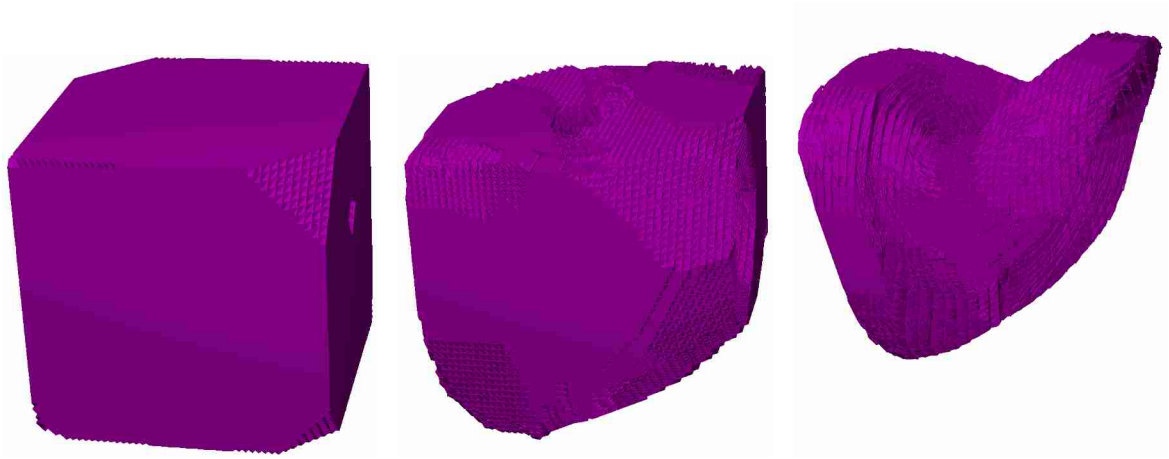
nearly recurrent components with vortices in a subsampled version of this dataset (Figure 4.25). Since the original dataset does not contain significant circulation, we have subtracted the average velocity from this dataset to aid in extraction.

4.7 Conclusion & Publication

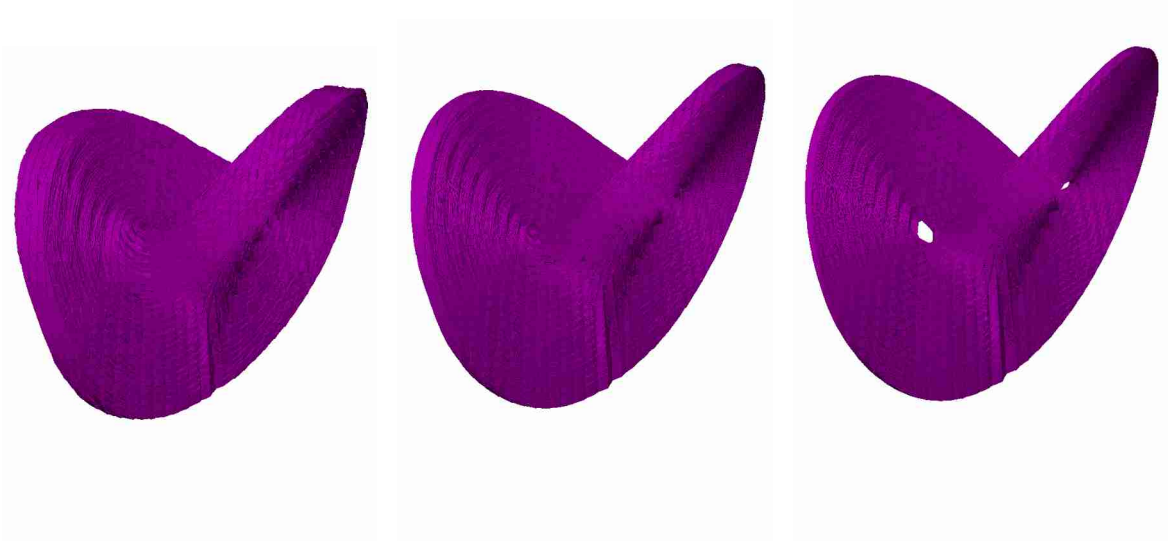
In this work, we have introduced a piecewise continuous method for extracting the upper bounds of recurrence from a three dimensional vector field, using computational geometry as the workhorse instead of more computationally intensive numerical methods. Results show that this method is able to detect a variety of features from analytic and simulated datasets.

As a drawback, our algorithm has high memory usage and high run-times, which may prohibit the study of larger datasets that have great complexity.

This work has been published [8], and was presented at EuroVis 2012.



(a) No refinement. **(b)** First refinement. **(c)** Second refinement.



(d) Third refinement. **(e)** Fourth refinement. **(f)** Fifth refinement.

Figure 4.20: Refinement progression. Figure 4.20(a) shows the results of zero refinement, increasing to five rounds of refinement in Figure 4.20(f).

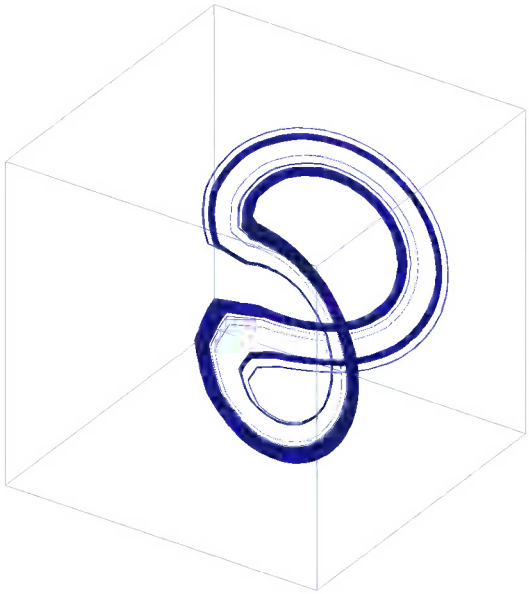


Figure 4.21: Lorenz with R=260 and eight rounds of refinement.

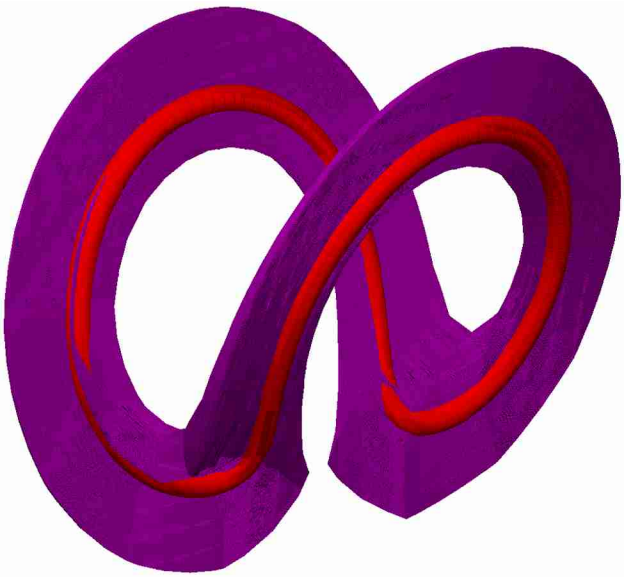
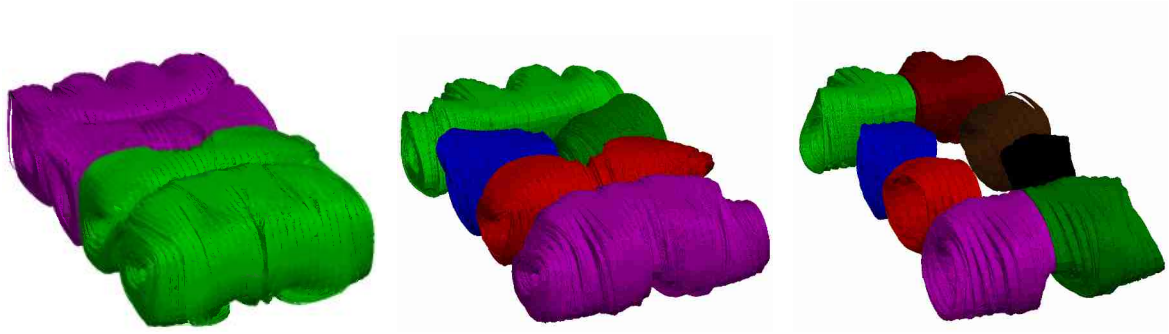


Figure 4.22: Lorenz with R=350 and seven rounds of refinement. The red tube structure is the neighborhood of the periodic trajectory found by [6] (using numerical methods). The purple nearly recurrent component found by our algorithm encloses it.



(a) **(b)** **(c)**

Figure 4.23: Bénard-Rayleigh convection dataset. On the left, two components are apparent. Since there is circulating flow everywhere in this volume, the components are large. By removing voxels from the boundaries of the input volume, we obtain more, smaller, components. The middle and right images have one and two layers of voxels removed, respectively.

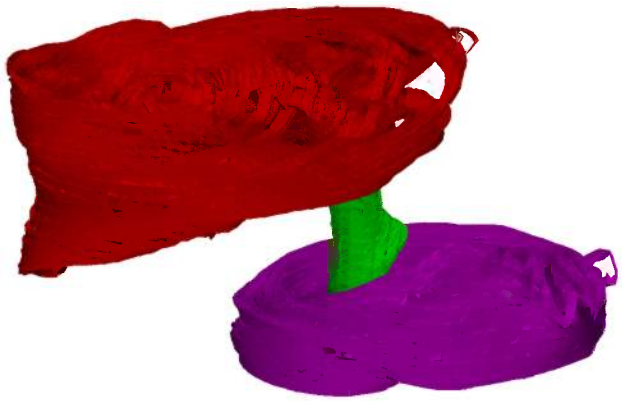
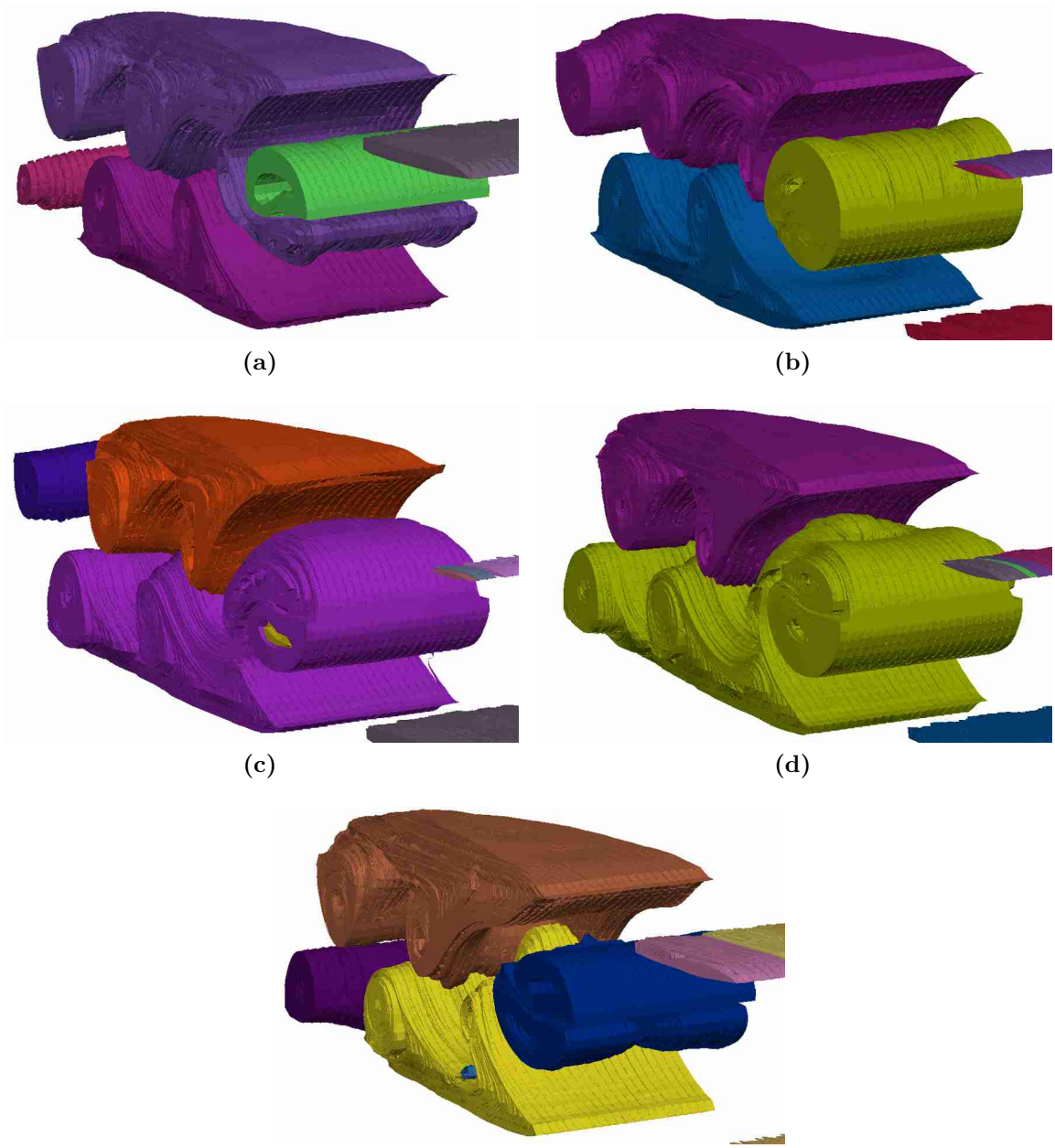


Figure 4.24: Three main components can be found in this dataset; these are equivalent to a clockwise spin of atmosphere at the higher altitude, a green central vortex, and counter-clockwise intake formation at the lowest altitude. Four rounds of refinement were used.



(a) **(b)** **(c)** **(d)** **(e)**

Figure 4.25: Selected time slices of the Cylinder Flow dataset. Slices are consecutive. Vortex shedding can be observed in subfigures a, c, and e.

CHAPTER 5

SHARED-MEMORY PARALLEL ALGORITHM

In this chapter, we put forth an alternate formulation of the algorithm presented in Chapter 4. The refinement process is replaced with a multi-pass procedure that is friendly to concurrent implementations, and improves memory locality. The ideas presented in this paper apply to both 2D and 3D PC vector fields, however, we focus on the 3D case.

The algorithm we have presented so far is relatively efficient. It was also able to be naively parallelized for a modest speedup on multicore CPUs. However, by reworking the approach into a universal, parallel and cache aware model, higher performance can be achieved, allowing the exploration of larger datasets that are common in today's research to be accomplished in less time.

