$y_N = b_N - p_N y_{N-1}$. Reformulating in vector form, we find that

$$\begin{pmatrix} y_i \\ 1 \end{pmatrix} = \begin{pmatrix} -p_i & b_i \\ 0 & 1 \end{pmatrix} \begin{pmatrix} y_{i-1} \\ 1 \end{pmatrix}$$

for $i \geq 1$ (defining $p_1 = 0$), and thus that

$$\begin{pmatrix} y_i \\ 1 \end{pmatrix} = H_i \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

where $H_i = G_i \cdots G_1$ and $G_i = \begin{pmatrix} -p_i & b_i \\ 0 & 1 \end{pmatrix}$ for $1 \leq i \leq N$. Thus, we can compute the y_i's as a simple application of parallel prefix on 2×2 matrices. The solution to $U\vec{x} = \vec{y}$ can be computed in a similar fashion, except that we compute x_N first and then work backwards from there with the recurrences $x_i = (1/q_i)(y_i - u_i x_{i+1})$ for $1 \leq i \leq N-1$. In both cases, the prefix approach tends to be stable since we are dividing by diagonal elements at every step, instead of by off-diagonal elements.

The more challenging task is to find L and U such that $A = LU$. Multiplying L and U together as in Equation 1.22, we find that $q_1 = d_1$, $\ell_i = p_i q_{i-1}$, and $d_i = p_i u_{i-1} + q_i$ for $2 \leq i \leq N$. Reformulating, we find that $p_i = \ell_i / q_{i-1}$ and $q_i = d_i - \left(\frac{\ell_i u_{i-1}}{q_{i-1}}\right)$ for $2 \leq i \leq N$. The hard part is solving the recurrence for the q_i. Once this is done, computing each p_i is straightforward since $p_i = \ell_i / q_{i-1}$.

In order to express the recurrence $q_i = d_i - \left(\frac{\ell_i u_{i-1}}{q_{i-1}}\right)$ as a parallel prefix problem, we make the substitution $q_i = r_i / r_{i-1}$ where $r_0 = 1$ and $r_1 = d_1$. This results in the recurrence

$$r_i = d_i r_{i-1} - \ell_i u_{i-1} r_{i-2}$$

for $2 \leq i \leq N$, which is in a form that can be solved (as before) with parallel prefix. After solving for the r_i, we can plug back in and compute $q_i = r_i / r_{i-1}$ for $i \geq 2$ in a single step. Provided that the q_i are nonzero, the r_i will also be nonzero, and we will never have to worry about dividing by zero. Hence, if the LU-decomposition exists, and A is nonsingular, then it can be computed in $O(\log N)$ steps on an N-leaf complete binary tree.

1.3.4 Gaussian Elimination ⋆

In order to solve arbitrary systems of equations, we need to use a more robust (and expensive) technique known as *Gaussian elimination.* Gaussian elimination is one of the oldest and most widely known techniques for solving general systems of linear equations and inverting arbitrary nonsingular matrices. In this subsection, we describe how to implement a parallel version of Gaussian elimination on a two-dimensional array. As a result, we will be able to solve a variety of problems for $N \times N$ matrices in $O(N)$ steps.

We start with an algorithm for solving a system of linear equations $A\vec{x} = \vec{b}$. For simplicity, we will assume that A is an $N \times N$ nonsingular matrix, and, hence, that $\vec{x}$ has a unique solution. Extending the algorithm to the case when A is nonsquare or singular, or when $\vec{x}$ has no solution, is straightforward and left to the exercises.

Gaussian elimination is a process by which the matrix A is reduced to an upper triangular matrix U by a series of elementary row operations. In the case when A is nonsingular, the reduction continues until $U = I$, the identity matrix. An *elementary row operation* consists of multiplying a row by a scalar, switching two rows, or adding a multiple of one row to another. In each case, the result of applying a row operation to a matrix A can be expressed as a matrix product RA where R is a matrix associated with the row operation. For example, the matrix

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

serves to switch the first two rows of a 3×3 matrix, and

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -2 & 0 & 1 \end{pmatrix}$$

serves to subtract twice the first row from the third row.

