Streaming Connected Component Computation for Trillion Voxel Images

Martin Isenburg
LAStools

Jonathan Shewchuk
UC Berkeley

Abstract

We describe a highly scalable algorithm for computing connected components in large binary 3D grids. We employ a streaming version of the union-find algorithm that—as it processes the voxel grid row-by-row and layer-by-layer—only maintains the component structure along a single layer of *active rows*. We can process voxel grids whose sizes are three orders of magnitude larger than what the best previously reported method is capable of and still use less than half the memory. For example, given a random binary grid of 10,000×10,000×10,000 (= one trillion) voxels our method can count the number of components and report the volume and surface area of each using no more than 350 MB of main memory.

Keywords: connected components, image labeling, streaming union-find, finalization, large data sets, out-of-core algorithms

1 Introduction

Imagine a regular grid of binary numbers in three dimensions where a binary 1 corresponds to a solid voxel and where a binary 0 corresponds to an empty voxel or vice versa. Now consider the following questions: How many solid connected components are there? What is their volume? What is the area of their boundary surface?

These are just some of the questions that arise in typical image analysis applications. For example, in order to study cracks in concrete that is stressed to failure, Landis et al. [12] create high-resolution three-dimensional images of concrete specimens under load with microtomographic scanning. They threshold these images into binary grids and detect crack opening by measuring the increase in crack volume over the increase in crack surface area.

The common approach for finding connected components among a set of elements is to use a union-find or disjoint-set data structure [4]: process the elements one at a time, loop over all processed elements that are connected to the current element, find their component, and merge the current element with their component. Employing tree-based union-find structure with path compression and union-by-rank heuristic leads to linear run-times in practice.

The structured nature of binary grid of voxels can be exploited to make the algorithm even more efficient. The number of necessary adjacency checks and union-find operations can be significantly reduced by using run-length encoding as the first step [7]: group adjacent solid voxels of the same row into runs and perform adjacency checks and union-find operations per run rather than per voxel.

Franklin and Landis [6] describe a concise union-find data structure for finding connected components in large binary 3D images. It is implemented as a single array of voxel run records that index each other as the union operations merge them into components. On difficult random input where half the voxels are solid and the average run-length is two, their algorithm is capable of processing images of nearly 900^3 voxels before exhausting 2 GB of memory.

After Franklin and Landis' algorithm has processed the entire image it stores all runs and how they are connected in main memory. Only then it starts discovering and reporting connected components. This makes it possible to output a description for each component, for example in form of a labeled image or as a list of voxels. But this also means that the memory requirements are proportional to the number of runs in the input, effectively limiting this approach to binary 3D grids in the billion elements range.

We describe a streaming approach to computing connected components that pushes the scalability up by three orders of magnitude. Our implementation can process random binary grids with up to $10{,}000^3$ voxels—putting us in the trillion element range—on a standard laptop using no more than 350 MB of main memory.

The key idea is to employ a *streaming* version of the union-find algorithm. As we process the voxel grid row-by-row and layer-by-layer we only maintain the component structure along a single layer of *active rows*. We *finalize* an active row when we know that its voxels cannot not be adjacent to any future row. In this moment we deallocate (or recycle) all data structure that will not part take in future union-find operations. We find a connected component when we delete the last reference to a root node. We immediately output its volume and surface area. Hence, after a single processing pass over the entire grid we have reported all components. Several other image processing queries can also be answered this way.

If necessary, we can also output a description of components in form of a labeled image by operating in three passes: two passes of the streaming connected component algorithm over the whole image and one pass over a (typically smaller) temporary file.

2 Connected Components and Labeling

Computing connected components is a fundamental task in image processing. Applications in pattern recognition or computer vision, for example, often start out by *labeling* all connected components in an image. A report by Wu, Otoo, and Suzuki [3] gives a good summary on recent work in connected component labeling. Labeling connected components not only involves finding connected components, but also producing a description of each component, usually in form of a labeled image where all voxels of the same component are given the same color or simply as a list of voxels.

