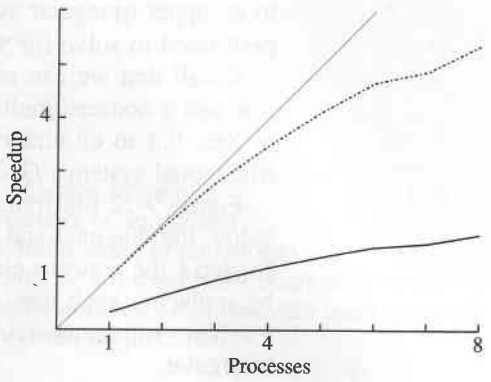


FIGURE 9-11 Speedup (solid line) and parallelizability (dotted line) of parallel odd-even reduction on Sequent Symmetry on a tridiagonal system of size 65,636.

As Fig. 9-11 makes clear, although the algorithm is easily parallelizable, its speedup is poor. Let's explore the reasons for this.

The total number of floating-point operations performed by the sequential algorithm of Fig. 9-7 is

$$\sum_{i=1}^{n-1} 6 + \sum_{i=2}^{n} 3 + 1 = 9n - 8 \tag{9.19}$$

The total number of floating-point operations performed by the odd-even reduction algorithm is

$$\sum_{i=1}^{\log n-1} \left(\left(\sum_{j=1}^{n/2^i-1} 13 \right) + 7 \right) + 1 + \sum_{i=1}^{\log n-1} \left(\left(\sum_{j=1}^{n/2^i-1} 5 \right) + 3 \right) =$$

$$13\,(n - 2 - (\log n - 1)) + 7(\log n - 1) + 1 + 5(n - 2 - (\log n - 1)) \tag{9.20}$$
$$+3(\log n - 1) = 18n - 8\log n - 27$$

Taking the ratio of equations 9.19 and 9.20

$$\lim_{n\to\infty} \left(\frac{9n - 8}{18n - 8\log n - 27} \right) = \frac{1}{2} \tag{9.21}$$

We should not expect a parallel implementation of odd-even reduction to exhibit an efficiency greater than 50 percent.

9.4 GAUSSIAN ELIMINATION

In this section we describe the parallelization of gaussian elimination, a well-known algorithm for solving the linear system $\mathbf{Ax} = \mathbf{b}$ when the matrix $\mathbf{A}$ has nonzero elements in arbitrary locations. Gaussian elimination reduces $\mathbf{Ax} = \mathbf{b}$

to an upper triangular system $\mathbf{Tx} = \mathbf{c}$, at which point back substitution can be performed to solve for $\mathbf{x}$.

Recall that we can replace any row of a linear system by the sum of that row and a nonzero multiple of any row of the system. We used this technique in Sec. 9.3 to eliminate nonzero elements from below the main diagonal in tridiagonal systems. Gaussian elimination uses the same technique.

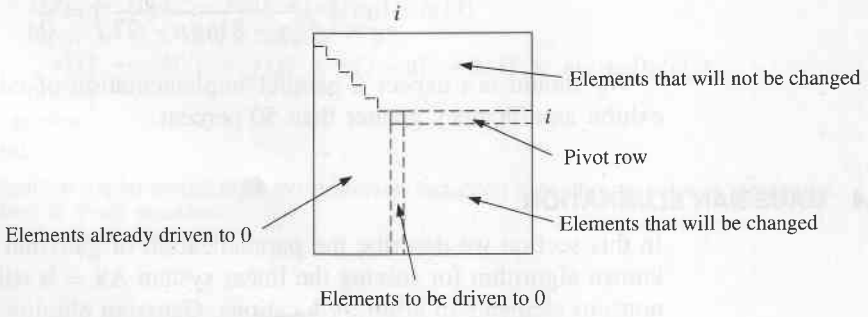
Figure 9-12 illustrates one iteration of the algorithm. All nonzero elements below the diagonal and to the left of column i have already been eliminated. In step i the nonzero elements below the diagonal in column i are eliminated by replacing each row j, where $i + 1 \leq j \leq n$, with the sum of row j and $-a_{j,i}/a_{i,i}$ times row i. After $n - 1$ such iterations, the linear system is upper triangular.