Since $SA\vec{x} = S\vec{b}$ for any S, notice that applying the same sequence of row operations $S = R_r \cdots R_2 R_1$ to A and $\vec{b}$ results in an equivalent system of equations $A'\vec{x} = \vec{b}'$ where $A' = SA$ and $\vec{b}' = S\vec{b}$. If A is nonsingular and the row operations are chosen so that $SA = I$, then $\vec{x} = S\vec{b}$ and the system is solved.

The sequence of matrices and vectors formed during Gaussian elimination is easily represented by a sequence of $N \times (N+1)$ matrices of the form $\mathcal{A} = [A \mid \vec{b}]$. Notice that a row operation on A and $\vec{b}$ can be represented as a single (identical) row operation on $\mathcal{A}$. Hence, given a nonsingular A

and arbitrary $\vec{b}$, our goal is to perform a sequence of row operations R_1, $\ldots$, R_r on $\mathcal{A} = [A \mid \vec{b}]$ so that $R_r \cdots R_1 \mathcal{A} = [I \mid \vec{b}']$. Then the solution to the original system will simply be $\vec{x} = \vec{b}'$. In other words, the solution to $A\vec{x} = \vec{b}$ is precisely the last column of $R_r \cdots R_1 \mathcal{A}$.

An example of how Gaussian elimination can be used to solve a system of equations of this form is shown in Figure 1-47. At each step, we choose an elementary row operation that moves the first N columns of the current $\mathcal{A}$ closer to I. The first step in the example is to produce a 1 in the $(1,1)$ entry of $\mathcal{A}$. This is accomplished by multiplying the first row by a scalar, in this case 1/2. By subtracting appropriate multiples of the modified first row from the other rows, we then zero out the remaining entries in the first column. This is accomplished in steps 2 and 3, and the result is denoted by $\mathcal{A}^{(1)}$ in Figure 1-47. We next desire to produce a 1 in the $(2,2)$ entry of $\mathcal{A}^{(1)}$. Unlike before, this cannot be accomplished by simple scalar multiplication, since the $(2,2)$ entry of $\mathcal{A}^{(1)}$ is 0. Hence, we must first switch the second row with another (in this case the third) that contains a nonzero entry in the second column. This entry is then converted to a 1 in step 5 by scalar multiplication. Since the second entry of the new third row of $\mathcal{A}^{(1)}$ is already known to be zero, we don't have to worry about making it zero. Rather, we only need to worry about zeroing out the second entry of the first row. This is accomplished in the usual way to form $\mathcal{A}^{(2)}$ in step 6. The algorithm is completed by normalizing the $(3,3)$ entry of $\mathcal{A}^{(2)}$ to be 1 in step 7, and zeroing out the third entry of the first and second rows in steps 8 and 9. At this point, we have produced the identity matrix in the first three columns of $\mathcal{A}^{(3)}$, and, hence, the solution to the system of equations is contained in the last column.

In general, Gaussian elimination on an $N \times (N+1)$ matrix $\mathcal{A}$ consists of N phases. In the first phase, we identify the uppermost nonzero item in the first column of $\mathcal{A}$ and move the row containing that item ahead to become the first row. We then multiply this row by a scalar to produce a 1 in the $(1,1)$ position and subtract multiples of the new first row from the remaining rows so that all entries below the first in the first column become zero. The resulting matrix is called $\mathcal{A}^{(1)}$. In the second phase, we perform the same operations on the lower-right $(N-1) \times N$ submatrix of $\mathcal{A}^{(1)}$. We also subtract a multiple of the second row from the first in order to zero out the second entry of the first row. The resulting matrix is called $\mathcal{A}^{(2)}$.

