A Parallel and Memory Efficient Algorithm for Constructing the Contour Tree

A Project Report

Submitted in partial fulfilment of the

requirements for the Degree of

Master of Technology

in

Computational Science

by

Aditya Acharya

Supercomputer Education and Research Centre

Indian Institute of Science

BANGALORE – 560 012

JULY 2014

Acknowledgements

I am grateful to everyone who stood beside me during the course of my project and the writing of my thesis. It would have been impossible to complete my project without the unfledging support and invaluable help of the kind people around me, only some of whom I can mention here.

First I would like to thank my project guide, Dr. Vijay Natarajan, without whose support and patience, this thesis would not have been possible. Besides completing my project under him, his guidance and suggestions have helped me gain an immense knowledge about my research field and academic research in general.

I thank Dr. Yogesh Simmhan and Dr. Sathish Vadhiyar for their constructive criticisms and helpful suggestions.

The library and the computing facilities of the Institute and my Department have been indispensable.

I would like to thank my labmates especially Dilip, Nithin, Preeti, Talha and Vidya for their help and support, throughout the duration of my project, and wish them all the best for their future endeavors. I would badly miss the comfortable couch in my lab, which enabled me to catch a quick nap on those long and arduous nights.

I would also like to thank all my classmates and colleagues for their unconditional support throughout my course. I will cherish their friendship, for years to come.

Last but not the least, I would like to express my sincere gratitude to all those who have directly or indirectly helped me in making this happen.

To Paul, DP and Lucy, for giving me a reason

Abstract

The contour tree is a topological structure associated with a scalar function that tracks the connectivity of the evolving level sets of the function. It efficiently stores the topological highlights of the scalar function and supports intuitive and interactive visual exploration and analysis. This thesis describes a fast, parallel, and memory efficient algorithm for constructing the contour tree of a scalar function on shared memory systems. Comparisons with existing implementations show significant improvement in both the running time and the memory expended. We observe near linear scaling of our implementation with increasing number of processors. The proposed algorithm is also particularly suited for large data sets that do not fit in memory. For example, the contour tree for a scalar function defined on a 8.6 billion vertex domain (2048X2048X2048 volume data) can be efficiently constructed using less than 10GB of memory.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

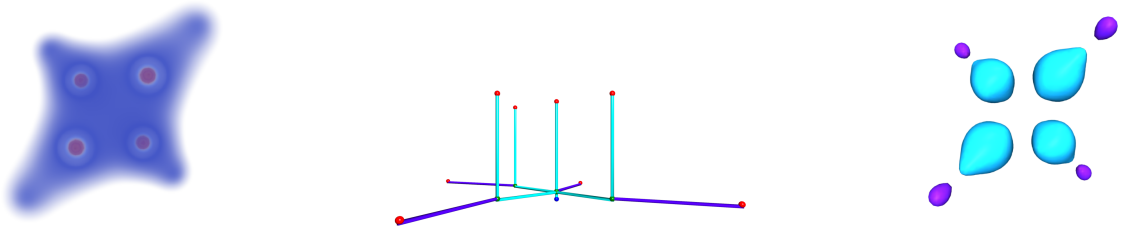
Scientific data obtained from simulations and measurement devices is often represented as a scalar function over a two, three, or higher dimensional domain. A scalar function, also called as a scalar field, maps each point on the domain to a real value. A popular method for visualizing scalar functions is via extraction of isosurfaces or level sets. A level set of a scalar function consists of all points where it attains a given real value. The contour tree tracks topology changes in level sets of a scalar function defined on a simply connected domain, and therefore serves as a good abstract representation of the data. In this paper, we propose a parallel algorithm for fast and memory efficient construction of the contour tree for large data sets.

1.1 Motivation

Topological information about a data set have been used in a variety of methods and techniques in exploring and visualizing scalar functions. Contour trees have been widely used to encode such information [1],[2]. Further, they are used to efficiently compute and explore level sets and isosurfaces [3],[4] in three dimensional scalar functions. Flexible isosurfaces [3] exploit the relation between an iso-surface and the arcs in a contour tree, effectively visualizing different components of the iso-surface.

Contour trees have also been used in various other applications including topography and GIS [5], in extraction of hierarchical landscapes[6], for surface segmentation and parametrization in computer graphics [7], [8], [9], image processing and analysis of volume data sets [10], [11], designing transfer functions for volume rendering in scientific visualization [12], [13], [14], [15], and exploring high dimensional data in information visualization [16], [17].

More recently, the contour tree has been used to identify and extract symmetric patterns in 3D scalar functions [18]. Figure 1.1 shows symmetric structures in a tetrahedrane molecule (C_4H_4) that is extracted from an electron density distribution over the molecule. The contour tree of the scalar function is analyzed to identify similar sub trees. Each collection of similar sub trees corresponds to a set of symmetric regions in the data. Symmetry plays an important role in defining "features" in this application. The contour tree enables such a feature-directed exploration of scientific data that is particularly useful when the data is large.



(a) (b) (c)

Figure 1.1: Application of the contour tree to symmetry identification. (a) Volume rendering of electron density distributed in a C_4H_4 molecule.(b) The contour tree of the electron density function. Similar sub trees are colored cyan and purple. (c) The regions corresponding to the similar sub trees are symmetric.

The exponential growth in compute power has facilitated the generation of higher fidelity simulation data and higher resolution imaging data, which in turn has resulted in a massive increase in the size of the data sets. Topology-based methods were developed with the aim of enabling analysis and visualization of these large data sets by providing abstract representations of the key features in the data. However, the construction of the topological structures is now increasingly becoming a bottleneck. This necessitates the

development of efficient algorithms that can additionally handle large data sizes. With multicore and many-core CPUs becoming ubiquitous, it is also essential to leverage their power in computing topological structures such as the contour tree. Further, the memory required for such an algorithm grows proportionally to the size of the input. So, it is imperative to ensure that the algorithm is also memory efficient. We address the above-mentioned challenges in this thesis.

1.2 Contributions

We describe a fast and memory efficient parallel algorithm for computing the contour tree of a piecewise trilinear function defined over large structured grids, on a shared memory system.

The input grid is first subdivided into blocks. The Join and Split trees for the scalar function restricted to individual blocks are constructed in parallel via monotone path computation. The Join trees (and Split trees) of the individual blocks are stitched together, again in parallel, by identifying the nodes of the tree that lie on the common boundary between the blocks. The contour tree of the input is constructed in a final step by merging the Join and Split trees.

The algorithm is output sensitive *i.e.*, the running time depends on the number of nodes in the final contour tree. We report results that demonstrate significant improvements in terms of time and memory over the existing parallel algorithms. The contour tree for a data set containing 8.6 billion points ($2048 \times 2048 \times 2048$ volume) can be constructed within 3 minutes in a 64-core shared memory environment. In an 8-core environment, the algorithm uses no more than 10GB of memory and computes the tree in approximately 14 minutes.

1.3 Outline of the thesis

Chapter 2 provides the necessary background for rest of the thesis, where we introduce necessary definitions and terms. Chapter 3 describes the related work and the existing algorithms for computation of the contour tree. Next we describe our algorithm in Chapter 4 providing details of our implementation and data structures used. The observation and results, along with the experimental setup is described in Chapter 5. We conclude with Chapter 6, by providing future directions towards further improvements in the computation of the contour tree.

Chapter 2

Background

In this chapter, we introduce the necessary definitions and terms used in this paper. For a detailed discussion readers can refer to texts on computational topology and algebraic topology [19, 20, 21, 22].

If S is a simplicial complex, and V its vertex set, then the scalar function $\hat{f}$ is a function, from the set V to $\mathbb{R}$, mapping every vertex to a real value. Denote by f the continuous analog of $\hat{f}$ which extends $\hat{f}$ by assigning real values to every point in the connected space containing S. If S is a structured grid, then the function is extended into the interior of the cells via trilinear interpolation. A level set is given by $S_a = f^{-1}(a)$, set of all the points in $\tilde{S}$ having function value a. For $x, y \in \tilde{S}$, the continuous analog of S we say $x \sim y$ iff they belong to the same level set, *i.e.,* $f(x) = f(y)$ and belong to the same *component* of the level set $f^{-1}(f(x))$. Then the Contour tree is the quotient space $\tilde{S}/\sim$ which glues all the points equivalent under the relation $\sim$. Or in other words every connected component of a level set is represented by a point on the contour tree. A *sub-level* set is the set $S_{\underline{a}} = f^{-1}((-\infty, a])$, which contains every point in $\tilde{S}$ having function value less than or equal to a. Similarly a *super-level* is given by the set $S_{\bar{a}} = f^{-1}([a, \infty))$.

As we sweep across a range of function value the corresponding level set undergoes change in connectivity. Points at which the topology of the level sets change during this evolution are known as *critical points*. Points that are not critical are called *regular*

points. The contour tree expresses the evolution of the connected components of these level sets as a graph whose nodes correspond to critical points of the function. The *Join tree* tracks the evolution of sub-level sets while the *Split tree* tracks the evolution of super-level sets.