A typical image processing task may want a list of the locations of all connected components above a certain size. Or the location and orientation of the largest three components. Or the centroids of all components whose bounding box has an aspect ratio below a certain cutoff. Using Matlab's Image Processing Toolbox one would use the function `bwlabel` or `bwlabeln` to create an image where connected components are labeled and the function `regionprops` to generate an array containing the area, centroid, and/or orientation of each component. In a final loop over this array one would then report the desired outputs.

Our streaming connected component approach can answer these and similar queries in one pass over the image and without explicitely constructing a labeled image.

2.1 Preliminaries

The answer to how many solid connected components there are depends on which neighboring cells we consider as being connected. Our implementation supports 6-connectivity (i.e. only solid cells that share a face are connected) as well as 26-connectivity (i.e. solid cells that share a vertex are connected) for 3D images. A single solid cell has a volume of one unit and a surface area of six units. Our implementation also supports the special case of 2D data. The corresponding scenarious are 4-connectivity and 8-connectivity and cells have an area of one unit and a boundary length of four units.

The outside of the grid can be considered either empty or solid. We consider the outside to be empty so that components end at the boundary of the grid. In the other case all components that touch the boundary of the grid would be connected.

3 Franklin and Landis' Algorithm

The algorithm of Franklin and Landis [6] reads and processes the grid one row at a time in row-by-row and layer-by-layer order. Each read row (x, y) is decomposed into a sequence of runs (x, y, z_{lo}, z_{hi}) of connected solid cells. These runs start out as roots of single-run components. The runs are then merged with the components of overlapping runs from previously processed, adjacent rows. Which rows are considered adjacent depends on which type of connectivity that is to be computed. For 6-connectivity these are rows $(x-1, y)$ and $(x, y-1)$ given that they exist. For 26-connectivity these are in addition rows $(x-1, y-1)$ and $(x-1, y+1)$.

The run sequences of two rows are checked for overlap in pairs of runs. The check starts at the two runs that have the smallest z_{hi} value in their respective sequence. After each check the sequence whose current run has the smaller z_{hi} value is advanced. When an overlap between the runs is detected the `find()` function is used to find their components. If the runs already belong to the same component their actual overlap is subtracted from the running total for surface area (note: for 26-connectivity the rows may not actually have an overlap in surface area). Otherwise the `union()` function is used to merge the components with their running totals for surface area being summed and their actual overlap being deducted.

The `find()` and `union()` functions of the tree-based union-find data structure are implemented with a single array of run records. In addition to x, y, z_{lo}, and z_{hi} the records contain a *parent* field. This number is either negative marking the run as root of a component and specifying the running total for its surface area or is positive and indexes the parent of this run in the array of run records.

After the last row was processed the array of runs records contains c (extremely shallow) trees of runs that correspond to the c connected components. Another pass over the run array computes the volume of each component and compresses all paths (i.e. each run either indexes or is the root). In the final pass the volume and the surface area of each component is reported (note: the implementation by Franklin and Landis first sorts the components by volume). Because all runs are stored in memory the algorithm can optionally output a description of each component as a list of runs.

4 Streaming Connected Components

Just like Franklin and Landis we process the grid row after row and decompose each row into a sequence of runs. The main difference is that we immediately delete data structures that will not part take in future union-find operations. At any time we only keep in memory a layer of *active rows* that are composed of *active runs* plus a varying number of *active nodes* that describe how active runs are connected with each other by the rows that have been processed before.

Runs are active as long as their rows are active. The record for an active run (shown in Figure 2) contains a *parent* pointer to an active node that is zero until the run merges with another run and two indices z_{lo} and z_{hi} that specify the z coordinate of where the voxel run starts and ends. The x and y coordinate of each run are implicit due to the order in which the rows are processed.

Rows become active in the moment they are processed and remain active as long as they are adjacent to some unprocessed row.