$$A = \begin{pmatrix} 2 & 4 & -7 & 3 \\ 3 & 6 & -10 & 4 \\ -1 & 3 & -4 & 6 \end{pmatrix}$$

$$\text{step 1} \quad \begin{pmatrix} 1 & 2 & -7/2 & 3/2 \\ 3 & 6 & -10 & 4 \\ -1 & 3 & -4 & 6 \end{pmatrix}$$

$$\text{step 2} \quad \begin{pmatrix} 1 & 2 & -7/2 & 3/2 \\ 0 & 0 & 1/2 & -1/2 \\ -1 & 3 & -4 & 6 \end{pmatrix}$$

$$\text{step 3} \quad \begin{pmatrix} 1 & 2 & -7/2 & 3/2 \\ 0 & 0 & 1/2 & -1/2 \\ 0 & 5 & -15/2 & 15/2 \end{pmatrix}$$

$$\text{step 4} \quad \begin{pmatrix} 1 & 2 & -7/2 & 3/2 \\ 0 & 5 & -15/2 & 15/2 \\ 0 & 0 & 1/2 & -1/2 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 2 & -7/2 & 3/2 \\ 0 & 1 & -3/2 & 3/2 \\ 0 & 0 & 1/2 & -1/2 \end{pmatrix} \quad \text{step 5}$$

$$\begin{pmatrix} 1 & 0 & -1/2 & -3/2 \\ 0 & 1 & -3/2 & 3/2 \\ 0 & 0 & 1/2 & -1/2 \end{pmatrix} \quad \text{step 6}$$

$$\begin{pmatrix} 1 & 0 & -1/2 & -3/2 \\ 0 & 1 & -3/2 & 3/2 \\ 0 & 0 & 1 & -1 \end{pmatrix} \quad \text{step 7}$$

$$\begin{pmatrix} 1 & 0 & 0 & -2 \\ 0 & 1 & -3/2 & 3/2 \\ 0 & 0 & 1 & -1 \end{pmatrix} \quad \text{step 8}$$

$$\begin{pmatrix} 1 & 0 & 0 & -2 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1 \end{pmatrix} \quad \text{step 9}$$

Figure 1-47 *Using Gaussian elimination to solve the system of equations* $\begin{pmatrix} 2 & 4 & -7 \\ 3 & 6 & -10 \\ -1 & 3 & -4 \end{pmatrix} \vec{x} = \begin{pmatrix} 3 \\ 4 \\ 6 \end{pmatrix}$. *Examining the rightmost column of* $A^{(3)}$, *we find that* $\vec{x} = \begin{pmatrix} -2 \\ 0 \\ -1 \end{pmatrix}$.

We continue in this fashion for N phases, at which point we will have produced the identity matrix in the first N columns of $\mathcal{A}^{(N)}$. The solution to the original system of equations will then reside in the last column of $\mathcal{A}^{(N)}$.

It is a simple fact of linear algebra that this process of successively interchanging rows, normalizing diagonal entries to 1, and zeroing out nondiagonal entries is always guaranteed to work provided that A is nonsingular. This is because the inability to find a nonzero diagonal element at the beginning of any phase would imply that the determinant of the original matrix is zero, thereby implying that A is singular.

The implementation of Gaussian elimination on an $N \times (N+1)$ array is quite straightforward, although the notation involved can be a bit tedious. To simplify matters, we will start by describing how to implement the first phase on an $(N+1)$-cell linear array.

The first step in the first phase is to find the uppermost nonzero entry a_{t1} in the first column of $\mathcal{A}$. The row containing a_{t1} (row t) then becomes the first row of $\mathcal{A}^{(1)} = (a_{ij}^{(1)})$. After finding this row, we normalize it so that $a_{11}^{(1)} = 1$. This is accomplished by multiplying each value in the row by $1/a_{t1}$. In other words, $a_{1j}^{(1)} \leftarrow a_{tj}/a_{t1}$ for $1 \leq j \leq N+1$ (where for simplicity we define $a_{i,N+1}$ to be b_i for $1 \leq i \leq N$). Next, we subtract multiples of this row from subsequent rows of $\mathcal{A}$ so as to produce zeros in all subsequent entries of the first column. This is accomplished by subtracting a_{i1} copies of the new first row from the ith row for $t < i \leq N$. More precisely, we compute $a_{ij}^{(1)} \leftarrow a_{ij} - a_{i1}a_{1j}^{(1)}$ for $i > t$ and $1 \leq j \leq N+1$. Note that entries in previous rows ($i < t$) are left unchanged since their first-column entries are already known to be zero.

