1989

Parallel Asynchronous Connected Components in a Mesh

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

Michael Luby

Report Number:
89-888

Hambruch, Susanne E. and Luby, Michael, "Parallel Asynchronous Connected Components in a Mesh" (1989). *Department of Computer Science Technical Reports*. Paper 756.
<https://docs.lib.purdue.edu/cstech/756>

PARALLEL ASYNCHRONOUS CONNECTED
COMPONENTS IN A MESH

Susanne Hambruch
Michael Luby

CSD-TR-888
July 1989

**Parallel Asynchronous Connected
Components in a Mesh**

Susanne Hambruch
Michael Luby

CC-89-9 (August 1989)

Purdue University
West Lafayette, Indiana 47907

This research was supported in part by the Office of Naval Research under University Research Initiative grant number N00014-86-K-0689.

Parallel Asynchronous Connected Components in a Mesh

Susanne Hambrusch *

Department of Computer Sciences
Purdue University

Michael Luby †

International Computer Science Institute
Berkeley, California

July 4, 1989

Abstract

Levaldi [6] introduced a parallel synchronous algorithm for counting the number of connected components in a binary image embedded in an $n \times n$ mesh of processors that runs in time $O(n)$. We describe a parallel *asynchronous* algorithm for the same problem achieving the same time bound.

*Research done while visiting the International Computer Science Institute, Berkeley, California, research supported in part by ONR under contracts N00014-84-K-0502 and N00014-86-K-0689, and by NSF under Grant MIP-87-15652.

†On leave of absence from the University of Toronto, research partially supported by NSERC operating grant A8092.

1 Introduction

The problem of counting and labeling connected components in a binary image is a fundamental problem in image processing [1,3,9,10] for which numerous solutions on different architectures and computational models have been developed. An architecture well-suited for operations on images is the mesh. Mesh algorithms for problems on images can generally be classified into local or random-access algorithms. In a *local algorithm* every step consists of the processors updating registers depending on the values of their neighboring processors. In a *random-access algorithm* a step may involve performing more complex operations such as sorting, broadcasting, or a prefix computation. While local algorithms are more desirable from a practical point of view, not all problems can be efficiently solved by a local algorithm.

Leviadi [6] introduced a synchronous local algorithm for counting the number of components in $O(n)$ time. Synchronous random access mesh algorithms for this problem, as well as the problem of labeling the connected components, have been developed in [2,4,7,8]. Leviadi's local algorithm and the random-access algorithms do not run correctly in an asynchronous model (i.e., a model in which the processors operate at different speeds). We describe an asynchronous local algorithm for counting the number of components in $O(n)$ time. Our algorithm is based on a modification of Leviadi's algorithm.

Designing and verifying synchronous parallel algorithms is, in general, easier than designing asynchronous parallel algorithms. At the same time, providing a central clock for synchronization does not come for free. It is

the exception rather than the rule, that a problem has a synchronous and asynchronous algorithm based on the same general idea and with the same asymptotic complexity.

2 The Problem

A *mesh* is an $n \times n$ array of points with a processor assigned to each point. The points in the mesh are indexed from $(0, 0)$ in the upper lefthand corner to $(n - 1, n - 1)$ in the lower righthand corner. We use (r, c) to index a point in the mesh, where r is the row number and c is the column number. (See Figure 1.) To simplify the notation in the explanations, we refer to a point by the single variable i . Each point has up to eight neighbors: above, above-right, right, below-right, below, below-left, left and above-left. The neighbors of $i = (r, c)$ are referred to as $i_{ab} = (r_a, c_b)$, where $a, b \in \{-, +, =\}$, and r_+ is $r + 1$, $r_ =$ is r , r_- is $r - 1$, c_+ is $c + 1$, $c_ =$ is c , c_- is $c - 1$. For example, i_{+-} is the point below and to the left of i , and $i_{=+}$ is the point to the right of i . (See Figure 2.)

Each point in the mesh is initialized to a value, either 0 or 1. Let I be the $n \times n$ image array that contains the initial 0/1 values assigned to the points. For simplicity, we assume that there is a *border* around the mesh, i.e. an extra row above and below the mesh and an extra column to the left and to the right of the mesh, as shown in Figure 1.¹ All points on the border are assigned the initial value 0. The image array I determines the *input graph* G as follows. Every point with value 1 corresponds to a

¹The row coordinates of the top and bottom border rows is -1 and n , respectively. Similarly, the column coordinates of the top and bottom border columns is -1 and n , respectively.

vertex of G and two vertices i and j are adjacent in G if and only if the corresponding points are neighbors in the mesh.