In the straightforward gaussian elimination algorithm just described, row i is the **pivot row**, i.e., the row used to drive to zero all nonzero elements below the diagonal in column i. This approach does not exhibit good numerical stability on digital computers. However, a simple variant, called **gaussian elimination with partial pivoting**, does produce reliable results. In step i of gaussian elimination with partial pivoting, rows i through n are searched for the row whose column i element has the largest absolute value. This row is swapped (pivoted) with row i. Here the algorithm uses multiples of the pivot row, now stored as row i, to reduce to zero all nonzero elements of column i in rows $i + 1$ though n.

A sequential gaussian elimination algorithm appears in Fig. 9-13. Rather than actually swapping the pivot row and row i in each iteration, the algorithm makes use of indirection. Array element $pivot[i]$ contains the iteration number in which row i was used as the pivot row. Another array is introduced to make it easy to determine if a particular row has been used as a pivot row; array element $marked[i]$ is set to 1 when row i is chosen as a pivot row.

Let's determine how well-suited gaussian elimination is to parallelization. First, we count every arithmetic operation and comparison involving floating-

FIGURE 9-12 During iteration i of the gaussian elimination algorithm, all elements in column i for each row j below row i are driven to 0 by subtracting a multiple of row i from row j.



```
GAUSSIAN.ELIMINATION (SISD):

Global   n                  {Size of linear system}
         a[1...n][1...n]    {Coefficients of equations}
         b[1...n]           {Right-hand sides of equations}
         marked[1...n]      {Indicates which rows have been pivot rows}
         pivot[1...n]       {Indicates iteration each row was used as pivot}
         picked             {Row picked as pivot row}

begin
  for i ← 1 to n do
    marked[i] ← 0
  endfor
  for i ← 1 to n – 1 do
    tmp ← 0
    for j ← i to n do
      if marked[j] = 0 and |a[j][i]| > tmp then
        tmp ← |a[j][i]|
        picked ← j
      endif
    endfor
    marked[picked] ← 1
    pivot[picked] ← i
    for j ← 1 to n do
      if marked[j] = 0 then
        tmp ← a[j][i] / a[picked][i]
        for k ← i + 1 to n do
          a[j][k] ← a[j][k] – a[picked][k] × tmp
        endfor
        b[j] ← b[j] – b[k] × tmp
      endif
    endfor
  endfor
  for i ← 1 to n do
    if marked[i] = 0 then
      pivot[i] ← N
      break
    endif
  endfor
end
```

FIGURE 9-13 Sequential gaussian elimination algorithm with partial pivoting. The algorithm assumes the linear system is nonsingular, i.e., has a solution.

point numbers in the algorithm of Fig. 9-13. The algorithm has $n-1$ iterations, where the iteration number i varies from 1 to $n-1$. During iteration i there are $n-i$ comparison steps to determine the pivot row. Once the pivot row is known, the algorithm must reduce $n-i$ rows, where each row-reduction step requires $2(n-i)$ floating-point operations to modify the coefficients of **A**, and two more floating-point operations to modify the row's entry in **b**. Hence the total number of floating-point operations and comparisons is

$$\sum_{i=1}^{n-1}\left((n-i+1)+\sum_{j=i+1}^{n}\left(1+\sum_{k=i+1}^{n+1}2\right)\right)=$$

$$\sum_{i=1}^{n-1}\left((n-i+1)+\sum_{j=i+1}^{n}(2n-2i+3)\right)= \tag{9.22}$$

$$\sum_{i=1}^{n-1}\left(2n^2+4n+1+2i^2-4i(n+1)\right)=$$

$$\frac{2n^3+3n^2-2n-3}{3}$$

Most of these operations occur inside the innermost for loop. A study of the algorithm's data dependencies reveals that both the innermost for loop indexed by k and the middle for loop indexed by j can be executed in parallel. In other words, once the pivot row has been found, the modifications to all unmarked rows may occur simultaneously. Within each row, once the multiplier $a[j][i]/a[picked][i]$ has been computed, modifications to elements $i+1$ through n of each row may occur simultaneously. Since most of the operations counted in the last equation occur inside these for loops, the algorithm seems to be well-suited to parallelization.

Let's consider the implementation of gaussian elimination on a multicomputer. Assume n is a multiple of p. How should we distribute the elements of matrices a and b to the memories of the individual processors? If we examine the data flow of the algorithm for iteration i (Fig. 9-14), we see that determining the pivot row $picked$ requires that data items in column i be compared, while determining the new value of a particular element $a[j][k]$ requires three values besides the current value of $a[j][k]$. These values are $a[j][i]$, $a[picked][i]$, and $a[picked][k]$.