All of these operations can be performed quite simply by an $(N+1)$-cell linear array. The inputs are arranged so that the jth column of $\mathcal{A}$ enters the top of the jth cell (counting from left to right) of the array starting with a_{1j} at step j. The first cell of the array scans for the first nonzero entry, ignoring zero entries. When a nonzero entry a_{t1} is found, it is inverted and sent rightward. The jth cell in the array ($j > 1$) simply passes downward the inputs received from above (after holding each for one step) until it receives a value ($1/a_{t1}$) from the left. This will happen at step $j+t-1$, at which point, the cell multiplies the current input from above (a_{tj}) by the input from the left ($1/a_{t1}$) and then saves the result. The value from the left ($1/a_{t1}$) is passed rightward, but nothing is passed downward. At this point, the jth cell has just computed $a_{1j}^{(1)} = a_{tj}/a_{t1}$.

After seeing and inverting the first nonzero input, the first cell in the array simply passes remaining inputs rightward. The subtraction operations are performed by the interior cells of the array. In particular, the jth cell subtracts the product of the saved value ($a_{1j}^{(1)}$) times the left input (a_{i1}) from the top input (a_{ij}) at each step following the calculation of $a_{1j}^{(1)}$. The result is output below, and the left input (a_{i1}) is passed rightward. Hence, the jth cell computes and outputs

$$a_{ij}^{(1)} = a_{ij} - a_{i1}a_{1j}^{(1)}$$

at step $j + i - 1$ for $i > t$. As an example, we have illustrated this process for a 3×4 matrix in Figure 1-48. Following the notation adopted in Subsection 1.3.2, notice that only the circular cell need perform divisions.

The preceding algorithm takes $\mathcal{A}$ as input from above, saves the first row of $\mathcal{A}^{(1)}$, and outputs the lower-right $(N-1) \times N$ submatrix of $\mathcal{A}^{(1)}$ below. By placing an N-cell linear array just below the rightmost N cells of the $(N+1)$-cell linear array, we can also save the second row of $\mathcal{A}^{(2)}$, and output the lower-right $(N-2) \times (N-1)$ submatrix of $\mathcal{A}^{(2)}$. The computation proceeds exactly as before. The only task remaining in Phase 2 is to subtract $a_{12}^{(1)}$ times the second row of $\mathcal{A}^{(2)}$ from the first row. This is accomplished in the same fashion as with the other rows by simply passing the values saved by the first linear array downward once all other rows have passed through them. The output from the second linear array will then consist of the lower $N-2$ rows of $\mathcal{A}^{(2)}$ in a staggered fashion, followed by the first row. So that later phases can proceed in a similar way, this data is followed by the second row of $\mathcal{A}^{(2)}$, and then by a row of end-of-matrix markers to let the cells in the subsequent linear arrays know when to pass on their stored values downward.

All N phases can be performed by the upper-right portion of an $N \times (N+1)$ mesh, such as that shown in Figure 1-49. The kth phase of the algorithm is performed by the kth row of the mesh. For each k, the kth row

1) takes $\mathcal{A}^{(k-1)}$ as input starting with rows $k, k+1, \ldots, N$ and finishing with rows $1, 2, \ldots, k-1$,
2) computes and stores the kth row of $\mathcal{A}^{(k)}$ by computing $1/a_{tk}^{(k-1)}$ in cell (k,k) and $a_{kj}^{(k)} = a_{tj}^{(k-1)}/a_{tk}^{(k-1)}$ in cell (k,j) for $j > k$ where t is the smallest value in $[k, k+1, \ldots, N]$ such that $a_{tk}^{(k-1)} \neq 0$, and

