Title:
A Hybrid Solution to Calculating Augmented Join Trees of 2D Scalar Fields in Parallel

Authors:
Paul Rosen, prosen@usf.edu, University of South Florida
Junyi Tu, junyi@mail.usf.edu, University of South Florida
Les A. Piegl, lespiegl@mail.usf.edu, University of South Florida

Keywords:
Data Analysis, Computational Topology, Scalar Field

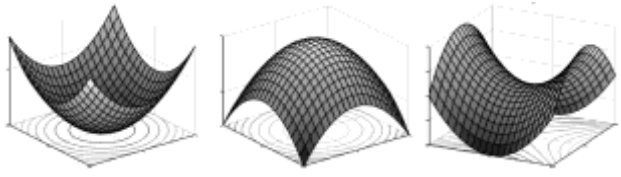
DOI: 10.14733/cadconfP.2017.xxx-yyy

Introduction:

In CAD applications, scalar fields are used to describe a variety of details from photographs, to laser scans, to x-ray, CT or MRI of machine parts. These scalar fields are invaluable for a variety of tasks, such as fatigue detection in parts. However, analyzing scalar fields can be quite challenging due to their size, complexity, and the need to understand both local details and global context. Join trees are the key data structure used in the computation of merge trees, split trees, and contour trees [1]. By capturing geometric properties, including local minima, local maxima, and saddle points (Fig. 1), these trees are useful in the evaluation and simplification (i.e. denoising) of scalar field data. However, computing these trees is expensive, and their incremental construction makes parallel computation nontrivial.

In its naïve implementation, the algorithm to compute join trees seems efficient. It has an $\mathcal{O}(n \lg n)$ sort phase and $\mathcal{O}(n + k)$ computation phase, where n is the number of elements in the scalar field and k is the aggregate cost of the find operation of a disjoint-set data structure. However, this algorithm has three practical challenges. First, as the dimensions of the field are doubled, the number of elements, n, in 2D fields grow quadratically. Secondly, the compute time per element in the computation phase is very high. Third, the computation phase requires ordered incremental construction, making it challenging to parallelize.

Three strategies have been proposed to parallelize join tree calculations: pruning [2], spatial-domain parallelization [3], and value-domain parallelization [4]. Each of these approaches have pros and cons, and up until now, these strategies have not been effectively integrated. In this paper, we have combined these strategies in an OpenCL join tree implementation. The implementation results in an $\mathcal{O}(n + k)$ pruning phase, an $\mathcal{O}(n)$ critical point extraction phase, an $\mathcal{O}(c \lg c)$ sorting phase, and $\mathcal{O}(c)$ propagation phase, where c is the number of critical points. What's more, these phases are designed to be parallelized such that they require at worst $\mathcal{O}(k)$, $\mathcal{O}(1)$, $\mathcal{O}(\lg c)$, and $\mathcal{O}(\lg c)$ parallel iterations, respectively. The result is a significant speedup, making computation of trees on large data practical on even modest commodity hardware.



(a) (b) (c)

Fig. 1: Functions representing (a) local minimum, (b) local maximum, and (c) a saddle point.

1	7	12	13
9	2	6	5
14	8	4	3
15	16	11	10

Fig. 2: Example 2D scalar field.