5.0.1 Naive Parallelization

As an initial attempt to gain a performance enhancement, the refinement process of the previous chapter was naively parallelized. This approach added multi-threading to the refinement pass, with a large number of critical sections (forced serialization) to prevent consistency issues.

Here, t threads (usually equal to the available logical cores of the host CPU) were spawned, each with an allotment of nodes to refine. Since the refinement process requires the addition and removal of arcs in the graph, it is disadvantageous to have two different threads operate on adjacent nodes. Thus, a preprocessing stage was run in an attempt to create a list of independent nodes, where no node n_i shares an arc with node n_j in the list.

This list is then consumed by the threads, each taking a node, refining it, adding the subdivided nodes to the graph, updating the arcs, and then deleting/unlinking the original node.

The problems with this approach are many. In order to maintain consistency, it is necessary to specify arc operations (adding or deleting arcs) to be critical sections as well as when adding or removing nodes to the graph. As there are many arcs and many nodes that must be added, the algorithm will spend a significant portion of the processing time waiting for the mutex to be unlocked. There is also a large amount of time spent merely finding independent nodes. As we refine, and the number of independent unrefined nodes change, we must run the parallel refinement in a loop, first computing independent nodes, refining, etc to maximize the number of parallel processable nodes. Further, while the number of nodes that can be processed in parallel is usually a large fraction of the overall graph, there will be a number of nodes left over that must be done in serial. A more optimal formulation would be able to process all nodes with no such expensive preparatory step.

While the situation varies from dataset to dataset, it is not uncommon to experience only an approximate 150% speedup on eight core processors. To solve these problems, a reformulation of the algorithm is required.

5.1 Algorithm

In this section, we describe our parallel implementation of transition graph subdivision. The basic idea is to construct a new graph for each refinement round instead of executing refinements for one node at a time. In our implementation, we describe the 3D case, but the process readily applies to 2D as well.

5.1.1 Transition Graph Representation

Before we discuss the new algorithm, we first describe an alteration to the underlying graph data structures used. The graph consists of many lists of nodes; each list consists of nodes that were obtained by subdividing the same grid cell. More precisely, with nodes organized into multiple lists, addition of nodes to the graph no longer requires a single graph global mutex as in previous implementations; we use a separate mutex for each list. The objective of this representation is to maximize concurrent use by taking advantage of our

problem context.

While this is not fully optimal, adding a mutex for each node to maximize concurrency may cause memory problems as the graphs can grow to several million nodes with ease.

5.1.2 Transition Graph Construction

Here, we detail changes for the construction of the coarse transition graph used at the onset of our algorithm.

First, we construct a new empty transition graph G, whose nodes correspond to grid cells of dimension no more than 2 as described in Chapter 4. The compute device will then receive a list of projection matrices to be used in the refinement operation. For i rounds of refinement (specified by the user) we cycle through the main loop of our algorithm.

To complete a single iteration of refinement, we first create an empty transition graph H, containing no nodes and no arcs. For each node A in the SCCs of G, we subdivide A and insert all its children into H. Again, we use the data structure outlined in Section 5.1.1, using a node list per grid cell. In particular, nodes belonging to different lists can be subdivided in parallel.

To populate H, we process the nodes in the strongly connected components of G. For each node A in an SCC of G, we subdivide A into child nodes $A'_1, \ldots, A'_n$. In 3D, this means edge n-sets being split in two and face n-sets being split into four pieces. These child nodes are inserted in H. The parent node A still possesses its connectivity information to other nodes in G. This is presented in Figure 5.1.

Because of the change to graph implementation, node insertions can be done in parallel without the need for any critical sections by having threads operate on a cell level. A single thread subdivides all nodes that are owned by cell C, before moving to the next cell. In this manner, no thread will conflict with another over a cell's node list.

With H populated with nodes, we now determine which are to be connected by an arc. To do this, we examine the parent nodes. For each parent node A, we loop over all outgoing arcs $A \to B$ (with owner cell C). Our goal is to insert an arc from child node A' to B' in

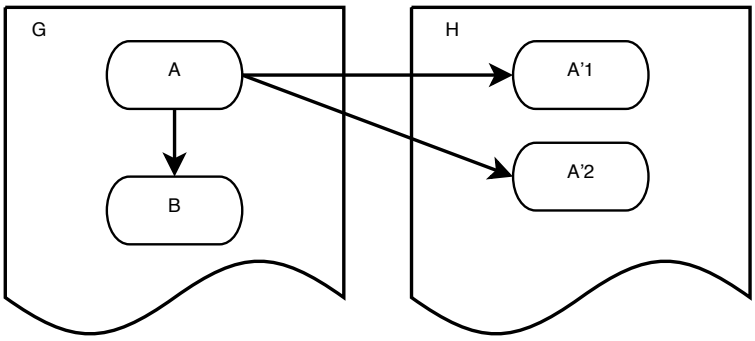


Figure 5.1: Multiple graphs during subdivision. Node A is subdivided into $A'1$ and $A'2$, with pointers to these new child nodes. The parent node A is not removed; G remains a consistent transition graph while H is under construction.

H if the connectivity check returns true. To do this, we add all A, B, as well as C, to the parallel processing list L. The goal of L is to provide a concise representation of all arcs along with arc owner information in a parallel-ready structure.

Each record in L is then processed in parallel, for example, by a kernel function. Since L is the only information sent to the device, before connectivity can be computed, the children of A and B are reconstructed on the compute device (this is not required if the compute device shares memory address space with the graph data structure). While this duplicates the subdivision work of the previous step of the algorithm, the alternative is to fill the list with children nodes and their projection contexts. In the case of two 2D nodes (faces), this means sixteen records (along with redundant projection information) must be added to the list and possibly transferred to the compute device. By only sending nodes and duplicating the child computation, we limit the effect of the memory transfer bottleneck between host and device, which is the commonly the slowest portion of the compute process. Further, the children of a node are relatively inexpensive to compute. The kernel/thread will look up the associated projection information associated with C, project the children of A and B, and compute intersection tests to determine which children of A should be connected to which children of B. It then encodes this information in a bitmap O (each tuple owns a portion of O). An example of processing a record is in Figure 5.2.

The device takes each child of A and computes the connectivity to each child of B, using the projection information of C. If the connectivity test between the two n-sets is true, we encode a 1 into the bitmap. In Figure 5.2, we can see a single arc that must be constructed.

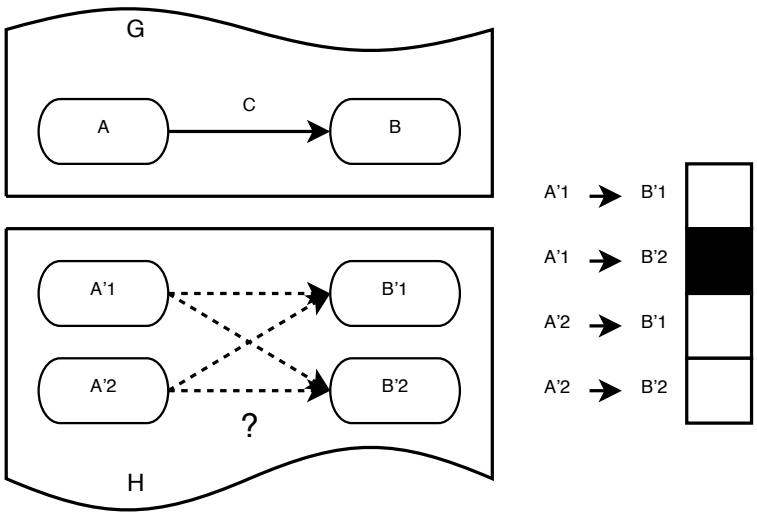


Figure 5.2: Example of children arc testing and encoding. Nodes A, B and arc/owner cell C (which provides projection information) are the input elements, and the bitmap is returned as the connectivity output. In this example, there are two children each for nodes A and B. The children nodes here are temporarily constructed on the compute device, but their connectivity will be provided to the child nodes in H afterwards. The output bitmap shows one arc is to be constructed between the child nodes A'_1 and B'_2.

The bitmap O is returned from the compute device. We now insert arcs into H. For each pair of nodes $A \to B$ in the upload list, we will connect the children $A'_i \to B'_j$ if their bit is true in the bitmap. This operation can also be done roughly in parallel. Adding an arc requires the exclusive access of each nodes' arc lists. When processing a single bitmap slot in serial, this is a non-issue. For parallelization of the entirety of the bitmap, the thread will have to lock the mutex of A's owning grid cell, as well as the mutex of B's owning grid cell.

Next, O is returned from the device, where the bitmap is decoded. Whenever the bitmap indicates that the projections of two refined nodes intersect, arcs connecting them are inserted into H. G is now destroyed.

We compute the strongly connected components of H. As in the original algorithm [8], we use Tarjan's algorithm[28] to compute SCCs. Although this step could potentially be

parallelized [34, 35], this is not the focus of this work.

Finally the contents of G are replaced with H (at most this requires swapping a pointer), to prepare for the next round of refinement. This process can be repeated as many times as the user desires.

5.2 Implementation & Results

In this section, we present two implementations of this algorithm, and their results on selected data sets.

5.2.1 Implementations

Our first implementation has been applied in part to run on a multi-core computer for 3D input data using multi-threading in a shared memory environment. The implementation has dual graph support, and fully parallelized connectivity testing. A buffer system has been added in preparation for OpenCL powered testing. Arc addition has been partially parallelized through the use of threading and mutexes.

In our second, experimental, implementation, we have added support for a CPU OpenCL compute device to build the arc connectivity bitmap. It shares the same technology of the first implementation, with the addition of pooling of nodes in memory. Because of the design of OpenCL, this implementation will always use the maximum number of logical cores available on the platform to build the bitmap. As our graph implementation is contained in CPU addressable memory and is in a separate address space from OpenCL, we interpret and connect the arcs using threading and mutex exclusion.

5.2.2 Preliminary Results

To show improvements in the previous version of this algorithm, we test our approach with the same 3D datasets as from Chapter 4.

Lorenz This dataset is a $24 \times 24 \times 24$ grid vector field originally presented in [5], where the vector field is described by $\dot{x} = \sigma(x - y)$, $\dot{y} = \rho x - y - xz$, $\dot{z} = xy - bz$. Using the

common parameters $\sigma = 10$, and $b = \frac{8}{3}$, we use $\rho = 350$, which, by [6], will provide a periodic orbit within the domain $[-150, 150] \times [-150, 150] \times [200, 500]$.

Bénard-Rayleigh Convection This set is the simulation of convection by Daniel Weiskopf [31]. We have subsampled the dataset to $128 \times 32 \times 64$. Further, to reveal more detail in the structure of this dataset, we have striped one and two voxels from each side to create three datasets in total.

Square Cylinder Prepared by Camarri et al. [32], this simulation places a square cylinder in a fluid flow. By subtracting the average velocity from the flow, we extract interesting swirling structures. This set has subsampled dimensions of $96 \times 32 \times 24$.

Table 5.1 provides timing of processing these datasets (graphical representation in Figure 5.3). The number of refinement iterations is listed, as well as the original processing times offered in Chapter 4 (the 'Old' algorithm, with naive parallelization, and the time needed to write the output excluded). Our improved algorithm was run several times on a compute node with differing numbers of assigned cores ('P-n', where n is the number of available cores for that run). We also show timings for our experimental OpenCL powered algorithm as 'PCL-24-4', using 24 cores to compute the arc connectivity bitmap, and 4 cores to connect arcs.

The algorithm was tested on a 24-core 2.4GHz Intel® Xeon® E5645 node with 48 gigabytes of RAM. Overall, the CPU used here and in Chapter 4 are comparable.

Table 5.1: Timing results in seconds.

Dataset	Ref.	Old	P-1	P-2	P-4	P-8	P-24	PCL-24-4
Lorenz ($\rho = 350$)	7	437s	270s	217s	153s	156s	174s	96s
Bénard-Rayleigh	4	419s	241s	189s	132s	153s	185s	77s
Bénard-Rayleigh (-1)	4	287s	154s	120s	87s	110s	118s	50s
Bénard-Rayleigh (-2)	4	158s	91s	69s	50s	62s	72s	30s
Square Cylinder	4	446s	263s	201s	142s	172s	207s	82s

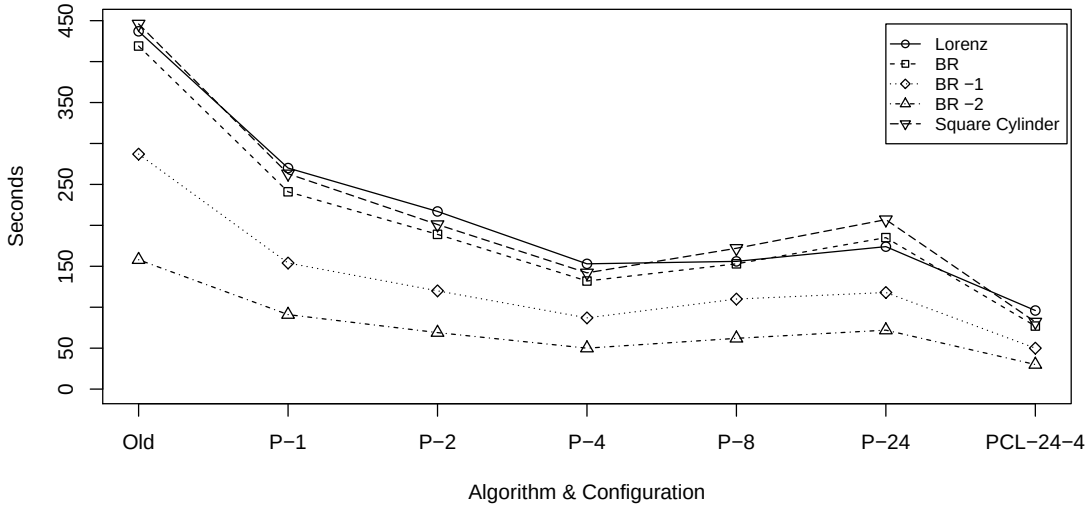


Figure 5.3: Graph of algorithm timings, comparing old times and new.

5.2.3 Discussion

Table 5.1 demonstrates improved performance for all input data sets, even in the case where only a single core was used, indicating better use of system resources when in serial mode. In the case of the Lorenz dataset, we reduced execution time from almost eight minutes to under three minutes for our first implementation, and less than two minutes for the OpenCL implementation. This is promising, as there are still a number of improvements to make to the implementations to bring them fully in line with the described algorithm. Memory usage of these algorithms is roughly comparable with the algorithm in Chapter 4; our new algorithm adds a statically sized buffer for computation, but also uses a more RAM efficient graph representation.

