LBNL-56864

Optimizing Connected Component Labeling Algorithms

Kesheng Wu, Ekow Otoo and Arie Shoshani

Lawrence Berkeley National Laboratory, Berkeley, CA, USA

ABSTRACT

This paper presents two new strategies that can be used to greatly improve the speed of connected component labeling algorithms. To assign a label to a new object, most labeling algorithms use a scanning step that examines some of its neighbors. The first strategy exploits the dependencies among the neighbors to reduce the number of neighbors examined. When considering 8-connected components in a 2D image, this can reduce the number of neighbors examined from four to one in many cases. The second strategy uses an array to store the equivalence information among the labels. This replaces the pointer based rooted trees used to store the same equivalence information. It reduces the memory required and also produces consecutive final labels. Using an array based instead of the pointer based rooted trees speeds up the connected component labeling algorithms by a factor of 5 $\sim$ 100 in our tests on random binary images.

Keywords: Connected component labeling, Union-Find, optimization

1. INTRODUCTION

Our goal is to speed up the connected component labeling algorithms. Since connected component labeling is a fundamental module in medical image processing, speeding it up improves the turn-around time of many medical diagnoses and procedures.[1–5] Improving these labeling algorithms also benefits other applications in computer vision and pattern recognition.[6–8] Given an image, connected component labeling assigns labels to a pixel such that adjacent pixels of the same features are assigned the same label.

In this paper, we only consider the problem of labeling binary images stored as 2D array of pixels.[9–11] Such a binary image is typically an output from a preceding image analysis step that has identified pixels of specific features. Limiting to 2D binary images allows us to isolate the connected component labeling problem for a more detailed study. The techniques discussed in this paper can be easily be applied to cases where the labeling procedure is integrated into other analysis. To further limit the scope of this paper, we only use 8-connected components of 2D images for our experiments and illustrations.[9–11] The approaches used in this paper should apply to higher dimensional images as well.

In a paper by Suzuki et al.,[11] approaches for connected component labeling are categorized into four groups: (1) methods by repeated passes over the data, (2) methods of two passes over the data, (3) methods using hierarchical tree equivalent representations of the data, (4) parallel algorithms. These approaches share one common step, known as the scanning step, which examines the neighbors that have already been assigned a label to determine a label for the current pixel. The methods in the first group simply repeat this scanning step back and forth until each pixel has a stable label. Most of these methods can be implemented in-place without any additional work space. However, the number of iterations could be arbitrarily high. An efficient variations of this basic approach is an algorithm by Suzuki et al.,[11] which uses an additional array to store some label equivalence information.

A general approach to handle the label equivalence information is to use Union-Find algorithms with pointer based rooted trees.[10, 12, 13] The second group of methods generally take this approach. They scan the image once to assign provisional labels and establish equivalence information, and pass through image again to assign the final labels. An algorithm by Fiorio and Gustedt is a representative we choose from this group.[10] In this paper, we concentrate on strategies to improve the first two groups of methods and will not discuss any image format issues or parallelization issues.

A shortcoming of the pointer based rooted trees is that the time to manage the pointers and the associated dynamic data could easily dominate the whole labeling algorithm.[14] Recently, Chang et al. proposed a new labeling algorithm based on contour tracing, which they show to be more efficient than most known algorithms.[9]

Further author information: Kesheng Wu: E-mail *KWu@lbl.gov*, Telephone 1 510 486 6609. Ekow Otoo: E-mail *ekw@data.lbl.gov*. Arie Shoshani: E-mail *Arie@lbl.gov*.

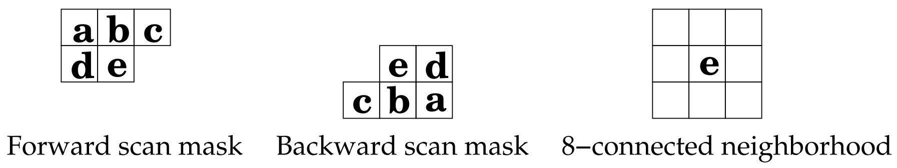

Figure 1. The masks and the neighborhood of pixel **e**. Notice that all the pixels in the forward and backward scan masks are in the neighborhood of pixel **b**.

To improve speed of the labeling procedure, we propose two strategies; the first reduces the number of neighbors examined during scanning steps and the second reduces the cost of Union-Find algorithms. The pattern formed by the neighbors scanned already plus the current pixel is called a *scan mask*,[11] see an illustration in Figure 1. The first strategy comes out of a realization that the neighbors in a scan mask are not independent. When considering 8-connected components, all pixels in the scan mask are neighbors of one of them. With appropriate supporting data structures, only one neighboring pixel is needed to determine the label of a new pixel. This can reduce the number of neighbors examined from four to one. Clearly, this is not the case for every pixel, however, the average number of neighbors scanned is usually less than four. For 2D images, this approach does not apply to the 4-connected components. However, for high dimensional images, the potential gain of exploiting the similar dependency can be very significant.

The second contribution of this paper is that we implement Union-Find algorithms with an array rather than pointers. Our implementation uses less memory than the usual pointer based implementations. More importantly, it significantly reduces the overall execution time of the labeling algorithm. When all data structures fit into memory, the labeling algorithm with array based Union-Find is about five times faster than the similar algorithms with pointers. If the memory is smaller, the difference can be much more dramatic.

This paper does not provide a rigorous analysis of the new strategies. To verify their correctness, we implemented a number of different algorithms with and without our new proposed strategies. In addition, we implemented a version of the algorithm based on contour tracing. Unlike the original one proposed by Chang et al., our implementation does not copy the image into a larger array.[9] In fact, we require no auxiliary storage at all. We also implemented Fiorio's algorithm without the flatten step that comes after scanning every line.[10] The flatten step was crucial in the theoretical analysis, however, in practice, removing it usually speeds up the labeling programs. We also implemented a labeling algorithm that first connects pixels into blocks.[15] On the set of random images we tested, this block scan approach does not outperform the pixel based scans because there are very few large blocks.