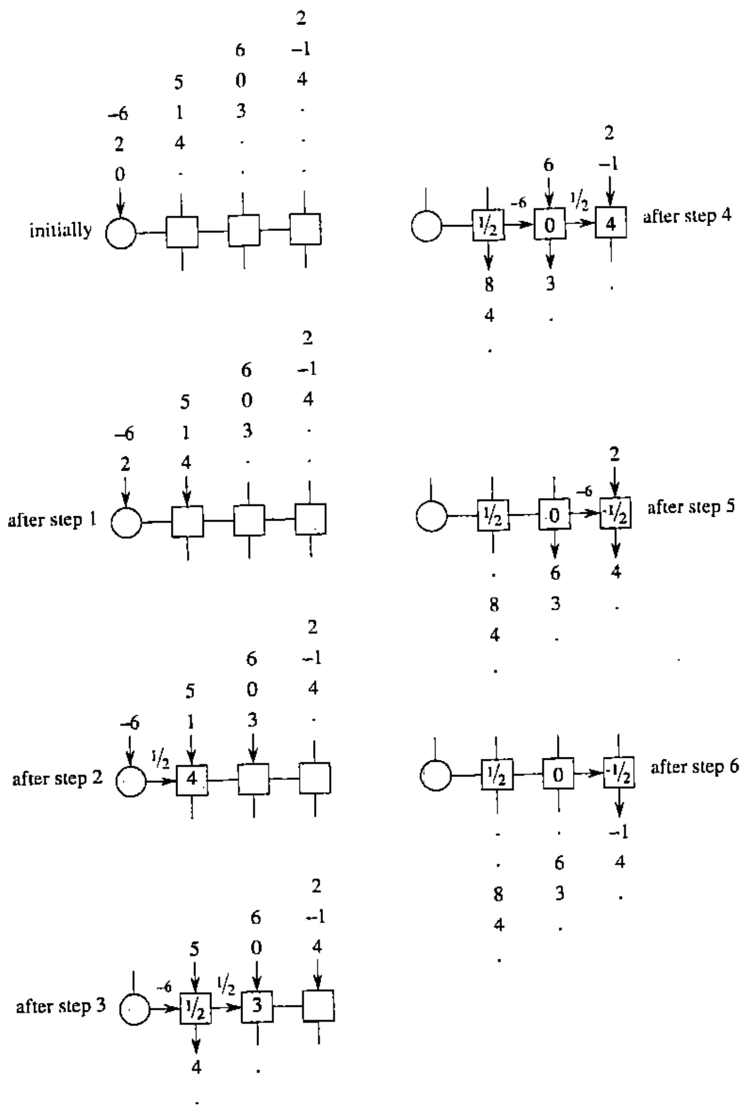


Figure 1-48 *Computing the first phase of Gaussian elimination on a 3×4 matrix A. The first row of $A^{(1)}$ is stored in the array. Subsequent rows of $A^{(1)}$ are output in a staggered fashion. Since the entries in the first column of $A^{(1)}$ are zeroed out, they do not need to be output by the leftmost cell.*

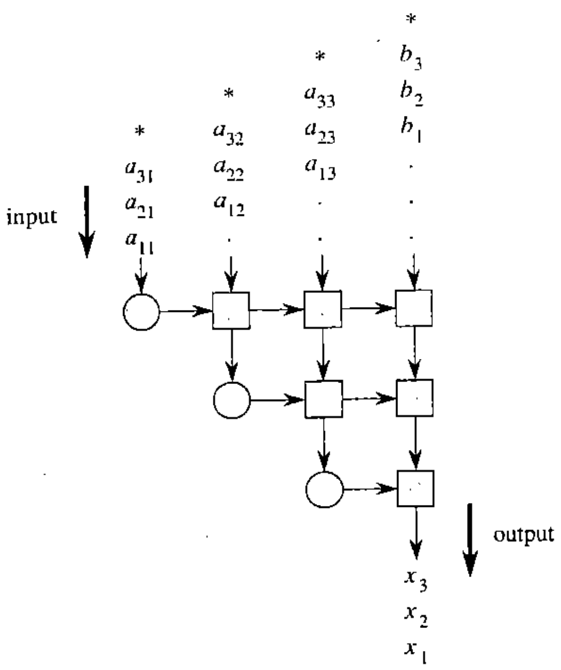


Figure 1-49 *Network used to solve a* 3×3 *system of equations* $A\vec{x} = \vec{b}$. *Circular cells perform division. Square cells multiply and subtract. Asterisks denote an end-of-matrix marker.*

3) computes and outputs $A^{(k)}$ starting with rows $k+1, \ldots, N$ and finishing with rows $1, \ldots, k$ by computing

$$a_{ij}^{(k)} = a_{ij}^{(k-1)} - a_{ik}^{(k-1)} a_{kj}^{(k)} \qquad (1.23)$$

for $i \neq k$ in cell j for $j > k$.