While there are significant gains, the results show that there is not a linear improvement as we increase the number of cores used. This is ultimately the effect of Amdahl's Law [36]; with only the bitmap generation being perfectly parallelized, the weight of a highly contentious arc insertion procedure and a serial SCC algorithm become nontrivial. The OpenCL version is able to achieve better times by being able to use the full 24 cores available

to compute connectivity, while only having 4 contentious threads to insert arcs, as opposed to the first implementation, which would use the same number of cores for both tasks.

Figure 5.4 shows an example CPU profile of an eight core processor node running the first parallel algorithm on the Bénard-Rayleigh dataset over four refinement iterations. The features we note are blocks of large spikes of parallelism of 800% usage, separated by small troughs that extend to $\approx 200\%$ usage. These blocks are separated by larger troughs at 100% usage (single core saturation). This last feature represents the strongly connected component algorithm. As this is not yet parallelized on our implementation, it runs on a single core. The time spent in this serial procedure (up to 20% in some cases) is large enough to justify a parallel SCC addition to the software. Of greater interest are the troughs at 200%. In our current implementation, we repeatedly fill a buffer to be processed and then compute the arc connections, repeating this every time the buffer is full. Here we see the buffer being processed (the large 800% spikes) and the arcs being connected (200% troughs). The arc connection portion is threaded, but uses mutexes to prevent concurrency related issues; low processor usage here denotes that the program is waiting for these mutexes to be unlocked by other threads.

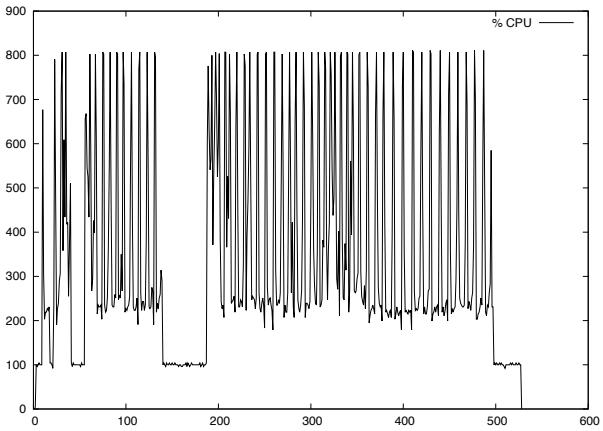


Figure 5.4: Example runtime of the Bénard-Rayleigh (4R) dataset on eight processors ('P-4'). The x-axis represents the sample number (samples taken around four times a second), while the y-axis denotes the percent of system CPU activity.

Thus, while the larger issue of computing intersections in parallel has been solved, these two smaller issues (SCC computation and parallel arc insertion) must be strongly considered for revision in future implementations of this algorithm. As mentioned, parallel SCC algorithms have already been developed and are readily available. The problem of parallel arc insertion is more involved; double buffering, queues, and other methods could be used as a remedy. A possible solution involves the use of strong partitioning, where each thread or process owns a portion of the graph outright, so that no mutexes are used at any stage of the algorithm. This approach is specifically being explored as we consider distributed systems (see Chapter 6).

5.3 Conclusion & Publication

In this document, we have presented a parallel reformulation of the refinement process for piecewise constant transition graphs. The new approach separates connectivity testing from the arc insertion and node insertion procedures, showing improvements in computation performance even when run in serial. When run in parallel, run times are greatly reduced, facilitating study of larger datasets.

This work has been presented at *Topoinvis '13*, and has been invited to be published as a book chapter [9].

CHAPTER 6

DISTRIBUTED TRANSITION GRAPH & MEMORY USAGE REDUCTION

In Chapters 4 and 5, we introduced an algorithm for computing recurrence in 3D piecewise constant vector fields. We then described a parallel implementation of this algorithm for a shared memory machine. In this chapter, we describe two improvements to the algorithm. First, we describe a new representation of transition graph that decreases memory requirements. Secondly, we describe a distributed version of the algorithm that allows us to apply our technique to data sets of considerably larger size than any prior implementation could handle.

The main issue with our framework is high memory usage. In practice, the transition graph size tends to be exponential in the number of refinement iterations. Figure 6.1 illustrates this phenomenon for the Bénard-Rayleigh dataset. Note that even before any refinement begins, the number of nodes in the graph is on the order of 10^5. The size of the graph would grow by a factor close to 4 if all nodes were uniformly subdivided (since the face piece number would dominate the graph size). Due to the fact that we do not retain nodes in the graph that are not part of a strongly connected component, the graph size grows slower, but still exponentially.

This has limited our ability to examine large scale datasets. Most sets examined to date are either subsampled versions or sub-volumes of more common data set dimensions, yielding initial coarse transition graph sizes on the order of 10^6 nodes. In most cases, refinement of the transition graphs was terminated primarily because of lack of memory. Larger datasets could easily broach several billion nodes in a coarse graph, and imply that no refinement of the transition graph could take place. In the following sections we put forth two contributions with an aim to bring these datasets into reach.

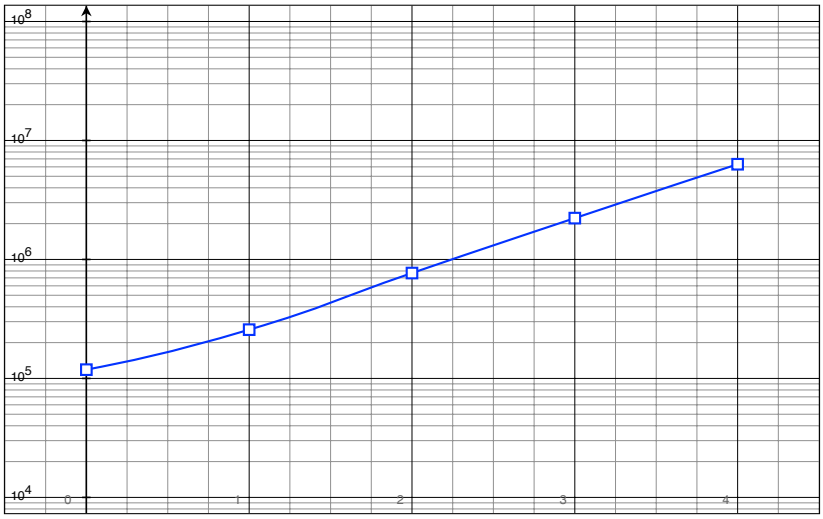


Figure 6.1: Graph size growth. The x axis denotes the number of refinement steps, while the y axis shows the number of nodes. Note that the y axis is *logarithmic*, and the graph size depends greatly on the underlying dataset.

6.1 Memory-Efficient Transition Graph Representation

Each transition graph node represents a portion of a mesh cell. Current implementations realize this by storing a set of points representing the subset of the mesh cell in which they are contained. For example, a node representing a 2D mesh cell portion will have four 3D points describing the subset (this is represented by four three-element single precision floating point vectors, or 48 bytes, plus an additional 8 or more bytes for a pointer to the array and heap metadata). A node representing a 1D mesh cell piece will have two 3D points.

Face nodes (those representing pieces of 2D mesh cells), having the most points, are also the most numerous in the transition graph. When nodes are subdivided, face nodes are subdivided into four more pieces, and edge pieces into two. Eventually, the transition graph size is be dominated by a quadrupling in the number of face nodes. This can easily cause memory requirements to stretch into the tens of gigabytes, overwhelming common desktop hardware, even for smaller datasets.

Arcs in the transition graph contain a reference the projection environment (a mesh cell) through which they pass. These references are commonly represented by a pointer (usually

8 bytes on 64 bit hardware). Graph arcs, while typically more numerous than graph nodes, consume less memory due to their small individual RAM consumption, but are still one of the contributors to high memory usage.

By reducing RAM storage requirements for both arcs and nodes, significant memory savings could be realized. To remedy the issue of memory use, we do away with explicit storage of points, reducing the transition graph to nodes containing single unique identifiers represented by an integer. We also reduce the size of cell references stored in arcs.

All information necessary for our algorithm to function can be derived from this identifier. Points representing the mesh cell piece (either whole or subdivided) can be computed when they are needed (the refinement and output phases). Node type (face, edge, vertex) is readily computable. Allocation and deallocation are also greatly simplified; in previous implementations either a dynamic array would need to be requisitioned, or a memory-wasting statically sized array would need to be filled. In the implementation proposed here, only a single integer must be assigned, avoiding heap allocation, etc. The IDs are assigned, using an coding scheme based on space subdivision techniques (e.g. quad-trees [37]).

This approach has the drawback that it imposes a limitation on the number of refinements that can be executed on the transition graph. In our current implementation with a signed 64-bit integer, this limitation is 14 rounds of refinement with a $3000 \times 3000 \times 1000$ input volume. If more refinement iterations are required, or for very large data sets, this approach is easily extended by using more bits to represent an identifier.

We wish to have a unique integer identifier (ID) for a node in the graph. We base our addressing on *spaces*, or integer ranges. Converting an ID from one space to the next is accomplished by translating the ID by the appropriate boundary.

We first split the 64-bit integer range (for discrimination, the *world space*) into four *type spaces*; C, V, E and F, one for each class of node or mesh cell type. Each space starts at 0 and extends to a chosen maximum. The initial world space division is shown in Figure 6.2; we wish to provide an integer from this space for each mesh cell and graph node. A given

ID in world space is translatable to a type by determining which type space the ID is within. For example, an ID that is between 8×10^9 and $4.500000008 \times 10^{18}$ would fall into the E (or edge) type space. Note that when the coarse graph is constructed, the world IDs of mesh cells are inserted into the transition graph to provide the initial graph nodes (see Section 6.3 for details).

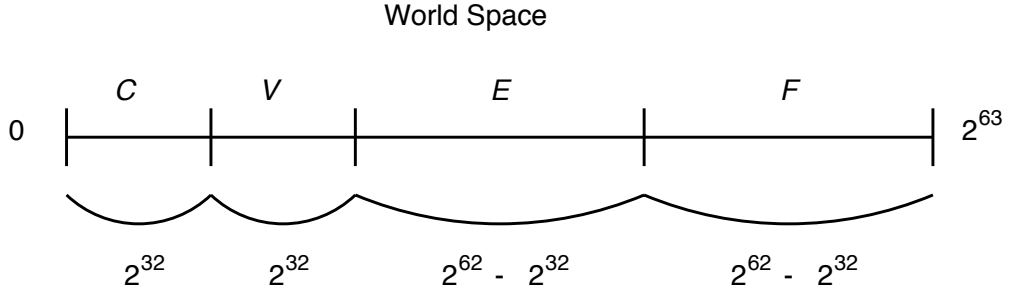


Figure 6.2: 64-bit integer space subdivision for World Space. Sizes are approximate. Note that this integer is signed, thus providing 2^{63} unique IDs; this was chosen to provide an effective null ID (negative numbers). Our algorithm is not restricted as to the type of underlying integral, signed or unsigned.

C is dedicated to providing identifiers for cube mesh cells and V is dedicated for vertex mesh cells and 0D graph nodes. The C and V type spaces are allotted 2^{32} values to distribute, which are assigned based on their spacial location in the 3D volume, treating the assigned space as a flattened 3D array. Equation 6.1 shows the mapping between spacial coordinates and their associated index; note that this mapping uses the x and y dimensions of the input vector field volume, and a coordinate triple. The coordinate corresponding to a 3D mesh cells is the corner of the cell with minimum $x-$, $y-$ and $z-$ coordinates; the output of this function is used for the type space ID, that is then translated into world space.

$$\text{map}(dim_y, dim_z, x, y, z) \rightarrow x \cdot dim_z \cdot dim_y + y \cdot dim_z + z \tag{6.1}$$

As edge and face nodes are more diverse and can be subdivided, a more complex addressing must be used for spaces E and F. Each is allotted $2^{62} - 2^{32}$ identifiers. These spaces are further subdivided into *dimension spaces*. For face nodes and 2D mesh cells, these refer to the axis on which the normal lies. For edges and 1D mesh cells, this refers to the axis

in which the edge node sits. The coordinate of the cell, either a corner for 2D cells or an endpoint for 1D cells, is used for an ID in this space, that is then translated into world space.

We then subdivide the dimension spaces further into *tree spaces*; each of these tree spaces contains a flattened breadth-first tree, which matches the structure of subdivisions. In this manner, the parent of a graph node (i.e. the coarse parent) and children (i.e. the subdivided children identifiers) can be easily computed by following the tree.

An example is presented in Figure 6.3. In general, an edge node identifier N is decoded by using simple formulas in each of the nested address spaces that contain N; first the world space, then the edge space, then the dimension space, and finally the tree space. The parent, children, and neighborhood of N in the refinement tree can also be determined using simple formulas.

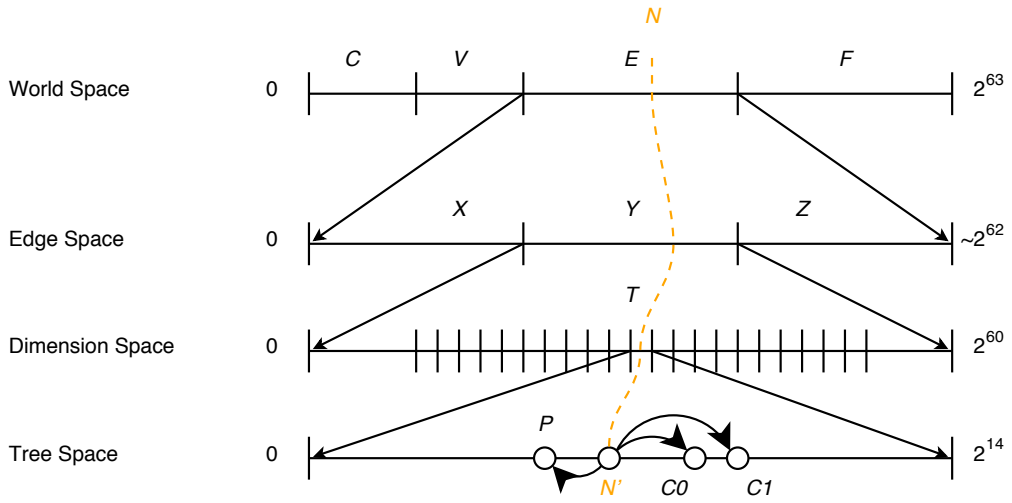


Figure 6.3: Integer identification space division for edge nodes. This is similar for face nodes. Limits are approximate.

To determine the parent ID in world space for a face or edge node, we first transform the node's world ID into a tree ID. If the tree space ID is 0, this node is the root. If it is non-zero, the parent is computed by Parent ID $= \left\lfloor \frac{\text{Tree ID}-1}{d} \right\rfloor$ where d is either 4 for face nodes or 2 for edge nodes. The parent ID is then transformed back into a world ID. Children are computed in a similar fashion; the world ID must be transformed into a tree ID. Once in tree space, they are found by computing Children IDs $= 2 \cdot \text{Tree ID} + \{1, 2\}$ for edges or

Children IDs $= 4 \cdot$ Tree ID $+ \{1, \ldots, 4\}$ for face nodes.