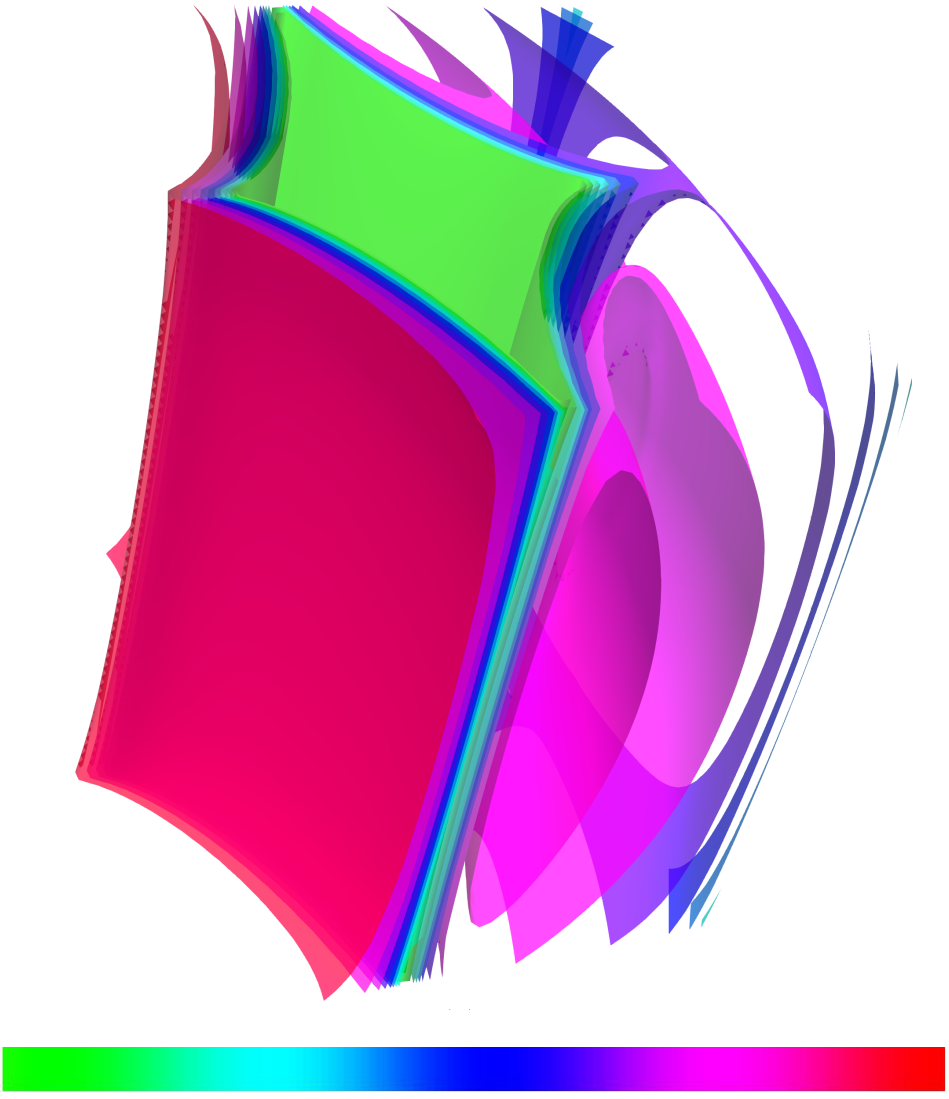


Figure 2.1: An analytic function sampled on a structured grid and visualized using multiple level sets. Each level set consists of one or more connected components;

Figure 2.1 shows multiple level sets extracted from a synthetic scalar function defined on a structured grid. Figure 2.2 shows the contour tree, Split tree, and Join tree for this scalar function. Each connected component of the level set, called a *contour* maps to a

different arc in the contour tree. The *maxima* are shown in red and the *minima* in blue. The other critical points are shown in green, which are called *saddles* where a single component of level set either splits or multiple components merge.

If all critical points of f are isolated and non-degenerate, then f is a Morse function [22, 21]. Critical points of a Morse function can be classified based on the behavior of the function within a local neighborhood. This condition typically does not apply for piecewise trilinear functions, like a scalar function defined on a 3D grid. However, a simulated perturbation of the function [19, Section 1.4] imposes a total order on the vertices and helps in consistently identifying the vertex with the higher function value between a pair of vertices. Consequently, critical points can be identified and classified based on local information about the function.

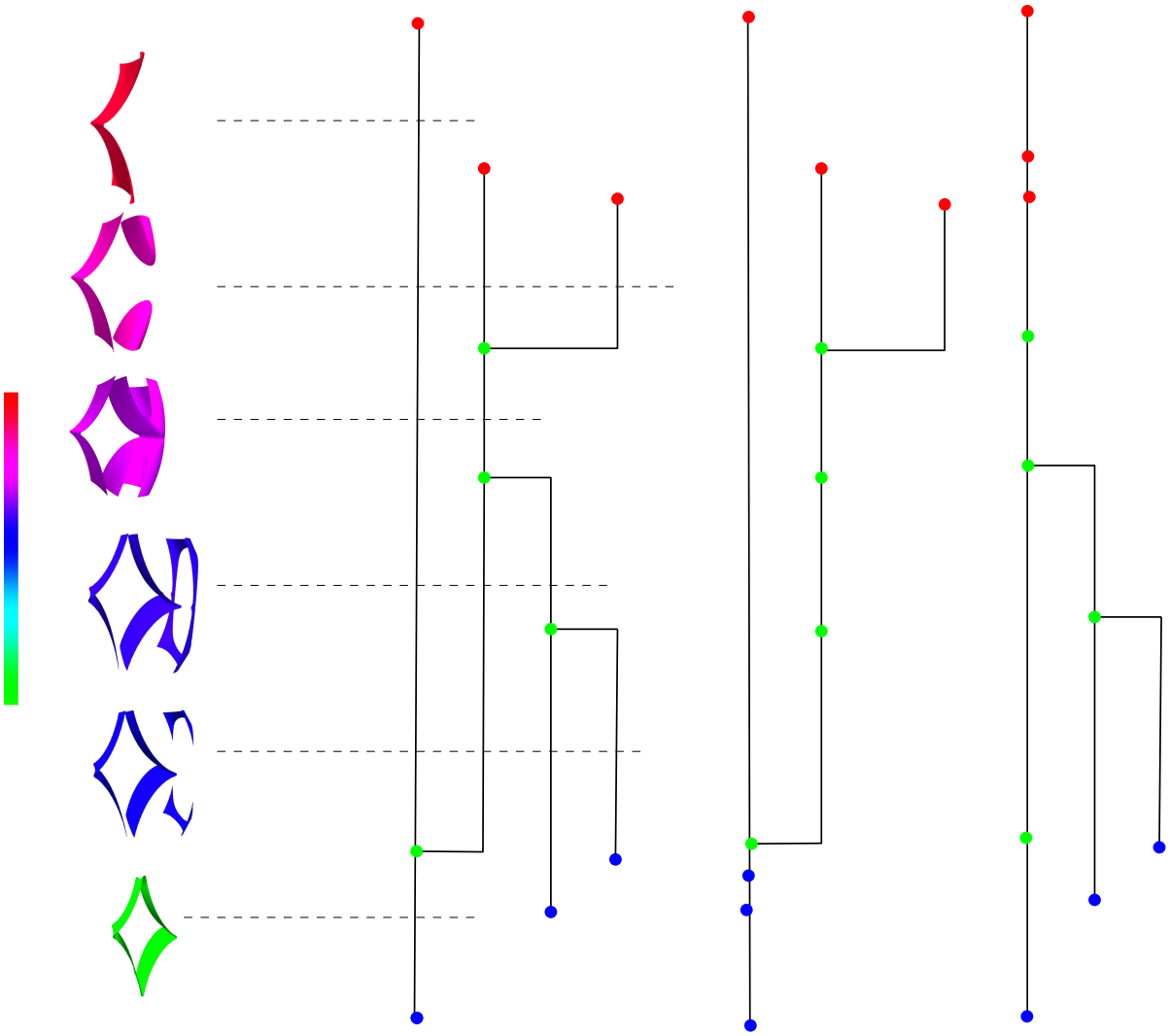


Figure 2.2: The contour tree for the analytic function shown in Figure 2.1. (a) Level sets at different function values (b) The contour tree tracks the evolution of connected components of the level sets. (c) The Split tree tracks connected components of the super-level sets (d) The Join tree tracks connectivity of sub-level sets

Chapter 3

Related Work

de Berg and van Kreveld [23] were among the first to develop an algorithm for computation of the contour tree and apply it to GIS and elevation queries. Using divide and conquer strategies they compute the contour tree of a scalar function defined over a two dimensional space in $O(n \log n)$ time, n being the number of triangles. van Kreveld et al. [4] developed an algorithm that maintained evolving level sets in order to compute the contour tree in $O(n \log n)$ for two-dimensional input, and $O(n^2)$ time for three-dimensional input.

Tarasov and Vyalyi[24] described an $O(n \log n)$ algorithm that computes the contour tree of a three-dimensional scalar function by performing two sweeps over the input in decreasing and increasing order of function value to identify the joins and splits of the level set components. The contour tree is computed by merging the results of the first two sweeps. Carr et al.[1] simplified this approach to develop an algorithm which is arguably the most elegant and widely used algorithm for computation of a contour tree. In two sweeps over the input, their algorithm computes a Join tree and a Split tree, which tracks the evolution of sub-level and super-level sets respectively. These two trees are then merged to obtain the contour tree. This algorithm has a running time of $O(v \log v + n\alpha(n))$, where v is the number of vertices in the input, n is the number of tetrahedras and α is the inverse Ackermann function.

