1988

A Study of Connected Component Labeling Algorithms on the MPP

Susanne E. Hambrusch
Purdue University, seh@cs.purdue.edu

Lynn TeWinkel

Report Number:
87-721

A STUDY OF CONNECTED COMPONENT LABELING ALGORITHMS ON THE MPP

Susanne Hambrusch
Lynn TeWinkel

CSD-TR-721
January 1988

A Study of Connected Component Labeling Algorithms on the MPP

Susanne Hambrusch and Lynn TeWinkel

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907, USA.

CSD-TR-721
January 1988

Abstract.

In this paper we consider the problem of labeling the connected components of a binary image. We describe three different algorithms and variations thereof which we implemented on NASA's MPP. These algorithms include a simple label propagation algorithm, a version of Levialdi's algorithm, and an asymptotically optimal divide-and-conquer algorithm. We discuss the performance of these algorithms on the MPP and provide insight into how special hardware features of the MPP influence the design of parallel algorithms. We also address the issue of how to handle images that are larger than the processor array of the MPP (i.e.; images larger than 128×128).

This work was supported by the Office of Naval Research under Contracts N00014-84-K-0502 and N00014-86-K-0689, and by the National Science Foundation under Grant DMC-84-13496.

I. Introduction

The Massively Parallel Processor (MPP) is a Single Instruction, Multiple Data Stream (SIMD) parallel processor consisting of a 128×128 mesh of processing elements, each of which is a bit serial processor with 1024 bits of storage [B]. The MPP was originally intended for image processing and analysis problems, but has proven to be suitable for a number of other scientific applications [F]. In this paper we consider the connected component labeling problem of binary images, a fundamental problem in image analysis. We discuss three different algorithms for this problem and their implementation and performance on the MPP. All algorithms are programmed in a high level Parallel Pascal [R1].

In the connected component problem the 1-pixels (i.e., the pixels of value 1) define patterns to be analyzed and the 0-pixels serve as the background in the image. Every 1-pixel is assigned an integer label so that 1-pixels x and y have the same label if and only if they are in the same connected component. Two forms of connectivity, 4- and 8-connectivity, are commonly considered. Two 1-pixels x and y are 4-connected if and only if there is a path from x to y consisting of horizontal and vertical segments and every position traversed by the path contains a 1-pixel. Two 1-pixels x and y are 8-connected if diagonal, as well as horizontal and vertical segments, can be used in a path. Our algorithms were implemented for 8-connectivity, but they can easily be changed to handle 4-connectivity.

Because of its importance as a mid-level task in computer vision, numerous sequential and parallel algorithms have been proposed for the connected component labeling problem. We briefly discuss some parallel algorithms relevant to our work. Throughout, we measure space in terms of the number of bits needed per processor. Assume a binary image of size $n \times n$ is stored, one pixel per processor, in a mesh of size $n \times n$. The most straightforward parallel algorithm for a mesh is a propagation algorithm in which every 1-pixel starts off with a unique label. At each successive iteration, each 1-pixel updates its label to the minimum among its label and its neighbor's labels. At the conclusion of the algorithm, all 1-pixels have as their label the minimum label of any 1-pixel in the same connected component as themselves. This algorithm requires either $\Theta(n^2)$ or $O(n^3)$ time and uses $\Theta(\log n)$ space per processor. Levialdi [L]

$f(n)=O(g(n))$ if for some constant $c \in R^+$ and some integer n_0, $f(n) \leq cg(n)$ for all $n \geq n_0$; $f(n)=\Theta(g(n))$ if for some constant $c \in R^+$ and some integer n_0, $f(n) \geq cg(n)$ for all $n \geq n_0$.

developed an elegant algorithm for counting the number of connected components in $O(n)$ time using $\Theta(\log n)$ space. His algorithm can easily be extended to label the pixels of the connected components by using an additional $O(n)$ space [CSS]. In their paper, Cypher, Sanz, and Snyder [CSS] also present an algorithm based on Levialdi's that uses $O(n)$ time and $\Theta(\log n)$ space per processor. Nassimi and Sahni [NS] and Miller and Stout [MS] developed algorithms based on divide-and-conquer techniques that run in $\Theta(n)$ time and use $\Theta(\log n)$ space, respectively. However, their algorithms require sorting and complex data movements. Using techniques of a systolic flavor described in [H], one can achieve the same asymptotically optimal time bounds without using sorting, using only simple data movements.