Determining the boundary points for an un-subdivided node or mesh cell from a World ID is straightforward. The node's ID type (cube, face, edge, or vertex) is quickly established by which region of the world space the ID is within. In each, extracting coordinates can be accomplished by inverting the mapping of Equation 6.1. This provides a 3D point; the rest of the points needed are then built for the needed type. Axis alignments for edges and faces is again determined by their appropriate space.

For a node that has been subdivided (that is, has a non-zero Tree ID), the boundary points are determined as follows. First, the points for the root of the tree are constructed. The root is found by recursively finding the parent of the Tree ID of the node until the root (Tree ID of 0) is reached in tree space. We record child numbers as we proceed, creating a path to the root node. An example of this is presented in Figure 6.4. Here, we are attempting to compute the points that represent node 45 in tree space (shown as the red portion on the right of the figure). The path to the root is $45, 11, 2, 0$; where 45 is the first child of 11, 11 is the third child of 2, and so on. The path demonstrates the precise subdivision order required to reconstruct the points of the subdivided node from the root. Once the root has been reached, the points needed are obtained by the inverted mapping of Equation 6.1 as previously described, then subdivided from the path determined.

By inserting nodes into the graph by following the world space division (vertices first, then 1D nodes, etc), our node list becomes sorted; operations on sorted ranges provide a number of optimization opportunities, especially in associative arrays. This benefit is compelling over other solutions, such as bit addressing, which may reduce the complexity of the node computation cost. In practice, however, the impact our chosen scheme has on runtime is negligible.

Regardless of scheme, reconstruction of points for nodes, subdivided or not, represents a small performance loss over previous implementations. This is a cost we accept for memory reduction. However, the mapping of points for a graph node (regardless of its subdivided state)

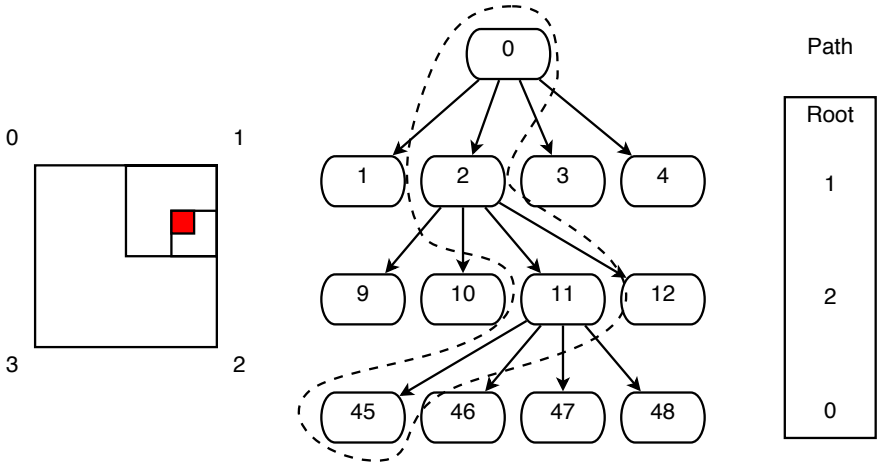


Figure 6.4: Point computation, from the point of view of Tree Space. The red square represents node 45 in tree space, following the path of 0, 2, 11, 45. The path corresponds to the four quadrants of a subdivided face; 2 implies that the lower right quadrant of the face should be kept, 0 means the upper left.

can be cached. This is essentially the condition of the graph for our previous implementations; the whole graph was cached. Here we can choose just how many points are to be stored in memory at a given time. For our purposes, storing 100 points in a cache provides an acceptable tradeoff in performance versus memory.

Arcs are stored as in previous implementations; in a per-node outgoing list that records the node handle of the endpoint. Instead of a per-arc reference to projection information (8 bytes on our 64-bit platform), we instead use integer maps keyed by a 32-bit unsigned integer to a projection environment, taking advantage of the common situation where there are less than 2^{32} projection environment records. This is shown in Figure 6.5. In this case, we store all projection information into arrays based on type (i.e. we have three lists for cube projection information, face projection information, and edge flow direction), which is accessed by a simple lookup from the integer.

6.2 Distributed Graph

To exploit the memory resources available on supercomputers, we update our algorithm for use in an MPI-based [38] environment. MPI defines a communication infrastructure to facilitate multiprocess execution and communication, commonly taking place across many

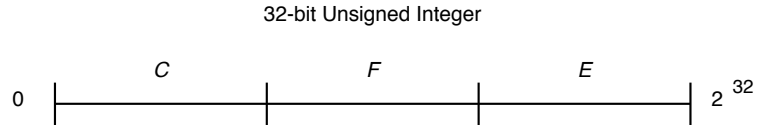


Figure 6.5: Arc reference replacement map. Each arc will carry an integer that falls into one of the three ranges, indicating which projection environment this arc is under. This is a similar approach to node ID assignment; in this case, however, the boundaries are fluid and based on the number of projection environments for each mesh cell type, since, after the construction of the parallel mesh, their number remains constant.

discrete supercomputer nodes. As many instances of the software will be executing in parallel, we will refer to a single instance as a process in this text.

The first steps toward this environment hinge around promoting our transition graph for distributed execution. We thus will need a parallel graph implementation to serve as a foundation.

6.2.1 Prior Work in Distributed Graph Implementations

The Parallel Boost Graph Library (PBGL) [39] is a widely used implementation of a distributed graph using MPI. It is intended as a research platform and as a common base to explore graph algorithms, with an API designed to closely match the shared memory graph implementation already in the Boost Library. It features a well-tested SCC implementation as well as parallel graph construction with node balancing.

This was initially considered as an alternative for a distributed graph implementation to form the foundation of our distributed transition graph. However, our experience showed that this graph library would not be suitable for our needs.

The first point of contention is that the BPGL is highly generic, as it is designed to solve a wide scale of problems. Capabilities such as arc weighting, as well as graph node deletion are not needed in our algorithm, but their inclusion in the library results in a higher memory usage than our own implementation; in some cases twice as much memory was needed to represent equivalent graphs.

The second issue is the PBGL's strongly connected components algorithm. It uses the same DCSC approach from [35] (detailed in Section 6.2.3). However, it exhibited much lower performance than our message based implementation, sometimes taking an order of magnitude more time to complete the same SCC pass on equivalent graphs.

Finally, our problem case provides structure and optimization opportunities (detailed in 6.2.4) which would be difficult to integrate into the Boost codebase.

Because of these issues, we have constructed our own distributed graph implementation, with a message based framework to power the SCC algorithm.

We note that an experimental message based version of the PBGL, named PBGL2 [40, 41], is being developed which, with the extensive developer resources of the Boost project, may turn out to be more competitive in future in terms of performance and robustness.

6.2.2 Custom Implementation Details

In our system, each node in the graph has a global identifier, described in detail in the previous section. Each process is given a number of the total nodes of the graph to store locally, distributing the nodes across the entire parallel system. Some nodes are considered to be *shared*; these nodes have connectivity to parts of the distributed graph that are non-local to a process. Shared nodes are marked as such, and are duplicated for every process that shares that node. Each process provides each local node with a handle (a 32-bit integer), which is an index into the local node list. A mapping for global identifiers to local handles is present for each process.

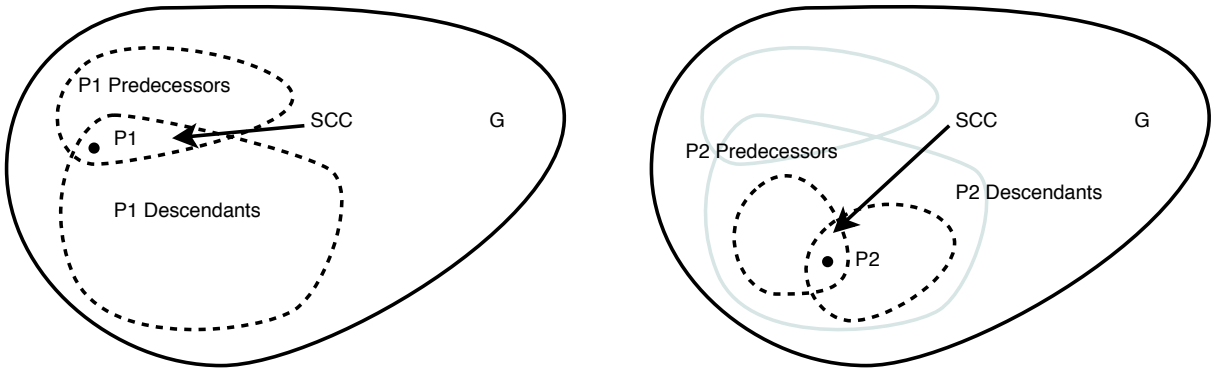
Arcs are stored in a per-node list for outgoing arcs. There also exists an incoming-arc list that is the reverse of this graph, which is generated on demand (for example, in the SCC procedure).

Note that processes can only add arcs and nodes to their own portion of the graph; no process can insert a node into another processes graph piece. Only the most important operations for our algorithm, such as the SCC algorithm, have been implemented. As this is procedure is complex, we devote the rest of this section to our SCC approach.

6.2.3 Parallel Strongly Connected Components Algorithm Review

In our previous implementations, we used the serial strongly connected components (SCC) algorithm [28], which posed parallelization limitations. Further, Tarjan's approach is not well suited for a supercomputer environment.

A concurrent-capable version of the SCC algorithm, called the Divide and Conquer SCC approach [35] (DCSC) shows promise in this area. In this algorithm, the graph is recursively partitioned; a node in the graph or *pivot* is chosen at random from the graph G, as shown in Figure 6.6(a) as P1. The predecessor and descendant subgraphs are computed from this point, providing a partitioning of G three sets; the *predecessor*, *descendant*, and *remainder*. The intersection of the *predecessor* and *descendant* subgraphs is a strongly connected component. The algorithm is then recursively run on these subgraphs, partitioning each one further, as seen in Figure 6.6(b). These further partitions, and all the ones that follow, can be undertaken concurrently.



(a) For graph (or subgraph) G, a single node $P1$ is selected at random as a pivot, and the predecessor and descendants sub-graphs are detected. The intersection of these sets is a SCC.

(b) The algorithm proceeds recursively with each subgraph and the remainder of G.

Figure 6.6: Graph partitioning example.

A potential drawback of this approach is that it is not work efficient: its expected running time is $O(m \log n)$ (where n is the number of nodes in the graph and m is the number of arcs), as opposed to the $O(n + m)$ runtime of the serial Tarjan's approach. The hope is that enough parallel pivots can be made in the graph such that the longer runtime is spread

across many concurrently executing processes, resulting in less execution time overall.

Mclendon et al. [42] improved DCSC with the addition of a trim operation, which removes (in an iterative fashion) nodes which cannot possibly be in a non-trivial SCC, or an SCC with more than one node. The goal is to reduce the number of potentially poor pivot choices that would limit the effective parallelism. Trim operation removes any graph node that has no incoming arcs or has no outgoing arcs (this can be implemented by giving the node a flag to avoid any graph modification). Since this will expose new nodes that can be trimmed, this process is repeated again and again until no nodes are removed. The efficacy of the trimming step depends greatly on the underlying graph.

The combination of these two methods, i.e. a trim phase followed by the DCSC pass, was shown by Hong et al. [43] to be a potent approach to parallel strong component identification, and is thus one we intend to follow.

6.2.4 Parallel SCC Implementation

Our SCC procedure begins with a trim (node marking) phase. This is executed directly after the last arc has been inserted in the graph (no arc may be deleted after insertion; a whole new graph should be allocated instead). We mark them similarly to Hong et al. [43]; the nodes are not deleted, but merely marked. This retains the graphs immutability after construction as justified in Chapter 5. Marking shared nodes requires additional effort. Each processor constructs a list of shared nodes that it believes should be marked; that is, if the node is adjacent to only marked nodes, or has no incoming or outgoing arcs. These lists are then communicated in batch to the other processes that share a node. These processes then reply for a given node, only if they, too, would mark the node. If all agree to mark that node, then the node is stricken. With this complete, we can proceed to the rest of the SCC phase.

Our problem context presents opportunities for greater parallelism beyond simply selecting an SCC algorithm. For the coarse transition graph case, we have no prior knowledge of graph structure; the SCC algorithm has to be run on the whole graph (except for nodes removed in the trimming step). Each subsequent refinement phase may split the graph into a number of

disjoint SCCs. If we have multiple non-empty SCCs determined in the previous steps, their refined versions can be obtained by independent analysis of the coarser components.

As an example, Figure 6.7 shows a graph that has previously detected SCCs of a graph (distributed across eight processors), leaving sub-graphs A-F. These sub-graphs have had their nodes subdivided, and each is ready for the next SCC pass as dictated by the algorithm. These different subgraphs can each be processed independently. Subgraphs B and D-F are entirely local to a processor; for these, we run a local SCC algorithm, namely the serial Tarjan [28] approach which is more efficient. Subgraphs A and C, however, span multiple processors, and will require the DCSC algorithm. To handle these cases, we first determine an owner (or controller) for each subgraph. The controller is responsible for managing DCSC algorithm for each subgraph. The master process (processor 0) is in charge of issuing these subgraph assignments to controllers.

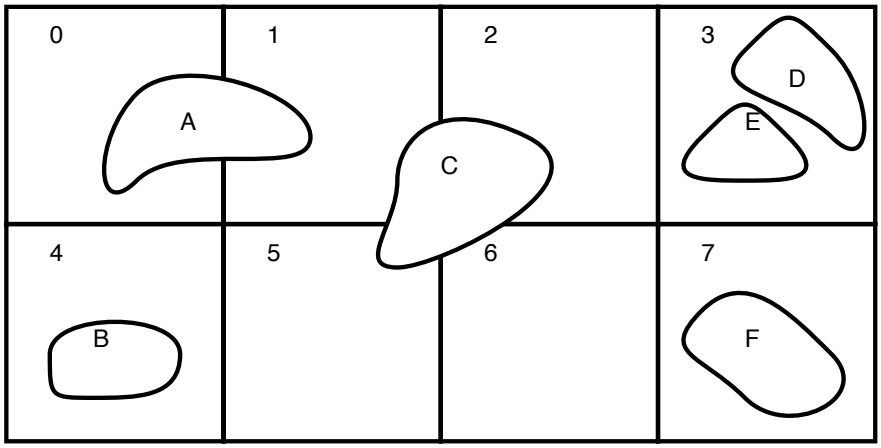


Figure 6.7: Component assignment example. Here, eight processors (0-7) are dividing a 2D graph. A-F are strongly connected components previously detected within the graph that have their nodes subdivided. Each of these subgraphs can be processed (that is, have new SCCs computed) in parallel. Shared components (A and C) will have a controller assigned to manage them. All other components can be processed locally.

