Here the last equation has a single unknown, and we can solve for x_4 directly. Once x_4 is known, it is easy to solve for x_3. The algorithm continues in this fashion until the values for the rest of the variables have been determined.

Let's consider the form of the equations in the tridiagonal system. Except for the first and last equations, each equation has three variables in it.

$$\begin{aligned} g_1x_1 + h_1x_2 &= b_1 \\ f_ix_{i-1} + g_ix_i + h_ix_{i+1} &= b_i \qquad 2 \le i \le n-1 \\ f_nx_{n-1} + g_nx_n &= b_n \end{aligned} \tag{9.13}$$

A sequential algorithm to solve a tridiagonal system of equations appears in Fig. 9-7. The algorithm has two for loops, each performing $n-1$ iterations (assuming the size of the linear system is n). The steps within each for loop require constant time. Hence the complexity of the sequential algorithm to solve a tridiagonal system of linear equations is $\Theta(n)$.

FIGURE 9-7 Sequential algorithm to solve a system of tridiagonal equations. The algorithm performs $9n - 8$ floating-point operations, assuming the expression $f[i+1]/g[i]$ is evaluated only once per iteration of the first for loop.

TRIDIAGONAL.SYSTEM.SOLVER (SISD):

```
{This algorithm solves the set of equations
   g1 x1 + h1 x2 = b1
   fi xi-1 + gi xi + hi xi+1 = bi for 1 < i < n
   fn xn-1 + gn xn = bn
}

  Global   n                          {Size of tridiagonal system}
           f[2...n], g[1...n], h[1...(n - 1)]   {Coefficients of x}
           b[1...n]                   {Constant vector}
           x[1...n]                   { Solution vector }
begin
  for i ← 1 to n - 1 do
    g[i + 1] ← g[i + 1] - (f[i + 1]/g[i]) × h[i]
    b[i + 1] ← b[i + 1] - (f[i + 1]/g[i]) × b[i]
  endfor
  for i ← n downto 2 do
    x[i] ← b[i] / g[i]
    b[i - 1] ← b[i - 1] - x[i] × h[i - 1]
  endfor
  x[1] ← b[1] / g[1]
end
```

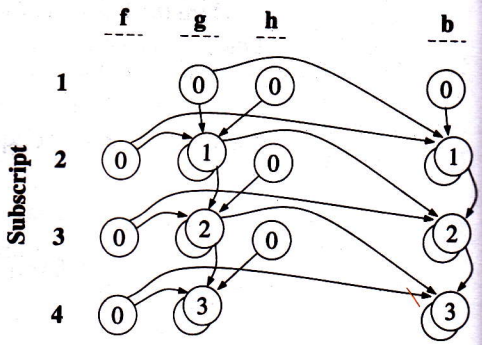


FIGURE 9-8 Data flow diagram illustrating data dependencies in the first loop of the tridiagonal-system solver of Fig. 9-7, for a system of size 4. The number inside each vertex is the length of the longest path preceding the vertex. The overlapping circles represent instances where the new value of a variable is computed from an expression containing its previous value.

We can determine if an algorithm can be parallelized by exploring its data dependencies. Figure 9-8 is a data-flow diagram for the first loop of the algorithm solving a system of size 4. The overlapping circles represent instances where the new value of a variable is computed from an expression containing its previous value. The data flow diagram makes clear that this particular algorithm is not amenable to parallelization.

To achieve some parallelism, we must take another problem solving approach. First, we want to represent all n equations the same way. We can do this by introducing pseudovariables x_0 and x_{n+2}, both having value 0. Now we are able to write

$$f_i x_{i-1} + g_i x_i + h_i x_{i+1} = b_i \qquad 1 \le i \le n \tag{9.14}$$

Rewriting equation 9.14 to solve for x_i we get

$$x_i = (b_i - f_i x_{i-1} - h_i x_{i+1}) / g_i \qquad 1 \le i \le n \tag{9.15}$$

If we introduce two more pseudovariables x_{-1} and x_{n+2}, both having value 0, we can use equation 9.15 to substitute for x_{i-1} and x_{i+1} in equation 9.14:

$$f_i \left(\frac{b_{i-1} - f_{i-1} x_{i-2} - h_{i-1} x_i}{g_{i-1}} \right) + g_i x_i +$$

$$h_i \left(\frac{b_{i+1} - f_{i+1} x_i - h_{i+1} x_{i+2}}{g_{i+1}} \right) = b_i \qquad 1 \le i \le n \tag{9.16}$$

To simplify this equation we define

$$\begin{aligned} \gamma_i &= f_i / g_{i-1} \quad 1 \le i \le n \\ \delta_i &= h_i / g_{i+1} \quad 1 \le i \le n \end{aligned} \tag{9.17}$$

Now we can rewrite it as

$$-\gamma_i f_{i-1} x_{i-2} + (g_i - \gamma_i h_{i-1} - \delta_i f_{i+1}) x_i - \delta_i h_{i+1} x_{i+2} =$$