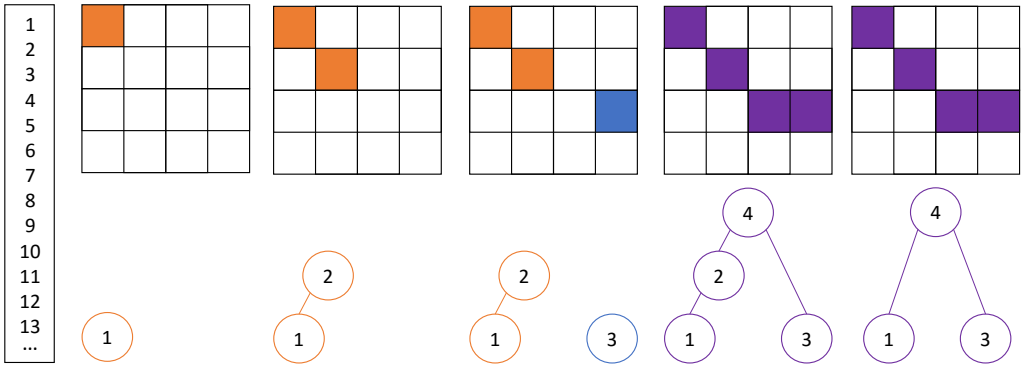


Fig. 3: Description of the conventional Augmented Join Tree algorithm. (a) Scalar field values are first sorted. Then, the points are added to the tree one-by-one. (b)(d) If no neighbors are in the tree, a leaf is created. If any neighbors are in subtrees, the node is connected to the top of the subtree. (c) If connected to one subtree, the subtree is just lengthened. (e) If connected to more than one subtree a saddle point is created. (f) Finally, only leaves and saddles are retained.

Conventional Join Tree Construction:

The conventional join tree construction process [1] starts with a scalar field (Fig. 2). The elements of the field are first sorted (Fig. 3 (a)) in ascending order for a merge tree or descending order for a split tree.

The elements are then processed one-by-one. The top element of the list is selected. A tree node is created for that element ((Fig. 3 (b) bottom) and a color assigned (Fig. 3 (b) top). Next, the neighborhood of 8 surrounding elements is searched. If none has been assigned a color (Fig. 3 (b) and (d)), the operation is complete. If one (Fig. 3 (c)) or more (Fig. 3 (e)) neighbors has already been assigned a color, those neighbor subtrees are connected to current tree node as children, and all nodes in the subtree are assigned the same color (Fig. 3 (e) top).

At this point, a Join Tree has been formed. An Augmented Join Tree is formed by removing non-critical point nodes from the tree. This is done by checking each child node in the tree. If the child only has one child of its own, then that point is not critical and can be skipped. In Fig. 3 (e) bottom, the node #4 has children #2 and #3. Node #2 has only one child, node #1, while node #3 has zero children. Having a single child means node #2 is not critical, and thus it can be removed. It is removed by connecting node #4 to node #1 (Fig. 3 (f) bottom).

From an implementation standpoint, the sorting can be handled by any $\mathcal{O}(n \lg n)$ sorting algorithm. The coloring of the nodes is made efficient using the disjoint-set data structure. Other operations are constant time per element. Unfortunately, the strictly-ordered bottom up construction of the tree means that each operation relies upon the results of the prior operation making parallelization challenging.

Methods:

Due to the complicated bottom up construction, efficient parallelization requires deconstructing and reordering the operations of the Augmented Join Tree algorithm. The first two phases of the new implementation are pruning and critical point extraction phases, which uses a spatial-domain decomposition to exclude many of the non-saddle point elements from further computation. In the third phase, the saddles must be sorted. Finally, the critical points are connected by using a value-range decomposition, building subtrees in parallel and propagating their join information globally.

Phase 1: Pruning

The pruning phase has two main objectives. The first objective is to eliminate as many non-critical points as possible from further computation by using a water shedding approach. The second is to perform a spatial-domain decomposition of the data, by taking the 2D scalar field and splitting it into subfields that can be processed in parallel.