Once controller assignments have been received, all processes enter a message loop, as seen in Figure 6.8. Subgraph controllers select an involved process (for the example in Figure 6.7, subgraph C may be assigned to the processor 2; the involved processes would be $1, 2, 5$ and 6), and send a partition request. Each partition request is handled by picking a pivot node

in the graph, and traversing the graph to determine its *predecessor* and *descendent* sets. Our traversals are based on a coloring scheme. Colors (integer labels) are used to indicate which nodes are in the pivot sets; each color has a *forward*, *backward*, or *other* flag to match the partitioning scheme, and new labels are provisioned for new pivots. Each of the involved processes for this partition thus execute a "pre-color" phase, to first determine the extent of the nodes that can be traversed (recall that the traversal is limited to nodes within a certain partition). These pre-colored nodes are given the *other*-flagged color of the new pivot, overwriting any previous coloring. Then the single selected process will begin traversing with the new pivot's color, dyeing nodes until no further nodes can be reached. If any of the dyed nodes are shared, a forward or backward color message is sent to the appropriate processes, which continue the traversal and send a reply upon completion. Once all the color requests are replied to, the selected process can report this to the controller, as the partition is complete.

The controller then determines which processes have nodes belonging to the *predecessor*, *descendant*, and *other* sets for that partition. Once these replies are collected, new partitions in the three sets (if there are sufficient nodes in the set) are ordered, and added to the work queues for the selected processes.

When no more partitions can be issued and all existing partitions are complete, the controller reports this to the master process, which will terminate the SCC loop as soon as the rest of the controllers are done.

6.3 Distributed Transition Graph & Algorithm Details

In this section, we detail our changes to the algorithm to support distributed systems, using the distributed graph foundation developed in the previous section.

In this scheme, we divide the input volume into a rectilinear grid of sub-volumes in a pre-processing phase, that are then consumed by our modified algorithm in parallel, followed by a post-processing phase to rebuild the algorithm's output into a coherent volume.

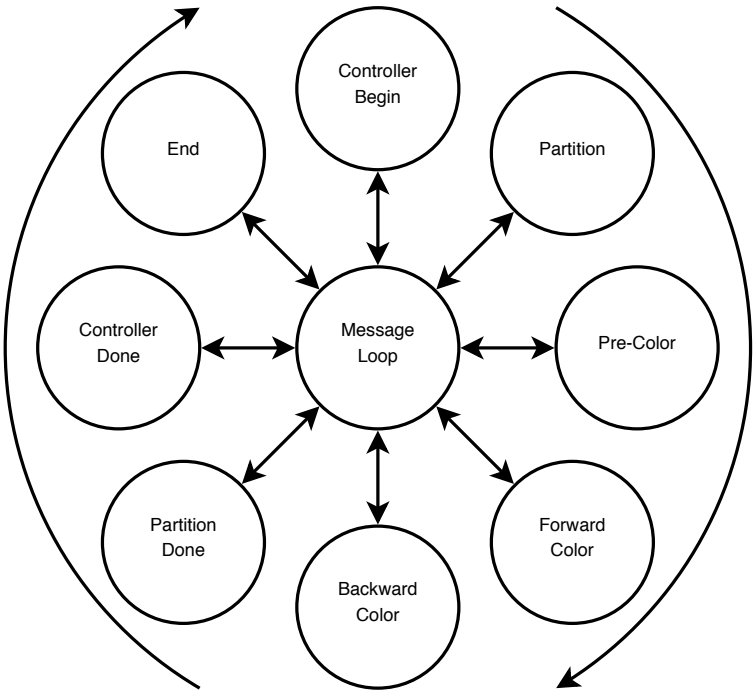


Figure 6.8: Message loop. Stages are executed in clockwise fashion from top.

6.3.1 Case Construction

Input to the algorithm is a case folder that contains data directories, one for each processor. This form closely matches the case system used by other parallel applications; for example, the OpenFOAM computational fluid dynamics toolkit [44] uses this method, as it can better accommodate directory locking mechanisms of distributed file systems.

A decomposition tool builds this directory structure by consuming a given vector field volume and dividing it into a 3D grid according to a user-specified dimensions. An example of this division is shown in Figure 6.9. This split takes place over the vector field voxels, and so voxels that are adjacent to another processor are specified as being 'shared'; this is critical during later graph traversal stages as when following graph paths, the algorithm must know which processor continues the graph in that direction.

This grid decomposition provides a spacial relationship between processors and their graph contents. The number of neighbor processes that a process must communicate with is bounded (the bound is $3^3 - 1 = 26$ in the 3D case), so the communication bottlenecks are

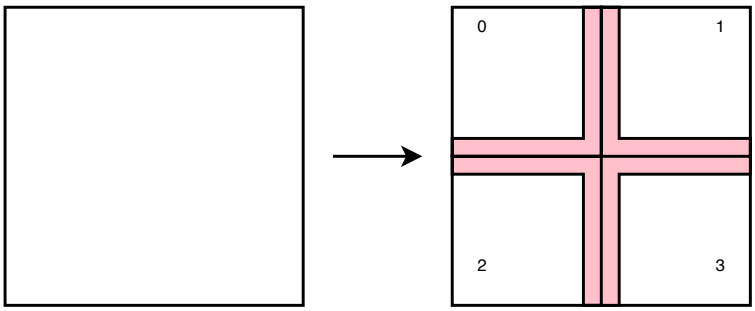


Figure 6.9: Volume division and assignment. The left is the input problem vector field volume, which is then split into four components for a four processor case. Adjacent cells (in red) are marked as shared.

reduced. This choice is also intended to group processes with adjacent portions of the graph to the same supercomputer nodes to further reduce network communications.

6.3.2 Parallel Mesh & Coarse Graph Construction

The algorithm follows the outline of previous versions to build the mesh and distributed coarse transition graph.

Each processor constructs a mesh, complete with 0D to 3D cells. Some care must be taken to propagate the status of shared 3D cells; lower dimensional cells should also be marked as shared. Projection information for 1D to 3D cells is also computed and stored. Each cell is given its computed world space ID as discussed in Section 6.1.

The coarse transition graph is then built in a two-phase approach. First, all new nodes are inserted. In this case, a new node is inserted for every 0D, 1D or 2D cell, with the node data being the cell's world space ID. Next, arcs are inserted into the coarse transition graph using the same projection based principles from previous implementations; nodes are connected based on intersections in projected spaces. When all arcs have been inserted, we execute a trimming operation to remove nodes from SCC processing and proceed to execute the parallel strongly connected components algorithm, as described in 6.2.4.

6.3.3 Refinement

Refinement continues the multi-graph approach from previous implementations; subdivided nodes of the current distributed transition graph are inserted into a new, empty distributed transition graph, with the goal of improving locality and eliminating costly node deletions that arise from mutating a single graph over the algorithm's lifetime.

In Chapter 5, we devoted much effort to speeding up the subdivision process, as the node representation required actual cartesian points. These had to be divided and inserted into the new graph. Here, subdivision merely consists of computing child node IDs from the parent node ID and inserting them into the graph. Care is taken to mark children nodes shared if their parent was shared as well.

Arc connections take place as in previous algorithms. We examine arcs of the previous transition graph; using projection information and points generated on demand, we determine which child nodes should be connected in the new transition graph. There is a minor performance loss as the points for a node might need to be recomputed multiple times. Caching can mitigate this loss.

After refinement, the trim operation is executed again before the SCC algorithm detects new components.

6.3.4 Output

In previous implementations, visualization of the strongly connected components of the transition graph was accomplished through a triangular representation of the surface of each SCC. For each SCC, each arc between nodes was emitted as a convex hull of the graph nodes' points.

This approach suffers from the same scalability issues as the transition graph. As our problem size increases and the number of refinement steps increases, the number of triangles emitted becomes overwhelming. Even simpler problems can result in a multi-million triangle representations. Before, attempts were made to control this output by a coarsening phase or

the use of convex hulls.

To avoid this issue, we now render the results of the analysis to a per-process voxel volume, which is then converted to a Vapor Data Collection (VDC) [45] file format in a post-processing step.

This approach has numerous benefits. First, the output file size is now well controlled by the underlying dataset. The results are rendered into a file that is based on the size of the input volume and number of refinement steps. If an input voxel volume is of dimension $\{x, y, z\}$, we choose to output voxel volumes for a refinement level r as $\{x, y, z\} \cdot 2^r$ to capture sufficient detail.

As the VDC file format is wavelet based, even when the output voxel volume grows in size, we can take advantage of its ability to progressively access data. This permits us to view even the largest datasets on desktop class hardware, with a minor corresponding loss in image fidelity.

6.4 Results

To evaluate our new implementation, we have run it for a variety of datasets of different input sizes and provenance. All times reported include only the time for nearly recurrent component computation. Time needed to convert the result into a volumetric representation is excluded. This is consistent with the previous results of this thesis. Unless otherwise stated, all images of the results were generated from the VAPOR viewing application [46].

We have explored two kinds of datasets to examine our two improvements of the algorithm. The first contribution is to reduce memory consumption. This is useful for the desktop use-case, where memory is restricted and limits the number of refinements that can be done on smaller datasets. To demonstrate our ability decrease RAM usage, we have selected the same datasets that were used in previous chapters. These sets were processed on the same server node used in previous chapters (a 24-core 2.4 GHz Intel® Xeon® E5645 node with 48 GB of RAM) with our improved algorithm. While the goal of this chapter is not to reduce runtimes, they are reported for comparison. As the DCSC process has a random component,

execution durations were obtained for this section by running the dataset multiple times and selecting the average of those runtimes. An overview of the results for this section is shown in Table 6.1.

Table 6.1: Desktop class dataset results summary with comparison. Key: n is the number of nodes in the final transition graph, m is the corresponding number of arcs, and t is the time taken, excluding output processing. We have selected the best runtimes obtained from previous algorithms to use as a comparison here. The table shows that we are able to consistently reduce memory usage, regardless of the dataset.

Dataset	# Ref	n	m	RAM (Old)	RAM	t (Old)	t
Periodic Band	4	320 K	1.3 M	785 MB	250 MB	27s	27s
Bénard-Rayleigh	4	18 M	81 M	12 GB	5 GB	30s	75s
Square Cylinder	4	19 M	79 M	12 GB	4.6 GB	82s	375s

The second contribution is aimed at the study of large 'real world' datasets. To demonstrate our ability to process these types of datasets, we have examined a number of vector fields with large numbers of voxels or that result in oversized graphs that require the resources of distributed clusters to analyze. The improved algorithm was executed on the Peregrine supercomputer at the National Renewable Energy Laboratory (NREL), a petaFLOP machine consisting of a spread of E5-2470, E5-2670, and E5-2695 v2 Intel® Xeon® processors (from 2.3 to 2.6 GHz). These results from the supercomputer are summarized in Table 6.2.

Table 6.2: Supercomputer dataset results summary. Datasets consuming 40 GB or more are typically out of the range of current desktop hardware; datasets are now able to be processed on computer clusters.

Dataset	# Ref	n	m	RAM	t
Lillgrund Turbine	2	83 M	365 M	40 GB	48m
Hurricane Isabel	3	313 M	1375 M	107 GB	617m
Ocean Circulation	3	539 M	1949 M	244 GB	143m

The rest of this section details input and output characteristics for each dataset, along with any distinctive runtime behaviors and notes.

6.4.1 Periodic Band

We start with a simple periodic trajectory using the same analytical definition as in Table 4.1 at an input voxel resolution of $32 \times 32 \times 32$ (somewhat larger than the original), as this was our test case throughout development, and provides a direct comparison to the results in Chapter 4. Using two cores and four rounds of refinement, we achieved the same results (as shown in Figure 6.10) for this dataset consuming approximately 250 MB of RAM versus 785 MB in our initial results. The running time (27 seconds) was the same amount of time required for the timing obtained in Chapter 4. With four rounds of refinement, the output volume is 512^3 voxels in size.

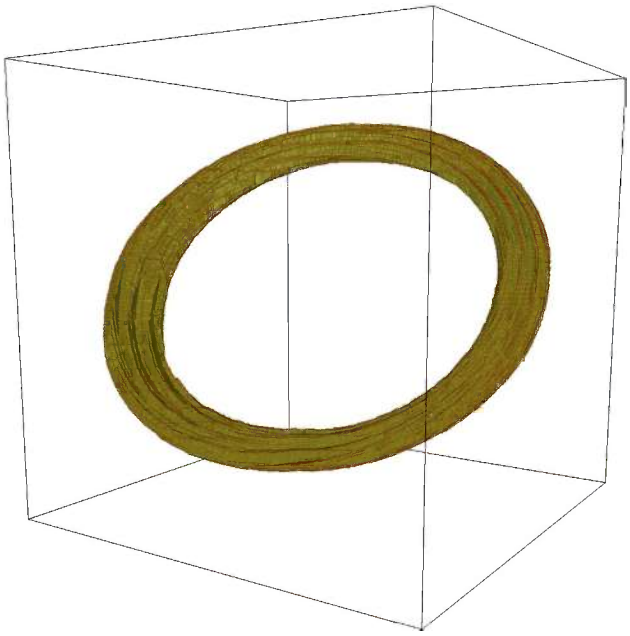


Figure 6.10: Periodic band dataset example.

6.4.2 Bénard-Rayleigh Convection

For the Bénard-Rayleigh simulated convection datasets (again, from Weiskopf et al. [31]), we present example output for the cases that were explored in previous chapters in Figure 6.11. The dataset was kept to the same dimensions as before, using a $4 \times 1 \times 1$ processor grid for each. Similar convection cells are apparent, and removing top and bottom layers of voxels

shows convection cell structures similar to those presented in Section 4.6.2.

Resources consumed are listed in Table 6.3. Three different versions of the dataset were used. "BR" refers to the subsampled dataset, with no voxels removed. "BR -1" and "BR -2" refer to this same dataset with one and two layers of voxels removed, respectively. For reference, previous memory consumed was on the order of 12 GB for the "BR" dataset; our algorithm now consumes only 5 GB. We note that the execution time for the "BR-2" example took around 57 seconds compared to a runtime of 30 seconds achieved in Chapter 5. This is suitable for a distributed system.