Chiang et al.[25] proposed an output sensitive approach that first finds all component

critical points that represent the nodes in the contour tree, using the local neighborhood information. Monotone paths are constructed from these critical points, and later combined to give us the Join and Split tree containing only the component-critical points. Their algorithm has a running time of $O(t \log t + n)$, where t is the number of critical points of the input. Van Kreveld et al.[26] showed a $\Omega(t \log t)$ lower bound for the construction of contour trees. Since reading the input takes $O(n)$ time, the output sensitive algorithm is optimal.

Pasucci and Cole-McLaughlin [27] proposed the first known parallel algorithm that computes the contour tree of a piecewise trilinear function defined on a three dimensional structured mesh. It was also the first output sensitive algorithm for computing the contour tree. The sequential version for a 3D structured grid has a time complexity of $O(n + t \log n)$, where n is the no. of vertices and t is the no. of critical points. This contour-tree algorithm is based on the divide-and-conquer paradigm. At each step, the volume is recursively subdivided into two halves of roughly equal number of vertices, with the common boundary (the separator) having $O(n^{\frac{2}{3}})$ vertices and edges. In other words they compute the contour tree on each voxel and recursively merge them to eventually obtain the contour tree over the entire domain. They claim to have obtained near linear speedup but do not report the actual timings. Although their algorithm is well suited for coarse grained parallelism, computing the contour tree for each voxel individually may result in huge overheads.

Maadasamy et al [28] provide an output sensitive, work efficient parallel implementation which modifies the monotone paths algorithm [25] suitably to make it efficient for parallel architectures. They compute the monotone paths in parallel and arbitrary order rather than sequentially to compute the contour tree. Although the method scales well for small unstructured grids, there is a significant dip in the speedup for large structured grids as the number of available processors increase. Moreover the required memory to compute the contour tree is huge owing to the additional auxiliary structures needed to compute the Join and Split tree.

Another recent approach towards computing the join/Split tree in parallel, by Morozov et al.[29] proceeds by constructing the local Join tree or Split tree on each processor and merging them across sub domains, but keeping only a certain part of the tree relevant to the sub domain. Every merging is followed by a pruning of unimportant vertices, a process they call *sparsifying.* So eventually the merge tree is spread across all processors which in turn helps in parallel processing of queries on the tree.Their method is best suitable for distributed memory systems with the final Join and Split trees being distributed across nodes. The final computation of the contour tree and even the applications requiring a contour tree have to be significantly modified to work in a distributed memory paradigm.

Chapter 4

Algorithm

We now describe our parallel algorithm for the computing the contour tree. We assume that the input is available in the form of a structured grid with scalar values associated with each vertex. We further assume that the vertices have unique function values. This may be achieved via a simulated perturbation using the vertex index that imposes a total order on the vertices.

Major steps involved in our algorithm are as follows

1. Split the domain into sub domains of appropriate size and assign each sub domains to a different processor.

2. Identify the critical points in each sub-domain

3. Compute the *Join tree,* which tracks the connectivity of the sub-level sets, and the *Split tree,* which tracks the connectivity of the super-level sets, for each sub-domain.

4. Prune each Join and Split tree by removing nodes that do not represent a change in the number of connected components of the level set.

5. Stitch the trees across neighboring sub domains hierarchically to construct the Join and Split tree for the entire domain.

6. Merge the global Join and Split tree to give the global *contour tree.*

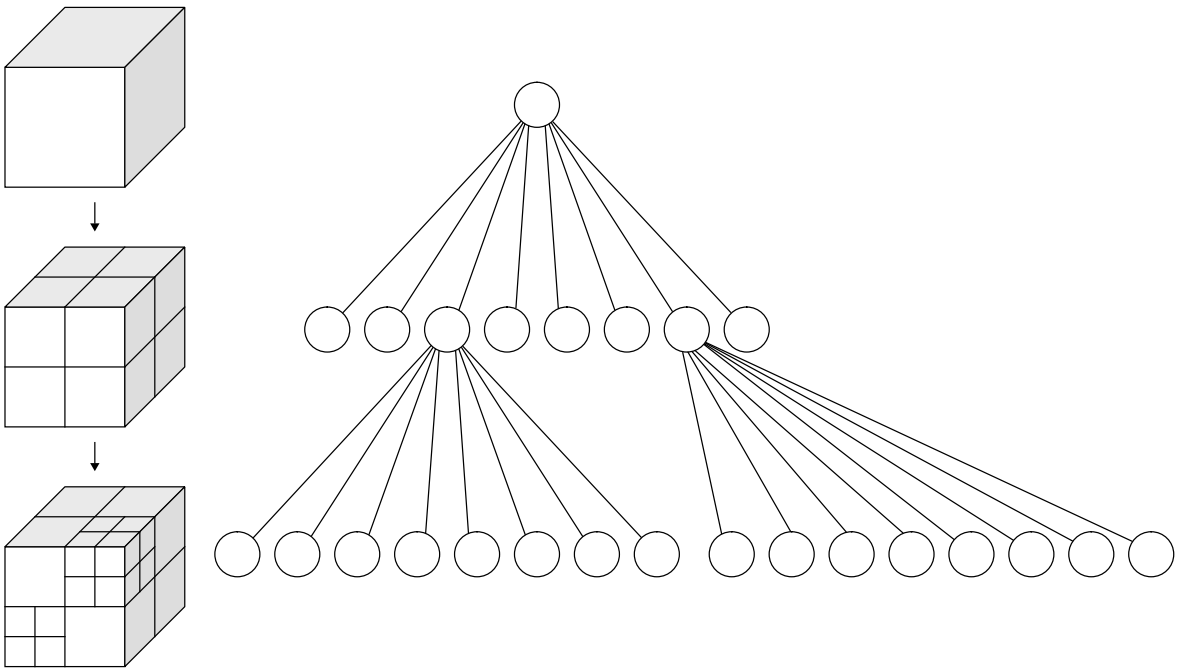


Figure 4.1: Octree based division of domain

4.1 Splitting into sub-domains

We decompose our domain into sub domains following an octree based division. Figure 4.1 illustrates this division.

In other words we halve the sub domain every iteration along the largest dimension of the block but sharing a common plane across both sub domains. For example if a domain D has dimensions (dim_x, dim_y, dim_z) with $dim_x \geq dim_y \geq dim_z$ we divide D into D_1 and D_2 with dimensions $(\lceil dim_x/2 \rceil, dim_y, dim_z)$ and $(dim_x - \lceil dim_x/2 \rceil + 1, dim_y, dim_z)$ essentially sharing the YZ plane having the x coordinate $\lceil dim_x/2 \rceil$. The D_is are further subdivided and by the end of i iterations we have 2^i sub domains which are processed in parallel by different processors.

4.2 Critical Point Identification

We classify the points as critical or regular depending upon the local neighborhood information. Edlesbrunner et al. [30] provide a combinatorial description of the critical

points of a piecewise linear function, which are always located at the vertices of the mesh. But to define an appropriate mesh for a structured grid additional points on the faces of the cells and within the body of a cell need to be taken care of. These *face saddles* and *body saddles* can be explicitly computed by the information about the face or the body as shown by [27]. For every face and body saddle we add it to the vertex list and connect it with the vertices on their respective face or body. We extend the edge set of the grid by inserting the above-mentioned edges.

The local neighborhood link of a vertex u in the original structured grid is described by the triangulation of its neighboring six vertices. The link of a body/face saddle v is the set of its neighboring vertices on the mesh along with the induced edges and triangles. Adjacent vertices with lower function values along with their induced simplices form the *lower link* while the same with higher function values together with their induced simplices form the *upper link.*

A vertex is *regular* if its upper-link and one lower-link have exactly one component or in other words, there is no change in the connectivity of the level set when we sweep through the function value of the vertex. All other vertices are called critical. A critical point is identified as *maximum* if its upper link is empty, and as a *minimum* if its lower link is empty. All other critical points are classified as *saddle.* Every upper link and lower link is represented by a point contained within the respective link.

Since we divide our domain into sub domains it is essential to classify as critical, those points on the boundary, which would otherwise have been classified as critical when the entire domain is processed as a whole. It is easy to see that a vertex is a saddle only when its lower (upper) link lies on a plane normal to one of the axes and its upper (lower) link consists of two isolated vertices on either side of it. Figure 4.3(a) illustrates such a point, along with the separating plane. A major consequence of the observation is that a saddle can have at most two upper-link and lower-link components.

A point that is critical in the global domain would not be classified as such in the sub domain using the classification scheme of the previous paragraph, only if the separating plane lies on the boundary. Such a boundary point is an extremum on the boundary.