The rest of this paper, describes the above mentioned strategies as modifications to existing labeling algorithms. In Section 2, we describe how to modify the scanning phase of the algorithm by Suzuki and others. In Section 3, we describe how to modify Fiorio's algorithm to use an array based Union-Find. In Section 4, we describe a modified version of Chang's algorithm and a block based algorithm similar in spirit to an algorithm by Shima et al..[15] Performance measurements are given immediately following the description of the modified algorithms. A short summary is given in Section 5. Pseudo-code segments of the array-based Union-Find algorithms are given in the appendix.

2. REDUCING THE NUMBER OF NEIGHBORS SCANNED

In this section, we briefly review Suzuki's algorithm and describe a decision tree to exploit the dependencies among the neighbors. The decision tree allows us to reduce the number of neighbors examined during the scanning steps of Suzuki's algorithm. We will also describe the test setup and performance measurements.

2.1. Outline of Suzuki's Algorithm

To show how to use the decision tree, we describe a particular example, i.e., applying it to the algorithm proposed by Suzuki et al., referred to subsequently simply as *Suzuki's Algorithm*.[11] Suzuki's algorithm performs repeated passes over a binary image $b(x, y)$, alternating in forward and backward directions. The binary image $b(x, y)$ consists of pixel values F_O and F_B indicating foreground object and background values of the image where $F_O > F_B$. Like in the simple scanning step, it accesses the pixels sequentially in raster scan order using the scan mask shown in Figure 1. At the same time, it also updates a one-dimensional table called the *label connection table* T to memorize label equivalence information. Similar local operations in the backward scan order are performed using the scan mask shown in Figure 1. A key feature of the Suzuki's algorithm is the use of the label connection table to reduce the number forward and backward scans. The assignment of provisional labels not only propagates on the connected components but also in the label connection table.

In the scanning step, Suzuki's algorithm determines the provisional label of a pixel at position (x, y) (marked **e** in the scan masks of Figure 1) as follows:

$$g(x,y) = \begin{cases} F_B & \text{if } b(x,y) = F_B, \\ m, (m = m+1) & \text{if } \forall (i,j) \in M_s, g(x-i, y-j) = F_B, \\ T_{min}(x,y) & \text{otherwise, where} T_{min}(x,y) = \min\{T[g(x-i,y-i)] | (i,j) \in M_s\}. \end{cases}$$

where m is initialized to 1, $(m = m + 1)$ increments the label m and M_s denotes the region of the mask except **e**. The label connection table is updated, simultaneously with the assignment of the provisional labels as follows:

$$\begin{cases} \text{no-operation} & \text{if } b(x,y) = F_B, \\ T[m] = m & \text{if } \forall (i,j) \in M_s, g(x-i,y-j) = F_B, \\ T[g(x-i,y-j)] = T_{min}(x,y) & (i,j) \in M_s, g(x-i,y-j) = F_O. \end{cases}$$

Operations on the backward scans are similarly defined with appropriate modifications to the formulas. See Suzuki et al.[11] for details. To save space, we refer the readers to the original papers for other algorithms. The basic version of the algorithm that always examines four neighbors is labeled as 'S4' in the later tests. Suzuki et al. also presented an inproved version that examine two neighbors in most cases.[11] We have implemented a version that mimics their proposed behavior using two if-blocks rather than table lookups. In later tests, we refer to this version as 'S2'.

2.2. Decision tree

While determining the provisional label of a pixel **e**, one may always examine four neighbors, however this is not necessary as Suzuki et al. have indicated. In fact it is possible to reduce the number of neighbors examined to one. In Figure 1, it is clear that all the neighbors in the scan masks are neighbors of **b**. If there is enough equivalence information to access the correct label of **b**, there is no need to examine the rest of the neighbors. Without proving the correctness, we simply present the decision tree as a way to organize the scan operation in a specific order.

Instead of examining all four neighbors of **e**, i.e., **a**, **b**, **c** and **d**, we propose to examine the neighbors according to a decision tree, see Figure 2. Let **L** denote the 2D array storing the labels and let **E** denote equivalence array, which was termed the label connection table T in Suzuki's algorithm.[11] The three functions used by this decision tree are defined as follows.

1. The one-argument copy function, copy(a), contains one statement:

   L(e) = **E**(**L**(a)).

2. The two-argument copy function, copy(c, a), contains three statements:

   L(e) = min(**E**(**L**(c)), **E**(**L**(a))), **E**(**L**(c)) = **L**(e), **E**(**L**(a)) = **L**(e).

3. The new label function sets m as **L**(e), appends m to array **E**, and increments m by 1.

Our modified version of Suzuki's algorithm does not reproduce the original algorithm. In particular, the provisional label for **c** may be different from that of **b** if they both pixels have value 1. In this case, using the decision tree would miss the opportunity to update the equivalence array. This may cause the modified version to use more iterations than the original version. However, they usually use the same number of iterations and and the modified version is faster per iteration than the original version. In the next section, we present a different version these copy functions that would fully capture the equivalence information. In that case, we only need to scan the image once.

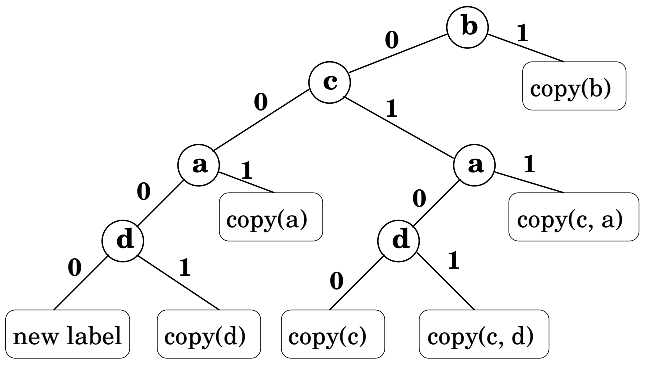

Figure 2. The decision tree used in scanning for 8-connected neighbors.

Table 1. Information about the test machines.