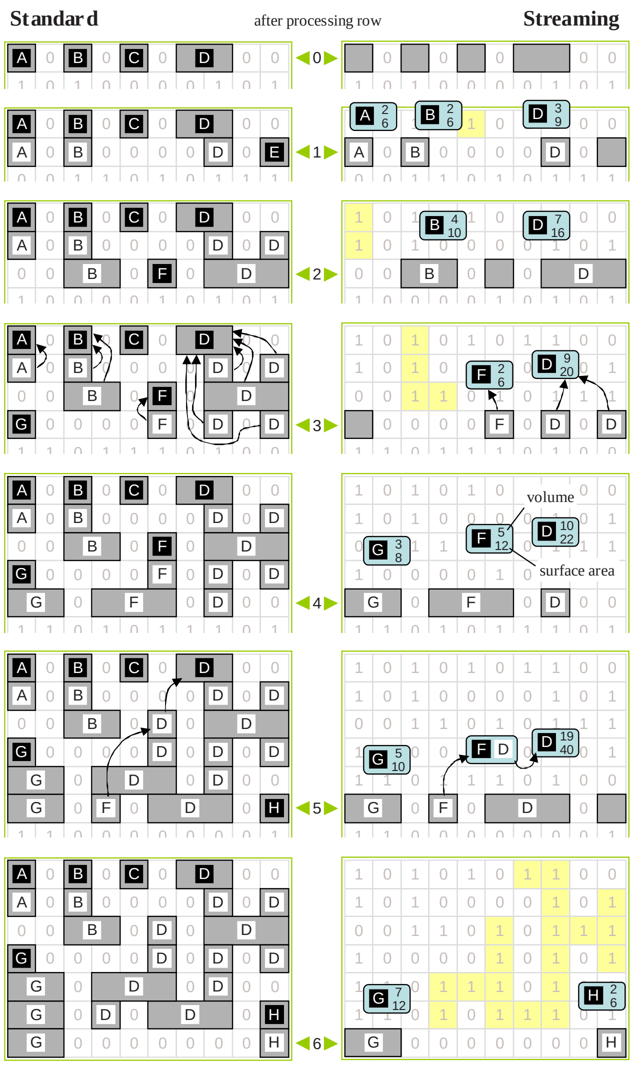

Figure 1: Computing connected components of a 10 by 7 grid with standard (left) and streaming union-find (right). Standard union-find merges all runs into components. Streaming union-find maintains only active runs and active nodes: (0) first row is read. it has 4 runs. there is no adjacent row yet. (1) second row has also four runs. the first three merge with runs from the previous row creating nodes A, B, and D. the first row is finalized. its third run is reported as a single cell component. (2) the third row has three runs. the first merges with node B, the third merges first with D and then with a previously unmerged run from the previous row. the second row is finalized. this finalizes root node A and a two cell component with a surface area of six is reported. (3) the fourth row has four runs. one merges with F and two merge with D. the third row is finalized. this finalizes root node B and a four cell component with a surface area of 10 is reported. (4) the fifth row has three runs that merge with G, F, and D respectively. the fourth row is finalized. (5) the sixth row has four runs. the second run merges into F. the third run first merges into F and then merges F into D. the fifth row is finalized. (6) the seventh row has two runs that merge into G and H. the sixth row is finalized. this finalizes node F and root node D and a nineteen cell component with a surface area of 40 is reported. now shown is finalizing the last row where two more components are output.

After their last adjacent row was processed we *finalize* an active row. In this moment we can usually delete a fair amount of data structure (or rather recycle it for the next row that we read). The maximal number of rows that have to be active at the same time is one layer worth of rows plus one (or plus two when we compute 26-connectivity). That is in 3D—in 2D the maximal number is two.

Nodes are active as long as they are pointed to by an active run or by another active node. An active node record (see Figure 2) contains two numbers *vol* and *area* for the running totals in volume and surface area of the connected runs for which this node is the root. It also contains a *parent* pointer that points to another active node if this node is not the root and a *ref_count* that counts how many runs and nodes reference this node as their parent.