The three algorithms we implemented are:

(1) a simple propagation algorithm which we refer to as algorithm SIMPLE,

(2) Levialdi's algorithm extended to labeling and using an additional $O(n)$ space, which we call algorithm LEV, and

(3) an asymptotically optimal divide-and-conquer algorithm based on techniques developed in [H] and which we refer to as algorithm 4QUAD.

We also implemented variations of algorithms SIMPLE and 4QUAD which are described in Section II. All of our algorithms handle unrestricted images. We thus did not compare our algorithms with the one reported in [R3] which was originally designed for images without holes.

We refer to [B] and [R2] for details of the architecture underlying the MPP. Features worth mentioning are the staging memory which is used to move data between a host computer and the MPP processor array and which is also capable of reformatting and storing data. Furthermore, the MPP has a feature which speeds up the computation of statistics performed on the entire array [GAC]. The array is divided up into 16 sub-arrays of size 32×32. Statistics are computed for each sub-array independently and the 16 sub-results are combined using special hardware.

On the data that we used algorithm LEV performed better than any of the other algorithms. For a large number of images (namely images with a diameter less than 10* 128) algorithm SIMPLE and its variation outperform algorithm 4QUAD and its variation. This is due to

the fact that the additional hardware for performing statistics available on the MPP improves the theoretical running time of algorithm SIMPLE and that a mesh of size 128×128 is, for most images, not large enough to reflect the asymptotic optimality of algorithm 4QUAD. In Section II we describe the algorithms in more detail and in Section III we discuss the input data used. The performance of the algorithms is discussed in Section IV. Section V discusses how to handle images larger than 128×128.

II. The Algorithms

This section describes the three connected component algorithms and their variations that were implemented on the MPP. For all algorithms the input consists of an $n \times n$ image of 1's and 0's. In the MPP implementations we have $n=128$. We assume throughout that the image is stored in a boolean array A. The output consists of the array L, where $L[i,j]$ contains the component number of the component containing the pixel at position $[i,j]$, if $A[i,j] = 1$. Each algorithm assumes that every processor contains a unique ID and we use the index of a processor in row-major numbering order as its ID. The array L is initialized with $L[i,j]$ = ID if $A[i,j] = 1$ and $L[i,j]$ = MAX (where MAX is an integer value not used to label a connected component), if $A[i,j] = 0$.

1. Algorithm SIMPLE

Algorithm SIMPLE updates the connected component label $L[i,j]$ of every 1-pixel to be the minimum of $L[i,j]$ and the labels in the eight processors adjacent to processor $[i,j]$. The algorithm runs until no further changes occur in array L. The final label of a connected component equals the smallest ID of any 1-pixel in the component.

Let the diameter d of a connected component be defined as $\max_{x,y} d(x,y)$, where x and y are any two 1-pixels in the connected component and $d(x,y)$ is the length of the shortest 8-connected path between x and y. The number of iterations executed by algorithm SIMPLE equals the largest diameter of any connected component in the image. The maximum diameter is n^2, and it is achieved, for example, by a large spiral-like connected component. If the termination test is done after every iteration, the running time is $O(n^3)$ (since the termination test requires $\Theta(n)$ time on a mesh without additional hardware). If the algorithm runs for n^2

iterations without performing termination tests, the running time is $\Theta(n^2)$, which is more than actually needed for most images. Our first version of SIMPLE, which from now on we call SIMPLE1, performs the termination test after every iteration. The observed running time of this version of SIMPLE does not appear to be close to n^3, since an average image is unlikely to contain large spirals and the additional hardware on the MPP executes the termination test relatively fast.

The second version of algorithm SIMPLE, which we call SIMPLE2, makes use of the assumption that most input data contains connected components of sufficient size so that many iterations of algorithm SIMPLE are required. The termination test is hence executed only at specified intervals, thereby eliminating the cost of reduction operations. The size of the intervals decreases during the execution (e.g, $n/2$, $n/4$, $n/8$, ... $\sqrt{n}$, $\sqrt{n}$). In our implementation we check for changes in array L after 64,32,16,10,10, ... iterations. The worst case running time of algorithm SIMPLE2 is $O(n^{5/2})$.