A set of points C form a (maximal) component of the input graph if: (1) $I(i) = 1$ for all $i \in C$; (2) for all pairs of points $i, i' \in C$ there is a path of points between i and i' such that all points along the path are in C ; (3) C is maximal with respect to conditions (1) and (2). The task at hand is to count the number of connected components in the input graph G . In one time step each processor can communicate with each of its eight neighbors and it can make local computations based on its values and the values of its neighbors.

3 Review of the Synchronous Algorithm

We briefly review the synchronous algorithm for counting the number of connected components as developed by Levialdi [6]. Our description is different from the one given in [6] and is more conducive to explaining our asynchronous algorithm. Associated with each processor is a variable $count(i)$, which is initialized to zero and is used to count the components that disappear at point i .

Let $S(i)$ be the 2×2 square of values of points with i as the upper-left point. Let $*$ denote an unspecified value which can be either 0 or 1. If $S(i) = \begin{bmatrix} 0 & 1 \\ 1 & * \end{bmatrix}$ we say that i has *grow potential* and if $S(i) = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$ we say that i has *shrink potential*. One time step consists of every processor i determining $S(i)$ and then determining whether or not to execute one of the following actions.

- When $S(i)$ has grow potential at time $t - 1$, the value of point i

becomes 1 at time t . We say that i executes the *grow rule* at time t .

- When $S(i)$ has shrink potential at time $t - 1$, the value of point i becomes 0 at time t . We say that i executes the *shrink rule* at time t . In addition, if the value of each one of the eight neighbors of i is 0 at time $t - 1$, processor i increments $count(i)$ by one at time step t .

It is shown in [6] that executing the grow and shrink rules in a synchronous model does not cause two components to grow together or one component to be disconnected. Every component C moves up and to the left. It disappears and is counted in square $(r_{\min}, c_{\min})$, where $r_{\min}$ is the smallest row coordinate in I to contain a point of component C and $c_{\min}$ is the smallest column coordinate to contain a point of C . At each time step the value of $\max\{r + c : (r, c) \text{ is a point in component } C \text{ at time } t\}$ decreases by one until it equals $r_{\min} + c_{\min}$. At this point, which occurs at time $r_{\max} + c_{\max} - r_{\min} - c_{\min} + 1$, component C is counted and it disappears from the mesh. All components are counted after at most $2n - 1$ time steps and the total number of components is $\sum_{i=0}^{n^2-1} count(i)$.

It is easy to find examples for which this algorithm, when run asynchronously, does not perform correctly, i.e. it may merge a number of components into one or split one component into a number of components.

4 The Asynchronous Model

In the synchronous model of computation, it is assumed that there is a global clock and that all processors are synchronized with respect to this clock, i.e. all processors read the values of neighboring processors at time t —

1 before any changes in values are made at time step t . In the *asynchronous model*, there is no such synchronization. The processors run independently at possibly very different speeds, and even a single processor is allowed to run at different speeds at different times.

We design algorithms for counting the connected components in a mesh for two versions of the asynchronous model which differ in the way the processors communicate with their neighbors. Assume that in one step of the algorithm processor p communicates with neighbors $n_1, \dots, n_k$, $k \leq 8$. In the *deterministic neighbor communication* model processor p completes its communication with neighbor n_i before initiating its communication with n_{i+1} . In the *random neighbor communication* model processor p performs the communication in a random order. Intuitively, in the random model, processor p does not need to remain idle until it has received values from neighbor n_i . Processor p is allowed to continue collecting information from its other neighbors in the meantime.

Although processors do not synchronize globally, it is convenient for the analysis to introduce a natural measure of running time called a *global time step*. Global time step t starts at the first point in global time when all processors have executed at least one step of their local algorithms since the start of global time step $t-1$. Thus, the actual time for a global time step is basically the processor cycle time for the slowest running processor. Since processors may run at different speeds and vary their speed over time, the global behavior of the algorithm may vary widely for different executions of the algorithm. The running time of the algorithm is the maximum over all possible executions of the algorithm of the number of global time steps

till completion of the algorithm.

5 The Asynchronous Algorithm

In Levialdi's algorithm, the connected components move up and to the left until they are counted. Our algorithm is based on a simple modification of this synchronous algorithm and it inherits the good properties of the synchronous algorithm. Each processor needs only a constant number of bits (except for the local variable $count(i)$ that counts the number of connected components at point i) and the running time is still $O(n)$. We omit the description on how to perform the final summation of the $count$ variables (it is straightforward to do this in an additional $2n - 2$ global time steps).

