DRAWING STRAIGHT LINES WITH A PYRAMID CELLULAR AUTOMATON

Quentin F. STOUT

Mathematical Sciences, State University of New York, Binghamton, NY 13901, U.S.A.

Received 15 March 1982; revised version received 6 June 1982

Keywords: Pyramid cellular automaton, parallel computing, picture processing, exact line problem, self-organizing automata

1. Introduction

This paper gives an efficient algorithm for drawing a straight line through two given points in a digitized picture, where the picture is stored one pixel per processor at the base of a *pyramid cellular automaton* (PCA). Dyer [1] defines a PCA to be a finite-state automaton which is replicated at each node of a finite, complete 4-ary tree, but for our purposes it is easier to take the equivalent view that each node contains a copy of a random access machine with a fixed number of memory cells, each of which can store only a fixed amount of information. In unit time a processor at height k can send a fixed amount of information to any of its neighbors: four sons at height $k-1$, a father at height $k+1$, or four adjacent processors at height k. (The base is at height 0, and there are obvious communication restrictions for processors along the sides, see Dyer [1].) All nine communication links can be used simultaneously independent of each other. An input to a PCA consists of initializing the memory of the base processors, setting all processors except the apex into a quiescent state, and putting the apex into a start state. PCAs have appeared in [1,3,4,5,6] and elsewhere.

Pick coordinates for the processors in the base so that, when viewed from the apex, the processor in the lower left is at position (0, 0) and the processor in the upper right is $(n-1, n-1)$. The height of the PCA is lg(n), where lg is $\log_2$. We call the column coordinates x and the row coordinates y. Sakoda [3] recently posed the *exact line problem*: suppose two base processors (corresponding to pixels) are initially 'marked', with all others 'unmarked'. The PCA must mark the processors representing the straight line between the centers of the two initially marked processors. There are various criteria used to draw a digitized line, and Sakoda considered the following one: in each column between the marked processors mark the square containing the intersection of the column's centerline with the line between the endpoints, using the upper square if the intersection occurs at a boundary. (This applies only if the difference in x coordinates between the marked processors equals or exceeds the difference in y coordinates. Using techniques given below, in O(log n) time it is possible to decide if this condition holds, and to rotate the entire procedure if it does not.)

Sakoda gave an algorithm which produced an approximation of the line in O(log(n)**2) time, and here we give an exact solution in O(log n) time. Further, the solution method can be applied to other problems. For example, in O(log n) time we can solve Sakoda's circle problem [3], in which one processor is initially red and one is black, and the PCA must mark the processors on the circle passing through the red processor with the black one as center. We note that if one only has an $n \times n$ mesh-connected computer then the exact line problem and circle problem each require O(n) time.

0020-0190/82/0000-0000/$02.75

2. Overview of the algorithm

Temporarily assume that each processor is a RAM of unlimited capacity, with all operations taking unit time. To draw the straight line, first use a top-down approach to have each base processor determine its x and y coordinates. Then have the marked processors send their coordinates to the apex, which sends them down to the processors on the base. Each base processor computes whether it lies on the line segment between the marked processors, and if so becomes marked. The initialization and broadcasting of the marked processors' coordinates takes O(log n) time, and the final calculations take O(1) time, giving O(log n) total time.

The heart of the PCA algorithm consists of building subunits, called *clerks*, that simulate the base processors in the above model. Each clerk lies in a single column of the base and consists of L consecutive processors, where $L = 2 ** [lg(k + 1 + k * lg n)]$ and k is a constant, independent of n, to be determined later. Each column contains n/L clerks in its base processors, and we consider only the case $n > L$. In each clerk, the processor of lowest row index is called the *controller*. Immediately following the controller are k *pseudo-registers*, denoted PR1,..., PRk, each consisting of $1 + lg(n)$ consecutive processors. Each processor in a pseudo-register uses one of its memory cells to store one bit of information for the pseudo-register. Thus a pseudo-register can store the binary representation of any integer from $-(n - 1)$ to $n - 1$. The remaining processors in a clerk, if any, are unused during normal clerk functions (see Fig. 1).