CPU type	Clock (MHz)	Cache (KB)	Memory (MB)	OS	Compiler
UltraSPARC	450	4096	4096	Solaris 8	Forte workshop 7
Pentium M	1500	512	512	Windows XP	Visual Studio .NET
Pentium 4	2200	512	512	Linux 2.4	gcc 3.3.3

2.3. Test setup

To measure the performance of the various labeling algorithms, we use random 2D binary images. If the majority of the pixels are 0, we initializes all pixels to 0 and then uses a uniform random number generator to pick a specified number of pixels to set them to 1. Conversely, if the majority are 1, it initializes all bits to 1 and sets some random pixels to 0. These test images are generally harder to label than real medical images. However, they make reasonable test images to measure the performance of connected component labeling algorithms.[10, 11]

To ensure that our measurements are not biased by a particular hardware environment, we have selected to run the same test cases on three different machines listed in Table 1. On each machine, we also choose to use different compilers. These choices should make the performance differences more likely due to algorithmic differences rather than other factors.

Majority of the test images have either 5,000 x 6,000 pixels or 10,000 x 10,000 pixels. They vary in the number of pixels that are 1. In all tests, the input image is overwritten with labels. All algorithms leave 0 pixels as 0 and assign a positive integer as the label of each pixel that was 1 on input. In later discussions, the *work space* refers to the space required in addition to this image array. All time values reported later are elapsed time in seconds. All reported time values are from a single test run and therefore may contain random variations due to both randomness in the test images and random activities on the test machines. Altogether there are 214 test cases, with 30 images with 5,000 x 6,000 pixels, 64 images with 8,000 x 8,000 pixels, 100 images with 10,000 x 10,000 pixels and 20 images of different sizes but constand fraction of pixels that are 1. When reporting total time or average time, we include all 214 test cases.

2.4. Performance of Suzuki's algorithms

Figure 3 shows the timing measurements of the three versions of Suzuki's algorithm on three different machines and Table 2 shows the total time used by each algorithm on all test images. The speedup reported in Table 2 is simply the total time used by version S4 divided by the total time used by the modified versions. Our version using the decision tree is labeled as S1. On two out of three test machines, the speedup of S1 is nearly two. The improved version S2 is faster than the basic version S4 as expected. S1 is about 20% faster than S2 on the average.

Among the four neighbors **a**, **b**, **c** and **d** in the scan mask, if none of them is 1 or only one pixel is 1, using the decision tree will not save any time. However, if there are more than one pixel that is 1, using the decision tree may save time. In random images, if there are more than one quarter of pixels that are 1, using the decision tree is expected to save time. In

Table 2. Total time of all test cases used by two versions of Suzuki's algorithm and the average speedup of the modified version.

	S4	S2		S1	
	Time (sec)	Time (sec)	Speedup	Time (sec)	Speedup
UltraSPARC	8570	5280	1.6	4490	1.9
Pentium M	1630	1280	1.3	1090	1.5
Pentium 4	1510	1020	1.5	842	1.8

Figure 3, the modified algorithm uses less time than the original version if the pixels in the connected components is more than 8×10^6 for 5,000 x 6,000 images or 2×10^7 for 10,000 x 10,1000 images. This agrees with expectation.

Since using the decision tree in the scanning step never costs more time than S2 and S4, we will always use the decision tree in the rest of this paper.

3. IMPROVING UNION-FIND WITH AN ARRAY

In this section, we consider two options to modify the connected component labeling algorithm by Fiorio and Gustedt.[10] The particular algorithm used here is Algorithm 2 in their paper. The first option is to replace the pointer based rooted trees for Union-Find with an array. The second option is to remove the flatten operation after scanning each line. This section contains three subsections, the first two describe the above two options and the last describes the performance of four different implementations of Fiorio's algorithm.

3.1. Array based Union-Find

We have mentioned that the equivalence array **E** used in Suzuki's algorithm does not capture all equivalence information. One way to improve the algorithm is to fully capture the equivalence information. The equivalence information is often represented in as rooted trees in a Union-Find algorithm. Representing the same information in an array provides an alternative for Union-Find algorithms.[12, 14, 16–19]

An arbitrary forest of trees can be suitably relabeled and represented with an array, see an example in Figure 4. First, nodes of these trees need to be relabeled with consecutive integers. In the example shown, the first integer used is 0 to make it directly usable as an index to an array in C/C++. This relabeling of nodes can be done with any ordering of the nodes. To make it easier to produce consecutive labels, we use an ordering that is equivalent to a pre-order traversal of the trees, which labels a node before its descendants. After relabeling, an element of the equivalence array represents a node and the value of the array element is the index of its parent. The array can be interpreted as the same rooted trees.

Since the trees used in Union-Find typically do not contain links from a parent to its children, the above relabeling procedure can not be performed after the rooted trees have been created. However, the provisional labels generated during the scanning step are consecutive integers that can be used as the indices to array **E**. This array **E** would have exactly the same size as the label connections table T used by Suzuki et al..[11]

It is relative straightforward to implement all the algorithm associated with Union-Find using an array instead of a set of pointer based rooted trees. The detailed algorithms are given in the Appendix A. Next, we briefly describe how to use it with the decision tree shown in Figure 2.

The leaf nodes of the decision tree contains three functions: a new label function, a one-argument copy function and a two-argument copy function. The new label function and the one-argument copy function remain the same as before. The two-argument copy function, copy(c, a), now contains one statement:

L(e) = union(**E**, **L**(c), **L**(a)).

After the scanning step to assign provisional labels and building the equivalence array, Fiorio's algorithm uses the trees to assign the final labels to all pixels in the connected components. Using the array based Union-Find that ensures the root of each tree has the minimal label, we can implement a procedure, called *flattenLabel* in the appendix, that not only flattens the trees but also assigns consecutive final labels to the components in one single pass of the array.