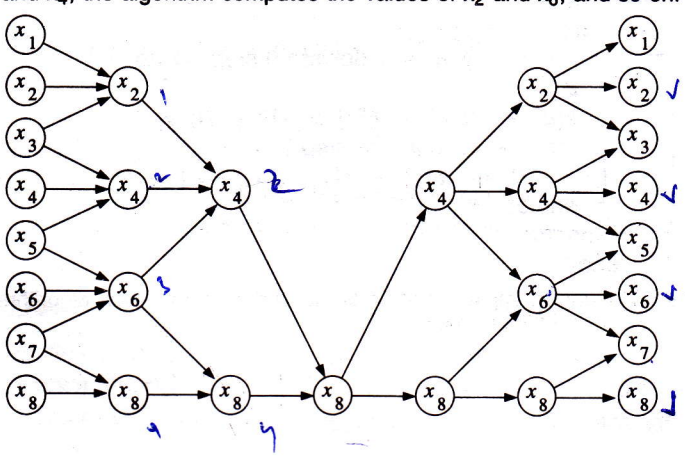
$$b_i \mp \gamma_i h_{i-1} \mp \delta_i h_{i+1} \qquad 1 \le i \le n \tag{9.18}$$

By this transformation we have expressed the value of x_i in terms of x_{i-2} and x_{i+2}. Taken as a whole, the equations for $x_2, x_4, x_6, \ldots, x_n$ form a tridiagonal system with $n/2$ variables. Applying this technique recursively yields a divide-and-conquer algorithm, called **odd-even reduction** or **cyclic reduction**, to solve a tridiagonal system of linear equations. The odd-even reduction algorithm was first published by Hockney (1965).

Figure 9-9 illustrates the odd-even reduction algorithm for a system of size 8. In the first step the coefficients of x_1, x_3, x_5, and x_7 are eliminated; the remaining system is tridiagonal and has only the variables with even indices. In the second step the coefficients of x_2 and x_6 are eliminated. In step three x_4 is eliminated, and we are left with a single equation in a single unknown. Once the value of x_8 is known, the value of x_4 can be found. With the values of x_8 and x_4 computed, the algorithm can determine the values of x_2 and x_6, and once x_2, x_4, x_6, and x_8 are known, the algorithm finds values for x_1, x_3, x_5, and x_7.

The odd-even reduction algorithm appears in pseudocode in Fig. 9-10. The time complexity of odd-even reduction is $\Theta(n)$, the same as that of the sequential algorithm presented earlier. However, odd-even reduction is much more amenable to parallelization, as shown in Fig. 9-9, as all three inner for loops can be executed in parallel. We have implemented a parallel odd-even reduction algorithm on the Sequent Symmetry system and measured its performance solving a tridiagonal system of size 65,536. The parallelizability and speedup of this algorithm are shown in Fig. 9-11.

FIGURE 9-9 This diagram illustrates how the odd-even reduction algorithm eliminates variables in a tridiagonal system of size 8. In the first step variables x_1, x_3, x_5, and x_7 are eliminated. In the second step variables x_2 and x_6 are eliminated. In step three variable x_4 is eliminated. Here the equation has a single variable—x_8—whose value can be computed directly. Once the value of x_8 is known, the algorithm solves for x_4. Given values for x_8 and x_4, the algorithm computes the values of x_2 and x_6, and so on.



```
ODD.EVEN.REDUCTION (SISD):

{This algorithm solves a system of tridiagonal equations}

Global  n                                   {Number of equations in linear system}
        f[1...n], g[1...n], h[1...n]        {Coefficients of the tridiagonal equations}
        b[1...n]                            {Constant vector}
        new.f[1...n], new.g[1...n],         {Newly computed coefficients}
        new.h[1...n], new.b[1...n]
        x[1...n]                            {Solution vector}
        d                                   {Distance between terms being combined}
        γ[1..n], δ[1..n]                    {Temporaries}

begin
  for i ← 0 to log n − 1 do
    d ← 2^i
    for j ← 2 × i + 1 to n − 1 step 2 × d do
      γ[j] ← f[j] /g[j − d]
      δ[j] ← h[j]/g[j + d]
      new.f[j] ← −γ[j] × f[j − d]
      new.g[j] ← − δ[j] × f[j + d] − γ[j] × h[j − i]
      new.h[j] ← δ[j] × h[j + d]
      new.b[j] ← b[j] + γ[j] × b[j − d] + δ[j] × b[j + d]
    endfor
    γ[n] ← f[n] /g[n − d]
    f[n] ← −γ[j] × f[n − d]
    g[n] ← g[n] − γ[n] × h[n − d]
    b[n] ← b[n] + γ[n] × b[n − d]
    for j ← 2 × i + 1 to n − 1 step 2 × d do
      f[j] ← new.f[j], g[j] ← new.g[j], h[j] ← new.h[j], b[j] ← new.b[j]
    endfor
  endfor
  x[n] ← b[n] / g[n]
  for i ← log n − 1 downto 0 step −1 do
    d ← 2^i
    x[d] ← (b[d] − h[d] × x[2 × d])/g[d]
    for j ← 3 × d to n step 2 × d
      x[j] ← (b[j] − f[j] × x[j − d] − h[j] × x[j + d])/g[j]
    endfor
  endfor
end
```

FIGURE 9-10 Sequential implementation of the odd-even reduction algorithm to solve a tridiagonal system of linear equations.