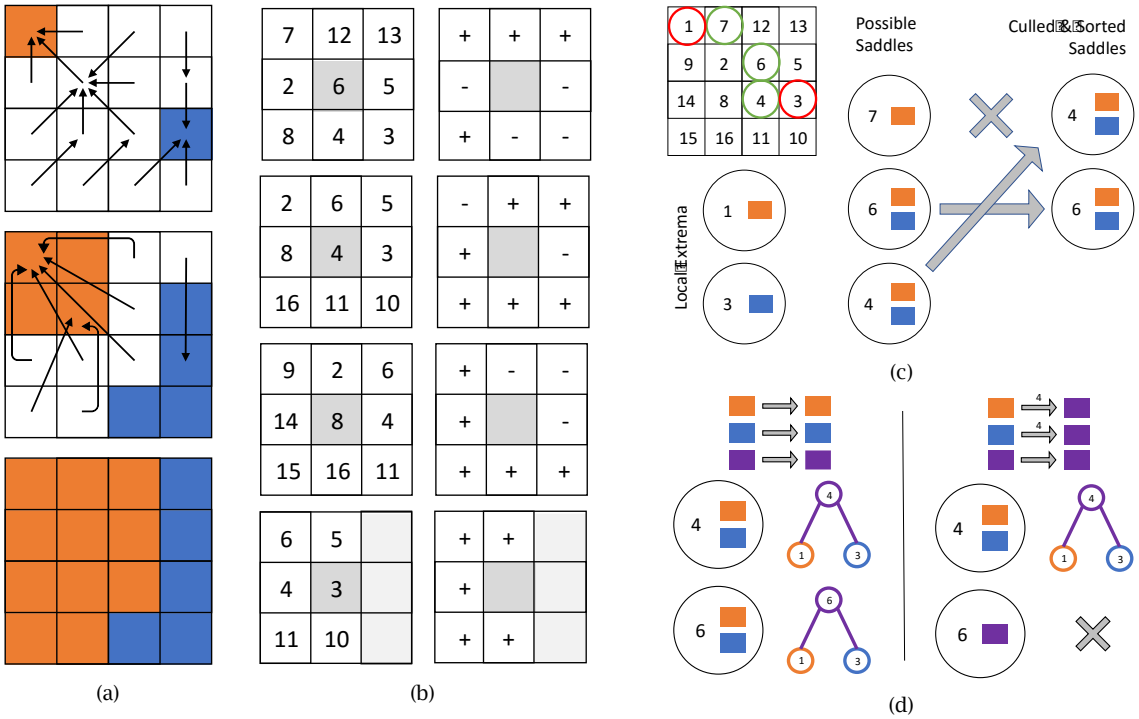


Fig 4: The four phase of our parallel implementation include (a) spatial-domain decomposition and pruning, (b) potential critical point extraction, (c) potential saddle point sorting, and (d) value-domain decomposition of saddle points and propagation of joins.

The water shedding approach is described in Fig. 4 (a) using the scalar field from Fig. 2. The first step is to point each element towards its largest (or smallest, depending upon merge or split tree) neighbor. If an element is larger than all its neighbors (i.e. a local maxima), it points to itself and is assigned a color. Next, one step of the find algorithm in a disjoint-set is taken. Essentially each element is updated to the pointer of its pointer. This process is repeated until the pointer reaches a colored element, at which point, the element receives that color.

Spatial-domain decomposition is accomplished by dividing the scalar field into 2D blocks. To complete the processing, neighboring blocks of elements only need to share boundary information. In other words, all elements are computed up to the boundary of their block, all boundaries are synchronized, and then element processing is finalized.

Phase 2: Potential Critical Point Extraction

The Augmented Join Tree will only contain critical points, so extracting potential critical points early in the process will save computation time. Local minima, maxima, and possible saddle points can be identified by looking at the value of an element relative to its neighbors.

Fig. 1 shows functions which have a local minimum, local maximum, and a saddle point, respectively. A simple observation helps us understand how to detect these 3 cases. For the minimum and maximum, notice that all regions surrounding the critical point are higher or lower, respectively. So, if the value of an element is smaller than all its neighbors, it is a local minimum. If the value of an element is larger than all its neighbors, it is a local maximum. The saddle point is a little more complicated to understand. Notice that around the saddle point, the function value goes up in two directions and down in two other directions. Therefore, if the neighbors of an element are larger in two or more disjoint directions and smaller in two or more disjoint direction, then the point may be a saddle.

This criterion does not guarantee a saddle point because of interpolation error. However, it can be used to exclude non-saddle points.