We create a new active node whenever we merge two active runs that have not merged with any run before (i.e. whose *parent* pointers are both zero). We initialize the active node with the combined volume and surface area (minus the overlap) of these two runs. We merge two active nodes whenever we merge two runs that have these two nodes as their respective root. We delete an active node when its reference count reaches zero. In case this active node is the root of a component (i.e. its *parent* pointer is zero) we have completely traversed a connected component and we record—or preferably immediately output—its volume and its surface area. The small 2D example in Figure 1 illustrates the difference between standard and streaming connected component computation.

We implement two standard heuristics to assure the total runtime of union find can be expected to be linear: path compression and weighted merge. Whenever we `find()` the root node of an active run we perform path compression. Whenever we `merge()` two root nodes we pick the node with the larger volume or the node with the larger surface area as the new root. The pseudocode in Figure 3 illustrates the entire algorithm for the 2D case. Note that this code does not include path compression or weighted merge.

```
struct ActiveRun {
  ActiveNode* parent;
  unsigned short z_lo;    struct ActiveNode {
  unsigned short z_hi;      ActiveNode* parent;
};                          unsigned int area;
struct ActiveRow {          unsigned int vol;
  int num;                  unsigned int ref_count;
  int allocated;          };
  ActiveRun* runs;
};
```

Figure 2: Data structures used for streaming connected components.

5 Results

We compare our streaming connected component implementation (that is available here [1]) to that of Franklin and Landis (that is linked here [2]). For a more fair comparison we modified their implementation so that it would neither sort the components nor output them so that both algorithms only report component statistics.

We run both implementations on the largest test case used by Franklin and Landis, a binary grid with dimensions 1,024 by 1,088 by 1,088 totalling 1,212,153,856 voxels. About 50 percent of the voxels are solid and runs have an average length of 30 voxels. Our code is about 25 percent faster, taking only 15.3 instead of 19.7 seconds to process the data. Our code also uses about 50 times less memory, with the total footprint being only 3.5 instead of 240 MB.

Random input constitutes a near worst-case scenario in terms of memory requirements for non-streaming approaches. With half the voxels being solid and the average run length being two voxels, Franklin and Landis have to allocate one run record for every four voxels in the input. Given 2 GB of main memory and each run using 12 bytes they can store at most 178 million runs which limits processing of random grids to a size of 894^3 voxels.

```
void sconnect2d(FILE* file, int nx, int ny) {
  ActiveRow rows[2];
  read_row_of_bits_as_runs(file, nx, rows[0]);
  for (y = 1; y < ny; y++) {
    read_row_of_bits_as_runs(file, nx, rows[y&1]);
    connect_row(rows[y&1],rows[(y-1)&1]);
    finalize_row(rows[(y-1)&1]);
  }
  finalize_row(rows[(ny-1)&1]);
};
void connect_row(ActiveRow& new, ActiveRow& old) {
  int n = 0, o = 0;
  while (n < new.num && o < old.num) {
    check_adjacency(new.runs[n], old.runs[o]);
    if (new.runs[n].z_hi < old.runs[o].z_hi) n++;
    else o++;
  }
};
void check_adjacency(ActiveRun &n, ActiveRun &o) {
  if ((n.z_hi-o.z_lo) < 0 || (o.z_hi-n.z_lo) < 0)
    return;
  ActiveNode* nroot = (n.parent ? find(n.parent) : 0);
  ActiveNode* oroot = (o.parent ? find(o.parent) : 0);
  if (nroot == 0 && oroot == 0) {
    nroot = n.parent = o.parent = new ActiveNode;
    nroot->vol = n.z_hi-n.z_lo+1+o.z_hi-o.z_lo+1;
    nroot->ref_count = 2;
  } else if (nroot == 0) {
    n.parent = oroot;
    oroot->vol += n.z_hi-n.z_lo+1;
    oroot->ref_count++;
  } else if (oroot == 0) {
    o.parent = nroot;
    nroot->vol += o.z_hi-o.z_lo+1;
    nroot->ref_count++;
  } else if (nroot != oroot) {
    oroot.parent = nroot;
    nroot->vol += oroot->vol;
    nroot->ref_count++;
    if (oroot->ref_count == 1) delete oroot;
    else oroot->ref_count--;
  }
}
void finalize_row(ActRow &row) {
  for (r = 0; r < row.num; r++)
    if (row.runs[r].parent)
      if (row.runs[r].parent->ref_count == 1)
        finalize_node(row.runs[r].parent);
      else
        row.runs[r].parent->ref_count--;
    else
      print(``component with vol %d'',
          row->runs[r].z_hi-row->runs[r].z_lo+1);
}
void finalize_node(ActiveNode* node) {
  if (node->parent)
    if (node->parent->ref_count == 1)
      finalize(node->parent);
    else
      node->parent->ref_count--;
  else
    print(``component with vol %d'', node->vol);
  delete node;
}
```