2. Algorithm LEV

Our second algorithm, algorithm LEV, is based on an algorithm by S. Levialdi [L]. While the algorithm originally proposed by Levialdi counts the number of connected components in $O(n)$ time, it can easily label the components within the same time bound when an additional $2n$ boolean arrays are used [CSS]. We denote the additional arrays by $IM[0], \cdots, IM[255]$. Parallel array $IM[0]$ contains the original image and $IM[k][i,j]$ denotes pixel $[i,j]$ of parallel array $IM[k]$. The coordinate axis used in the description has (0,0) as the top left position.

Algorithm LEV consists of a forward and a backward phase. In the forward phase we run Levialdi's algorithm, which applies shrinking operations to the pixels in the image such that all 1-pixels in the original image eventually disappear. The image obtained from the shrinking operations at the i-th-iteration is stored in array $IM[i]$. The shrinking operation of the forward phase is as follows [L]:

$$IM[k+1][i,j] = h\,[\,h(IM[k][i,j-1] + IM[k][i,j] + IM[k][i+1,j] - 1) + h(IM[k][i,j] + IM[k][i+1,j-1] - 1)\,]$$

where $h(t)$ is defined as follows:

$h(t) = 0$ for $t \leq 0$, and

$h(t) = 1$ for $t>0$.

The expression given above will compress the pattern toward the top right.

The backward phase consists of a backward expansion during which all pixels in the original image are assigned their correct labels. Let *iter* be the iteration at which the forward phase terminates (i.e., array $IM[iter]$ contains no 1-pixel). The algorithm for the backward phase is as follows. The where statement is a parallel statement in which the code controlled by the statement is executed at every processor at which the where condition is true.

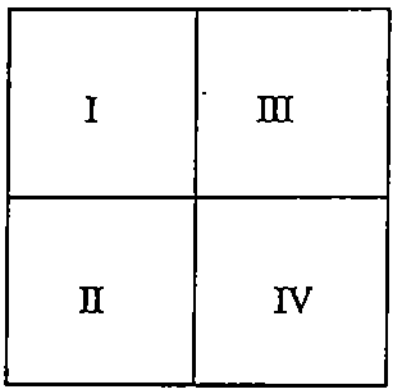
```
for t := iter to 0 do
begin
where IM[iter][i,j] contains an isolated point (i.e., a 1-pixel is surrounded by zeroes) do
      create a unique label using i, j, and t, and store this label in L[i,j]
where (IM[t][i,j] = 1) do   { both isolated and non-isolated 1's }
      begin
      send L[i,j] to neighbors [i+1,j], [i+1,j-1], and [i,j-1]
      where (IM[t-1][i,j] = 1) do  store the received value in L[i,j]
      end;
t := t - 1
end;
```

Assume a connected component is circumscribed by the smallest possible rectangle. Let $(x1,y1)$ and $(x2,y2)$ be the lower left and upper right corner of this rectangle, respectively. Then, the forward phase shrinks the component into the isolated point $(x2,y2)$ in $|x1-x2|+|y1-y2|$ steps, and the backward phase expands it in the same amount of time. Since the running time of algorithm LEV depends on the connected component with the largest value of $|x1-x2|+|y1-y2|$, the running time of algorithm LEV is $O(n)$.

From a theoretical point of view, the use of the $2n$ additional arrays is not desirable. As already mentioned, another variation of Levialdi's algorithm can label the connected components using only $\Theta(\log n)$ additional bit arrays [CSS]. However, this algorithm is more complex. Since for n=128 the additional storage of algorithm LEV did not pose a problem, we did not consider the space-saving approach.

3. Algorithm 4QUAD

Algorithm 4QUAD is based on a divide-and-conquer approach, a common problem solving technique for a parallel environment [AH, NS, U]. Let the 4 quadrants of an $n \times n$ mesh be as shown in Figure 1.



Numbering of Quadrants

Figure 1

Algorithm 4QUAD labels the images in quadrants I, II, III, and IV simultaneously and independently. It then merges, in parallel, the solutions of I and II and those of III and IV. A final merge combines the two halves. Since all the merges can be done in $\Theta(n)$ time, the running time $T(n)$ of algorithm 4QUAD is $T(n) = T(n/2) + cn$, which is $\Theta(n)$.