The solution to the original system of equations will be output at the bottom of the bottom-right cell of the mesh in order $x_1, x_2, \ldots, x_N$. The total time required is $4N - 1$ steps.

Although the overall implementation of Gaussian elimination may have seemed complicated, the individual action of each cell is quite simple. Each circular cell simply waits for the first nonzero input from above, whereupon it inverts this input and passes it rightward. Subsequent inputs are also passed rightward. Each square cell simply passes inputs from above downward (after holding them for one step) until it encounters an input from

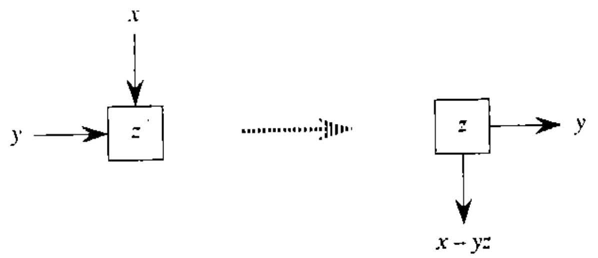
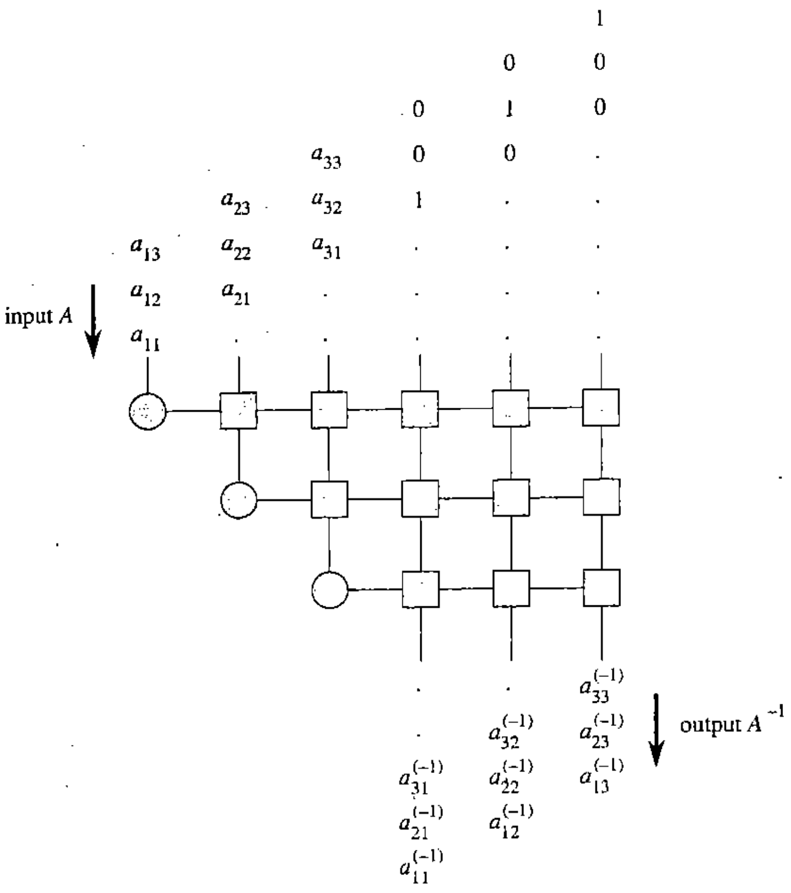


Figure 1-50 *The basic operation of a nondiagonal cell in Gaussian elimination.*

the left, whereupon it multiplies the left input by the top input and saves the product. The left input is passed rightward, and nothing is passed downward. In subsequent steps, square cells multiply the left input by the stored value and subtract the product from the top input. The result is immediately passed downward (i.e., the value from the top is no longer held for one step before being passed downward since there is already a value being stored in memory), and the left input is passed rightward. For example, see Figure 1-50. Eventually, an end-of-matrix marker is detected, whereupon the cell passes its stored value downward, followed by the marker at the next step.