To describe how the asynchronous algorithm works, we use the concept of a time wave. "Time waves" are sent out from the upper-left corner and move down and to the right. For purposes of talking about a time wave, we think of each consecutive time wave sent out as being labeled by consecutively increasing positive integers, although in the actual algorithms we don't explicitly keep track of these integer values. These time waves are used to coordinate the order in which the grow and shrink rules are applied. The intuitive idea is that when time wave w reaches point i , i performs the same action as it would have in Levialdi's synchronous algorithm at time step w . The movement of the time waves and the order of execution of the grow and shrink rules are regulated in such a way that the asynchronous algorithm exactly simulates the behavior of Levialdi's synchronous algorithm. Our algorithm can be viewed as a data-driven algorithm in which the presence of data of a certain value is the signal for a processor to take

a new action. Hence, our algorithm is a type of wavefront algorithm [5].

All points i that are not border points are initialized to start at time wave 0. The movement of time waves and the execution of processor actions obey the following rules. If wave w is currently at i then i can only *act* if wave $w + 1$ is at $i_{=-}$, $i_{=-}$ and i_{--} and if wave w is at $i_{=+}$, $i_{+=}$ and i_{++} . These conditions do not apply to neighbors of i that are in the border, i.e. i can act independently of the neighbor points in the border. Points on the border of the mesh never act. Figure 3 shows the situation allowing i to act. If wave w is at i and i can act, processor i reads $I(i_{=-+})$, $I(i_{+=})$ and $I(i_{++})$, performs a grow or shrink rule as appropriate, and then changes the wave number at i to $w + 1$. Notice that in the period of time from when i can act until the time when i changes its wave number from w to $w + 1$, the points $i_{=-}$, $i_{=-}$, i_{--} , $i_{=+}$, $i_{+=}$ and i_{++} cannot act because they do not satisfy the criteria for acting in this period. Thus, in this critical period when processor i reads the values of $I(i_{=-+})$, $I(i_{+=})$ and $I(i_{++})$ and possibly changes $I(i)$, all of the neighbors of i that either affect whether $I(i)$ changes value or are affected by the change in value of $I(i)$ are inactive.

It can be easily verified that if these rules are maintained, then the wave fronts move from the upper-left corner down and to the right, that only consecutive waves have a common border, and that if wave w is at i and i can act, then processor i applies exactly the same rule as in Levialdi's synchronous algorithm at time step $w + 1$.

We now address the implementation of these rules for the two asynchronous models. One way to directly implement these rules in the deterministic neighbor communication model is to use a counter at each proces-

sor to keep track of the wave number. This takes $O(\log n)$ extra bits per processor. Since two non-consecutive wave fronts are never next to each other, a mod 3 counter suffices. Thus we can count the number of connected components in this asynchronous model by using only two additional bits. As will be shown, the running time (measured in global time steps) remains $O(n)$.

The implementation using the mod 3 counter can be turned into an implementation for the random neighbor communication model by using an additional six bits per processor. Associate with each point i six boolean lock variables: $L_{-}(i)$, $L_{=}(i)$, $L_{++}(i)$, $L_{+-}(i)$, $L_{-+}(i)$, $L_{++}(i) \in \{0, 1\}$ and let processor i set $L_{ab}(i)$ to 1 when point i_{ab} has the correct wave number for i to act. The lock variables are used as local memory by i to keep track of which neighbors have the correct wave number to allow i to act. Having a lock variable set to 1 means that, with respect to the associated neighbor, i is allowed to go ahead and perform the next action. When all six lock variables are set to 1, i is allowed to act. After i has acted, it changes the values of the six lock variables to 0. If i has neighbors on the border, then the lock variables corresponding to the border points are initialized to 1 and always remain 1. For example, if i_{-} is a border point, then $L_{-}(i)$ is initialized to 1 and never changes its value. We next describe an implementation of this algorithm that does not require the mod 3 counter and uses only the 6 boolean lock variables. We eliminate the counter by handling the lock variables as follows. The value of $L_{ab}(i)$ can only be changed from 0 to 1 by the processor at i_{ab} , whereas only the processor at i is allowed to change these six values from 1 to 0. Initially, for all points

i with no neighbor in the border, $L_{+=}(i) = L_{=+}(i) = L_{++}(i) = 1$ and $L_{-=}(i) = L_{=-}(i) = L_{--}(i) = 0$. If i has neighbors on the border, then the setting of the lock variables for non-border neighbors is exactly the same as just described. For neighbors that are on the border the associated lock variable is initialized to 1 and remains 1 throughout the algorithm.