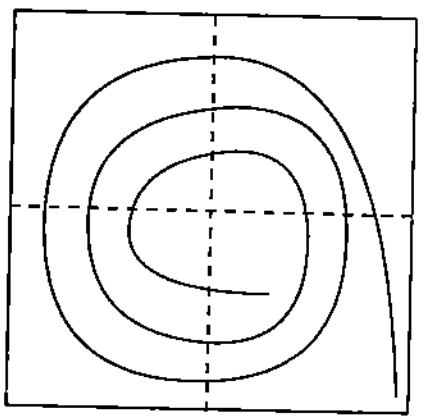
The heart of algorithm 4QUAD is the merging of quadrants. Assume the algorithm merges quadrants I and II. We first construct an undirected bipartite graph using the 1-pixels on the border between the two quadrants. Every such 1-pixel corresponds to a vertex in the graph and an edge is created for every two adjacent 1-pixels in different quadrants. We then apply the systolic-like procedure UPDATE which determines in $\Theta(k)$ time the connected components of a graph whose k edges are stored in a linear array of size k. Procedure UPDATE is described in detail in [H]. Note that UPDATE does not use sorting or random-access read or write operations. The result of UPDATE is a list of pairs $x \rightarrow y$, where x and y are component numbers. The final step of the merging is a broadcasting step in which this list of pairs is broadcast to every processor in quadrants I and II. The purpose of this step is to assign to every 1-pixel its correct component number. If a processor receives the pair $x \rightarrow y$ and if the current label in that processor is x, then the label is changed to y. At the end of this broadcasting step, all proces-

sors in quadrants I and II contain the correct labels for the merged quadrants I and II.

We also implemented a version of algorithm 4QUAD in which algorithm SIMPLE is used on blocks of size 32×32 and these eight blocks are merged with two iterations of algorithm 4QUAD. For the rest of the paper we refer to this version of 4QUAD as 4QUAD2 and to the original one as 4QUAD1. In 4QUAD2 we used algorithm SIMPLE1, due to the fact that there was not much difference in the timing results between SIMPLE1 and SIMPLE2. By running SIMPLE1 on blocks of size $c\sqrt{n}$, for a small constant c (in our case, c= 2.83), 4QUAD2's $\Theta(n)$ timebound is preserved and several of its iterations are replaced by an algorithm that is in many situations faster than 4QUAD.

III. Input Data

The input data we used consisted of images of size 128×128 containing 1's and 0's. Algorithm 4QUAD is data independent, and therefore it is sufficient to test it on one input image. The running times of algorithms SIMPLE and LEV are data dependent, but do not depend on the number of connected components in the image, only on the size of the "largest" component. We tested both algorithm SIMPLE and algorithm LEV on a variety of input images, each containing one connected component. The data we used contained images of alphanumeric characters and spirals. Using this data, we saw exactly at which point one algorithm outperforms the others. The alphanumeric characters we used were: *B*, *S*, *W*, *@*, and *#*. These characters covered between 1/6th and 1/3rd of the 128×128 array.



Example of a 3-spiral

Figure 2

A spiral is an image that starts in one of the quadrants and in order to reach the quadrant from which it started, the other three quadrants must be crossed. We call a spiral a k-spiral if, in order to reach the second endpoint of the spiral, each quadrant is crossed k-times. An example of a *3–spiral* is given in Figure 2. The k-spirals we used have a diameter of roughly $2k \times 128$ and their smallest enclosing rectangle has dimensions close to 128×128. We tested our algorithms on the *1–spiral*, *2–spiral*, *4–spiral*, *5–spiral*, *6–spiral*, *8–spiral*, and the *12–spiral*.

IV. Performance

Figure 3 contains the performance results for the five algorithms.

