www.cadanda.com
Computer-Aided Design AND APPLICATIONS™

A Hybrid Solution to Parallel Calculation of Augmented Join Trees of Scalar Fields in Any Dimension

Paul Rosen[0000-0002-0873-9518][1], Junyi Tu[2] and Les A. Piegl[0000-0003-0629-8496][3]

[1] University of South Florida, prosen@usf.edu
[2] University of South Florida, junyi@mail.usf.edu
[3] University of South Florida, lespiegl@mail.usf.edu

ABSTRACT

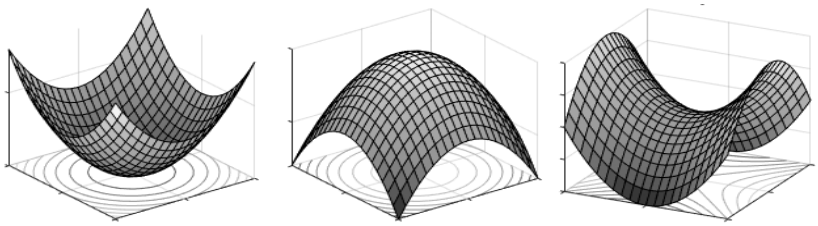
Scalar fields are used to describe a variety of details from photographs, to laser scans, to x-ray, CT or MRI scans of machine parts and are invaluable for a variety of tasks, such as fatigue detection in parts. Analyzing scalar fields can be quite challenging due to their size, complexity, and the need to understand both local and global details in context. Join trees are a data structure used to capture the geometric properties of scalar fields, including local minima, local maxima, and saddle points. Unfortunately, computing these trees is expensive, and their incremental construction makes parallel computation nontrivial. We introduce an approach that combines three strategies, pruning, spatial-domain parallelization, and value-domain parallelization, to parallelize join tree construction using OpenCL. The resulting implementation show a significant speedup, making computation of trees on large data practical on even modest commodity hardware.

Keywords: data analysis, computational topology, scalar field

1 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. Augmented join trees are the key data structure used in the computation of merge trees, split trees, and contour trees [2]. By capturing geometric properties, including local minima, local maxima, and saddle points (Fig. 1), these trees are useful in the evaluation and simplification of scalar field data. This is useful for tasks such as hierarchical visualization [1,3], segmentation [5,9], or tracing structures [10] in 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 augmented join trees seems efficient. No matter the dimension of the data, 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 dimension



(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.

of the field is doubled, the number of elements, n, grows quadratically in 2D fields and cubically in 3D fields. Secondly, although asymptotically small, the actual compute time per element in the computation phase is very high. Third, the computation phase requires loosely ordered incremental construction, making it a challenge to parallelize.

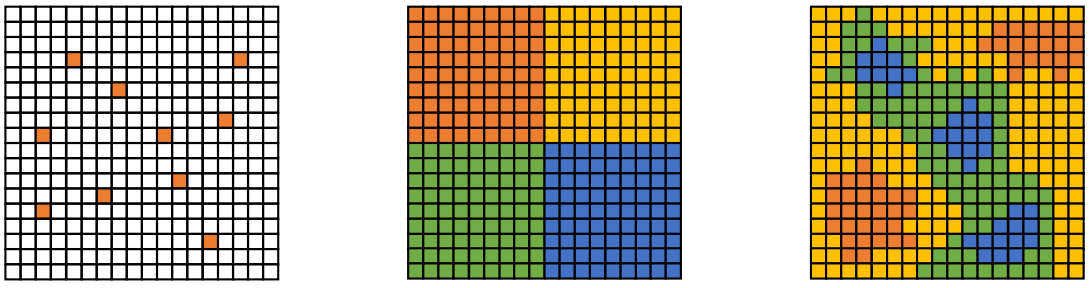
While the global sort can be avoided [8], the algorithm is still difficult to parallelize. Three strategies have been proposed to parallelize join tree calculations: pruning [4], spatial-domain parallelization [7], and value-domain parallelization [6]. Pruning (Fig 3 (a)) works by eliminating elements from the computation, which are predetermined not to be a minimum, maximum, or saddle point. This process can be done in parallel, but the compute phase still needs to be completed in serial. Spatial-domain parallelization (Fig 3 (b)) divides the scalar field into multiple smaller fields, each distributed to a different thread, processor, or computer. After each subfield has its tree computed, a messy tree merging process takes over. Finally, value-domain decomposition (Fig 3 (c)) distributes the scalar field to different threads, processors, or computers by selecting ranges of element values. This allows parallelizing the loosely ordered computation phase but still requires processing every element.