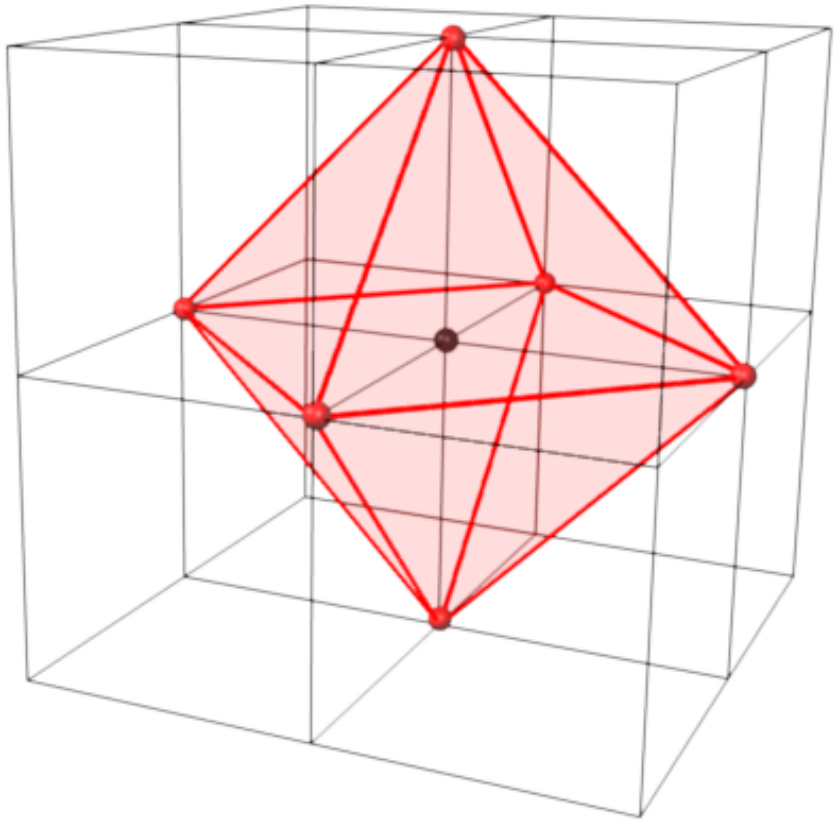
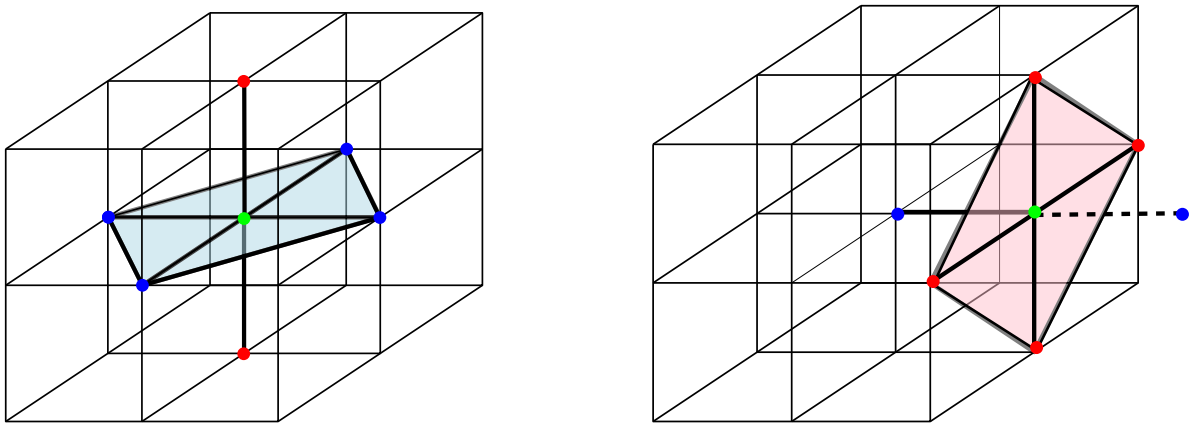


Figure 4.2: The triangulation of the adjacent vertices of u forms the link of the vertex u. Adapted from [28]



(a) (b)

Figure 4.3: Points classified as saddles in a sub domain. The cubes represent the sub domain (a) The green point represents an interior saddle with two upper link components(red points) and one lower link component (blue plane). The lower link component lies on a plane normal to one of the axes, with two upper link component on either side of it. (b) A point on the boundary with one upper link component (red plane) and one lower link component (blue point), within the sub domain. It is classified as critical because it is a boundary minimum

Hence, we add all the boundary extrema to the list of critical points. We note that some of them may be superfluous, as they might not appear in the final contour tree. Figure 4.3(b) represents such a boundary extremum.

4.3 Join and Split tree computation

For computing the Join and Split tree over each sub-domain we follow the output sensitive algorithm by Chiang et al. [25]. They compute the Join and Split tree by walking monotone paths from the critical points and combining them appropriately. We next discuss this computation in Algorithm 4.1

Before processing the sub domain the points are classified as described above. As a first step in the procedure `ConstructJoinTree()` the critical points are sorted in increasing order of their function values. For maintaining the connected components, a Union-Find data structure is used. The critical points form the ground set for the procedure and are processed in increasing order of their function values. The highest

Algorithm 4.1 ConstructJoinTree()

Input: Set of critical points $C=\cup c_i$
Input: Mesh M
Output: Join tree T_J

Initialize T_J as set of all critical points
$UF \leftarrow$ empty union find data structure
Sort C the set of critical points in ascending order of their function values
for i $\leftarrow$ 1 to $|C|$ **do**
  Mark vertex c_i as visited
  NewSet(c_i, UF)
  c_i.representative $\leftarrow c_i$
  for each Lower Link component L_j of c_i **do**
    Let R_j be the representative vertex of L_j
    Follow a descending path P from R_j along the mesh until a visited vertex v is hit
    Point every vertex on P to c_i
    Let c_r be the vertex v is pointing to
    $c'_r \leftarrow$ Find(c_r, UF)
    $c'_i \leftarrow$ Find(c_i, UF)
    if $c'_r == c'_i$ **then**
      continue
    end if
    $c_z \leftarrow c'_r$.representative
    Add edge (c_i,c_z) to T_J
    Union(c'_r,c'_i,UF)
    Let c_q be the parent of the newly formed set.
    c_q.representative $\leftarrow c_i$
  end for
end for
return Join tree T_J

valued vertex or the latest added vertex for a set is chosen as the representative of that particular set.

If v is a critical point, descending paths are constructed from each of the lower link components of v until an already visited vertex w is met. Every vertex on the paths are provided a pointer to v. Next the pointer from w is followed to find the critical point z which already had a descending path to w. Finally we unify the sets containing v and z and add an edge from v to the representative of the component containing z. Figure 4.4 illustrates this process. Essentially after v is processed all parts of the Join tree from

$-\infty$ to $f(v)$ are constructed. Therefore, after processing the final vertex we have the entire Join tree.

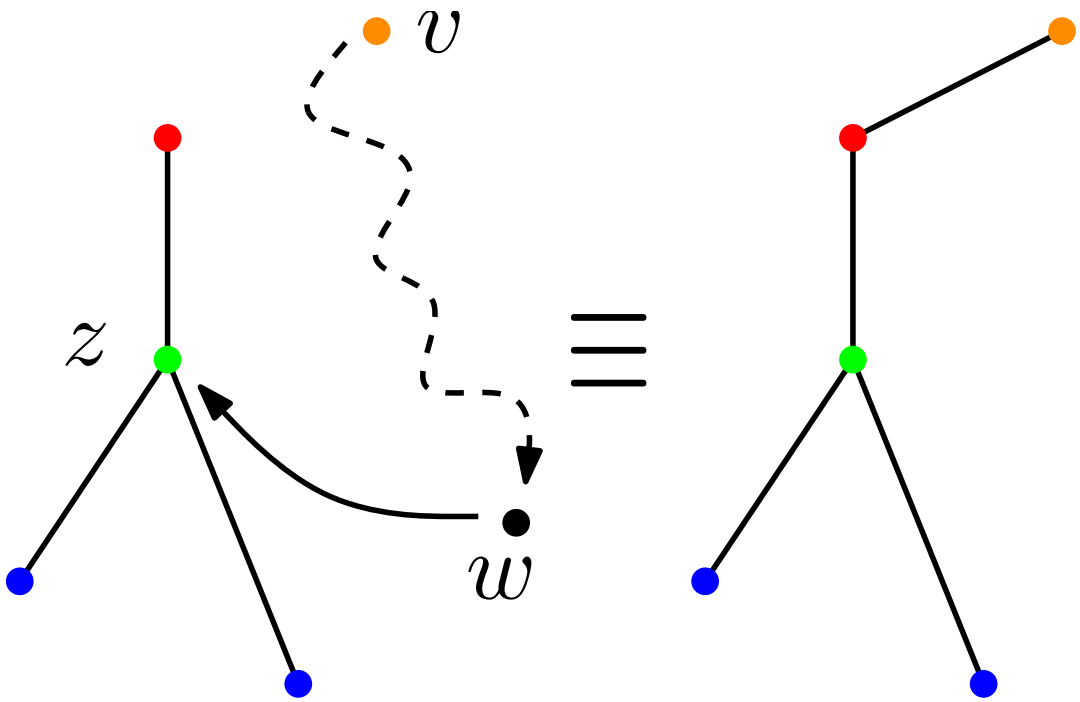


Figure 4.4: Illustrating the construction of the Join tree for a particular iteration when critical vertex v is processed.

The construction of the Split tree is analogous to the construction of the Join tree, and proceeds by processing the vertices in decreasing order of their function values and constructing ascending paths from the critical points. Another way of looking at Split tree is that it is the inverted Join tree of $-f$ defined over the same grid.