Algorithm	Execution speed - milliseconds				
	B	S	W	@	#
SIMPLE1	30.81	45.19	30.81	36.98	31.16
SIMPLE2	31.96	46.69	31.96	37.37	31.96
LEV	26.68	33.49	27.87	26.98	26.98
4QUAD1	365.94				
4QUAD2	292.4	294.91	302.08	291.9	293.93

(a) alphanumeric characters

Algorithm	Execution Speed - milliseconds						
	1-spiral	2-spiral	4-spiral	5-spiral	6-spiral	8-spiral	12-spiral
SIMPLE1	54.41	112.92	279.87	370.83	436.64	607.7	977.41
SIMPLE2	53.35	109.93	273.01	363.59	426.96	593.59	925.52
LEV	47.11	55.69	64.87	67.55	66.36	68.39	72.25
4QUAD1	← 365.94 →						
4QUAD2	291.91	296.5	301.59	298.52	302.61	303.62	305.66

(b) spiral data

Performance Results

Figure 3

The times do not include time for input and output operations, only the time taken for the actual connected component computation. All algorithms are written in Parallel Pascal. The parallel integer arrays used in the algorithms are 32-bit integer arrays. The algorithms, with the exception of LEV, do not require 32-bit integers, 15-bit integers are sufficient. However, when we began implementing the algorithms, some operations on subrange data did not work correctly in the MPP's Parallel Pascal. Arithmetic operations using subranges require significantly less time, as shown in Figure 4 which is taken from [B]. Algorithm LEV is implemented using boolean parallel arrays (with the exception of the integer array L containing the labels). Recently we retimed most of the algorithms using subrange data. The new timing results for algorithms SIMPLE1, SIMPLE2, and LEV were only slightly different. The use of subranges in algorithm 4QUAD1 resulted in an execution speed of 328.31 milliseconds (versus 365.94 without subranges). Algorithm 4QUAD2 was not retimed using subrange data, but we conjecture a speedup similar to that exhibited by 4QUAD1. Since the operations performed on the parallel arrays in all the algorithms consist mainly of shift and comparison operations, not arithmetic operations, the somewhat small improvement achieved by using subranges is not surprising.

Operation	Execution Speed, MOPS*
Addition of Arrays	
8-Bit Integers (9-Bit Sum)	6553
12-Bit Integers (13-Bit Sum)	4428
32-Bit Floating-Point Numbers	430
Multiplication of Arrays	
8-Bit Integers (16-Bit Product)	1861
12-Bit Integers (24-Bit Product)	910
32-Bit Floating Point Numbers	216
Multiplication of Array By Scalar	
8-Bit Integers (16-Bit Product)	2340
12-Bit Integers (24-Bit Product)	1260
32-Bit Floating-Point Numbers	373

*Million Operations Per Second

Speed of Typical Operations

Figure 4

From Figure 3 it is apparent that algorithm LEV performed better than any of the other

algorithms and we expect this to be true in general. Algorithm LEV uses fewer and simpler data movement operations compared to the other algorithms and its dependence on the largest enclosing rectangle is a definite advantage.

For an image containing components of diameter less than roughly 10×128 (which corresponds to a 5-spiral), both versions of SIMPLE performed better than either version of 4QUAD. With respect to SIMPLE1 and SIMPLE2, we initially expected SIMPLE2 to always outperform SIMPLE1, but this was not the case. This can be explained as follows. As mentioned earlier, the MPP has additional hardware to speed up the termination test which is a logical "or" operation. While SIMPLE2 does not perform this termination test at every iteration it accesses an integer scalar to count the number of iterations elapsed between termination tests. Hence, the time gained by not performing the boolean termination test is lost on accessing the scalar.

Although our results show that 4QUAD2 does run faster than 4QUAD1, the speedup is not that significant. This is due to the fact that the last two iterations of algorithm 4QUAD are the costliest iterations to execute and these are the iterations executed in 4QUAD2. Overall, the behavior of both 4QUAD algorithms is rather poor. A mesh of size 128×128 is not large enough to reflect the asymptotic optimality of algorithm 4QUAD. While divide-and-conquer is an elegant and powerful technique in the design of parallel algorithms, its practical value on fixed size architectures has to be investigated in every application. The merging step of algorithm 4QUAD requires communication between the processors on the border and the remaining processors in each quadrant. Due to the large diameter of a mesh, this communication is costly and results in a large constant compared to the one of algorithm LEV. In addition, the running time of divide-and-conquer algorithms are generally independent of the structure of the input. While we could have incorporated a number of small image-dependent optimizations into the merging step, our calculations showed that the speedup would not have been significant. It is thus not surprising that data dependent algorithms with a larger asymptotic running time, like algorithm SIMPLE, outperform a divide-and-conquer algorithm for relatively small n.