Each processor uses one of its memory cells to keep track of its *type*, which determines its behaviour during the algorithm. There are a fixed number of types, representing the fact that the processor is a controller, the apex, part of PR1, etc. Processors at height 1 through lg(L) are of type *supervisor*, and higher processors are of type *boss*. (The apex is a special type.) In each column each supervisor is above only one clerk or part of one clerk, but each boss is above two or more. When information is being sent from the apex to the clerks, each boss passes it to all of its sons, while each supervisor passes it only to its lower two sons. This protocol insures that information enters each clerk through its controller.

A clerk has a fixed set of instructions which it can execute. These include copying, addition, subtraction, multiplication (producing a double-word result), division and comparisons. To illustrate how instructions are executed, suppose the next in-

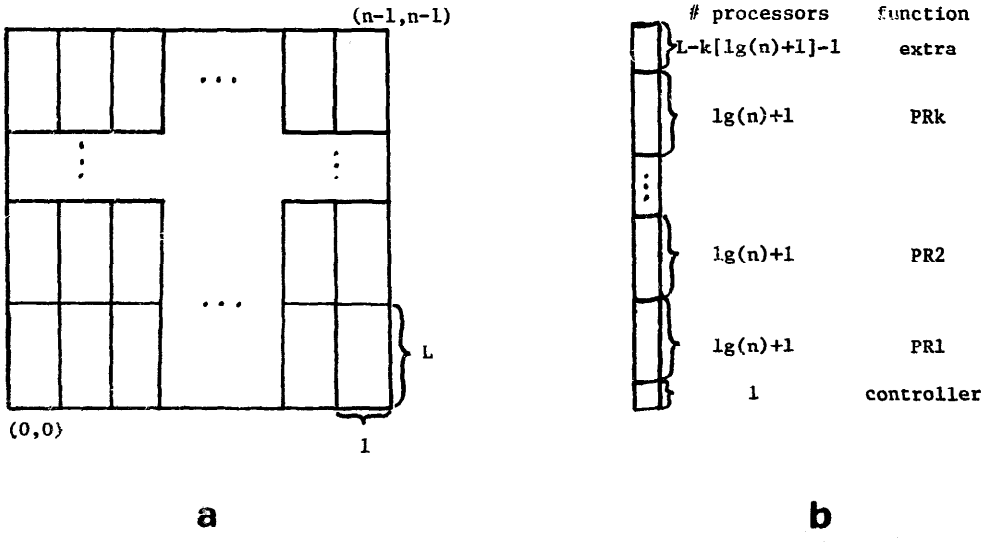

Fig. 1. Clerks. (a) Clerks on the base; (b) Processors within a clerk.

struction is to copy PR2 to PR4. The controller passes this instruction to its neighbor, which passes it to its neighbor, and so on. When the first processor in PR2 receives it, it passes it along, then sends a copy of the bit it is storing, and then sends an end-of-message (EOM) indicator. As this message passes along, when each processor in PR2 receives the EOM it adds its bit and then sends the EOM. When the stream reaches the first processor in PR4, it passes along the instruction, and then stores the next bit received (without passing it along). Each processor in PR4 acts similarly, later processors having to wait longer between the arrival of the instruction and their bit. We call this process *rolling* a number into PR4. When the last processor in PR4 receives EOM it sends back a signal that the instruction is completed. (This processor 'knows' that it is last because the next processor is of type PR5.) The signal reaches the controller in O(log n) time from the start of the execution.