For optimal performance and low memory utilization, we store the Join and Split tree as parent arrays. For example if Jt is the array corresponding to Join tree then $Jt[i]$ gives us the parent of the i^{th} critical point.

Moreover, we also store the corresponding children array Cjt for faster access of the children nodes of a vertex, which comes in handy in the later stages of the algorithm. Since each critical point has at most two upper-link or lower-link components, the maximum number of children a vertex in the Join tree can have is, two. Hence we can store Cjt as a fixed size array with $Cjt[2i]$ and $Cjt[2i+1]$ representing the two children of the

i^{th} critical point in the Join tree. we maintain similar arrays for the Split tree as well.

For maintaining and updating the union find data structure we use both *path compression* and *union by rank*.

4.4 Pruning Join and Split Trees

In this step, we prune the Join tree and Split tree for each sub domain in parallel. The pruning removes nodes that do not represent a change in the number of connected components. If v is a degree-2 node in the Join tree then the number of connected components of the sub-level set does not change when it crosses $f(v)$. Similarly, a degree-2 node in the Split tree contributes no additional information regarding the number of super-level set components. Hence, a node that is neither a join vertex nor a split vertex can be safely pruned away. We note that such nodes might represent other topological changes such as a change in genus. However, our aim is to only capture the number of connected components and we remove them as shown in Algorithm 4.2.

We do not prune the Join and Split trees independently. In other words, we do not remove all degree-2 vertices from the Join and Split trees. This is because, we would require this information while merging the two trees to construct the contour tree. Pruning the tree on-the-fly, when it is being constructed, is difficult. Therefore, we schedule the pruning after constructing the Join and Split tree for each sub domain. We do preserve the boundary points because they are required for correct stitching of the trees across the sub domains.

The pruning step contributes to the huge memory savings achieved by our algorithm. In our experiments, we observe a significant number of degree-2 nodes classified as critical. We find that the size of the tree reduces by a factor of 5-10 depending upon the data set after the pruning step. If the domain is divided into d sub domains and processed using p processors, then the memory requirement of our algorithm is p/d times the maximum memory utilized by any algorithm that does not partition the domain. We choose d such that it is significantly larger than p and hence require only a fraction of the

maximum memory utilized by other algorithms. The memory required for subsequent steps of the algorithm further reduces due to the pruning.

Algorithm 4.2 PruneTrees()

Input: List of critical points C, Join Tree T_J and Split tree T_S
Output: Pruned trees t_j and t_s

for every vertex $c_i \in C$ **do**
  if c_i is a degree two node in both T_J and T_S, and is not on the boundary **then**
    Remove c_i from C
  end if
end for
{*Prune Join Tree*}
for every vertex $c_i \in C$ **do**
  if c_i is the root of T_J **then**
    continue
  end if
  $p_i = c_i.JoinParent$
  while $p_i \notin C$ **do**
    $p_i = p_i.JoinParent$
  end while
  Add edge (p_i, c_i) to t_j
end for
Delete T_J
{*Prune Split Tree*}
for every vertex $c_i \in C$ **do**
  if c_i is the root of T_S **then**
    continue
  end if
  $p_i = c_i.SplitParent$
  while $p_i \notin C$ **do**
    $p_i = p_i.SplitParent$
  end while
  Add edge (p_i, c_i) to t_s
end for
Delete T_S
return t_j and t_s

4.5 Stitching Join and Split Trees

Join and Split trees of sub domains that share a common boundary are stitched together in parallel. A union-find data structure is again used to maintain connectivity. However, only the portions of the trees affected by the boundary nodes are updated. The remaining portions of the trees are carried over from the sub domain onto the next iteration. Algorithm 4.3 describes the stitching procedure for the Join tree. Split trees are stitched together using a similar procedure.

Let t_{j1} and t_{j2} denote the Join tree of the adjoining sub domains D_1 and D_2. Let T_1 and T_2 be the set of nodes on t_{j1} and t_{j2}. The nodes in T_1 and T_2 are already sorted. We merge these sorted lists in linear time to obtain a sorted list of nodes from $T_1 \cup T_2$. Duplicate nodes are retained to avoid reorganizing the data structures. The duplicate nodes would appear next to each other in the sorted list. We insert an edge between these duplicate nodes essentially creating a new mesh M_{12} whose vertex set equals the nodes in $T_1 \cup T_2$ and whose edge sets are the union of the arc sets of t_{j1} and t_{j2} together with the newly inserted edges.

The Join tree of the scalar function restricted to $D_1 \cup D_2$ is computed as the Join tree of M_{12} by maintaining a union-find data structure UF. First, the nodes and arc sets of t_{j1} and t_{j2} are merged. The vertices of M_{12} are processed in sorted order. The first set is created in UF when the first boundary node is processed. Subsequently, a new set is created only when another boundary node is processed or when a child of the node being processed belongs to UF. Union operations are triggered in both cases. Note that the several nodes of t_{j1} and t_{j2} are not inserted into UF because they remain unaffected after stitching. We again use union by rank and path compression for the union find operations, although we observe little to no improvements in case of union by rank acceleration. We observe in our experiments that the time required for stitching is indeed roughly proportional to the number of boundary nodes on the sub domains.

Figure 4.5 illustrates the stitching procedure where v' and v'' are the duplicated boundary vertices across neighboring sub domains. The final sub figure gives the Join tree over the union of the two sub domains.

Algorithm 4.3 StitchJoinTrees(t_{j1}, t_{j2})

Input: Sorted list of nodes T_1 and T_2
Output: Join tree t_j

Initialize $t_j \leftarrow t_{j1} \cup t_{j2}$
$UF \leftarrow$ empty union find data structure
$T \leftarrow Merge(T_1, T_2)$
for i $\leftarrow$ 1 to $|T| - 1$ **do**
  if v_i and v_{i+1} are boundary duplicate points **then**
    NewSet(v_i,UF)
    NewSet(v_{i+1},UF)
    Union(v_i,v_{i+1},UF) making v_{i+1} as the head
    v_i.JoinParent $\leftarrow v_{i+1}$
    Add v_i to v_{i+1}.JoinChildrenList
  end if
  for each child c_j of v_i **do**
    if c_j is present in UF **then**
      if v_i is not present in UF **then**
        NewSet(v_i,UF)
      end if
      Delete c_j from v_i.JoinChildrenList
      $c' \leftarrow$ FIND(c_j,UF)
      if $v_i \neq c'$ **then**
        c'.JoinParent $\leftarrow v_i$
        Add c' to v_i.JoinChildrenList
        Union(c',v_i,UF) ensuring v_i as the head
      end if
    end if
  end for
end for
return Join tree t_j

We keep stitching the trees across adjoining sub domains hierarchically, traveling up the domain decomposition octree. Independent stitching processes are performed in parallel. But as we proceed with the iterations and move up the octree the number of parallel jobs keep decreasing. As such, the stitching process is a top heavy one and does not scale well with the number of processors. In the final iteration, we merge the trees across two halves of the global domain, and by the end of it we have the Join and Split tree for the entire domain.

Split trees can be similarly processed and stitched across the sub-domains. In fact, if

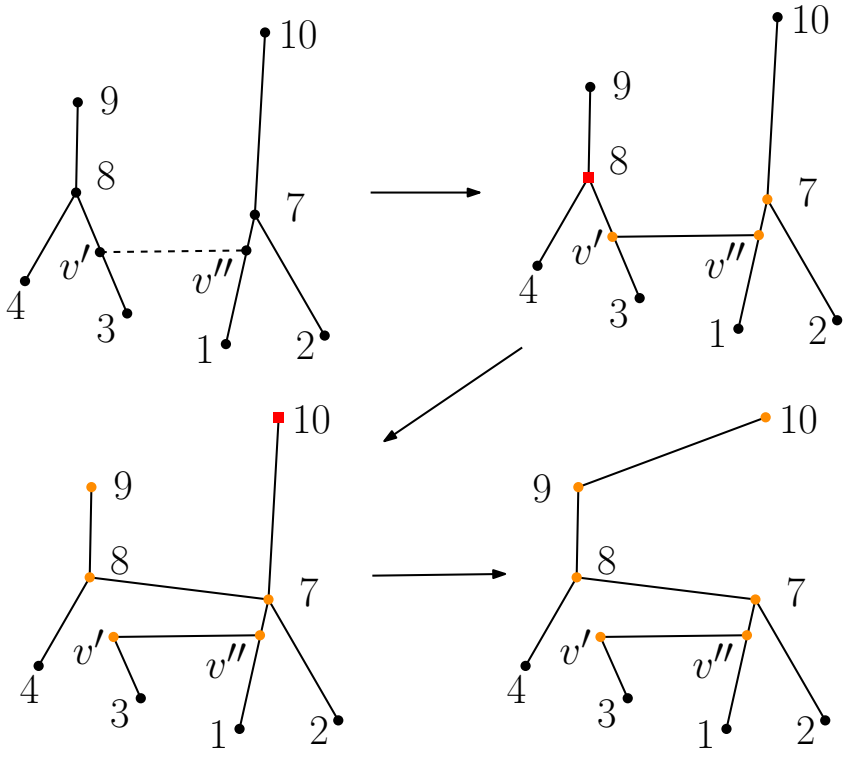


Figure 4.5: Stitching Join trees of neighboring sub domains:
v' and v'' are the duplicated boundary vertices across neighboring sub domains. v'' is initially made the parent of v'. The vertices are then processed in increasing order of their function values. The orange nodes represent the nodes present currently in the union-find data structure. The red node indicates the vertex currently being processsed.
In the second second sub figure, when vertex 8 is processed, it queries the representative of its child v' in the union find data structure. Since 7 is the representative of v' we add an edge (8,7) and remove (8,v') from the Join tree.
Next when it is the turn of the final vertex 10 to be processed it finds out that vertex 9 is the representative of its child 7. Hence (10,7) is removed and (10,9) is added to the Join tree.

there are available processors the stitching of the Split tree can be carried out in parallel with the stitching of the Join tree. We do not prune the duplicate vertices at the end of the step and defer it to the final stage just before computing the contour tree. In practice, the final pruning reduces the number of points in split and Join tree by only about 5%. It is expensive to scan the entire tree to find degree-2 nodes and therefore we avoid pruning the trees after every Stitch operation.