The method of merging quadrants as done by algorithm 4QUAD is of importance in another setting, namely when combining subimages of an image too large to be handled by the

processor array in its entity. The merging provides an efficient and natural way to combine subimages whose connected components have already been computed, as will be discussed in Section V. Using algorithms SIMPLE or LEV alone to compute connected components of larger images could result in inefficient algorithms.

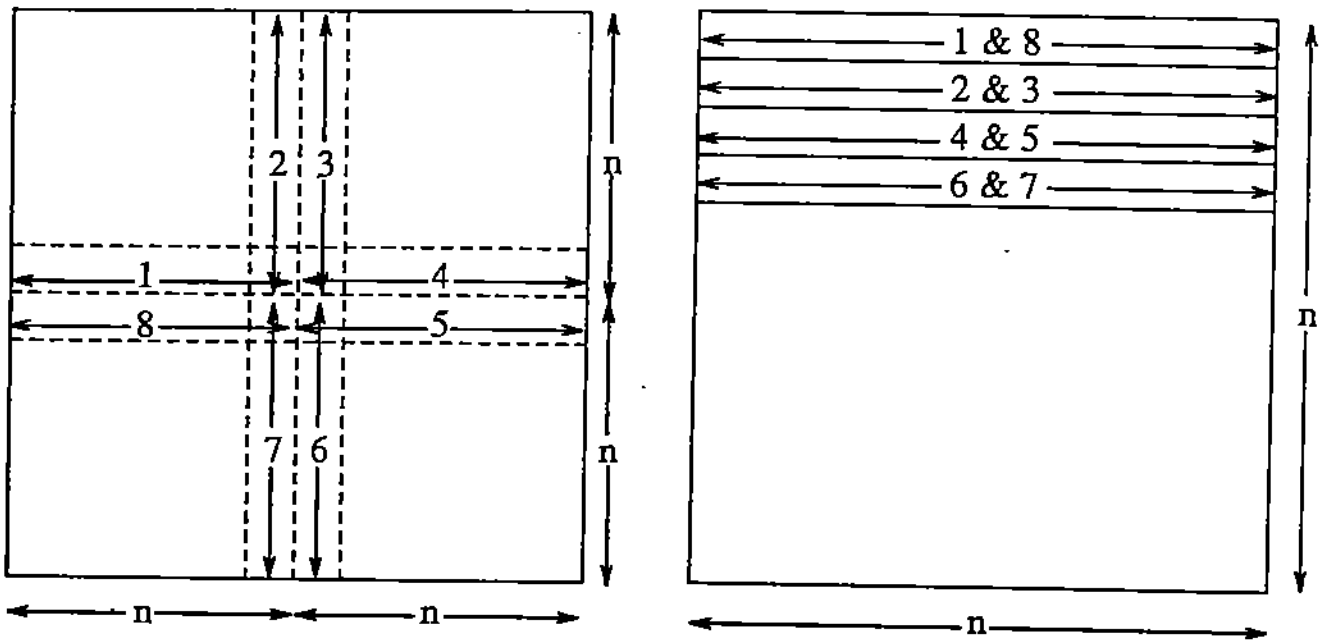
V. Extensions to Larger Images

Assume the size of the processor array is $n \times n$. In this section we discuss how to efficiently determine the connected components of images of size $m \times m$ with $m > n$. Consider first the case when the image is of size $2n \times 2n$ (which translates to images of size 256×256 on the MPP). One approach is to fold the image onto the $n \times n$ processor array so that every processor contains four pixels (they could be adjacent pixels or come from different quadrants) [R2]. A parallel algorithm labeling the entire image without handling subimages independently is likely to run into space problems. Extensions to even larger images might not be possible. We handle images of size $m \times m$ by partitioning them into subimages of size $n \times n$ (or smaller) before bringing them into the processor array. In order to do this efficiently, the abilities of the staging memory are crucial. After a subimage of size $n \times n$ is sent from the staging memory to the processor array, we find the connected components of the subimage using any of the connected component algorithms described in the previous sections (preferably, algorithm LEV). The connected component labels of the 1-pixels in the top and bottom rows and the left and right columns of the subimage are saved in the processor array for later use. The $n \times n$ array of labels generated by the connected component computation is moved from the processor array to the staging memory. After all of the subimages have been processed in this manner, the rows and columns relevant to compute the final connected component labels are stored in the processor array. A procedure reminiscent of the merging step done in algorithm 4QUAD determines the final labels.