Figure 3: Illustrative code for streaming connected component computation in 2D. The key ideas are the `finalize_row` function and the `ref_count` counters. We only compute the number of voxels for each component. Not shown is the path compression code that also decrements `ref_count` counters and deallocates nodes.

size of	components		avg.	utilized resources		per billion cells	
grid	total	singleton	area	time	RAM	time	RAM
$1{,}000^3$	8,967,435	87.7 %	167.4	49 sec	4.8 MB	49 sec	4.8 MB
$3{,}000^3$	241,074,040	87.7 %	168.1	23 min	36 MB	52 sec	1.4 MB
$9{,}000^3$	6,485,028,164	87.8 %	168.6	10 hrs	296 MB	51 sec	.41 MB
$10{,}000^3$	8,911,264,999	87.7 %	168.3	14 hrs	350 MB	51 sec	.35 MB

Table 1: Results for streaming connected component computation with 6-connectivity on random binary grids of increasing size. We report the total number of components, the percentage of components consisting of a single cell, the average surface area of the components, run-time and main memory use in total as well as per billion cells.

size of	components		utilized resources		per billion cells	
grid	total	singleton	time	RAM	time	RAM
$50{,}000^2$	164,441,796	47.5 %	86 sec	0.9 MB	34 sec	587 KB
$100{,}000^2$	1,292,871,491	50.2 %	6 min	1.7 MB	31 sec	176 KB
$500{,}000^2$	56,461,513,430	50.0 %	113 min	4.5 MB	27 sec	18 KB
$1{,}000{,}000^2$	237,918,756,449	50.0 %	7 hrs	7.8 MB	26 sec	8 KB

Table 2: Same measurements as done in Table 1 but for 2D images.

In Table 1 we demonstrate the scalability of our implementation on random binary voxel grids ranging from one billion to one trillion elements. The images are not read from disk but generated on the fly using repeated calls to the `rand()` function. In addition to the totals for run time and main memory requirement, we report timings and memory use per one billion processed elements. All experiments were run on a laptop with a 2.13 GHz Intel Pention processor and 1GB of main memory running Windows XP.

As expected, our computation costs grow linearly with the size of the problem whereas the amount of memory used only grows logarithmically. The CPU runs continuously at 100 percent for the one billion as well as the one trillion element grid.

For completeness we report in Table 2 scalability results for 2D images. Streaming connected components in 2D scales to incredibly large images in terms of memory requirements. In contrast to the 3D case where the memory footprint grows more or less with the size of one layer of rows of voxel runs, in 2D we only need to keep two rows of voxel runs in memory at any time.

For the worst case scenario in terms of memory footprint we must consider the maximum number of runs and nodes that could potentially be active at the same time. The number of active runs is maximal when each active row consists of single solid voxels separated by single empty voxels. The maximal number of active nodes is not as obvious. Because of path compression nodes have at least two nodes or runs pointing at them—unless they are root node of a single active run. That means the maximal number of active nodes either is the number internal nodes of a binary tree with as many leaves as there are active runs or it equals the maximal number of active runs. Both numbers are the same.

6 Extensions

The algorithm we have described so far can compute and output per component properties. We have described how to do this for volume and surface area, but other properties such as the bounding box, the center of mass, etc. could also be computed this way.

As presented our approach cannot report a per-voxel description of each component. Most runs that contribute to a component have long been finalized and deallocated by the time the root node of that component is finalized. We describe here how to make our algorithm report a per-voxel description.