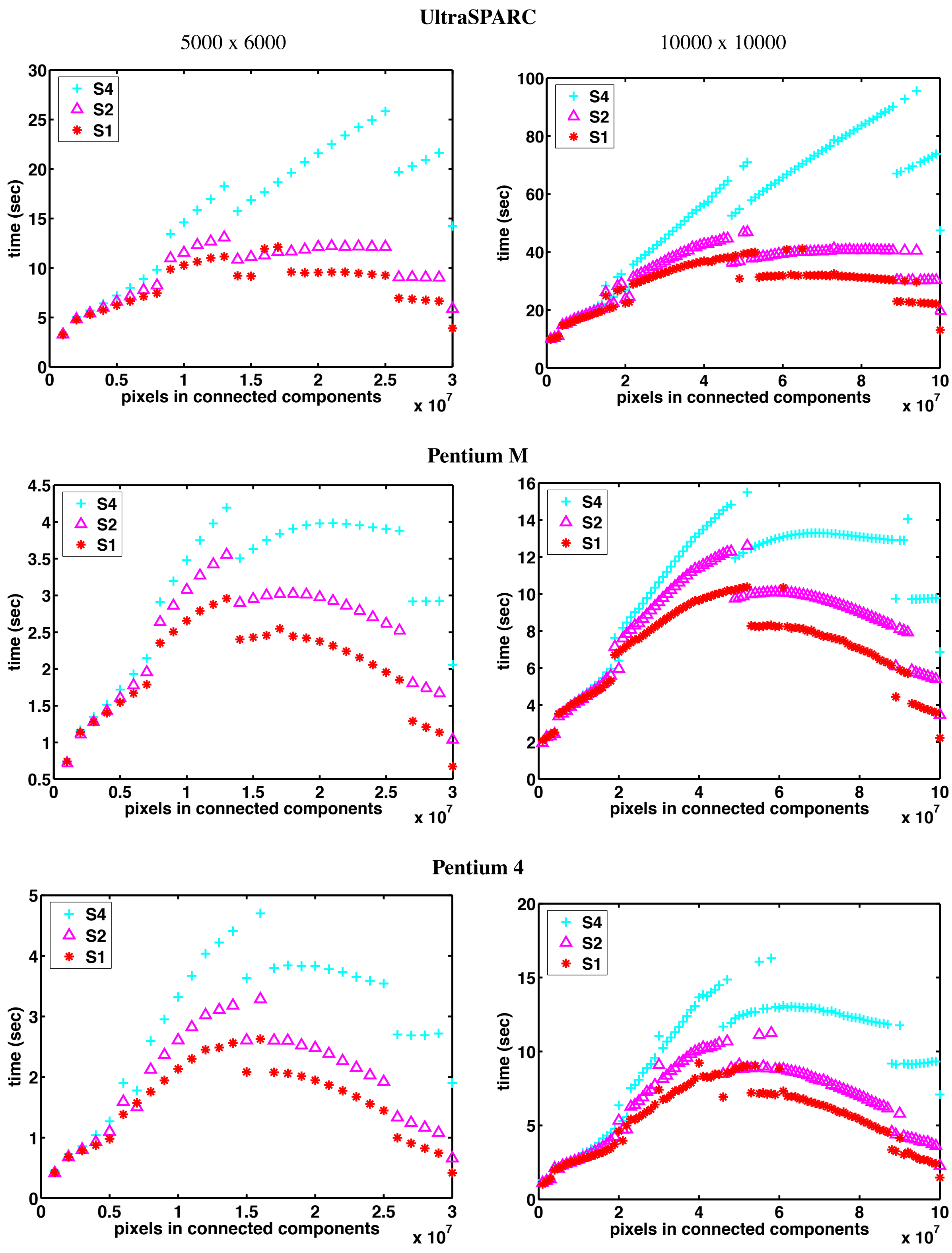

Figure 3. Timing results of two implementations of Suzuki's algorithm, where the modified version uses the decision tree shown in Figure 2 to reduce the number of neighbors examined.

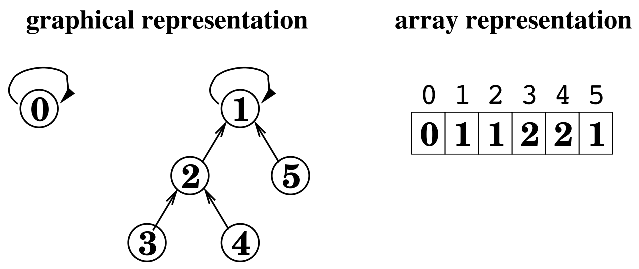

Figure 4. Any forest can be suitably relabeled and represented as an array. This example shows a forest with two trees and six nodes that is represented as an array.

3.2. Removing extra flatten operations

Algorithm 2 proposed by Fiorio and Gustedt in their paper[10] invokes a flatten operation after scanning each line. This flatten operation calls the find-compress function on every provisional label used in the current line. The same algorithm should run correctly without the flatten operations. When a particular provisional label is used later, the find-compress function would do the same job that would be performed in the flatten operation. Clearly, we don't expect this strategy to have very much impact on the overall performance of the labeling algorithm. However, in order to support the flatten operations, the provisional labels used in the current line need to be stored separately. The data structures for these labels is created and deleted for as each line of the image is scanned. Removing the flatten operations also removes these dynamic data, which could speed up the labeling algorithms.

3.3. Performance of four versions of Fiorio's algorithm

We implemented four variations of Fiorio's algorithm, the original algorithm, a modified version without any flatten operation in the scanning step, a version with the array based Union-Find, and a version with the array based Union-Find and without any flatten operation in the scanning step. In Figure 6 and Table 3, the four algorithms are labeled as 'Original', 'No Flatten', 'Array UF' and 'Array UF, No Flatten'. We do not show the timing results on **Pentium M** in Figure 6 to save space. As seen in Figure 3, timing results on **Pentium M** follow that of **Pentium 4** quite closely, with the former typically larger then the latter because the particular **Pentium 4** processor has a faster processor.

The total time of all test cases is shown in Table 3. The speedup shown is compared with the original algorithm. We see that in most cases, the speedup value is greater than one which indicate the modified version is faster than the original Fiorio's algorithm. The only exception is the version with pointer based rooted trees and without flatten operations, which shows an average slow down of about 7% on the **Pentium 4** machine. From Figure 6 we see that in these unusual cases where the modified version takes longer than the original version, both of them take much longer than Suzuki's algorithm. Figure 5 shows the number of provisional labels and the final labels assigned by Fiorio's algorithms. It is clear that in these unusual cases, the number of provisional labels is large, around 6 - 8 million. The array to store the labels requires about 400 MB. The pointer based rooted trees take four words per provisional label[*], which may take another 100 MB. The total memory requirement is close to the main memory size on the two Pentium based machines. In these cases, most of the execution time is spent on swapping. The observed slow down might also be specific to the **Pentium 4** machine because the same slow down is not observed on the **Pentium M** machine. In fact, when all data fit in memory as on the **UltraSPARC** machine, removing the unnecessary flatten operations increases the overall execution by about 55%.