4.6 Merging Join and Split Trees

The final contour tree is constructed from the Join and Split tree using a procedure similar to that described by Carr et al.[1], one that is amenable to a parallel implementation. Algorithm 4.4 describes this procedure.

Each iteration of this procedure identifies an arc of the contour tree that is incident on a leaf, removes the arc from the Join and Split tree, and inserts it into the contour tree. The procedure terminates when all arcs of the Join and Split trees are processed. The running time of the sequential version is linear in the number of critical points. The set of growing nodes G is processed in parallel with removal of nodes being processed atomically.

The final contour tree is stored as an abstract graph specifically as a set of nodes and edges.

Algorithm 4.4 MergeTrees()

Input: Join tree T_j and Split tree T_s
Output: Contour tree T_c

```
1:  G = Set of leaves in T_j and T_s
2:  while G ≠ φ do
3:    if c_i is a leaf in T_j or T_s then
4:      Process c_i and Remove it from G
5:      T = tree in which c_i is a leaf
6:      while c_i ≠ T.root and c_i is not processed do
7:        n_i = c_i
8:        c_i = parent vertex of c_i in T
9:      end while
10:     Remove n_i from T and G
11:     Add arc(n_i,c_i) to T_c
12:     if c_i is either a leaf in T_j or T_s then
13:       Add c_i to G
14:     end if
15:   end if
16: end while
```

4.7 Analysis

We assume there are v vertices in the structured grid and t critical points in the complete domain, with d being the number of sub domains. By t_i, $i = 1$ to d, we denote the number of critical points present in the i^{th} sub domain. Let b_i represent the number of boundary nodes classified as critical in the i^{th} sub domain

Classifying the grid and finding the face and body saddles take $O(v)$ time as it takes constant amount of time for every vertex to find the number of upper link and lower link components. Chiang et al. [31] show it takes $O(v + t \log t)$ time for constructing the Join and Split tree where t is the number of critical points. Since each sub domain has a maximum of $t_i + b_i$ points, the computation of the Join and Split tree for each sub domain takes $O\left(v/d + (t_i + b_i)\log(t_i + b_i)\right)$ time. Pruning the Join and Split trees again takes time proportional to the number of critical points in each sub domain. While stitching sub domain D_i and D_j we perform at maximum $(t_i + t_j + b_i + b_j)$ unions and finds which can be implemented using union-by-rank and path compression in $(t_i + t_j + b_i + b_j)\,\alpha\,(t_i + t_j + b_i + b_j)$ where α is the inverse Ackermann function [32]. This is a very conservative estimate since the majority of the nodes remain unaffected and hence do not feature in the union find operations. The merging of the two sorted lists takes $O\left(t_i + t_j + b_i + b_j\right)$ time. The final cleanup and the merging of the Join and Split tree to form the final contour tree is proportional to the number of critical vertices.

With d sub domains there are $\log d$ number of of stitching iterations. At worst case each of the iterations, process $z = t + 3d^{\frac{1}{3}} \cdot v^{\frac{2}{3}}$ points and hence take $O(z\alpha(z))$ time. Hence the net complexity is $O(v + \sum_{i=1}^{d}(t_i + b_i)\log(t_i + b_i) + \log d \cdot z\alpha(z)) = O(v + z\log z + \log d \cdot z\alpha(z))$. Hence if d is much smaller than v and t, we can write the final sequential running time as $O\left(v + \left(t + v^{\frac{2}{3}}\right)\log\left(t + v^{\frac{2}{3}}\right)\right)$. In our experiments, we observe that in a number of data sets, t is more than 10% of the number of vertices, v, and hence is much more than $v^{\frac{2}{3}}$ for larger data sets. So the final complexity can be further simplified to $O\left(v + t\log t\right)$, which is the optimal running time for construction of the contour tree.

If we assume perfect load balance among all processors with number of available

processors p equal to the number of sub domains d, we have a parallel running time of $O(v/p+(z/p)\log(z/p)+z\alpha(z))$. The first two terms, representing the time complexity for constructing the trees for each sub domain, scale linearly. But $z\alpha(z)$ term, a consequence of the stitching operations is independent of the number of processors and hence scales poorly.

Chapter 5

Experimental Results

5.1 Experimental Setup

We evaluate our algorithm on shared memory system with 64 cores. For that purpose, we use an AMD opteron 6274 processor with each of its 64 core running at 2.2GHz, with 64 GB of RAM. We use the data sets from volvis [33] for our experiments. All of the data sets are in binary .raw format, storing the function value for each of the points in an 8-bit unsigned integer format, ordered in an X by Y by Z layout, i.e. x-coordinate changes rapidly when stepping through consecutive memory locations. After reading in the data sets, we store the function values in float format. Details of the data sets are presented in Appendix.

Firstly, we report run times for the sequential version of our algorithm where the entire data is processed by a single processor. Then we report run times for increasing number of processors and compare it with an existing parallel algorithm ParallelCT [28], which computes the contour tree in a shared memory system without partitioning the domain. We call our procedure DivCT.

5.2 Single Core environment

On a single core environment DivCT behaves exactly like the output sensitive algorithm by Chiang et al. [31], but applied to the modified mesh corresponding to the structured grid input. libtourtre [34] is a publicly available and widely used serial implementation of the Carr et al. sweep algorithm of computing contour tree. The comparison between libtourtre, ParallelCT and DivCT is shown in Table 5.1. Both the DivCT and ParallelCT are faster than libtourtre for structured grids. This is expected since both DivCT and ParallelCT are output sensitive. It is clearly evident that DivCT outperforms ParallelCT. This can be attributed to ParallelCT's extra computational tasks of constructing the auxiliary data structures.

Model	#Vertices	libtourtre	ParallelCT	DivCT
Aneurism	$256 \times 256 \times 256$	15.2	9.4	7.7
Bonsai	$256 \times 256 \times 256$	21.9	18.9	16.1
Foot	$256 \times 256 \times 256$	31.5	19.8	17.2

Table 5.1: Time taken (in seconds) for computing the contour tree on a single core. DivCT outperforms both libtourtre and ParallelCT.

5.3 Multi-core environment

For processing a data set on p number of processors we divide our domain into at least $8p$ sub domains. If the sub domains are still large and do not fit in memory, they are further partitioned into blocks. Ensuring a minimum of $8p$ sub domains results in a reasonable balance of load among the cores while computing the Join and Split trees. In practice, we observe that this step scales almost linearly with increasing number of processors with an additional, but small, expense of handling greater number of boundary vertices. We observe in our experiments that the total number of boundary vertices including the duplicate points that are misclassified as critical points is roughly equal to 5% of the final size of the Join and Split trees. Therefore, the increase in number of sub domains does not adversely affect the computation time. Note that the domain is not partitioned

for the serial execution, for data sets of size up to $1024 \times 1024 \times 1024$. For even larger data sets, for serial execution we partition the domain into 8 sub domains, as we do not have sufficient memory for processing entire data sets as huge as $2048 \times 2048 \times 2048$.