To illustrate arithmetic instructions, suppose the instruction is to add PR8 to PR3. First a copy of the contents of PR8 is moved to PR3, using a second bit in each processor of PR3 to store this new number as well as the old. Then the first processor in PR3 checks the signs of the two numbers. If they are the same, then a signal is sent towards the last processor in PR3 to start a simple bit by bit addition, while if the signs differ then a comparison of the magnitude of the addends is started, with the result determining the sign of the answer and which addend is to be subtracted from the other. A suitable multiplication algorithm is in [2, p. 276], and with some effort the division algorithm in [2, p. 275] can be adapted to clerks, with each operation finishing in O(log n) time.

The algorithm consists of three parts: typing each processor, initializing pseudo-registers and performing calculations.

3. Setting the type of each processor

We first identify the supervisors and bosses, which requires us to determine lg(L). To do this, the apex sends a message of k 1's, one at a time, followed by EOM. This message is sent to its lower left son, which passes it on to its lower left son, and so on. Whenever a processor receives the EOM it adds k 1's before sending the EOM along. When this reaches base processor (0, 0) it is passed to processor (0, 1), which starts rolling the 1's onto the first row. When processor (0, 0) receives the EOM it adds its k 1's and then the EOM. The last processur to have a 1 rolled onto it is at (0, $1 + k + k * \log n$). This is the only processor not having other 1's pass over, and when it receives the EOM it starts it upwards. The first processor to receive the EOM which also participated in sending the original message downwards is at height lg(L). This is the highest level of a supervisor, and we use this processor to identify the other supervisors and bosses. It starts this process by sending up a 3. Each processor adds a 2 when the 3 is received, and then passes the 3. A stream of $\lg(n) - \lg(L) - 1$ 2's, followed by a 3, arrive at the apex. The apex sends this message down to all its children. Each processor receiving a downward message starting with 2 becomes a boss type, and sends down the rest of the message to all of its sons. Each processor receiving a downward message starting with 3 becomes a supervisor (unless it is on the base) and sends down the 3 to all of its sons. When the 3's reach the base (all simultaneously) a message is sent upward to start the next step.

Since the bosses and supervisors have been set, from now on, unless otherwise specified, all messages from the apex will enter a clerk only through the controller, and all messages are sent to all clerks. First the apex sends a 4, which when received by a base processor tells it that it is of type controller. To identify the processors in each clerk's PR1, the apex sends a 5 followed by EOM. This is passed on, each processor adding a 1 when the EOM is received. When it reaches the controller it is passed to the next processor and rolled onto the clerk, and when the controller receives EOM a final 5 is added and a message is sent upward to start the next step. Each previously untyped processor receiving a 5 is of type PR1. Each pseudo-register is built in similar fashion, finishing the typing of all processors in O(log n) time.

4. Initializing pseudo-registers

Into PR1 we put the x coordinates of the clerk, into PB2 the y coordinates of the controller, and into PR3 we put L. These are similar, so we describe only initializing PR1. The apex first sends a 0 sign bit to all children, then a 0 to its two left children and a 1 to its two right children, simultaneously, followed by EOM. Each boss and supervisor passes this along, similarly adding a 0 for left-hand children and 1 for right-hand, with the number being rolled into PR1. (To initialize PR3, each boss only adds 0's, while the supervisor receiving only 0's adds a 1, with all other supervisors adding 0's.) After each pseudo-register is initialized a signal is sent upward to start the next step. Into PR4 and PR5 we put the x and y coordinates of one of the marked points, with PR6 and PR7 containing the coordinates of the other. To do this, the apex sends a signal which is passed to all base processors (not just controllers), causing each marked processor to send an EOM to its parent. The parent passes along an ordered pair of bits giving the relative x and y position of the son, followed by EOM. E.g., if the message came from the lower right son, a (0, 1) is sent up. As this moves up each successor adds on the relative position of the corresponding the message. Eventually a processor is reached where two of its sons are sending up a message, at which point it sends up an ordered quadruple with the first two components representing one point and the next two components representing the other. When this reaches the apex it is sent to all clerks, with the apex adding the sign bit. These numbers arrive at the pseudo-register low-order bit first, so each processor keeps the first bit it receives, and if a later bit arrives it passes the old one and keeps the new. After O(log n) time the registers are ready.