The fastest version is the one with the array based Union-Find and without the extra flatten operations. When the test problem size fit in memory, the average speedup is more than five. In cases where the image size plus work space is close to the computer memory size, the speedup can be in the hundreds.

4. TWO MORE MODIFIED ALGORITHMS

In this section, we describe two more connected component labeling algorithms, one based on a contour tracing algorithm[9] and the other based on a block scan algorithm.[15] Our version of Chang's algorithm overwrites the input image array with

[*]The four words are: (1) one for the label value, (2) one for pointer to a parent, (3) one for the weight of the tree (required for weighted union algorithm[13]), and (4) one to point the data structure for a node of the rooted trees.

Table 3. Total time of all test cases used by four versions of Fiorio's algorithm and the average speedup of the modified versions.

	Original	No Flatten		Array UF		Array UF, No Flatten	
	time (sec)	time	speedup	time	speedup	time	speedup
UltraSPARC	11100	7160	1.55	4000	2.78	2040	5.4
Pentium M	27100	25000	1.08	1320	20.5	400	67.6
Pentium 4	5900	6340	0.93	1480	3.99	340	17.4

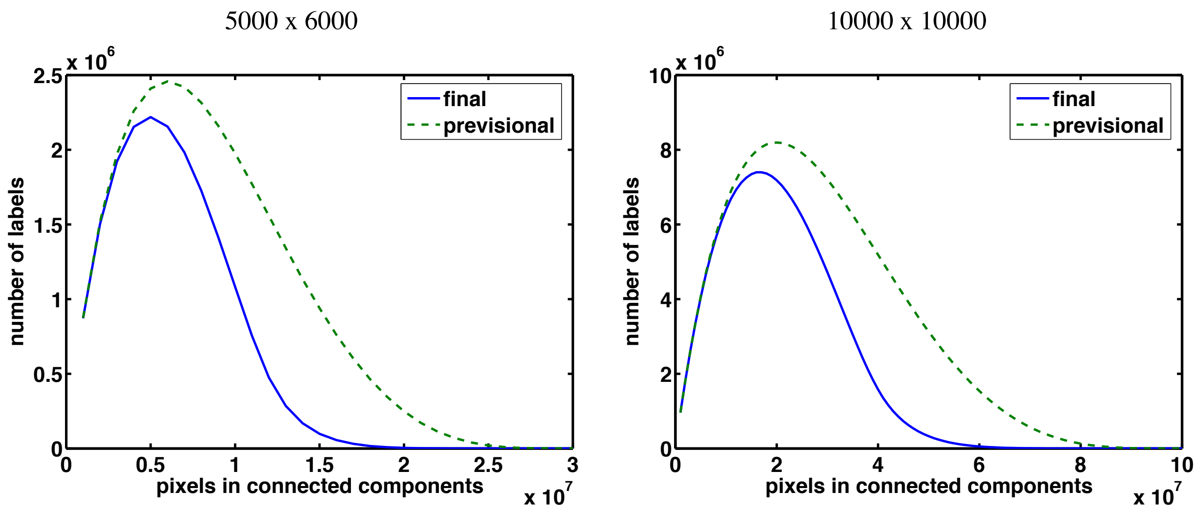

Figure 5. Number of final labels and provisional labels of the test images. Since all algorithms scan the images in the same order, the same number of provisional labels are used.

labels and does not use any additional storage. Our block scan algorithm builds blocks from input image and uses the array based Union-Find. This section has three subsections. The first two subsections briefly describe these two algorithms. The last subsection shows performance results of these two algorithms against the best versions of Suzuki's algorithm and Fiorio's algorithm.

4.1. An in-place contour tracing algorithm

The basic principle in contour tracing is to examine all neighbor pixels in a fixed order so that a neighbor on the boundary of the same connected component is visited first.[9] The algorithm described by Chang et al. uses a number of arrays as the work space. It is straightforward, although a little laborious to implement the same algorithm without the extra work space. What is needed is more if-statements to deal with the pixels on the boundaries of a image. Potentially, these extra if-statements would make the algorithm run slower than the original version if everything were to fit in memory. However, for large images like those with 10,000 x 10,000 pixels, the extra work space required by original algorithm would not fit in memory on two of the three test machines. This would make the original algorithm much slower than our version.

There are two main tricks we use to implement the in-place algorithm. One trick is to label the components starting from 2. We assume the input image contains only 0 and 1. Using the value 2 as the minimal label allows us to easily distinguish pixels that have been assigned a label and a pixel that have not. The other trick is to treat the integer in the image array as signed integer during contour tracing. This allows to mark some of the background pixels (i.e., those with 0 value) with the value -1 as suggested by Chang et al..[9] We could have used another value instead of -1, however, it appears to be easier to program the algorithm using -1. After all the pixels have been scanned, we need a second pass to reset all pixels marked -1 to 0. We also take the opportunity to reduce all positive labels by 1 so that the final labels starts with 1.