Each of these approaches take advantage of certain properties of Augmented Join Tree construction, but up until now, these strategies have not been effectively integrated.

In this paper, we have combined these strategies in an OpenCL Augmented 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.

2 CONVENTIONAL JOIN TREE CONSTRUCTION

The conventional Join Tree construction process [2] starts with a scalar field (Fig. 2). The elements of the field are first sorted (Fig. 4 (a)) in ascending order for a merge tree or descending order for a split tree.



(a) (b) (c)

Fig 3: The 3 existing strategies used to parallelize Augmented Join Tree construction. (a) In pruning, a parallel operation can prune away most non-critical points from computation. (b) In spatial-domain decomposition, regions of the original scalar field are split and distributed to different processes. (c) In value-domain decomposition, elements are distributed to processes based upon ranges of values.

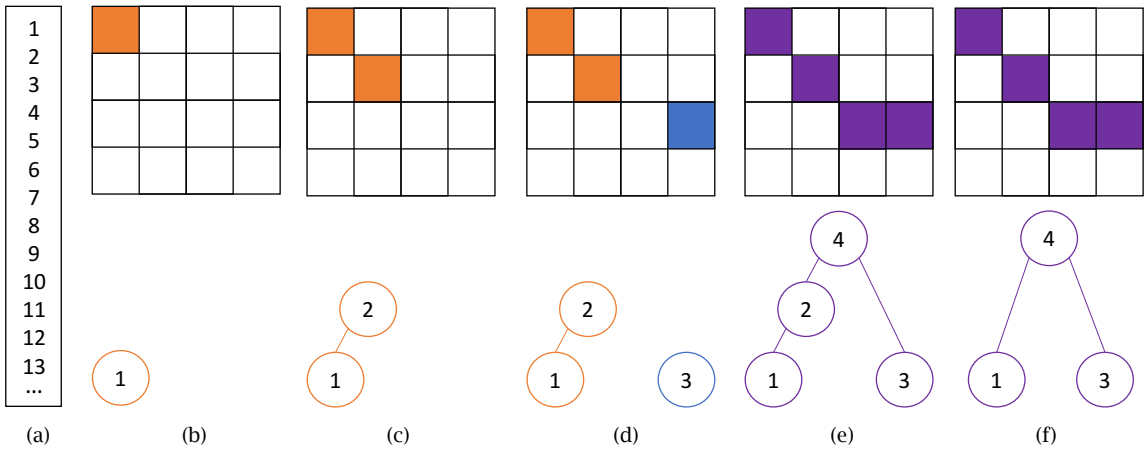


Fig. 4: Description of the conventional Augmented Join Tree algorithm. (a) Scalar field values are first sorted. Then, the points are added to the tree one-by-one. (b)(d) If no neighbors are in the tree, a leaf is created. If any neighbors are in subtrees, the node is connected to the top of the subtree. (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.

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. 4 (b) bottom) and a color assigned (Fig. 4 (b) top). Next, the neighborhood of 8 surrounding elements is searched. If none has been assigned a color (Fig. 4 (b) and (d)), the operation is complete. If one (Fig. 4 (c)) or more (Fig. 4 (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. 4 (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. 4 (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. 4 (f) bottom).

From an implementation standpoint, this operation relies on 2 components. 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, which has a cost of $\mathcal{O}(k)$ per lookup. Other operations are constant time per element. Unfortunately, this strictly-ordered bottom up construction of the tree means that each operation relies upon the results of the prior operation making parallelization challenging.

3 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.

3.1 Phase 1: Coloring

The coloring phase has two main objectives. The first objective is to prepare for eliminating 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.