5.4 Speedup and Scaling

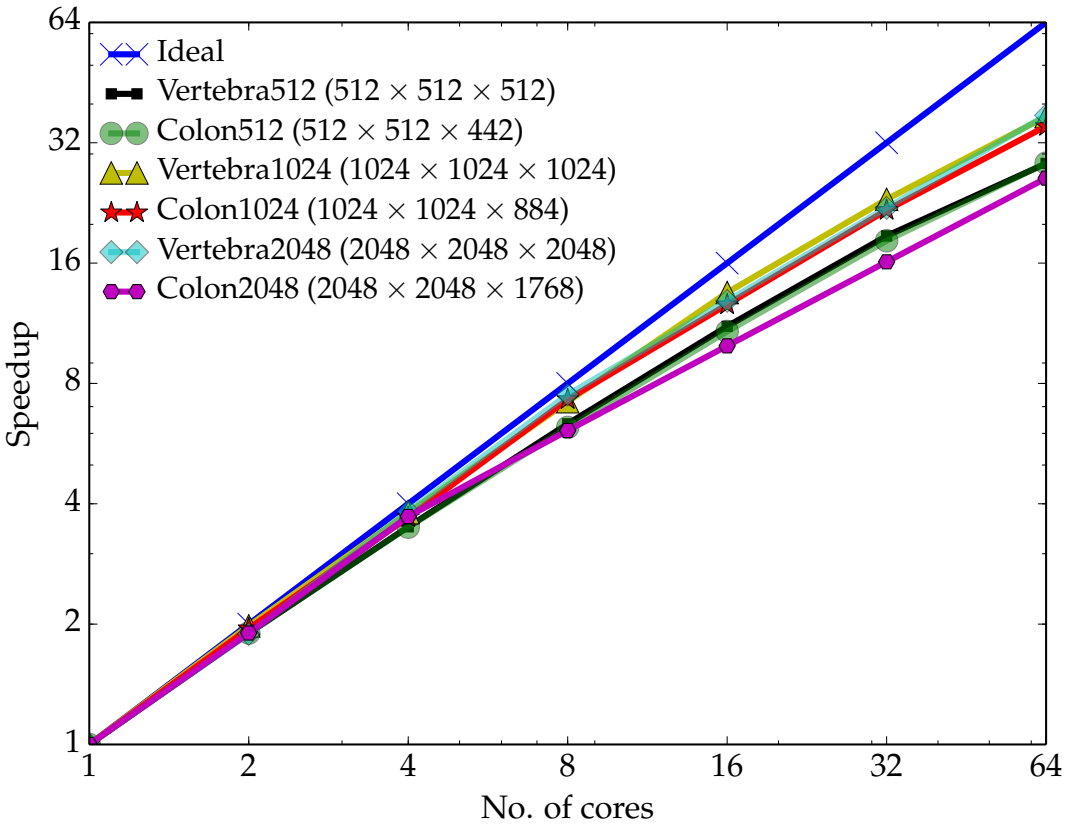


Figure 5.1: Speedup for large data sets with increasing number of cores, for the entire procedure of DivCT. It exhibits close to linear scaling behavior.

Figure 5.1 shows the scaling behavior of DivCT with respect to increasing number of processors on large data sets. The graph plots indicate that we achieve close to the ideal speedup shown in blue. As expected, the task of computation of the Join and Split trees for sub domains scales linearly and very close to the ideal speedup. This is primarily responsible for the overall near-linear speedup. Figure 5.2 shows the scaling result for

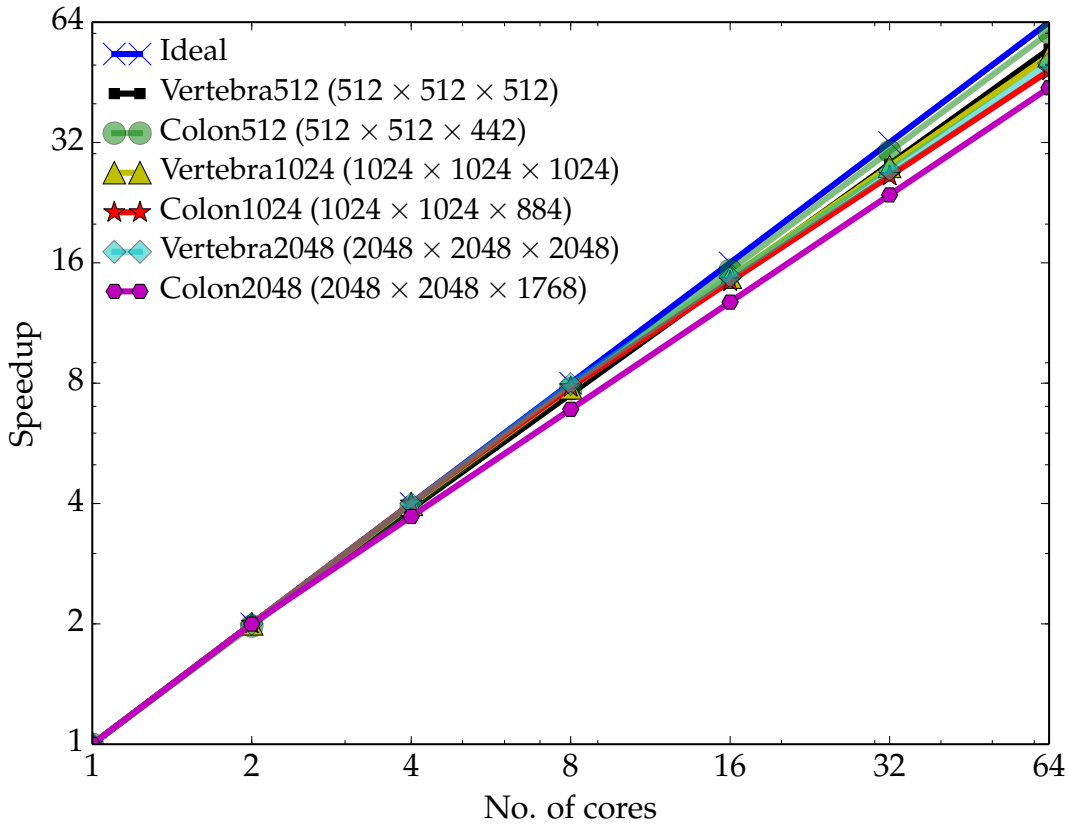


Figure 5.2: Speedup obtained for the computation of Join and Split trees of individual sub domains

this particular sub procedure. As evident, the scaling is linear and extremely close to the ideal speedup.

On the other hand, stitching the trees scales poorly. It is the primary contributor to the deviation from the ideal linear speedup in 5.1. For experiments run on 64 cores, the time taken for stitching together with the final merge to compute the contour tree is comparable to the time taken to compute the Join and Split trees for the all the sub domains.

We compare the performance of DivCT with ParallelCT over 64 cores in Table 5.2 and Table 5.3. We observe significant improvements both in terms of running time and speedup over ParallelCT. We observe a saturation in ParallelCT with increasing number of processors whereas DivCT exhibits good scaling.

Model	#Vertices	ParallelCT time (in seconds)		
		1 core	8 cores	64 cores
Vertebra	$512 \times 512 \times 512$	91.4	19.8 (4.6×)	8.8 (10.4×)
Colon	$512 \times 512 \times 442$	182.6	52.9 (3.5×)	15.4 (11.9×)
Vertebra1024	$1024 \times 1024 \times 1024$	-	-	-
Colon1024	$1024 \times 1024 \times 884$	-	-	-

Table 5.2: Time taken (in seconds) by ParallelCT for computing the contour tree in a many-core system. The speedup factor is shown within parenthesis. ParallelCT is unable to process larger data sets because it requires more memory than available.

Model	#Vertices	DivCt time (in seconds)		
		1 core	8 cores	64 cores
Vertebra	$512 \times 512 \times 512$	76.7	12.1 (6.3×)	2.7 (28.4×)
Colon	$512 \times 512 \times 442$	156.5	25.2 (6.2×)	5.5 (27.5×)
Vertebra1024	$1024 \times 1024 \times 1024$	532.5	74.0 (7.2×)	14.3 (37.2×)
Colon1024	$1024 \times 1024 \times 884$	1142.5	156.4 (7.3×)	32.5 (35.2×)
Vertebra2048	$2048 \times 2048 \times 2048$	6513.8	868.5 (7.5 ×)	173.3 (37.5 ×)
Colon2048	$2048 \times 2048 \times 1768$	12890.7	2113.2 (6.1 ×)	493.9 (26.1 ×)

Table 5.3: Time taken (in seconds) by the entire procedure of DivCT for computing the contour tree in a many-core system. The speedup factor is shown within parenthesis. DivCT exhibits better scaling and is faster in terms of total running time compared to ParallelCT, given in Table 5.2. It also processes larger data sets that ParallelCT cannot handle.

Table 5.4 shows the time taken and speedup obtained for the computation of Join and

Split trees restricted to individual sub domains. As observed in Figure 5.2 the speedup is extremely close to the ideal linear speedup.

Model	#Vertices	DivCt Time (in seconds)		
		1 core	8 cores	64 cores
Vertebra	$512 \times 512 \times 512$	76.7	10.2 (7.5×)	1.4 (54.8×)
Colon	$512 \times 512 \times 442$	156.5	20.2 (7.7×)	2.6 (60.2×)
Vertebra1024	$1024 \times 1024 \times 1024$	532.5	68.4 (7.8×)	10.1 (52.7×)
Colon1024	$1024 \times 1024 \times 884$	1142.5	146.1 (7.8×)	23.6 (48.4×)
Vertebra2048	$2048 \times 2048 \times 2048$	6404.7	804.4 (7.9 ×)	126.9 (50.5 ×)
Colon2048	$2048 \times 2048 \times 1768$	12491.2	1814.6 (6.9 ×)	284.8 (43.9 ×)

Table 5.4: Time taken (in seconds) for the computation of Join and Split trees of individual sub domains by DivCT in a many-core system with speedup values in bracket

5.5 Memory Efficiency

We also observe significant improvements in terms of memory consumption. For example ParallelCT requires approximately 12GB of memory to compute the contour tree for the Vertebra ($512 \times 512 \times 512$) data set. However DivCT requires only one-fifth as much because the data is partitioned into sub domains. In the case of larger data sizes, it is infeasible to use ParallelCT. For example, it requires more than 60GB of memory for a $1024 \times 1024 \times 1024$ data sets. DivCT consumes at most 11GB of memory for the same data set.