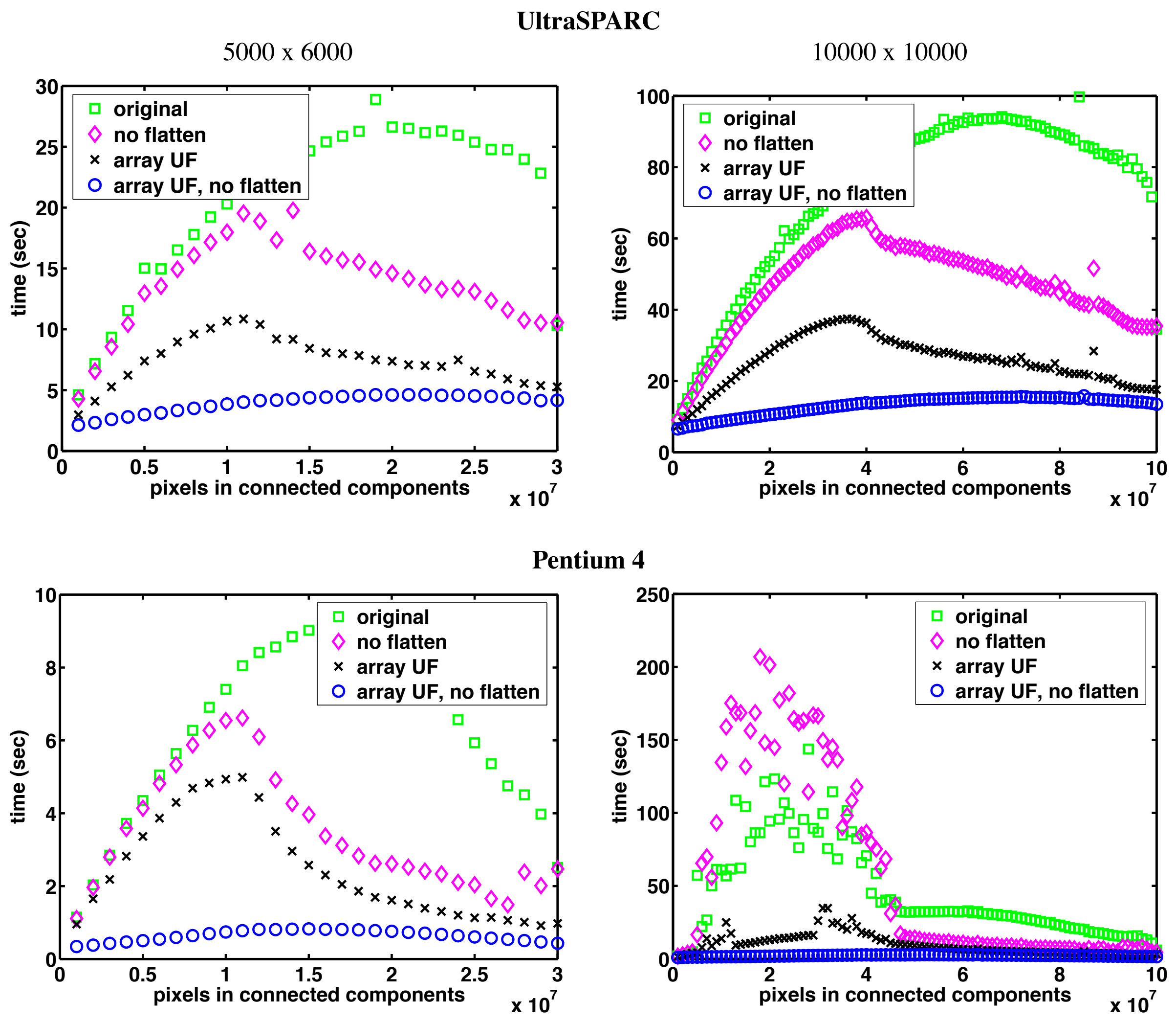

Figure 6. Timing results of four implementations of Fiorio's algorithm. The algorithms with array based Union-Find (marked 'Array UF') clearly require less time the those with pointers.

4.2. A block scan algorithm

Working with blocks instead of pixels has a number of obvious advantages if there are block structures in the image.[15] If most blocks contain more than two points, then a block representation would require less space. There are less objects to scan and the scanning step may use less provisional labels as well. In our implementation, we scan each line as usual, however, when scanning a line, we first turn consecutive pixels into blocks and store the starting position and the end position of each block along with its provisional label. This block data structure is used while scanning the next line and is discarded afterword.

In this algorithm, it may be necessary for us to perform a union on many labels. In this case, we use the function `find` to compute the root with the minimal label, and then use the function `set` to mark all the provisional labels involved equivalent to the new root. This should be more efficient than calling the pair-wise union function.

4.3. Comparing the best options

We measured the performance of the two above algorithms against the best variations of Suzuki's algorithm and Fiorio's algorithm. In Figures 7 and 8, our implementation of the contour tracing algorithm is marked as 'Chang'. The algorithm marked 'Suzuki' is the modified version that uses the decision tree shown in Figure 2. The algorithm marked 'Fiorio' is the one with the array based Union-Find and without extra flatten operations, i.e., the one marked 'Array UF, No Flatten' in Table 3 and Figure 6.

In Figure 7, each plot shows tests on images of the same size but with varying number of 1s. In Figure 8, each plot shows tests on images with a fixed fraction (1/6) of pixels being 1. In the first case, the total number of pixels in the images is fixed for each plot and in the second case, the total number of pixels is strictly proportional to the number of pixels in the connected components. All algorithms used are known to scale linearly as the number pixels in the image increases. We clearly see that the timing values follow straight lines in Figure 8. This confirms that our implementations have the expected linear property.

In Figures 7 and 8, our modified version of Fiorio's algorithm requires the least amount of time in most test cases. The modified version of Suzuki's algorithm takes about the same amount of time as the in-place version of Chang's algorithm in many test cases. The Suzuki's algorithm often uses twice as much time as Fiorio's algorithm and the block scan algorithm, because Suzuki's algorithm scan the image four times while Fiorio's algorithm and the block scan algorithm only access the image twice. On images with mostly 1s, the block scan algorithm and Chang's algorithm may be the fastest. We anticipate the block scan algorithm to perform even better if the input data is already in block structure, such as those output from a search algorithm.[20]

5. SUMMARY AND FUTURE WORK

We studied a number of strategies to speed up connected component labeling algorithms. The two most effective strategies are using a decision tree to minimize the number of neighbors scanned and using an array to implement Union-Find algorithms. The first can reduce the scan time by about half compared to the basic strategy, and the second can reduce the total execution time by a factor of five or more. If the image size is large, the speedup could be more than a factor of 100.