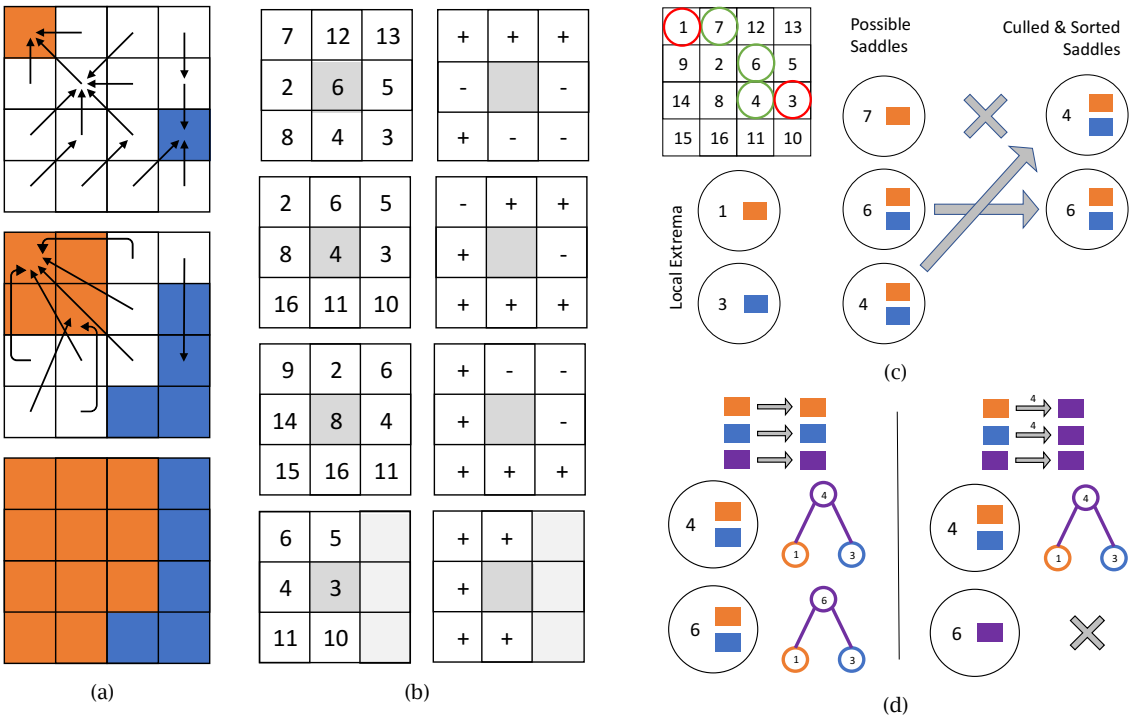


Fig 5: The four phase of our parallel implementation include (a) spatial-domain decomposition and pruning, (b) potential critical point extraction, (c) potential saddle point sorting, and (d) value-domain decomposition of saddle points and propagation of joins.

The water shedding approach is described in Fig. 5 (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. In the next step, each element is updated to the pointer of its pointer. This is essentially the find algorithm of a disjoint-set. 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.

3.2 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. 5 (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 one disjoint negative direction exist. This point can be excluded as non-critical. Finally, for element #3 all neighbors are larger indicating a local minimum.

3.3 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 5 (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 to. Therefore, #4 and #6 possibly 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.

3.4 Phase 4: Subtree Building and Join Propagation

The final phase of processing builds the tree by 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 occur.

Fig 5 (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 5 (d) top left). The two subtrees are built and the global join information updated (Fig 5 (d) top right). In the second pass (Fig 5 (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.

4 EXTENSION TO 3D AND HIGHER DIMENSIONS

Extending this approach to 3D or higher data is mostly trivial. Phases 1 and 2 do require modification.

For phase 1, the process is the same, except that now, the number of neighbors that must be searched grow to 26 for 3D or $3^d - 1$ for higher dimensions.

Phase 2 is problematic since saddle point detection in 3D or higher dimension is complex. This is because there are many more saddle point configurations in higher dimensions. To overcome this, phase 2 saddle detection can be skipped, and all points can be colored and treated as saddle points. Then, phases 3 and 4 can continue unmodified. The benefit of this is that complex saddle point detection is avoided. The downside is that a much larger number of saddle points are considered in phases 3 and 4.

5 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.

6 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.

6.1 Random Field Tests

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). A random field represents the most challenging case for calculating augmented join trees, as it is likely to produce a very dense set of critical points. To test our approach under less dense situations, we analyze those 130 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.

Fig. 6 (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. 6 (b) and 6 (c), respectively. These have 98 and 46 critical points, respectively.