Before describing the merging of the subimages, we take a closer look at the operations done in the merging step of procedure 4QUAD. In procedure UPDATE, a basic step in the systolic-like connected component computation consists of a five-part parallel case statement. Let *up–step* be the time taken by such a basic step in procedure UPDATE. Furthermore, let

br–step be the time taken by a basic step in the broadcasting done after procedure UPDATE. The time of this basic step is equivalent to about one-fifth of the time needed by a basic step in UPDATE.

Consider first the case when the image is of size $2n \times 2n$. The four rows and four columns containing the relevant labels of the four subimages are stored in the first four rows of the processor array as shown in Figures 5(a) and 5(b).



(a) Numbering of Boundary Rows and Columns in the Subimages

(b) Arrangement of Boundary Rows and Columns from (a) in the Mesh

Figure 5

A straightforward way to obtain the final labels is to consider the (no longer bipartite) graph consisting of the up to $4n$ edges formed by the four rows and four columns which are stored in a linear array of size $4n$ as shown in Figure 5(b). Performing procedure UPDATE on these edges to determine the connected components uses $4n$ *up–step* steps. The results of procedure UPDATE are then broadcast through the linear array so that every 1-pixel on the border of a subimage knows its correct label with respect to the $2n \times 2n$ image. This requires $5n$ *br–step* steps, and we thus spend a total of $4n$ *up–step* steps and $5n$ *br–step* steps on merging quadrants. For the remainder of the paper we call this solution the "linear array" solution.

An alternative, more parallel solution, which we call the "multiple merge" solution, is to

first perform UPDATE simultaneously on rows 1 and 8 and rows 4 and 5 of the subimages. The numbering of the rows is shown in Figure 5(a). The result of UPDATE on rows 1 and 8 (resp. 4 and 5) is then broadcast to rows 1 and 8 and also to columns 2 and 7 (resp. to rows 4 and 5 and also to columns 3 and 6). At this point the boundaries of quadrants I and II (resp. III and IV) of the image have been merged. Next UPDATE is performed on the $2n$ entries formed by the just updated entries in columns 2, 3, 7, and 6. The result of UPDATE represents the final answer and is broadcast to all boundaries of the subimages. The first UPDATE uses n *up–step* steps and it is followed by $2n$ *br–step* steps. The second UPDATE uses $2n$ *up–step* steps and it is followed by $3n$ *br–step* steps. Hence, we perform a total of $3n$ *up–step* steps and $5n$ *br–step* steps, not counting additional data movement needed between the rows of the processor array. We are currently implementing both approaches on the MPP. Not considering overhead, the multiple merge method appears to be more efficient.

Now consider the case when the image is of size $4n \times 4n$ (which translates to images of size 512×512 on the MPP). One way to obtain the final labels is to use the linear array method as follows. The twenty-four rows and twenty-four columns containing the relevant labels of the sixteen subimages are arranged so that they form a linear array of size $24n$ containing edges of a graph. Performing procedure UPDATE on this linear array uses $24n$ *up–step* steps. The results of procedure UPDATE are then broadcast through the linear array, which uses $25n$ *br–step* steps. Hence, a total of $24n$ *up–step* steps and $25n$ *br–step* steps are performed using the linear array method on an image of size $4n \times 4n$.

A second solution for computing the labels of a $4n \times 4n$ image uses the multiple merge method, where each quadrant is now of size $2n \times 2n$. Assume each such quadrant did update its labels by using one of the two methods described for images of size $2n \times 2n$. In addition, the border rows and columns of each quadrant (each being of length $2n$) also have their labels updated with respect to the quadrant they are in. This additional work can easily be done without increasing the time. Procedure UPDATE is now performed on the row of length $2n$ dividing quadrants I and II (resp. quadrants III and IV) of the $4n \times 4n$ image. The results of each UPDATE are then broadcast to the twelve border rows and twelve border columns of the $n \times n$ subimages within each half of the entire image. Next, procedure UPDATE is performed on the column of length $4n$ dividing quadrants I and II and quadrants III and IV and these results are