We also show resources consumed for the "BR" set taken to another round of refinement (five rounds instead of four), as shown in Figure 6.12. This would have been impractical to achieve using previous versions of this algorithm, as the transition graph grew to contain 65 million nodes and 280 million arcs. Of note is the shape of the features as they appear to transition between Figure 6.11(a) and Figure 6.11(b). This run used 14 GB, or very nearly the same amount of RAM for the previous algorithm to complete four rounds of refinement. The output VDC volume is $2048 \times 512 \times 1024$ voxels, or 4 GB in size. The VDC system can scale the output down to 8 MB of data needed for coarse visualization.

Table 6.3: Time and memory usage for the Bénard-Rayleigh convection dataset group. All sets were run using four processors.

Dataset	# Refinements	Memory	Time
BR	4	5 GB	75s
BR	5	14 GB	388s
BR -1	4	3 GB	62s
BR -2	4	2 GB	57s

6.4.3 Square Cylinder

As before, this dataset is the simulation of turbulence behind a square cylinder that is placed within a liquid flow. Provided by [32] and subsampled to dimensions of $96 \times 32 \times 24$ as in Chapter 5. We also subtracted the average velocity from the velocity value at every

(a) Subsampled Bénard-Rayleigh set.

(b) The same set with one voxel layer removed.

(c) The same set with two voxel layers removed.

Figure 6.11: Renders of the Bénard-Rayleigh convection sets.

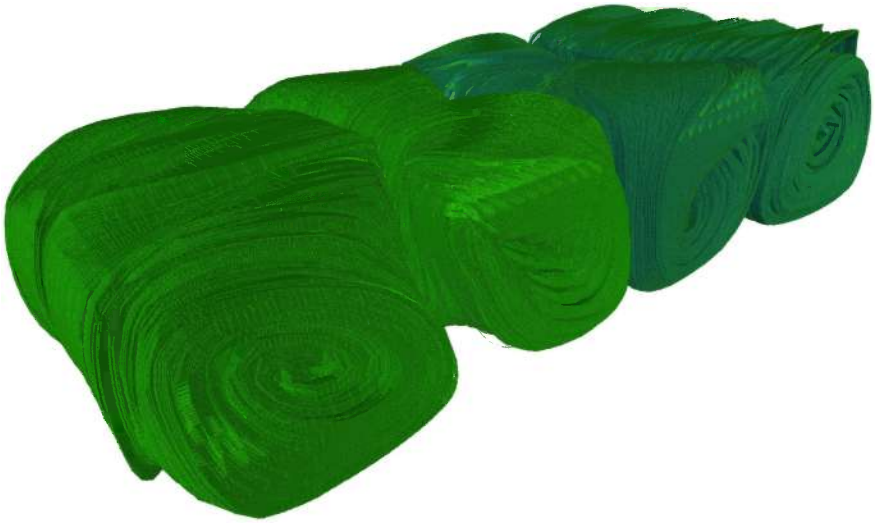


Figure 6.12: Render of Bénard-Rayleigh convection, with five rounds of refinement. Finer detail becomes more apparent, and the features begin to exhibit a shape more like those found in Figure 6.11(b)

point as done in Chapters 4 and 5. Example output of this set is presented in Figure 6.13. We used a $5 \times 1 \times 1$ processor grid. This lead to an imbalanced partition of the transition graph; some processes had much fewer nodes than others, resulting in an uneven memory distribution.

Figure 6.13: Render of the Square Cylinder dataset. The two smaller components visible on the left are situated next to where the square cylinder was placed.

We ran this same dataset to five rounds of refinement on eight cores (a $2 \times 2 \times 2$ processor grid), consuming $\approx$ 12 GB of RAM and taking $\approx$ 33 minutes to complete. The result is shown

in Figure 6.14. This same set, with four rounds of refinement, consumed approximately the same amount of memory with the previous algorithm implementations. The output volume is $3072 \times 1024 \times 768$, using ≈ 10 GB of disk space. This output volume size begins to pose a challenge to efficient visualization.

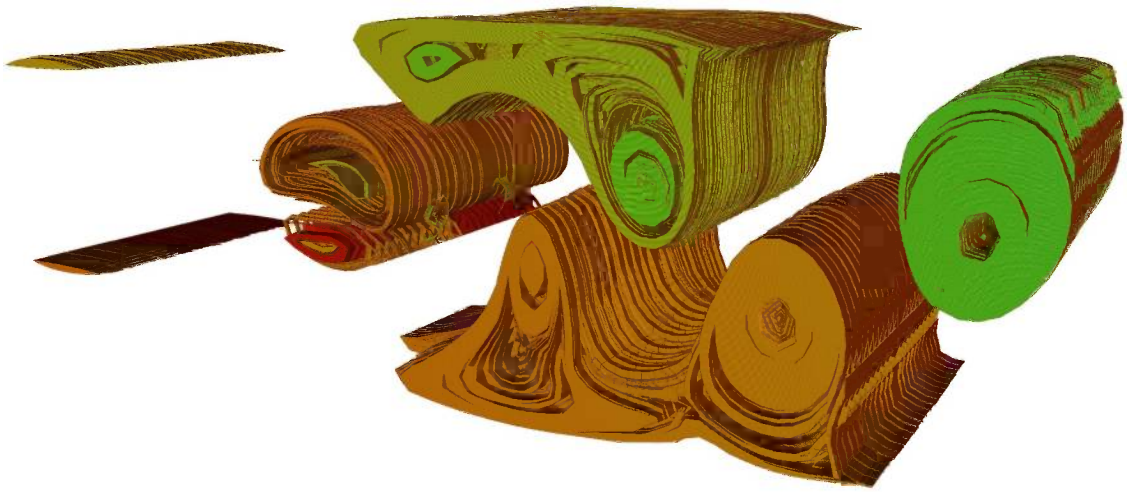


Figure 6.14: Render of the same dataset, with five rounds of refinement. Note the center components that emerge.

6.4.4 Lillgrund Wind Farm Turbine

This dataset is a vector field derived from a computational fluid dynamics simulation of the Lillgrund wind array, constructed using OpenFOAM[44] by Churchfield et al. [47]. The array itself is located off the coast of Denmark and consists of 48 turbines, each with a triplet of 93 meter long blades. The simulation of this array is used to study turbine placement, especially in regards to wakes created by the blades in high velocity wind. These meandering wakes pose a hazard to downstream turbines, as the vortices induced in the flow reduce the amount of power the downstream turbine can generate. The wakes also introduce additional stresses to the blades and gearbox, shortening turbine life. Study of these wakes are thus critical to ensuring proper energy production and recovery of investment from any wind farm.

The dataset is challenging, as the input data is extremely turbulent and our algorithm is likely to produce a large single component. Thus, we explore a portion of this dataset by

focusing on a single turbine (#15). The dimensions of the input volume is $142 \times 149 \times 84$. Due to the predominantly singular direction of the flow (generally in the $+x, +y$ direction), we use a similar tactic from the Square Cylinder dataset and subtract the average velocity from the vector field. This is a challenging dataset to analyze, as the velocity around the blades exhibits high turbulence.

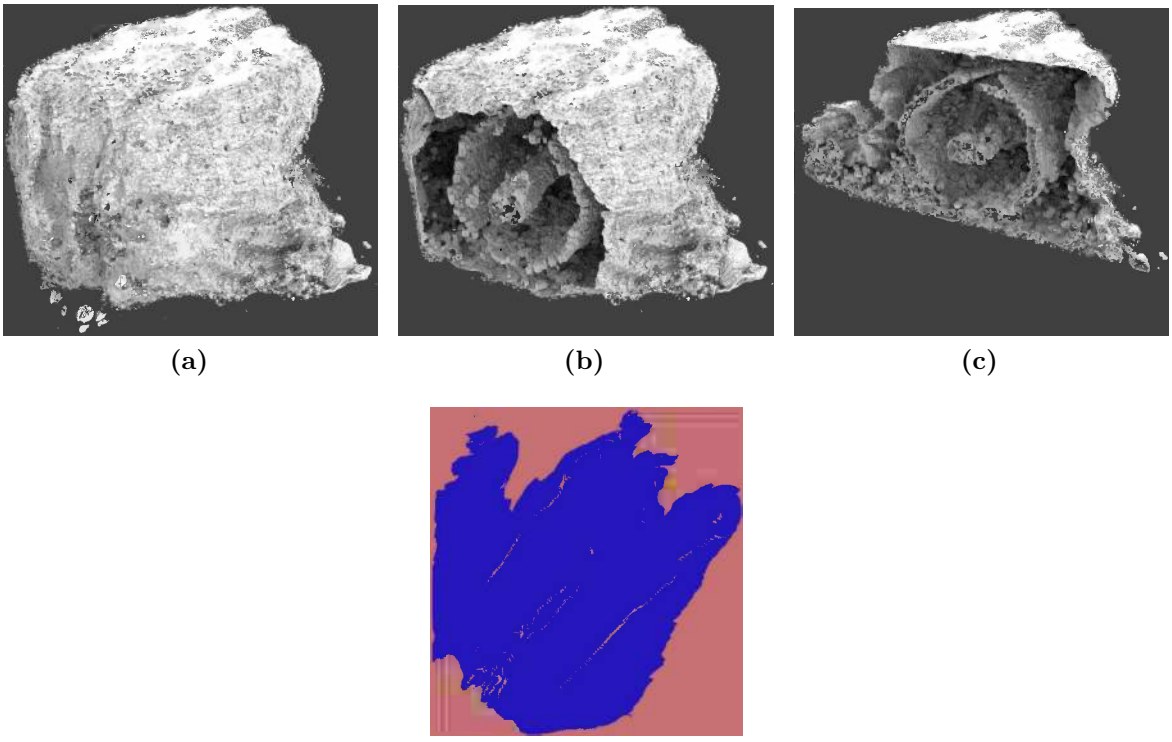
In Figure 6.15 we show a series of the results of our new algorithm on this dataset, using a $2 \times 2 \times 2$ processor grid (eight processors in total) and two rounds of refinement. Due to the noise in this dataset, volume rendering did not provide adequate clarity; we have constructed a geometric surface of the components. The surface was then rendered with ambient occlusion and global illumination using the open source 3D pipeline suite Blender® in order to provide detail that was obscured when the geometry was viewed under simpler interactive lighting methods. Due to floating point roundoff and thresholding, the surface generation phase has contributed to the rough look of the output.

This dataset grew to use approximately 83 million nodes (around 10 million per process), but, due to noise, had a large number of arcs that contributed to a nearly 40 GB tally of RAM. This dense connectivity also resulted in a 48 minute runtime.

The results demonstrate a complex structure. Using a slice of the output volume in Figure 6.15(d) as a guide, we can see the circular wake of the blade tips and a central vortex (the three-bladed turbine would be located at the approximate location of the center vortex in Figure 6.15(b). It is important to note that these features show an *absence* of any recurrence in those areas.

6.4.5 Hurricane Isabel Simulation

Previously visited in Chapter 4, this dataset is a simulation of Hurricane Isabel [23]. This simulation, produced by the Weather Research and Forecast model, courtesy of NCAR, and the U.S. National Science Foundation, was put forth as a way for researchers to demonstrate a variety of visualization techniques, and encourage inter-domain research collaboration.



(a) **(b)** **(c)**

(d)

Figure 6.15: Render of wind velocity dataset, centered around a turbine, which would be located approximately at the center vortex in Figure 6.15(b). The average velocity has been subtracted from the vector field. When the front of the resultant geometry is clipped, internal structure is revealed. Rendered with ambient occlusion and global illumination. Figure 6.15(d) shows a slice through the center of this dataset, oriented perpendicular to z. Blue indicates the presence of a component.

Our previous results were obtained for a heavily subsampled version of the dataset, so that it could be processed on a desktop class machine. As seen in Figure 6.16, the structure of the resulting component has changed with a larger input voxel size of $180 \times 180 \times 100$, processed on an $2 \times 2 \times 2$ processor grid with three rounds of refinement. As in the previous setup of Section 4.6, we have clipped the sides of the volume to expose recurrence around the central vortex. Subsampling likely removed much of noise that is present in this set, still, the general form of a hurricane is shown; a large disk at top, a smaller vortex in the center, and a large rotation at the bottom. The output dataset is $1440 \times 1440 \times 800$ voxels, and requires 6.3 GB for the on-disk format.

With the transition graph influenced by a single large component, the DCSC algorithm is not able to extract much parallelism. Almost all of the final transition graph's 313 million nodes and 1.375 billion arcs are used to represent this large central vortex, leading to a high runtime requirement of approximately ten hours. Most of this time was spent on computing the strongly connected component.

(a) Results appearing from the x axis.

(b) Results appearing from the y axis.

(c) Slice taken through the x axis, showing general vortex shape.

Figure 6.16: Isabel dataset results.

6.4.6 Ocean Circulation

This dataset is a model of oceanic circulation, provided by the *Estimating the Circulation and Climate of the Ocean, Phase II (ECCO2): High-Resolution Global-Ocean and Sea-Ice Data Synthesis* project [24]. In this project, worldwide ocean currents were reconstructed

with the aid of satellite imagery to provide a basis to study carbon cycle impact and polar ocean evolution.

We have constructed an input vector field volume containing the velocities of the surface of the ocean, stacked in the z direction in order of time. The slices are three days apart. Since we are processing time slices, our software was modified to disallow any inter-slice connectivity; removal of all xy planar 2D mesh cells and 1D mesh cells was sufficient to enforce this rule. This provides us the opportunity to examine evolution of eddies and circulation through time. Note that we are treating this as a planar vector field, rather than a field on a sphere; the components detected here are ocean eddies and with few exceptions are likely to be small.

The input volume has dimensions of $1440 \times 720 \times 30$ (thus, 30 time slices, or a total of 90 days). Three rounds of refinement were run, using a $3 \times 2 \times 2$ processor grid, consuming ≈ 244 GB of RAM. The dimensions of the output voxel volume is $11520 \times 5760 \times 240$, requiring 60 GB of storage. This dataset demonstrates the usefulness of our parallel-component processing system; Table 6.4 shows the runtime characteristics of the software; the counts of arc and nodes are lesser than equivalent sized runs, as we are using the time-as-stack modifications. The SCC computation time (with a transient) shows a general trend toward less time expended, even in the face of increasing nodes and arc counts. More components appear under refinement that can be processed either locally in complete independence using the generally faster Tarjan approach, or in parallel using cooperating processes.

Figure 6.17 shows the entire rendered dataset as it appears in VAPOR, with a backdrop of major landmasses. Note the strong currents in the Gulf of Mexico, corresponding to the Gulf Stream and the capture of the North, South and Counter Equatorial currents in the Pacific. In Figure 6.18 slices of the volume are shown. Of particular interest is the evolution of gulf currents and the same pacific equatorial circulation. Figure 6.19 shows the Pacific currents evolution in more detail with a time-slice view. Simply rendering the full 60 GB output volume now becomes infeasible; the volume shown has been rendered at half resolution.