5. Calculations

Finally, the apex sends down a signal ordering the clerks to start calculating. Let (x1, y1) be the coordinates of one marked processor, (x2, y2) the coordinates of the other, and (xc, yc) the coordinates of the controller. We first compute an approximation to the slope of the line, which by assumption is between -1 and 1. Each clerk computes $y1 - y2$ in PR8 and places zero in PR9. It computes $x1 - x2$ in PR10 and divides this into the double word in PR8 – PR9, with the single word result in PR8. This value, denoted m, is exactly n times the (approximation of the) slope, that is, if we imagine PR8 as having a binary point before the initial bit then PR8 contains the slope.

Next each clerk sees if it is in a column between the marked points. It checks if

$$\min\{x1, x2\} \le xc \le \max\{x1, x2\},$$

and if not signals the controller's parent that it is done. Otherwise, $m * (xc - x1)$ is computed in PR9 – PR10. Let V be the contents of PR9. It is approximately the processor in row $V + y1$ which should be marked in this column, but because m/n only approximates the slope it may be that the processor in row $V + y1 - 1$ or $V + y1 + 1$ should be marked. To see which is correct, compute

$$E(v) = (y2 - y1) * (xc - x1) - v * (x2 - x1)$$

for $v = V, V + 1$ and $V - 1$,

and let Y be the value of v that minimizes abs(E(v)). (In case of a tie, let Y be the larger value.) $Y + y1$ is the correct row to mark. If $Y + y1 < yc$ or $Y + y1 \ge yc + L$, then the $(Y + y1)$st processor is in another clerk, in which case the clerk signals that it is done. Otherwise it computes $y = Y + y1 - yc$, for it is the $(y + 1)$st processor in this clerk which should be marked. Convert y to unary notation (e.g., 3 becomes 111), just as we start at the controller. This uses the entire word, not just the registers, and can be done in $O(\log n)$ time. The first processor not receiving a 1 of this unary notation is the one which is marked, and then the clerk is done.

Whenever the lower two sons of a supervisor, or all four sons of a boss, signal that they are done, that processor in turn tells its parent that it is done. The algorithm is finished, in $O(\log n)$ time, when the four sons of the apex signal that they are done. Finally, we need to choose k large enough to give enough pseudo-registers to perform all the calculations. The value $k = 15$ suffices.

6. Conclusions

The idea of automata self-organizing to form more complex units is biologically motivated and quite old (e.g., von Neumann [7]), but does not seem to have been used in the manner given here. Basically, the use of clerks allows a PCA to behave as if it were composed of processors with registers of lg n bits, where each operation takes O(log n) time. This technique converts simple algorithms for powerful machines into fast algorithms for simple machines. Clerks can be used with other configurations, and forthcoming papers will use them to solve some open problems concerning mesh-connected computers.

References

[1] C.R. Dyer, A fast parallel algorithm for the closest pair problem, Inform. Process. Lett. 11 (1980) 49–52.

[2] D.E. Knuth, The Art of Computer Programming Vol. 2: Seminumerical Algorithms (Addison-Wesley, Reading, MA, 1969).

[3] B. Sakoda, Parallel construction of polygonal boundaries from given vertices on a raster, Tech. Rept. CS81-21, Penn. State Univ., Computer Sci. Dept.

[4] O.F. Stout, Pyramid computer solutions of the closest pair problem, to appear.

[5] S.L. Tanimoto, Towards hierarchical cellular logic: design considerations for pyramid machines, Tech. Rept. 81-02-01, Univ. of Wash., Dept. of Computer Sci.

[6] S.L. Tanimoto and A. Klinger, eds., Structured Computer Vision: Machine Perception Through Hierarchical Computation Structures (Academic Press, New York, 1980).

[7] J. von Neumann, The Theory of Automata: Construction, Reproduction, and Homogeneity, A. Burks, ed. (Univ. of Illinois Press, Urbana, 1966).