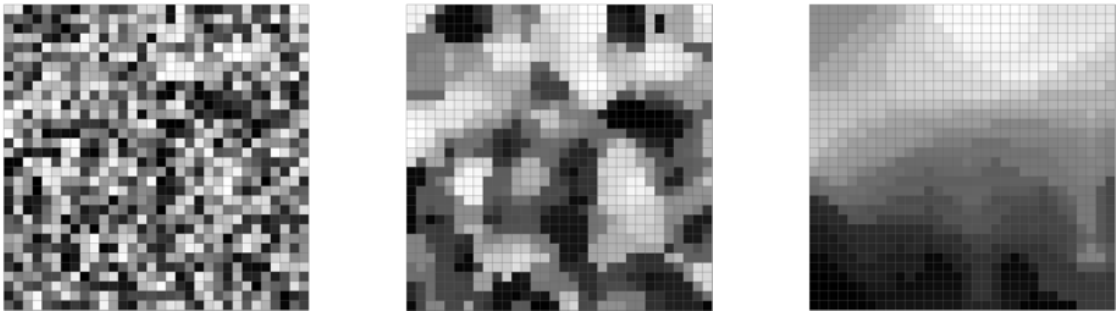
Fig. 7 (top) shows log-log charts highlighting the performance of various phases of our approach. Fig. 6(a) shows that in the average case, phases 1 and 2 grow linearly with respect to the number of elements in the field (R^2=0.96 and R^2=0.974, respectively). Similarly, Fig. 7 (b) shows that for the 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. 7 (middle) uses log-log charts to compare the performance of our approach to the CPU implementation. Fig 7 (c) shows the computational time against the number of elements, while Fig 7 (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 approximately 1 order of magnitude.

Fig. 7 (bottom) uses a log-linear charts to compare the speedup of our approach to the CPU implementation on the MacBook Pro. Fig 7 (e) shows the speedup against the number of elements, while Fig 7 (d) shows the speedup against the number of critical points. Interestingly, as the problem size grows, the GPU implementation speedup grows as well. We believe this is caused by the fact that the overall GPU performance is driven by the number of critical points, while the CPU performance is driven by the number of elements.

6.2 Contour Trees in Radio Astronomy Data

In radio astronomy, scalar fields are one of the primary data sources used to validate hypotheses. Radio telescopes capture 3D maps of the radio signals in the sky. Two dimensions of these maps are spatial



(a) (b) (c)

Fig. 6: Example 32x32 scalar fields: (a) random noise input, (b) after 2 smoothing iterations, and (c) after 4 smoothing iterations.