The maximum memory required by DivCT to construct the trees can be reduced by further subdividing the sub domains. In fact, the minimum available memory required by DivCT is comparable to the size of the final contour tree. The final contour tree can be computed for the Vertebra1024 ($1024 \times 1024 \times 1024$) data set using 2.5GB of memory on an 8-core machine by partitioning it into 512 sub domains of size $128 \times 128 \times 128$ each.

For data sets even larger in size, say a $2048 \times 2048 \times 2048$ containing about 8.6 billion points, we similarly divide the domain into 512 sub domains of size $256 \times 256 \times 256$. We compute the final contour tree in less than 14 minutes consuming roughly 10GB

of memory on 8 cores. With 64 cores, the computation time drops to approximately 3 minutes.

Chapter 6

Conclusions

We have presented a simple and memory efficient algorithm for parallel construction of the contour tree for a scalar function defined over a 3D structured grid in shared memory systems. We compute the the contour tree for extremely large data sets, that do not fit in memory, of size up to $2048 \times 2048 \times 2048$. The near linear speedup obtained over various data sets indicates that our implementation scales well with number of processors. We also report significant improvements in memory usage over existing shared memory based parallel algorithms for computing the contour tree.

It would be interesting to see if we can utilize GPUs or a CPU-GPU hybrid environment for faster computation of the contour tree. However, it is highly non trivial to design and implement such an algorithm in a GPU like environment to handle such large data sets.

One obvious way to improve upon our implementation is to parallelize the individual stitch operations themselves. However they might involve drastic changes in the data structures we use.

Appendix

Details of data sets used from volvis.org [33] for our experiments

- Aneurism ($256 \times 256 \times 256$): Rotational C-arm x-ray scan of the arteries of the right half of a human head. A contrast agent was injected into the blood and an aneurism is present.
- Bonsai ($256 \times 256 \times 256$): CT scan of a bonsai tree
- Foot ($256 \times 256 \times 256$): Rotational C-arm x-ray scan of a human foot. Tissue and bone are present in the data set.
- Vertebra ($512 \times 512 \times 512$): Rotational angiography scan of a head with an aneurysm.
- Colon ($512 \times 512 \times 442$): CT scan of a Colon phantom with several different objects and five pedunculated large polyps in the central object.
- Vertebra1024 ($1024 \times 1024 \times 1024$) and Vertebra2048 ($2048 \times 2048 \times 2048$) are up-sampled versions of Vertebra data set. Interpolation is done using a trilinear interpolant.
- Colon1024 ($1024 \times 1024 \times 884$) and Colon2048 ($2048 \times 2048 \times 1768$) are up-sampled versions of Colon data set. Interpolation is done using a trilinear interpolant.

Bibliography

[1] H. Carr, J. Snoeyink, and U. Axen, "Computing contour trees in all dimensions," *Comput. Geom. Theory Appl.*, vol. 24, no. 2, pp. 75–94, 2003.

[2] C. L. Bajaj, V. Pascucci, and D. R. Schikore, "The contour spectrum," in *Proc. IEEE Conf. Visualization*, 1997, pp. 167–173.

[3] H. Carr and J. Snoeyink, "Path seeds and flexible isosurfaces using topology for exploratory visualization," in *Proceedings of the symposium on Data visualisation 2003*, ser. VISSYM '03. Eurographics Association, 2003, pp. 49–58.

[4] M. van Kreveld, R. van Oostrum, C. Bajaj, V. Pascucci, and D. R. Schikore, "Contour trees and small seed sets for isosurface traversal," in *Proc. Symp. Comput. Geom.*, 1997, pp. 212–220.

[5] S. Takahashi, "Algorithms for extracting surface topology from digital elevation models," in *Topological Data Structures for Surfaces: An Introduction to Geographical Information Science*. John Wiley & Sons, 2006, pp. 31–51.

[6] D. Demir, K. Beketayev, G. H. Weber, P.-T. Bremer, V. Pascucci, and B. Hamann, "Topology exploration with hierarchical landscapes," in *Proceedings of the Workshop at SIGGRAPH Asia*. ACM, 2012, pp. 147–154.

[7] F. Hétroy and D. Attali, "Topological quadrangulations of closed triangulated surfaces using the Reeb graph," *Graph. Models*, vol. 65, no. 1-3, pp. 131–148, 2003.

[8] M. Mortara and G. Patané, "Affine-invariant skeleton of 3d shapes," in *SMI '02: Proceedings of the Shape Modeling International 2002 (SMI'02)*, 2002, p. 245.

[9] E. Zhang, K. Mischaikow, and G. Turk, "Feature-based surface parameterization and texture mapping," *ACM Trans. Graph.*, vol. 24, no. 1, pp. 1–27, 2005.

[10] J. Cox, D. B. Karron, and N. Ferdous, "Topological zone organization of scalar volume data," *J. Math. Imaging Vis.*, vol. 18, pp. 95–117, March 2003.

[11] S. Takahashi, I. Fujishiro, and Y. Takeshima, "Interval volume decomposer: a topological approach to volume traversal," in *Proc. SPIE*, 2005, pp. 103–114.

[12] I. Fujishiro, Y. Takeshima, T. Azuma, and S. Takahashi, "Volume data mining using 3d field topology analysis," *IEEE Computer Graphics and Applications*, vol. 20, pp. 46–51, 2000.

[13] S. Takahashi, Y. Takeshima, and I. Fujishiro, "Topological volume skeletonization and its application to transfer function design," *Graphical Models*, vol. 66, no. 1, pp. 24–49, 2004.

[14] G. H. Weber, S. E. Dillard, H. Carr, V. Pascucci, and B. Hamann, "Topology-controlled volume rendering," *IEEE Trans. Vis. Comput. Graph.*, vol. 13, no. 2, pp. 330–341, 2007.

[15] J. Zhou and M. Takatsuka, "Automatic transfer function generation using contour tree controlled residue flow model and color harmonics," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 6, pp. 1481–1488, 2009.

[16] W. Harvey and Y. Wang, "Topological landscape ensembles for visualization of scalar-valued functions," *Computer Graphics Forum*, vol. 29, pp. 993–1002, 2010.

[17] P. Oesterling, C. Heine, H. Jänicke, G. Scheuermann, and G. Heyer, "Visualization of high dimensional point clouds using their density distribution's topology," *IEEE Transactions on Visualization and Computer Graphics*, vol. 99, no. PrePrints, 2011.

[18] D. M. Thomas and V. Natarajan, "Symmetry in scalar field topology," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 17, no. 12, pp. 2035–2044, 2011.

[19] H. Edelsbrunner and J. Harer, *Computational Topology: An Introduction.* Amer. Math. Soc., Providence, Rhode Island, 2009.

[20] A. Hatcher, *Algebraic Topology.* New York: Cambridge U. Press, 2002.

[21] Y. Matsumoto, *An Introduction to Morse Theory.* Amer. Math. Soc., 2002, translated from Japanese by K. Hudson and M. Saito.

[22] J. Milnor, *Morse Theory.* New Jersey: Princeton Univ. Press, 1963.

[23] M. de Berg and M. J. van Kreveld, "Trekking in the alps without freezing or getting tired," *Algorithmica*, vol. 18, no. 3, pp. 306–323, 1997.

[24] S. P. Tarasov and M. N. Vyalyi, "Construction of contour trees in 3d in o(n log n) steps," in *Proceedings of the fourteenth annual symposium on Computational geometry*, ser. SCG '98. New York, NY, USA: ACM, 1998, pp. 68–75.

[25] Y.-J. Chiang, T. Lenz, X. Lu, and G. Rote, "Simple and optimal output-sensitive construction of contour trees using monotone paths," *Comput. Geom. Theory Appl.*, vol. 30, no. 2, pp. 165–195, 2005.

[26] M. van Kreveld, R. van Oostrum, C. Bajaj, V. Pascucci, and D. R. Schikore, "Contour trees and small seed sets for isosurface traversal," Department of Computer Science, Utrecht University, Tech. Rep. UU-CS-1998-25, 1998.

[27] V. Pascucci and K. Cole-McLaughlin, "Parallel computation of the topology of level sets," *Algorithmica*, vol. 38, no. 1, pp. 249–268, 2003.

[28] S. Maadasamy, H. Doraiswamy, and V. Natarajan, "A hybrid parallel algorithm for computing and tracking level set topology," in *High Performance Computing (HiPC), 2012 19th International Conference on.* IEEE, 2012, pp. 1–10.

[29] D. Morozov and G. Weber, "Distributed merge trees," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2013, pp. 93–102.

[30] H. Edelsbrunner, J. Harer, V. Natarajan, and V. Pascucci, "Morse-Smale complexes for piecewise linear 3-manifolds," in *Proc. Symp. Comput. Geom.*, 2003, pp. 361–370.

[31] Y.-J. Chiang, T. Lenz, X. Lu, and G. Rote, "Simple and optimal output-sensitive construction of contour trees using monotone paths," *Computational Geometry*, vol. 30, no. 2, pp. 165–195, 2005.

[32] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. MIT Press, 2001.

[33] D. Bartz, "Volren and volvis homepage," *URL: http://www. volvis. org*, 2005.

[34] libtourtre: A contour tree library. [Online]. Available: http://graphics.cs.ucdavis.edu/~sdillard/libtourtre/doc/html/