There are two ways to specify which voxels are part of which component: a labeled image of the same dimension as the binary image where voxels of the same component are given the same unique color (i.e. the label) or a detailed list of voxels (or runs of voxels) for each component. Due to the size of the images we are processing it can be impractical to report every component. For a labeled image, a very large number of components means a very large number of unique colors which in turn would require many bits per voxel to represent the many different colors. Similarly, for a detailed list of voxels the amount of data produced to describe billions of tiny components or several components containing billions of voxels may require many times the storage of the original binary voxel grid.

6.1 Labeled Image

By running our streaming connected component computation *twice* we can output a labeled image. During the first streaming connected component pass we record all merges between active components to a temporary *merge file*. In one *reverse* pass over this merge file we create a temporary *root file* that contains a mapping from active components to final root components (sorted in reverse order of creation). During the second streaming connected component pass we read the root file in reverse: whenever an active component is created we look up its root so that we can output correctly labeled voxels whenever a row is finalized.

To create the merge file we maintain an `index` field with each active node during the connected component computation. When a node is created this index field is set to the value of a global counter and the global counter is incremented. When two nodes merge we declare the one with the smaller index the root and output both indices to the merge file. In case a root node is finalized whose index was never output (i.e. it was not merged with any other node) we write its index twice to the merge file. We also tag indices when they are written for the *first* time. When we read the merge file in reverse these *finalization tags* tell us that an index has appeared for the *last* time.

To create the root file we read the merge file starting at last index pair. We maintain a hash table where we insert node records when their index is read for the first time and where we look-up node records every subsequent time their index is read. The parent pointer of the first node is set to point to the second node (e.g. the one it was merged into). When an index is tagged the corresponding node is removed from the hash table and inserted into a priority queue that is sorted by largest index. After processing an index pair we check whether the last component with the highest possible index is now in the priority queue. If yes we find its root by following its parent pointers and output its index to the root file. The actual algorithm is given in Figure 4.

To finally create the labeled image we run a second connected component computation. Whenever a new node is created we find its final root in the root file. Now, whenever an active row is finalized all its runs can already know their final label and we can raster its labeled voxels to disk.

In order to limit the resolution of the produced labeled image to, for example, 16 bit per voxel, we may want to label only the 65,335 largest components. To keep track of the size of each component we enhance the merge file by always outputting the index of a finalized root node together with the size of the corresponding component. We transfer this information when we create the root file. We can find the cutoff size for the smallest large component we want to keep either with one pass over the root file and a priority queue that always holds the currently largest 65,335 components or with repeated passes over the root file doing a binary search.

6.2 List of Voxels

In addition to maintaining the current total of voxels or a current bounding box with each active root node we could simply keep a list of run-length encoded voxels that is output when the component is finalized. However, the memory requirements of such an approach

```
struct TaggedIdx {           struct NodeWithIdx {
  unsigned int idx :  31;      int idx;
  unsigned int tag :  1;       NodeWithIdx* parent;
};                           };
void merge_2_root(fp* in, fp* out, int n_p, int n_n) {
  TaggedIdx idxs[2];
  NodeWithIdx* nodes[2];
  while (n_p--) {
    fseek(in, n_p*2*sizeof(TaggedIdx), SEEK_SET);
    fread(&idxs, sizeof(TaggedIdx), 2, in);
    if (idxs[0].idx == idxs[1].idx)
      prior->insert(new NodeWithIdx(idxs[0].idx));
    else {
      for (i = 0; i < 2; i++) {
        if (!(nodes[i] = hash->find(idxs[i].idx))) {
          nodes[i] = new NodeWithIdx(idxs[i].idx);
          hash->insert(nodes[i], idxs[i].idx);
        }
        if (idxs[i].tag) {
          hash->erase(idxs[i].idx);
          prior->insert(nodes[i])l
        }
      }
      nodes[0]->parent = nodes[1];
    }
    while (prior->top()->idx == n_n-1) {
      nodes[0] = prior->pop();
      while (nodes[0]->parent)
        nodes[0] = nodes[0]->parent;
      fwrite(&(nodes[0]->idx), sizeof(int), 1, out);
      delete nodes[0];
      n_n--;
    }
  }
}
```