broadcast to all twenty-four border rows and all twenty-four border columns of the $n \times n$ subimages. The first UPDATE uses $2n$ *up−step* steps followed by $3n$ *br−step* steps. The second UPDATE uses $4n$ *up−step* steps followed by $5n$ *br−step* steps. Let $T_{2n}(n)$ be the time to run either the linear array method or the multiple merge method on images of size $2n \times 2n$ and to update the border elements of the $2n \times 2n$ subimages. Then, the total time is $T_{2n}(n)$, $6n$ *up−step* steps, and $8n$ *br−step* steps. If the linear array method is used to compute the labels for the quadrants of size $2n \times 2n$, the resulting total time is $10n$ *up−step* steps and $13n$ *br−step* steps for images of size $4n \times 4n$. If the multiple merge method is used to compute the labels for the quadrants of size $2n \times 2n$, the resulting total time is $9n$ *up−step* steps and $13n$ *br−step* steps for images of size $4n \times 4n$.

Using the multiple merge method for images of size $4n \times 4n$ combined with either the linear array method or the multiple merge method for the $2n \times 2n$ quadrants of the image appears to be significantly better than using the linear array method on the entire $4n \times 4n$ image. There are a number of other methods that combine subimages using a combination of the methods we described. We only described what we consider to be the simplest method (namely, the linear array method) and the most efficient method (the multiple merge method). Our techniques for merging subimages can easily be extended to images larger than $4n \times 4n$ and to images of other dimensions.

After the borders of every subimage of size $n \times n$ know their correct labels with respect to the entire image, we bring each $n \times n$ labeled subimage from the staging memory into the processor array. We then broadcast the changes stored in the border rows and columns to every 1-pixel in the subimage. Doing so uses $6n$ *br−step* steps. After the subimage broadcasting has been done, each subimage is moved back to the host computer. After all $n \times n$ labeled subimages have been processed in this fashion using a total of $\lceil m/n \rceil^2 6n$ *br−step* steps, each labeled subimage contains the correct result of the connected component computation with respect to the entire image.

References

[AH] M.J. Atallah, S.E. Hambrusch, "Solving Tree Problems on a Mesh-Connected Processor Array", *Proceedings of 26th Annual Symposium on Foundations of Computer Science*, pp. 222-231, October 1985.

[B] K.E. Batcher, "Design of a Massively Parallel Processor", *IEEE Transactions on Computers*, Vol. C-29, No. 9, pp. 836-840, September 1980.

[CSS] R.E. Cypher, J.L.C. Sanz, L. Snyder, "Practical Algorithms for Image Component Labeling on SIMD Mesh Connected Computers", *Proceedings of 1987 Int. Conference on Parallel Processing*, pp 772-779, 1987.

[F] *Proceedings of First Symposium on the Frontiers of Massively Parallel Scientific Computation*, J. Fischer, Editor, NASA, September 1986.

[GAC] *General Description of the MPP*, Goodyear Aerospace Corporation, GER-17140, pp. 5-7, April 1983.

[H] S.E. Hambrusch, "VLSI Algorithms for the Connected Component Problem", *SIAM J. Comput.*, Vol. 12, No. 2, pp. 354-365, May 1983.

[L] S. Levialdi, "On Shrinking Binary Picture Patterns", *Communications of the ACM*, Vol. 15, No. 1, pp. 7-10, January 1972.

[MS] R. Miller, Q.F. Stout, *Parallel Algorithms for Regular Architectures*, manuscript, 1987 (to be published by MIT Press).

[NS] D. Nassimi, S. Sahni, "Finding Connected Components and Connected Ones on a Mesh-Connected Parallel Computer", *SIAM J. Comput.*, Vol. 9, No. 4, November 1980.

[R1] A.P. Reeves, "Parallel Pascal: An Extended Pascal for Parallel Computers", *J. of Parallel and Distributed Computing*, Vol. 1, pp. 64-80, 1984.

[R2] A. Reeves, "The Massively Parallel Processor: A Highly Parallel Scientific Computer", manuscript 1986.

[R3] H.K. Ramapriyan, "A Connected Component Algorithm on the MPP", manuscript, 1985.

[U] J.D. Ullman, *Computational Aspects of VLSI*, Computer Science Press, 1984.