Fig. 4 (b) shows four examples. In the first two examples, elements #6 and #4 are each surrounded by two disjoint positive and negative directions. This indicates that these points may be saddles. For element #8, only one disjoint positive and negative direction exist. This point can be excluded as non-critical. Finally, for element #3 all neighbors are larger indicating a local minimum.

Phase 3: Saddle Sorting

Join trees need be built bottom-to-top. At this point in the processing, the extracted coloring information and extracted saddle points (not the minima or maxima) come into play.

After the critical points are extracted, the critical points are colored by looking at the color of all neighboring elements. In Fig 4 (c), the possible saddle #4, #6, and #7 are extracted. They are colored with their neighbors, with #4 and #6 being colored orange and blue, and #7 being colored only orange. This coloring information identifies which extrema a saddle point potentially connects. Therefore, #4 and #6 possible connect the orange and blue extrema, #1 and #3. However, #7 only connects to orange, extrema #1. This means that #7 is not a true saddle point.

Once the coloring is complete, the remaining saddles are sorted by their values.

Phase 4: Subtree Building and Join Propagation

The final phase of processing builds the tree by first performing a value-domain decomposition, which is used to build subtrees and propagate joins. The value-domain decomposition divides the sorted list of critical points into groups, which are each processed in parallel.

Building the subtree and propagating joins is a 3-step process. First, the color of nodes is updated with the global recoloring information. Second, subtrees are built using their color information as a guide. Third, the global merge information is updated based upon the new subtrees. This process is repeated until no additional modifications to color occurs.

Fig 4 (d) shows this process. On the left, nodes #4 and #6 are value-domain decomposed into 2 processing groups with 1 node each. Each node is updated with the global join information, which is initially empty (Fig 4 (d) top left). The two subtrees are built and the global join information updated (Fig 4 (d) top right). In the second pass (Fig 4 (d) right), each group is updated with the new coloring information. For node #4 no changes occur. For node #6, it is only colored purple and is therefore excluded from further computation. At that point, the processing would stop.

OpenCL Implementation:

We have implemented the described methods using OpenCL for fast flexible cross-platform interoperability. For phases 1 and 2, each element of the scalar field receives its own thread. The spatial-domain decompositions are square and as large as the supported thread block size of the hardware. For phase 3, each potential saddle point receives its own thread. To sort points in parallel, we used a hybrid of histogram sorting for a rough global ordering and bitonic sorting for precise ordering. For phase 4, each potential saddle point receives its own thread, with the hardware thread block size defining the granularity of the value-domain decomposition.

Experiments:

We tested our implementation by comparing it to an optimized C++ implementation of the conventional approach. We used this conventional implementation to compare the performance and check the correctness of the output tree from our OpenCL approach.

To test our approach, we extract the split tree from randomly generated fields. For each of 13 levels of resolution (32x32 up to 2048x2048), we record the time for 10 different fields (130 tests). In addition, we analyze those random fields under 7 different levels of smoothing for a total of 910 tests. Random fields have high critical point density, while smoothed fields do not. We report the results from an early 2015 MacBook Pro with an Intel 2.7GHz i5 and an Intel Iris 6100 GPU and a Linux workstation with an Intel 3.4GHz i7 and NVIDIA Tesla K40 Accelerator.

(a)

(b)

(c)