Figure 4: Illustrative code for streaming computation of how to create the root file from the merge file.

are proportional to the combined number of runs of all components that are active at the same time (i.e. including all their already finalized runs). As long as components are small and and short-lived (i.e. do not span too many layers) this may be feasible, although the memory footprint would increase correspondingly. Alternatively, if there are few large components it is possible to operate in multiple passes (like discussed in the previous section) and immediately output the runs of those large components to separate files on disk. The problem cases would be those where many medium sized components are active at the same time or where more large components are active at the same time than there can be open files. However, in these cases there seems little utility in computing a list of voxels in the first place.

7 Discussion

We described a streaming implementation for finding connected components with union-find for extremely large data sets. We demonstrate the effectiveness of our approach with results on large binary grids. We push the scalability from input sizes in the $1{,}000^3$ range (billion of elements) that were possible before to input sizes in the $10{,}000^3$ range (trillions of elements) on similar data and with similar memory requirements. An implementation of our algorithm is available on the Web [1].

Instead of reporting only volume and surface area it is fairly straigh-forward to modify our implementation to also report the bounding box, the center of mass, the moments, or the genus of each component. For multi-signal images it would also be possible to integrate per connected component over the other signals.

By performing multiple streaming passes we can optionally construct large labeled images out-of-core. In the future we hope to use extend our streaming algorithm to include some ideas from [11] so that we can utilize all the computing power of all available cores on modern CPUs.

Addendum: There is another angle to this research that will be included in the final paper. We have streaming isosurface extraction software that takes as input a gigantic scalar volume field (represented either by a regular grid [8] or a by streaming volume mesh [9, 5]) and outputs an isosurface that corresponds to a particular isovalue. Using streaming can discard all the small "bubbles" that we often find in noisy data on-the-fly. Similarly, isolines extracted from a streaming TIN [10] can be directly cleaned from tiny contour lines that are usually not of interest.

References

[1] http://www.cs.unc.edu/˜isenburg/sconnect/streaming_connect.cpp.

[2] http://www.ecse.rpi.edu/Homepages/wrf/.

[3] K. Wu andE. Otoo and K. Suzuki. Two strategies to speed up connected component labeling algorithms. Technical report, 2005. Technical Report, LBNL-59102.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (2nd Edition)*. MIT Press and McGraw-Hill, 2001. Chapter 21: Data structures for Disjoint Sets, pp.498–524.

[5] C. Courbet and M. Isenburg. Streaming compression of hexahedral meshes. In *Computer Graphics Interface'10 Proceedings*, 2010.

[6] W. R. Franklin and E. Landis. Connected components on $1000\times1000\times1000$ datasets. In *16th Fall Workshop in Computational Geometry*, 2006.

[7] R. M. Haralick and L. G. Shapiro. *Computer and Robot Vision Volume I.* Addison-Wesley, 1992. pp.28–48.

[8] M. Isenburg and P. Lindstrom. Streaming meshes. In *Visualization'05 Proceedings*, pages 231–238, 2005.

[9] M. Isenburg, P. Lindstrom, S. Gumhold, and J. Shewchuk. Streaming compression of tetrahedral volume meshes. In *Graphics Interface'06 Proceedings*, pages 115–121, 2006.

[10] M. Isenburg, Y. Liu, J. Shewchuk, and J. Snoeyink. Illustrating the streaming construction of 2d Delaunay triangulations. In *Proceedings of the 2006 Symposium on Computational Geometry*, pages 481–482, 2006.

[11] M. Isenburg, P.Lindstrom, and H. Childs. Parallel and streaming generation of ghost data for structured grids. Computer Graphics and Applications, 2010.

[12] E. N. Landis, T. Zhang, E. N. Nagy, G. Nagy, and W. R. Franklin. Cracking, damage and fracture in four dimensions. *Materials and Structures*, 40(4):357–364, 2007.