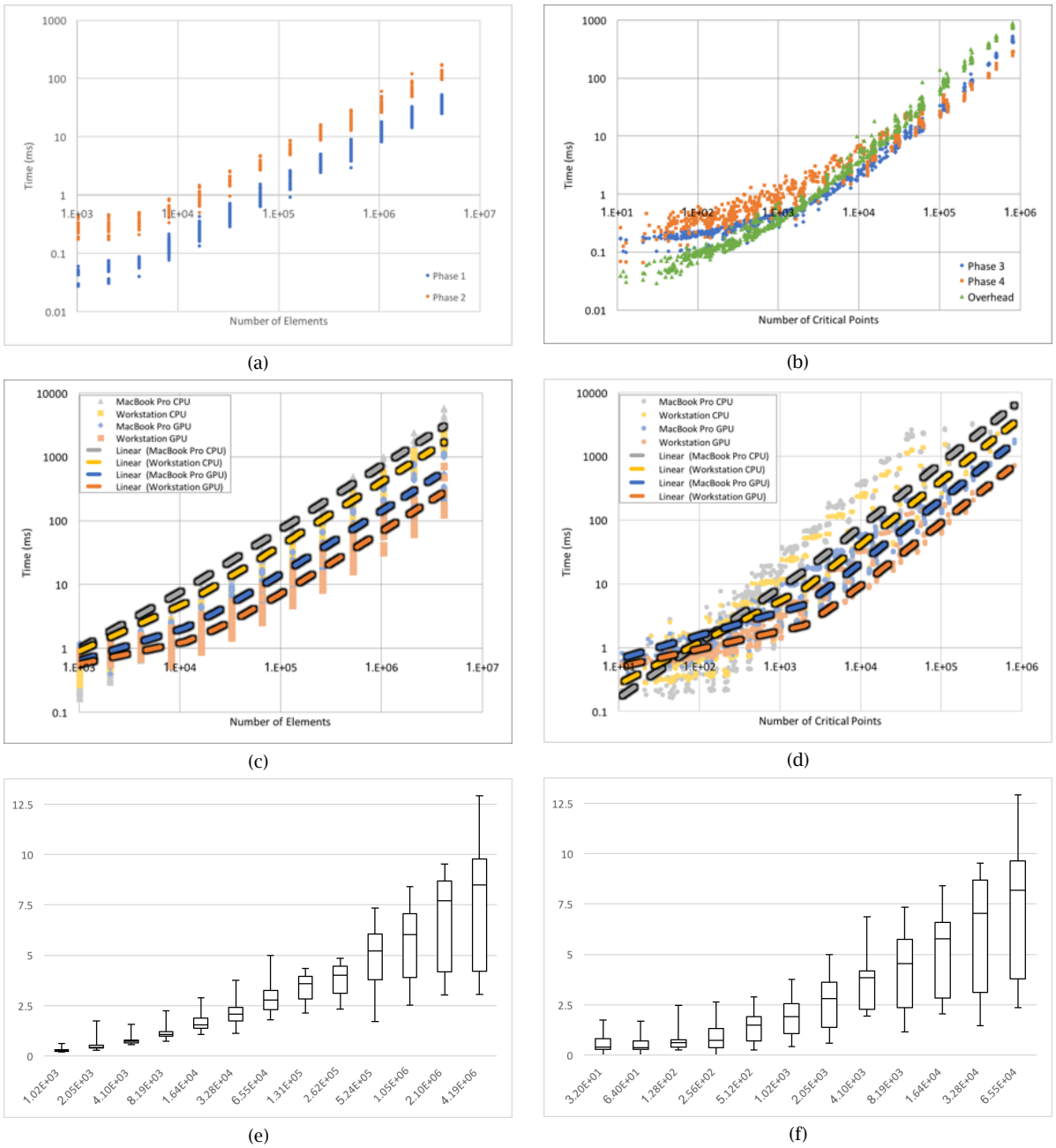
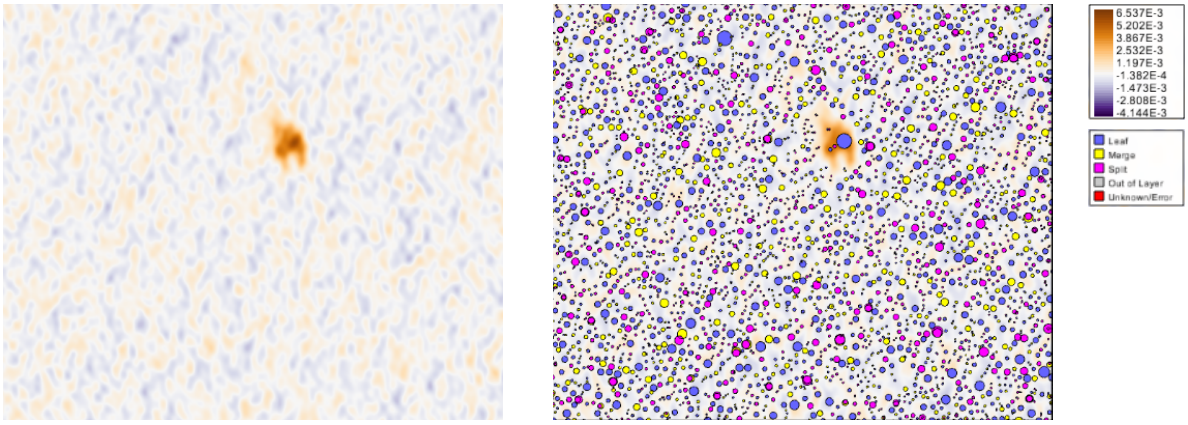


Fig. 7: Charts of the performance of our algorithm on random fields. (a) Log-log chart of the processing time of Phase 1 and 2 is highly linear against the number of elements. (b) Log-log chart of processing time of Phase 3, 4, and overhead is highly linear against the number of critical points. Log-log 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 on average slightly less than 1 order of magnitude improvement over the CPU counterpart. Log-linear boxplots of the speedup based upon the number of element (e) and based upon the number of critical points (f) show that as the problem size grows, the speedup increases as well.