The processor at point i continually executes the following program.

1. Processor i reads the values of $L_{-=}(i)$, $L_{=-}(i)$, $L_{--}(i)$, $L_{+=}(i)$, $L_{=+}(i)$ and $L_{++}(i)$ until all six values are 1. When this happens, processor i can act and does so by executing the following steps in sequence.
2. Processor i reads $I(i_{=+})$, $I(i_{+=})$ and $I(i_{++})$ and performs a grow or shrink rule as appropriate. In addition, if $I(i_{=+}) = I(i_{+=}) = I(i_{++}) = I(i_{=-}) = I(i_{=-}) = I(i_{--}) = 0$ and $I(i) = 1$ before the shrink rule is applied, then increment $count(i)$ by one to count the disappearing connected component.²
3. Processor i sets $L_{-=}(i) = L_{=-}(i) = L_{--}(i) = L_{+=}(i) = L_{=+}(i) = L_{++}(i) = 0$. (Except that the values of variables corresponding to border points remain at 1.)
4. Processor i changes the values of $L_{+=}(i_{=-})$, $L_{++}(i_{--})$, $L_{=+}(i_{=-})$, $L_{=-}(i_{+=})$, $L_{=-}(i_{=+})$, $L_{--}(i_{++})$ all to 1. Control then returns to step 1.

For each of the six variables at i we define the following *linked* variables:

$L_{-=}(i)$ is linked to $L_{+=}(i_{=-})$, $L_{=-}(i)$ is linked to $L_{=+}(i_{=-})$, $L_{--}(i)$ is linked

²It can be verified that if these six neighbors of i have value 0 at this point in time, then so do the other two neighbors of i .

to $L_{++}(i_{--})$, $L_{+=}(i)$ is linked to $L_{=-}(i_{+=})$, $L_{=+}(i)$ is linked to $L_{=-}(i_{=+})$ and $L_{++}(i)$ is linked to $L_{--}(i_{++})$. Because of the order of execution in steps 3 and 4 of the program, it can be seen that at most one of each of the variables in a linked pair can have value 1 at any point in time (except, of course, if one of the variables is associated with a border point).

This program is to be interpreted in the "time wave" terminology as follows. When processor i finds in step 1 that all six lock variables are set to 1 for the $(w + 1) - st$ time, i is at time wave w and is ready to act to make the transition to wave $w + 1$. This statement can easily be proved by induction on the number of times processor i finds the six lock variables equal to 1 in step 1. When the six lock variables are 1 for the $(w + 1) - st$ time, then the six lock variables at i_{--} , $i_{-=}$, and i_{--} have been 1 exactly $w + 1$ times and the six lock variables at $i_{+=}$, $i_{=+}$ and i_{++} have been 1 w exactly times. Furthermore, in the time period just before i starts to execute step 2 until the time when i completes execution of step 2 none of these six neighboring points are acting. This holds since each one of the six lock variables associated with i have value 1 during this period, all of the variables linked to these variables have value 0 during this period. Thus, the processors i_{++} , $i_{=+}$, $i_{+=}$, i_{--} , $i_{-=}$ and i_{--} are all stuck in step 1 of their respective programs during this period (or finishing up step 4 to go to step 1). Thus, among these points, i is the only processor that can act during this period.

Let diagonal d contain the points $i = (r, c)$ with $r + c + 1 = d$, $1 \leq d \leq 2n - 1$. In the following timing analysis, one global time step is defined to be the period of time in which every point i checks whether it can act

and, if it can, it completes the action. In particular, one global time step of the random neighbor communication algorithm using the six lock variables starts execution in step 1. If it finds all six lock variables set to 1, it executes steps 2, 3, and 4 within the same global time step.