Table 6.4: Ocean dataset runtime statistics. The number of arcs begin to weigh heavily on RAM consumption, yet we are able to process the graph faster due to the increasing number of small components that appear under refinement.

Refinement	# Nodes	# Arcs	SCC Time
Coarse	57,959,944	96,404,341	2174s
1	141,439,908	416,274,581	1159s
2	283,075,324	969,737,437	1837s
3	538,628,764	1,949,491,730	1321s

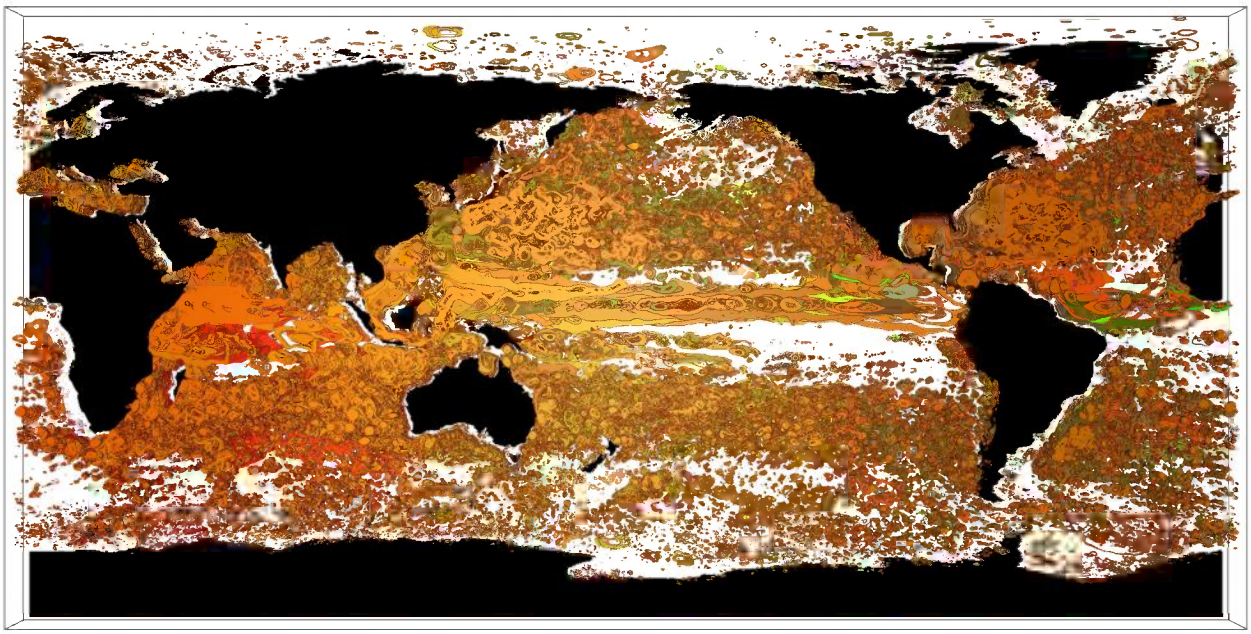


Figure 6.17: Ocean dataset results with three refinement steps, rendered at half resolution. Note that, due to the sheer number of components (more than 100,000), some components have been given similar colors. Pacific equatorial currents are prominent, as well as complex eddies near the Gulf of Mexico.

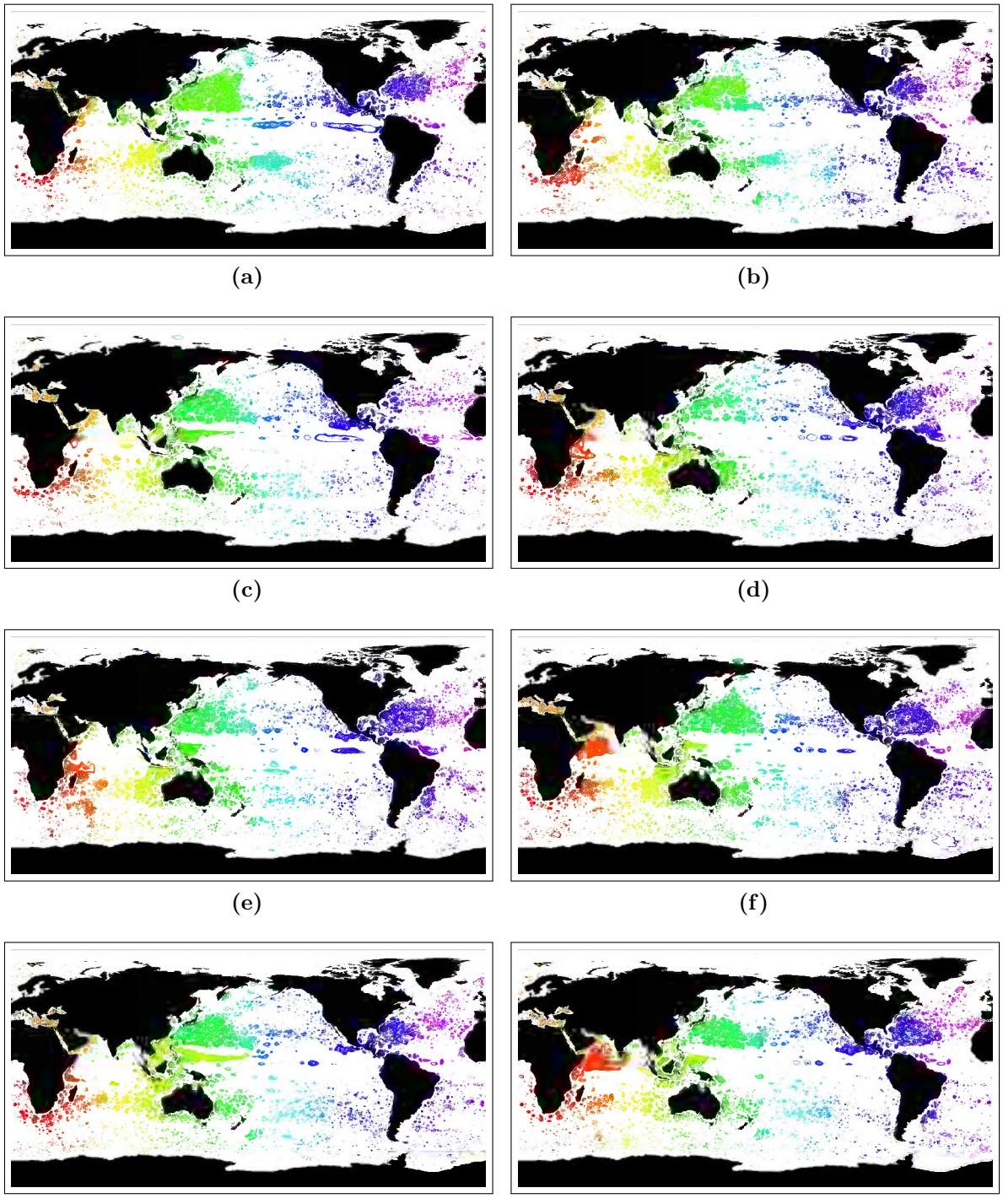
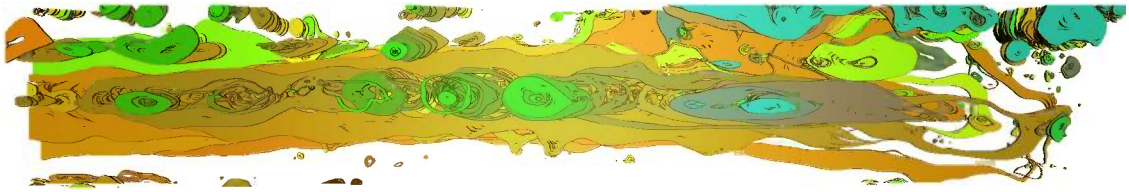
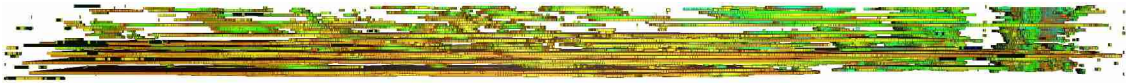


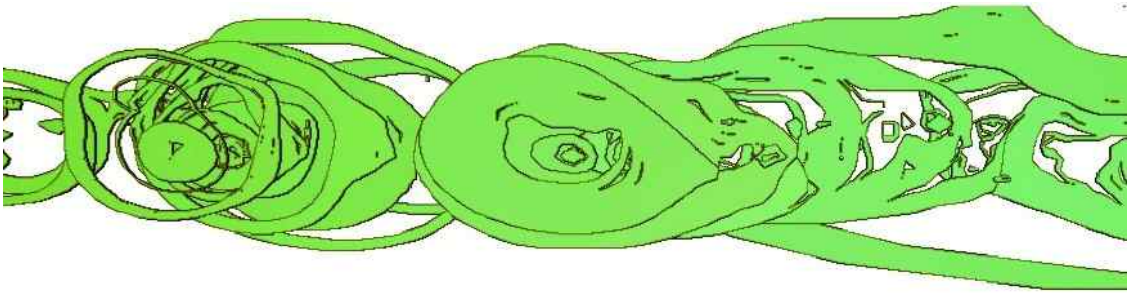
Figure 6.18: Time slices of ocean dataset, in increasing time. The outline of the African Continent is visible at the far left of each slice, with South America visible at the far right.



(a) Subvolume of the ocean dataset in the $-t$ direction.



(b) The same sub-volume shown from the side, demonstrating feature evolution through time.



(c) Detail of 10 time steps of the same sub-volume, top view.

(d) Detail of 10 time steps of the same sub-volume, side view.

Figure 6.19: Detail of Pacific Ocean equatorial circulation. Small eddies are shown to move through time in a horizontal fashion, or join/expand into larger areas of circulation.

6.5 Discussion

The results show that the implementation described in this chapter reduces the memory usage by at least 50%, permitting additional refinement rounds or larger datasets to be analyzed. In practice, one can expect to be able to perform an additional round of refinement for a typical dataset. This results in a more detailed and accurate output. With our memory reduction scheme for graph nodes, graph arcs now take the stage as the primary consumer of memory.

The addition of MPI provides the ability to analyze flow recurrence in real world datasets of unprecedented sizes by utilizing the vast resources of a supercomputer. Current desktop

class computers are not equipped with more than 32 GB of RAM; some of the datasets discussed in this section, particularly the Ocean Dataset, require considerably more than this to run. Given the memory reductions for the desktop data cases, we can expect that the original implementation would consume more than half a terabyte of RAM for this dataset. Desktop machines will not be able to handle such data sets for years to come. Supercomputers also allow the storage of large result data-files and the means to visualize them.

Performance is the main weakness of our MPI-based implementation. We stress that the goal of this chapter is not to decrease the running time of our algorithm, but to improve memory usage to allow analysis of larger datasets. This is somewhat different from the common use of HPC; we do not seek additional processors for a performance benefit, but for the memory provided. Thus, we do not provide common metrics like Karp-Flatt [48] or Gustafson's Law [49]. However, we would like to provide some insights into factors that influence the performance of our implementation.

Due to our choice of input volume decomposition, we can experience an imbalance in the distribution of nodes across processors. For example, in the Square Cylinder dataset, some processes experienced over three times the node load of others, simply due to the uneven spatial distribution of the components in the graph. A dynamic node balancing scheme might help improve the performance. However, effective node balancing is not easy to achieve in our application. Challenges include efficient and effective determination of node ownership and balancing projection information (as arcs would have to be moved from process to process as well). Consideration must also be given to the overhead inherent in using any messaging interface; every boundary has a cost, and in MPI every message sent contributes to additional runtime requirements [50]. This is an issue even when our implementation is run on a single machine. For the Ocean dataset case, every process sent about 6 GB of algorithm related data on average, mostly as small messages. Future implementations may consider additional schemes to reduce the number of messages that need to be sent.

There are also many dataset related factors that affect the runtime expense. These factors generally focus around the strongly connected component computation and DCSC. As an example, in the case of the Lillgrund dataset, over 93% of the time used was in computing the strongly connected components, as compared to the shorter amount of time spent with the shared memory implementation in Figure 5.4. The number of strongly connected components in the transition graph is another major dataset related factor. For example, our implementation is faster if there are many small SCCs, as each component is handled in parallel. However, large components are handled as a single task (essentially removing parallelism) and carry high communication overhead; for these large components to be detected, the nodes involved will be traversed *twice* (recall that an SCC in the DCSC algorithm is the intersection of two sets discovered by traversals), which will incur communication and processing costs. Further, after the partition and discovery of an SCC, we will have a number of nodes left behind. Without a post-partition trim phase, we must run the DCSC algorithm for these likely non-SCC nodes, which can be time-consuming, especially if they are on a shared boundary. More generally, the performance depends on the number of SCCs that are shared across processors; the more SCCs that can be processed locally without involving MPI or waiting for other processors, the better. Finally, there is an issue of component convergence rate. Components that converge quickly as the graph is refined might localize to a single process after a small number of refinement iterations, or require only a modest amount of communication to process if larger. Ideally, the input data partitioning should maximize the spread of local components so that each processor can process their work queue in parallel consuming roughly the same amount of time. A poor selection might cause transition graph balancing issues or excessive component sharing between processes. Optimal partitioning is typically hard to perform a priori.

As an example, we take a representative case by exploring the Square Cylinder dataset. In Figure 6.20 we show the dataset being analyzed with increasing numbers of processors in the x direction; i.e. a processor grid of $n \times 1 \times 1$. The dataset provides a small number of large

components that, with few exceptions, span the whole input volume; this is an unfavorable situation, but prevalent in the datasets we have analyzed. Note that in the single core case, we default to using the Tarjan SCC algorithm. At higher numbers of processors, we do not see any scaling; the runtimes are roughly the same for each successive processor grid, demonstrating poor *strong scaling* that results from this common scenario. We can see the opposite effect as demonstrated by the Ocean Dataset, when the dataset yields numerous small, local, components that are evenly spread to all processors. Although this dataset is larger than the Hurricane Isabel dataset, the analysis required much shorter time. The complex interplay of the number, distribution and convergence rate of components makes precise estimation of runtime difficult.