(a) (b)

Fig. 8: Visualization of Radio Astronomy data. (a) The noisy data shows the amplitude of the radio signal at multiple locations in the sky for a single wavelength. (b) The visualization of the contour tree shows the result of the union of a merge tree with a split tree. The density of critical points in this data is quite high.

positions in the sky. The third dimension is different radio frequencies. Unfortunately for astronomers, the radio signals collected are very low power and have a high signal to noise ratio. The problem was described best by one radio astronomer, "a cell phone on the moon would be a brightest signal in the sky".

Fig. 8 (a) shows an example of this data for a single radio frequency. The red blob towards the middle of the image is the feature of interest. In this dataset, this blob represents the signal put off by dust circling a black hole. For our experiments, we calculate the contour tree, which is just the union of a 2 augmented join trees (a merge and split tree). Fig 8 (b) shows a small region with the contour tree nodes highlighted. In this image leaves can be seen (both local minima and maxima) as blueish purple nodes and saddle points are yellow for merge saddles and magenta for split saddles.

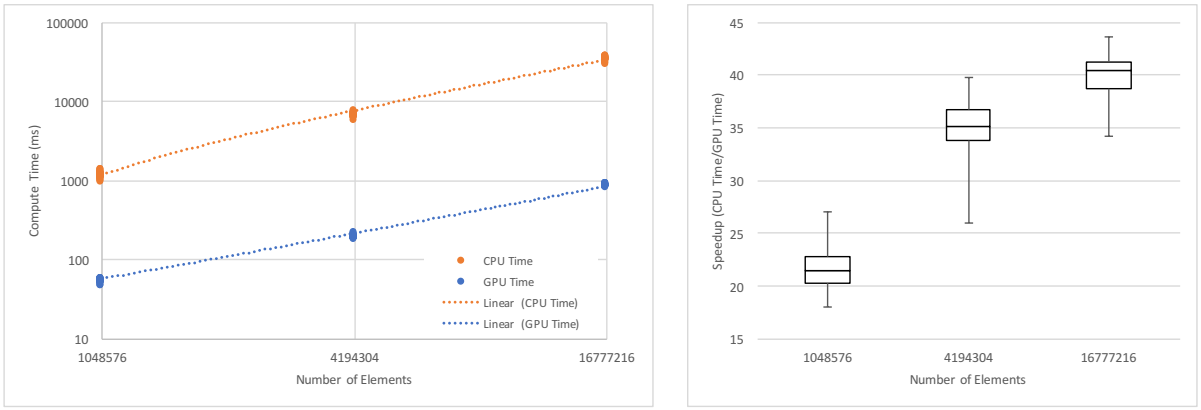
Fig 9 shows the performance results for our experiments. These experiments were only run on our MacBook Pro CPU/GPU. For each resolution (1024x1024, 2048x2048, and 4096x4096), we ran our tree construction on each of 38-2D slices (radio frequencies) of the data. Considered in our calculations are only the augmented join tree costs.

The log-log chart in Fig. 9 (a) shows that the time taken for our GPU implementation significantly outperforms the CPU implementation. Furthermore, both the CPU and GPU implementations performance grows linearly with the number of elements (R^2>0.99). The log-linear boxplots in Fig 9 (b) show the speedup for our implementation. As the number of elements grows, so too does our speedup, reaching on average 40x faster for the GPU implementation on the 4096x4096 example.

The speedup seen here is significantly better than that observed in the random field case. As mentioned in those tests, the random field example is the most challenging because of critical point density. For the random field tests, the median density was 1 critical point per 33.2 elements. For the radio astronomy data, the median density was 1 critical point per 107.5 elements, over 3x less dense. Given the strong relationship between the number of critical points and overall performance of our approach, this result makes sense.

7 CONCLUSIONS

In conclusion, we have presented an approach for efficiently calculating an Augmented Join Tree in parallel by combining 3 approaches, pruning, spatial-domain parallelization, and value-domain parallelization. This approach makes it feasible to quickly calculate merge trees, split trees, and contour trees for large scalar fields in any number of dimensions. These data structures are incredibly useful analyzing the scalar field data. We have evaluated our approach with a synthetic random field dataset and with a dataset from the discipline of radio astronomy. Although we have calculated the Augmented Join



(a) (b)

Fig. 9: Radio astronomy data compute time (a) and speedup (b) on the MacBook Pro CPU and GPU for 38 2D slices at 3 different resolutions, 1024x1024 (1048576 elements), 2048x2048 (4194303 elements), and 4096x4096 (16777216 elements).