The algorithms are tested using random binary images, it would be more interesting to see the performance on application data. It might be also interesting to provide more rigorious analyses of the algorithms presented.

APPENDIX A. THE ALGORITHMS FOR ARRAY BASED UNION-FIND

This appendix presents the algorithms for array based Union-Find in pseudo code with C++ syntax. In particular, we use `std::vector` from the Standard Template Library to store the equivalence array **E**.

```
// Combine two trees containing node i and j.
// Return the root of the union.
unsigned union(std::vector<unsigned>& E,
               unsigned i, unsigned j) {
    unsigned root = find(E, i);
    if (i != j) {
        unsigned rootj = find(E, j);
        if (root > rootj) root = rootj;
        set(E, i, root);
        set(E, j, root);
    }
    return root;
}

// Find the root of node ind.  Compress the path.
unsigned findCompress(std::vector<unsigned& E,
                      unsigned ind) {
    unsigned root = find(E, ind);
    set(E, ind, root);
    return root;
}
```

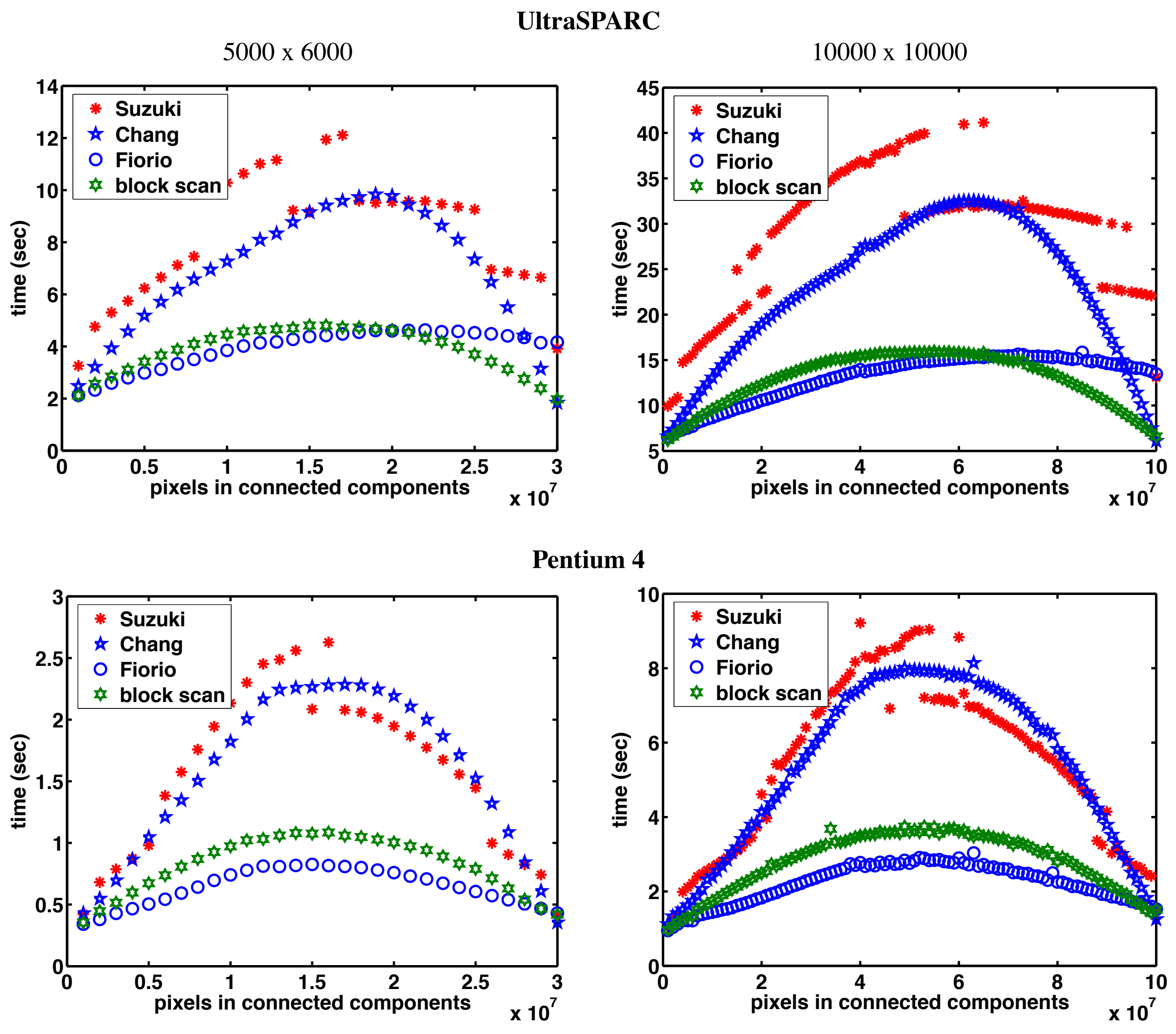

Figure 7. Timing results of four different algorithms, the modified version of Suzuki's algorithm, the version of Fiorio's algorithm with the array based Union-Find and without flattening, the in-place version of Chang's algorithm, and a block scan algorithm.

```
// Find the root of the tree from node ind.
unsigned find(const std::vector<unsigned>& E,
              const unsigned ind) {
    unsigned root = ind;
    while (E[root] < root)
        root = E[root];
    return root;
}

// Set all nodes to point to a new root.
void set(std::vector<unsigned>& E,
         unsigned ind, unsigned root) {
    unsigned i = ind;
    while (E[i] < i) {
        unsigned j = E[i];
        E[i] = root;
```