The algorithm for solving systems of equations just described can be easily extended to handle matrix inversion and other common problems in linear algebra. For example, we can invert an $N \times N$ matrix A by simply performing Gaussian elimination on the $N \times 2N$ matrix $\mathcal{A} = [A \mid I]$. If A is nonsingular, the sequence of elementary row operations used to reduce A to I will also transform I into A^{-1}. This is because if $SA = I$, then $SI = A^{-1}$. Hence, the algorithm can be implemented on the upper-right portion of an $N \times 2N$ mesh, as shown in Figure 1-51. The function of the cells is the same as before, and A^{-1} is output from the rightmost N columns of the array, leading rows first. The total time required is $5N - 2$ steps.

Notice that if we have several matrices to invert, then we can invert them at a rate of one matrix every N steps by simply *pipelining* the matrices into the network one after the other. More precisely, we start inputting the jth column of the mth matrix into the jth column of the mesh immediately after the jth column of the $(m-1)$st matrix has been entered $(1 \le j \le N)$. We must be careful to attach the end-of-matrix markers for



input A
a_{11} a_{12} a_{13}
a_{21} a_{22} a_{23}
a_{31} a_{32} a_{33}
1 0 0
0 1 0
0 0 1
output A^{-1}
$a_{11}^{(-1)}$ $a_{21}^{(-1)}$ $a_{31}^{(-1)}$
$a_{12}^{(-1)}$ $a_{22}^{(-1)}$ $a_{32}^{(-1)}$
$a_{13}^{(-1)}$ $a_{23}^{(-1)}$ $a_{33}^{(-1)}$

Figure 1-51 *Network used to invert a* 3×3 *matrix* A *by Gaussian elimination. The* (i,j) *component of* $A^{(-1)}$ *is denoted by* $a_{ij}^{(-1)}$.

the $(m-1)$st matrix to the entries in the first row of the mth matrix, and to reset the program in each cell whenever it sees such a marker. The *delay* of the resulting algorithm is still $5N-2$ since it takes that many steps to invert any particular matrix, but the solution *rate* is one problem for each N steps.

Gaussian elimination is known to be numerically stable for several natural classes of matrices. For example, when dealing with symmetric positive definite or diagonally dominant matrices, we don't even need to switch rows to ensure that the leading diagonal element in each phase is nonzero. For most classes of matrices, however, the stability of the algorithm is dramatically improved by making sure to pivot on large entries within each column. Unfortunately, the task of implementing Gaussian elimination with pivoting on an array is difficult. Although several variations of Gaussian elimination with pivoting have been devised, most take more than $\Theta(N)$ steps to implement on an $N \times N$ array. Several such variations are discussed in the exercises, as are modifications of the algorithm to compute the determinant, rank, or PLU-decomposition of a matrix. In addition, a more efficient implementation of Gaussian elimination with pivoting on a mesh of trees interconnection network will be discussed in Chapter 2.

In conclusion, we note how surprising it is that such a simple interconnection of simple cells can be used to compute such a powerful algorithm. Yet this is a phenomenon that we shall observe over and over throughout the text. In fact, this phenomenon will be especially apparent in Section 1.5, where we use a virtually identical network and algorithm to solve a variety of very-different-looking graph problems.

1.3.5 Iterative Methods ⋆

All of the algorithms discussed so far in this section find *exact* solutions to systems of equations and related problems provided that the calculations are done with infinite precision. In reality, of course, calculations are often done with imperfect finite precision, and the calculated solutions are only approximations to the real solutions. If approximations to the real solutions are acceptable, then it makes sense to consider *iterative methods* as an alternative to the exact methods described in Subsections 1.3.3 and 1.3.4. Iterative methods work by continually refining an initial approximate solution so that it becomes closer and closer to the correct solution. For example, the Newton iteration algorithm for division is an iterative algorithm. In some cases, iterative algorithms require substantially less