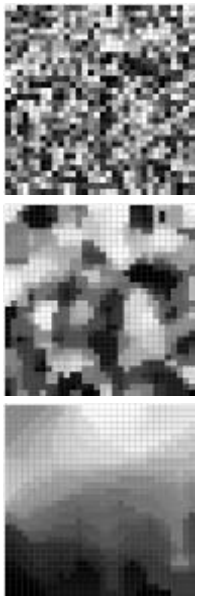
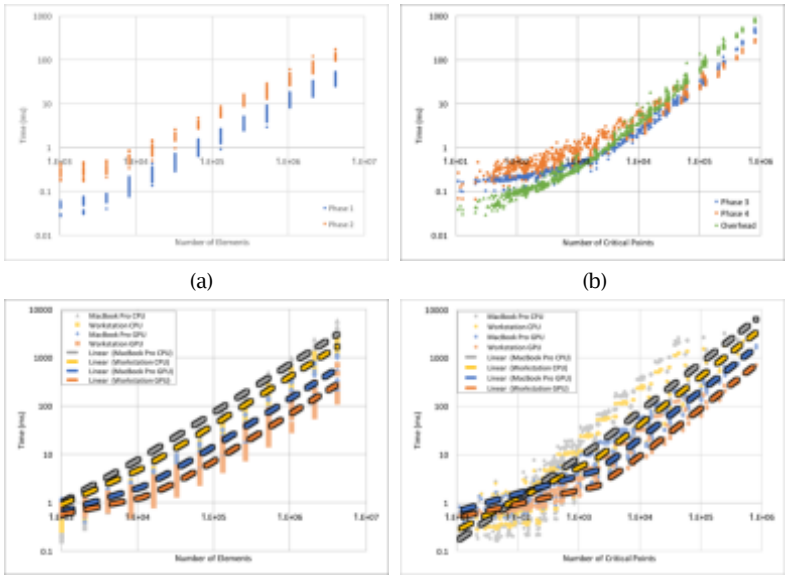


Fig. 5: Example 32x32 scalar fields: (a) random noise input, (b) after 2 smoothing iterations, and (c) after 4 smoothing iterations.



(a) (b) (c) (d)

Fig. 6: Log-log charts of the performance of our algorithm. (a) The processing time of Phase 1 and 2 is highly linear against the number of elements. (b) The processing time of Phase 3, 4, and overhead is highly linear against the number of critical points. Performance comparison of (c) the number of elements against time and (d) the number of critical points against time. For both hardware configurations, the OpenCL implementation shows approximately 1 order of magnitude improvement over the CPU counterpart.

Fig. 5 (a) shows an example 32x32 noisy scalar field. This scalar field has 206 critical points, making the tree difficult to display. The scalar field after 2 and 4 smoothing iterations can be seen in Fig. 5 (b) and 5 (c), respectively. These have 98 and 46 critical points, respectively.

Fig. 6 (top) shows log-log charts highlighting the performance of various phases of our approach. Fig. 6(a) shows that in the average case, Phases 1 and 2 grow linearly with respect to the *number of elements* in the field (R^2=0.96 and R^2=0.974, respectively). Similarly, Fig. 6(b) shows that for the average case, Phases 3, 4, and OpenCL overhead (data transfer, etc.) grows linearly with respect to the number of *critical points* in the field (R^2=0.98, R^2=0.992, and R^2=0.992, respectively).

Fig. 6 (bottom) uses log-log charts to compare the performance of our approach to the CPU implementation. Fig 6 (c) shows the computational time against the number of elements, while Fig 6 (d) shows the computational time against the number of critical points. The average time performance for both our algorithm and the conventional implementation is approximately linear. Our approach has the advantage of being highly parallel in nature. For both hardware configurations, our OpenCL implementation beat the CPU implementation by slightly less than 1 order of magnitude.

References:

[1] Carr, H.; Snoeyink, J., and Axen, U.: Computing contour trees in all dimensions, Computational Geometry 24.2, 2003, 75-94.

[2] Carr, H.; Weber, G.; Sewell, C.; and Ahrens, J.: Parallel Peak Pruning for Scalable SMP Contour Tree Computation, In 6th IEEE Symposium on Large Data Analysis and Visualization, 2016.

[3] Morozov, D.; Weber, G.: Distributed contour trees, In Topological Methods in Data Analysis and Visualization III, pp. 89-102. Springer International Publishing, 2014. https://doi.org/10.1007/978-3-319-04099-8_6

[4] Gueunet, C.; Fortin, P.; Jomier, J.; Tierny, J.: Contour Forests: Fast Multi-threaded Augmented Contour Trees, IEEE Symposium on Large Data Analysis and Visualization, 2016.