Lemma 1 *After point i on diagonal d has acted $2n - d$ times, $\text{count}(i)$ has its final value, $1 \leq d \leq 2n - 1$.*

Proof: After i has acted $2n - d$ times, it has applied the same rules as the first $2n - d$ steps of Levaldi's synchronous algorithm. At time $2n - d$ in Levaldi's algorithm the points on diagonal d have completed counting components and thus the lemma follows. $\square$

Lemma 2 *After global time step t every point on diagonal d has a wave number of at least*

$$w \geq \lfloor \frac{t - d}{3} \rfloor + 1.$$

Proof: The proof is by induction on the number of global time steps. When $t = 0$ the claim is vacuous. Let $t \geq 1$ and assume the induction hypothesis is true for all $t' < t$. This means that at the beginning of global time step t , for all $w \geq 1$, all points on diagonals

$$d \in \{t - 3w, t - 3w + 1, t - 3w + 2\}$$

are at wave at least w . Furthermore, all points on diagonal $d \geq t$ are at wave at least 0. We need to show that at the end to global time step t , for all $w \geq 0$, all points i on diagonal $t - 3w$ are at wave at least $w + 1$. By the induction hypothesis, at the beginning of global time step t the wave

at i is at least w . If the wave at i is $w + 1$ or more at the beginning of time step t then certainly the wave at i is $w + 1$ or more at the end of time step t . Consider the case when the wave at i is w at the beginning of time step t . Neighbors i_{-+} , i_{+-} , and i_{--} are on diagonals $t - 3w - 1$ and $t - 3w - 2$ and by the induction hypothesis the wave number at these neighbors is at least $w + 1$ at the beginning of global time step t . Because the wave at i is w , it must be the case that the wave number at all of these neighbors is exactly $w + 1$ at the beginning of time step t . By similar reasoning, the wave at neighbors i_{++} , i_{-+} , and i_{+-} is w at the beginning of time step t . Thus, i is able to act at during global time step t and is at wave $w + 1$ by the end of global time step t . $\square$

Theorem 3 *After global time step $6n - 5$ all components have been counted.*

Proof: By Lemma 1, a point on diagonal d needs to act at least $2n - d$ times in order to compute $\text{count}(i)$ correctly, $1 \leq d \leq 2n - 1$. By Lemma 2, a point on diagonal d has acted at least

$$\lfloor \frac{t - d}{3} \rfloor + 1$$

times by the end of global time step t . Thus, by the end of global time step $6n - 2d - 3$ a point on diagonal d has finished. The diagonal that takes the greatest number of global times steps to finish is $d = 1$, resulting in the bound of $6n - 5$. $\square$

6 Acknowledgements

We thank Stephen Omohundro for several helpful discussions.

References

- [1] Blanz, W.-E., Petkovic, D., Sanz, J., "Algorithms and Architectures for Machine Vision", Chapter 10 in *Signal Processing Handbook*, C.H. Chen, Editor, Marcel Dekker, 1989.
- [2] Cypher, R., Sanz, J., Snyder, L., "Practical Algorithms for Image Component Labeling on SIMD Mesh Connected Computers", Proceedings of *International Conference on Parallel Processing*, 1987, pp. 772-779.
- [3] Fairhurst, M., *Computer Vision for Robotic Systems*, Prentice Hall, 1988.
- [4] Hambruch, S.E., TeWinkel, L., "A Study of Connected Component Algorithms on the MPP", *Proceedings of 3rd Internat. Conference on Supercomputing*, 1988, pp 477-483.
- [5] Kung, S.Y., *VLSI Array Processors*, Prentice Hall, 1988.
- [6] Levialdi, S., "On Shrinking Binary Picture Patterns", *CACM*, 1972, Vol. 15, pp. 7-10.
- [7] Miller, R., Stout, Q., *Parallel Algorithms for Regular Architectures*, manuscript, to be published by MIT press, 1988.
- [8] Nassimi, D., Sahni, S., "Finding Connected Components and Connected Ones on a Mesh-Connected Parallel Computer", *SIAM J. on Computing*, 1980, 9(4), pp. 744-757.
- [9] Prasanna-Kumar, V.-K., Reisis, D., "Parallel Architectures for Image Processing and Vision", Proceedings of *Image Understanding Workshop*, 1988, pp 609-619.
- [10] Rosenfeld, A., Kak, A., *Digital Picture Processing*, Academic Press, 1982.

	-1	0	1	2	...	c	...	n-1	n
-1									
0									
1									
2									
...									
...									
r						i			
...									
...									
n-1									
n									

Image I and its border
Figure 1

i_{-}	i_{-}	i_{+-}
i_{--}	i	$i_{=+}$
i_{+-}	$i_{+=}$	i_{++}

Point i and its neighbors
Figure 2

$w+1$	$w+1$	
$w+1$	w	w
	w	w

Situation allowing i at wave w to act
Figure 3