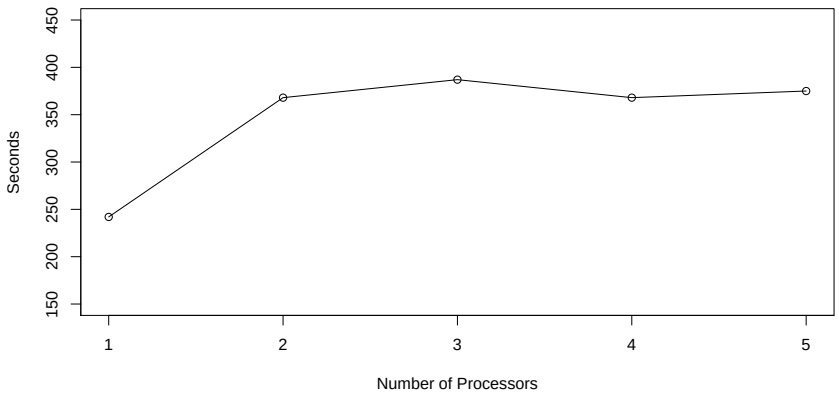


Figure 6.20: Scaling of Square Cylinder Dataset. Data collected using four rounds of refinement and increasing processor splits in x direction. Note that the one processor case defaults to the serial SCC implementation.

The implementation described in this chapter can be improved in several ways to address some of these issues. Notably, we do not do a post-partition trim phase. This step, while complex to implement, would further reduce the number of nodes that would be poor choices for further partitions. Hong et al. [43] shows that, while not providing a universal improvement to runtimes, an additional trim operation per-partition can reduce execution times for datasets that result in a small graph diameter. Further, some operations could be

parallelized with threads, in a hybrid multi-process and shared memory system. A multi-scale scheme where a coarse result guides the partitioning of the finer transition graph may also be considered. Finally, it would be interesting to come up with alternatives for the strongly connected component algorithm. For example, faster ways to compute reasonably tight upper bounds on SCCs could potentially replace the exact SCC computation.

6.6 Conclusion

In this chapter, we have shown an improvement to 3D PC vector field analysis using a distributed transition graph, and a new node representation to reduce memory usage. Large scale datasets, of a size that were impossible to study using previous implementations, were analyzed on a cluster, demonstrating the utility of a distributed approach. Changes to the transition graph representation not only reduced memory but reduced the complexity of the underlying graph data structure.

As a drawback to these advances, higher runtimes were experienced as a consequence of the DCSC procedure, graph load balancing, and use of message passing. Based on these factors, our algorithm appears most suitable for RAM rich shared memory machines. For datasets small enough to be analyzed on a desktop, the implementation of the previous chapter may be favored to avoid the complexity of a tuned MPI installation. However, it should use the updated transition graph representation to reduce memory pressure.

CHAPTER 7

CONCLUSION

In this thesis, we have presented three contributions to PC vector fields.

As the first contribution, we describe an algorithm for finding recurrence in 3D vector fields. Most vector fields of interest in science and engineering are three-dimensional. Our approach allows one to analyze them directly, without being restricted to slices or other two-dimensional domains.

The algorithm was then restructured to increase parallelism and improve performance on shared memory machines. This led to considerably better run times, but the sizes of the datasets that could be analyzed remained small because of high memory requirements.

Finally, we designed and implemented a distributed version of the algorithm, suitable for cluster systems. The main goal behind this effort was to enable analysis of larger datasets as well as to provide more detailed analysis of smaller sets by allowing more refinement iterations.

We have thus provided vector field researchers with two implementations to use as a guideline in constructing their own utilities to aid them in reaching their scientific goals. Our results demonstrate their usefulness; for example, in extracting oceanic currents or identifying convection cells.

Our contribution has also opened up new avenues of future work on this topic.

The primary disadvantage of our approach is an exponentially growing transition graph, leading to high resource usage. While, we expect new technologies like compressible RAM, non-volatile memory, and solid-state hard drives to mitigate the cost of memory requirements, compressed graph representations should be explored.

With our 3D formulation, the transition graph commonly contains a single large component, which may obscure detail; for example, in 6.4 in the Lillgrund Wind Array test

case, a series of 2D slices of the volume perpendicular to the prevailing wind velocity may provide more interesting structure. Thus, the 2D formulation should not be discarded in the face of our work, but instead the two may be able to work in concert on a single dataset to provide insight that a single method could not bring alone. Further, there are methods to remove noise and clean topology in 2D; moving these techniques to 3D could improve the visualization.

Another prospective area is in feature identification, matching, and evolution. As an example, in Section 6.4, we demonstrated extraction of circulating eddies in oceanic currents and observed their evolution through slices of time. Matching features in multiple transition graphs in an automated fashion would allow a researcher to explore how features grow, branch, and merge. Multiple transition graphs could also be used to explore the stability of features that are found, giving the researcher a control to perturb the vector field and explore what features remain or are created.

Concerning implementation; as our methods rely heavily on efficient strongly connected component algorithms, it may be prudent to develop new SCC methods that can better process the typically large components found in 3D transition graphs.

REFERENCES CITED

[1] Robert S. Laramee, Christoph Garth, Helmut Doleisch, Jürgen Schneider, Helwig Hauser, and Hans Hagen. Visual analysis and exploration of fluid flow in a cooling jacket. In *Proceedings IEEE Visualization 2005*, pages 623–630, 2005.

[2] Christoph Garth, Robert S. Laramee, Xavier Tricoche, Jürgen Schneider, and Hans Hagen. Extraction and visualization of swirl and tumble motion from engine simulation data. In H. Hauser, H. Hagen, and H. Theisel, editors, *Topology-Based Methods in Visualization (Proceedings of Topo-In-Vis 2005)*, pages 121–135, 2007.

[3] Guoning Chen, Konstantin Mischaikow, Robert S. Laramee, Pawel Pilarczyk, and Eugene Zhang. Vector field editing and periodic orbit extraction using morse decomposition. *IEEE Transactions on Visualization and Computer Graphics*, 13:769–785, July 2007. ISSN 1077-2626.

[4] Guoning Chen, Konstantin Mischaikow, Robert S. Laramee, and Eugene Zhang. Efficient morse decompositions of vector fields. *IEEE Transactions on Visualization and Computer Graphics*, 14:848–862, July 2008. ISSN 1077-2626.

[5] Edward N. Lorenz. Deterministic nonperiodic flow. *Journal of Atmospheric Sciences*, 20(2):130–148, 1963.

[6] Marian Mrozek and Pawel Pilarczyk. The conley index and rigorous numerics of attracting periodic oribts. In *Conference on Variational and Topological Methods in the Study of Nonlinear Phenomena*, Pisa, Italy, 2000.

[7] Andrzej Szymczak and Eugene Zhang. Robust morse decompositions of piecewise constant vector fields. *IEEE Transactions on Visualization and Computer Graphics*, 18(6):938–951, 2012.

[8] Andrzej Szymczak and Nicholas Brunhart-Lupo. Nearly Recurrent Components in 3D Piecewise Constant Vector Fields. *Computer Graphics Forum*, 31(3):1115–1124, 2012.

[9] Nicholas Brunhart-Lupo and Andrzej Szymczak. Parallel Computation of Nearly Recurrent Components of Piecewise Constant Vector Fields. In Bremer, Peer-Timo and Hotz, Ingrid and Pascucci, Valerio and Peikert, Ronald, editor, *Topological Methods in Data Analysis and Visualization III*, Mathematics and Visualization, pages 119–132. Springer International Publishing, 2014.

[10] James Helman and Lanbertus Hesselink. Representation and display of vector field topology in fluid flow data sets. *Computer*, 22:27–36, 1989.

[11] Thomas Wischgoll and Gerik Scheuermann. Detection and visualization of closed streamlines in planar flows. *IEEE Transactions on Visualization and Computer Graphics*, 7: 165–172, 2001.

[12] Holger Theisel, Tino Weinkauf, Hans-Christian Hege, and Hans-Peter Seidel. Grid-independent detection of closed stream lines in 2d vector fields. In *VMV'04*, pages 421–428, 2004.

[13] Holger Theisel, Tino Weinkauf, Hans-Christian Hege, and Hans-Peter Seidel. Saddle connectors – an approach to visualizing the topological skeleton of complex 3d vector fields. In *PROC. IEEE VISUALIZATION*, pages 225–232, 2003.

[14] Harsh Bhatia, Shreeraj Jadhav, Peer-Timo Bremer, Guoning Chen, Joshua A. Levine, Luis Gustavo Nonato, and Valerio Pascucci. Flow Visualization with Quantified Spatial and Temporal Errors Using Edge Maps. *IEEE Transactions on Visualization and Computer Graphics*, 18(9):1383–1396, September 2012.

[15] Harsh Bhatia, Shreeraj Jadhav, Peer-Timo Bremer, Guoning Chen, Joshua A. Levine, Luis Gustavo Nonato, and Valerio Pascucci. Edge maps: Representing flow with bounded error. In *Proceedings of IEEE Pacific Visualization Symposium 2011*, 2011.

[16] Robin Forman. A user's guide to discrete morse theory. In *Proceedings of the 2001 International Conference on Formal Power Series and Algebraic Combinatorics*. Advances in Applied Mathematics, 2001.

[17] Jan Reininghaus and Ingrid Hotz. Combinatorial 2d vector field topology extraction and simplification. In *Mathematics in Visualization (TopoInVis)*, Snowbird, Utah, 2009.

[18] Jan Reininghaus, Christian Lowen, and Ingrid Hotz. Fast combinatorial vector field topology. *IEEE Transactions on Visualization and Computer Graphics*, 99:1433–1443, 2010.

[19] Attila Gyulassy, Peer-Timo Bremer, and Valerio Pascucci. Computing morse-smale complexes with accurate geometry. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2014–2022, 2012.

[20] Charles Conley. *Isolated invariant sets and the Morse index*. Number 38 in Regional Conference Series in Mathematics. Conference Board of the Mathematical Sciences, 1978.

[21] D. E. Norton. The fundamental theorem of dynamical systems. *Commentationes Mathematicae- Universitatis Carolinae*, 36(3):585–598, 1995.

[22] Andrzej Szymczak. Stable morse decompositions for piecewise constant vector fields on surfaces. *Eurographics / IEEE Symposium on Visualization 2011*, 2011.

[23] Bill Kuo, Wei Wang, Cindy Bruyere, Tim Scheitlin, and Don Middleton. IEEE Visualization 2004 contest: Data set. `http://vis.computer.org/vis2004contest/data.html`, 2004.

[24] ECCO2: Estimating the Circulation and Climate of the Ocean, Phase II. `http://ecco2.org`.

[25] Herbert Edelsbrunner and Ernst Peter Mücke. Simulation of Simplicity: A Technique to Cope with Degenerate Cases in Geometric Algorithms. *ACM Transactions on Graphics*, 9:66–104, 1990.

[26] L. Gorniewicz. *Topological Fixed Point Theory of Multivalued Mappings*, volume 4 of *Topological Fixed Point Theory and Its Applications*. Springer, 2 edition, 2006.

[27] Paul E. Black. Strongly connected component. In Paul E. Black, editor, *Dictionary of Algorithms and Data Structures*. U.S. National Institute of Standards and Technology, August 2008.

[28] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[29] E. Nuutila and E. Soisalon-Soinen. On finding the strongly connected components in a directed graph. In *Information Processing Letters*, volume 49(1), pages 9–14. 1994.

[30] Z. Galias and P. Zgliczyński. Computer assisted proof of chaos in the lorenz system. *Physica D*, 115:165–188, 1998.

[31] Daniel Weiskopf, Tobias Schafhitzel, and Thomas Ertl. Texture-based visualization of unsteady 3d flow by real-time advection and volumetric illumination. *IEEE Transactions on Visualization and Computer Graphics 13*, 3:569–582, 2007.

[32] Simone Camarri, Maria-Vittoria Salvetti, Marcelo Buffoni, and Angelo Iollo. Simulation of the three-dimensional flow around a square cylinder between parallel walls at moderate reynolds numbers. In *XVII Congresso di Meccanica Teorica ed Applicata*, Florence, Italy, September 2005.

[33] International CFD Database. `http://cfd.cineca.it/`.

[34] Jiri Barnat, Petr Bauch, Lubos Brim, and Milan Ceska. Computing strongly connected components in parallel on cuda. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 544–555, Washington, DC, USA, 2011. IEEE Computer Society.

[35] Lisa Fleischer, Bruce Hendrickson, and Ali Pinar. On Identifying Strongly Connected Components in Parallel. In *Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, IPDPS '00, pages 505–511, London, UK, UK, 2000. Springer-Verlag.

[36] G.M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. *Proceedings of AFIPS Conference*, pages 483–485, 1967.

[37] R.A. Finkel and J.L. Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, 1974.

[38] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[39] Douglas Gregor and Andrew Lumsdaine. The Parallel BGL: A generic library for distributed graph computations. In *Parallel Object-Oriented Scientific Computing (POOSC)*, 2005.

[40] Nicholas Edmonds. *Active Messages as a Spanning Model For Parallel Graph Computation.* PhD thesis, Indiana University, 2013.

[41] Nicholas Edmonds and Andrew Lumsdaine. Parallel Boost Graph Library 2 (Active Messages Version). `http://www.crest.iu.edu/projects/pbgl2/`.

[42] William Mclendon, III, Bruce Hendrickson, Steven J. Plimpton, and Lawrence Rauchwerger. Finding strongly connected components in distributed graphs. *J. Parallel Distributed Computing*, 65:901–910, 2005.

[43] Sungpack Hong, Nicole C. Rodia, and Kunle Olukotun. On Fast Parallel Detection of Strongly Connected Components (SCC) in Small-World Graphs. In *2013 International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*, November 2013.

[44] Silicon Graphics International Corporation (SGI). OpenFOAM. `http://www.openfoam.org`.

[45] J. Clyne, P. Mininni, A. Norton, and M. Rast. Interactive desktop analysis of high resolution simulations: application to turbulent plume dynamics and current sheet formation, 2007.

[46] VAPOR, a product of the Computational Information Systems Laboratory at the National Center for Atmospheric Research. `www.vapor.ucar.edu`.

[47] M. J. Churchfield, S. Lee, P. J. Moriarty, L. A. Martinez, S. Leonardi, G. Vijayakumar, and J. G. Brasseur. Large-Eddy Simulation of Wind-Plant Aerodynamics. In *Proceedings of 50th AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition*, January 2012.

[48] Alan H. Karp and Horace P. Flatt. Measuring parallel processor performance. *Commun. ACM*, 33(5):539–543, May 1990.

[49] John L. Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM*, 31: 532–533, 1988.

[50] Douglas Doerfler and Ron Brightwell. Measuring MPI Send and Receive Overhead and Application Availability in High Performance Network Interfaces. In Bernd Mohr, JesperLarsson Träff, Joachim Worringen, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 4192 of *Lecture Notes in Computer Science*, pages 331–338. Springer Berlin Heidelberg, 2006.