Tree in parallel, in the future, parallelizing several additional computations would be exceedingly useful. For example, parallelizing the union of Augmented Join Trees (to form Contour Trees), calculating of Persistence (a stability measure), or the hierarchical simplification of a Merge, Split, or Contour Tree would all be very useful moving forward.

8 ACKNOWLEDGEMENTS

Paul Rosen, http://orcid.org/0000-0002-0873-9518
Junyi Tu
Les A. Piegl, http://orcid.org/0000-0003-0629-8496

REFERENCES

[1] Bajaj C.; Pascucci V.; and Schikore DR.: The contour spectrum, In Proceedings of the 8th Conference on Visualization, pp. 167-ff, 1997. https://doi.org/10.1109/visual.1997.663875

[2] Carr, H.; Snoeyink, J., and Axen, U.: Computing contour trees in all dimensions, Computational Geometry 24.2, 2003, 75-94. https://doi.org/10.1016/S0925-7721(02)00093-7

[3] Carr H.; Snoeyink J.; and Van De Panne M.: Flexible isosurfaces: Simplifying and displaying scalar topology using the contour tree, Computational Geometry, 43(1), pp. 42-58, 2010. https://doi.org/10.1016/j.comgeo.2006.05.009

[4] 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. https://doi.org/10.1109/ldav.2016.7874312

[5] Dillard S.: Contour trees and cross-sections of multiphase segmentations, University of California at Davis, 2009.

[6] Gueunet, C.; Fortin, P.; Jomier, J.; and Tierny, J.: Contour Forests: Fast Multi-threaded Augmented Contour Trees, IEEE Symposium on Large Data Analysis and Visualization, 2016. https://doi.org/10.1109/ldav.2016.7874333

[7] Morozov, D.; and 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

[8] Raichel B.; and Seshadhri C.: Avoiding the global sort: A faster contour tree algorithm, arXiv preprint arXiv:1411.2689, 2014.

[9] Szymczak A.: A categorical approach to contour, split and join trees with application to airway segmentation, In Topological Methods in Data Analysis and Visualization, pp. 205-216, 2011. https://doi.org/10.1007/978-3-642-15014-2_17

[10] Van Kreveld M.; van Oostrum R.; Bajaj C.; Pascucci V.; and Schikore D.: Contour trees and small seed sets for isosurface traversal, In Proceedings of the thirteenth annual symposium on Computational geometry, pp. 212-220, 1997. https://doi.org/10.1145/262839.269238

Please follow the format shown above. That is:

- author names are listed by last name;
- multiple authors are separated by a semi-colon;
- after the last author's name place a colon;
- list the title and the journal separated by a comma; and
- the journal details are Volume (Issue), Year, Pages.

For books, thesis, etc, please list the title, publisher, place and year of publication.

NOTE: Please do not change any of the settings in this file, and please do not remove any of the headers and footers! Thank you.

IMPORTANT NOTE: EFFECTIVE VOLUME 8, WE ARE REQUIRED BY CROSSREF TO INCLUDE HYPERLINK INTO EVERY REFERENCE THAT HAS DOI LINKING. PLEASE GO TO:

http://www.crossref.org/SimpleTextQuery/

REGISTER YOUR E-MAIL, CUT AND PASTE THE LIST OF REFERENCES INTO THE BOX, GET THE DOI LINKS AND PASTE THEM INTO YOUR PAPER! AS PER NEW CROSSREF PRACTICE, WE NEED TO USE HTTP AND DROP THE DX. EXAMPLE:

https://doi.org/10.1080/16864360.2014.881190

IMPOERTANT NOTE NO 2: EFFECTIVE VOL. 13, WE HAVE TO ADD YOUR ORCID TO YOUR PAPER. TO GET YOUR ORCID, PLEASE VISIT https://orcid.org/. AFTER YOU REGISTERED, THE SYSTEM GIVES YOU A 16 DIGIT ID WHICH YOU INSERT INTO THE BRACKETS. FOR EXAMPLE:

John C. Gotti[0000-0003-0629-8495]