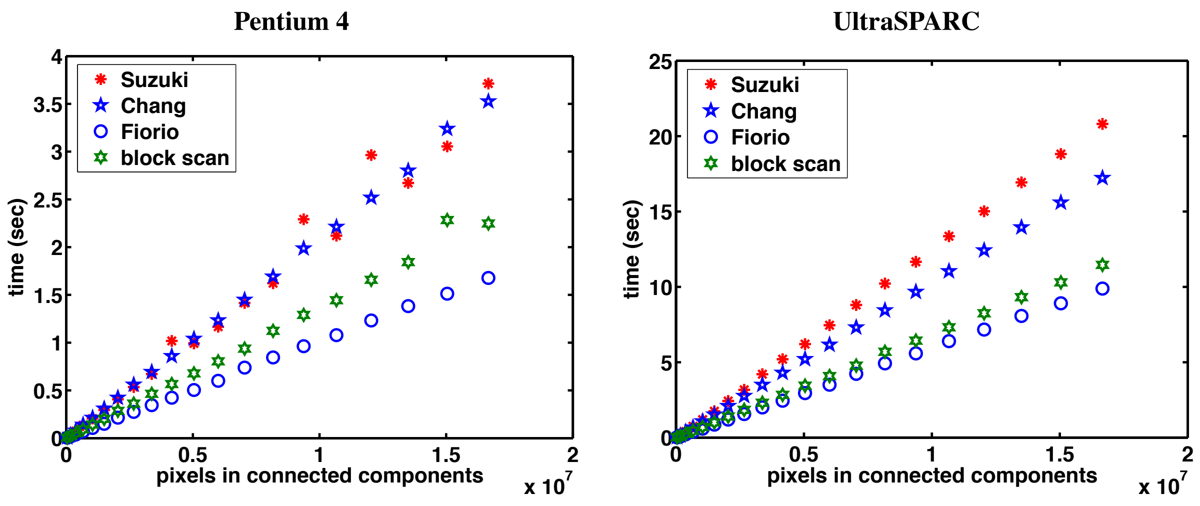

Figure 8. Timing results of four algorithms on different size images with fixed density (1/6).

```
            i = j;
        }
        E[i] = root;
}

// Flatten the Union-Find tree.
void flatten(std::vector<unsigned>& E) {
    for (unsigned i = 0; i < E.size(); ++ i)
        E[i] = E[E[i]];
}

// Flatten the Union-Find tree and relabel the components.
void flattenLabel(std::vector<unsigned>& E) {
    unsigned label = 0;
    for (unsigned i = 0; i < E.size(); ++ i)
        if (E[i] < i) {
            E[i] = E[E[i]];
        }
        else {
            E[i] = label;
            ++ label;
        }
}
```

REFERENCES

1. J. Freixenet, X. Muñoz, D. Raba, M. Marti, and X. Cufí, "Yet another survey on image segmentation: Region and boundary information integration," in *Proceedings of the European Conference on Computer Vision (ECCV 2002)*, pp. 408–422, 2002.
2. B. van Ginneken, B. M. ter Haar Romeny, and M. A. Viegever, "Computer-aided diagnosis in chest radiography: A survey," *IEEE Transcations on Medical Imaging* **20**(12), pp. 1228–1241, 2001.
3. J. Maintz and M. Viergever, "A survey of medical image registration," *Medical Image Analysis* **2**(1), pp. 1–36, 1998.
4. T. McInerney and D. Terzopoulos, "Deformable models in medical image analysis: A survey," *Medical Image Analysis* **1**(2), pp. 91–108, 1996.
5. T. W. Nattkemper, "Automatic segmentation of digital micrographs: a survey," in *Proceedings of MEDINFO 2004, San Francisco*, Americal Medical Informatics Association, 2004.
6. D. H. Ballard, *Computer Vision*, Prentice-Hall, Englewood, New Jesey, 1982.

7. N. Ezquerra, S. Capell, L. Klein, and P. Duijves, “Model-guided labeling of coronary structure,” *IEEE Transcations on Medical Imaging* **17**(3), pp. 429–441, 1998.
8. H. Samet, “Connected component labeling using quadtrees,” *J. ACM* **28**(3), pp. 487–501, 1981.
9. F. Chang, C.-J. Chen, and C.-J. Lu, “A linear-time component-labeling algorithm using contour tracing technique,” *Comput. Vis. Image Underst.* **93**(2), pp. 206–220, 2004.
10. C. Fiorio and J. Gustedt, “Two linear time union-find strategies for image processing,” *Theor. Comput. Sci.* **154**(2), pp. 165–181, 1996.
11. K. Suzuki, I. Horiba, and N. Sugie, “Linear-time connected-component labeling based on sequential local operations,” *Comput. Vis. Image Underst.* **89**(1), pp. 1–23, 2003.
12. M. B. Dillencourt, H. Samet, and M. Tamminen, “A general approach to connected-component labeling for arbitrary image representations,” *J. ACM* **39**(2), pp. 253–280, 1992.
13. R. E. Tarjan, “Efficiency of a good but not linear set union algorithm,” *J. ACM* **22**(2), pp. 215–225, 1975.
14. C. Fiorio and J. Gustedt, “Memory management for union-find algorithms,” in *Proceedings of 14th Symposium on Theoretical Aspects of Computer Science*, pp. 67–79, Springer-Verlag, 1997.
15. Y. Shima, T. Murakami, M. Koga, H. Yashiro, and H. Fujisawa, “A high-speed algorithm for propagation-type labeling based on block sorting of runs in binary images,” in *Proceedings of 10th International Conference on Pattern Recognition*, pp. 655–658, 1990.
16. S. Alstrup, A. M. Ben-Amram, and T. Rauhe, “Worst-case and amortised optimality in union-find,” in *Proc. 31th Annual ACM Symposium on Theory of Computing (STOC’99)*, pp. 499–506, ACM Press, 1999.
17. B. Bollobás and I. Simon, “On the expected behavior of disjoint set union algorithms,” in *STOC ’85: Proceedings of the seventeenth annual ACM symposium on Theory of computing*, pp. 224–231, ACM Press, 1985.
18. J. Doyle and R. L. Rivest, “Linear expected time of a simple union-find algorithm.,” *Inf. Process. Lett.* **5**(5), pp. 146–148, 1976.
19. H. N. Gabow and R. E. Tarjan, “A linear-time algorithm for a special case of disjoint set union,” in *STOC ’83: Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pp. 246–251, ACM Press, 1983.
20. K. Wu, W. Koegler, J. Chen, and A. Shoshani, “Using bitmap index for interactive exploration of large datasets,” in *Proceedings of SSDBM 2003*, pp. 65–74, 2003. A draft appeared as tech report LBNL-52535.