Clearly the data distribution determines the points in the algorithm where communication is required. Suppose we assign to each processor a contiguous group of rows of a and the associated elements of b (Fig. 9-15). Given this distribution of data, the processors must interact in order to determine the pivot row. Once the pivot row has been determined, the processor owning the pivot row must broadcast its elements to the other processors, so that they

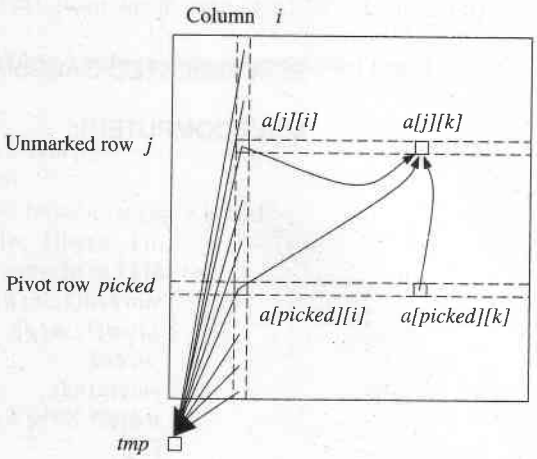
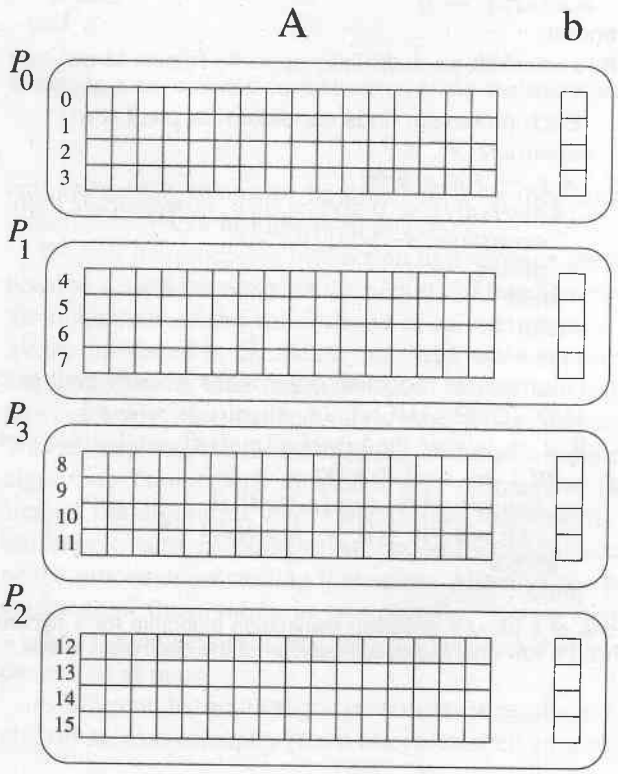


FIGURE 9-14 Data dependencies during a single iteration of the gaussian elimination algorithm. During iteration *i* the column *i* element of every unmarked row must be examined to determine the identity of the pivot row *picked*. Once the pivot row has been identified, every element *a[j][k]* of every unmarked row *j* must be modified, which requires accessing *a[j][i]*, *a[picked][i]*, and *a[picked][k]*.

FIGURE 9-15 A row-oriented decomposition of the data for the multicomputer-targeted gaussian elimination algorithm. In this example $n = 16$ and $p = 4$.



```
ROW.ORIENTED.GAUSSIAN.ELIMINATION (HYPERCUBE
MULTICOMPUTER):

  Local   n                  {Size of linear system}
          a[1...n/p][1...n]  {Coefficients of equations}
          b[1...n/p]         {Right-hand sides of equations}
          marked[1...n/p]    {Indicates which rows have been pivot rows}
          pivot[1...n/p]     {Indicates iteration each row was used as pivot}
          picked             {Row picked as pivot row}
          magnitude          {Pivot value}
          winner             {Processor controlling pivot row}
          i, j
begin
  for all P_id, where 1 ≤ id ≤ p do

    {Initially no rows have been used as pivot rows}
    for i ← 1 to n/p do
      marked[i] ← 0
    endfor
    for i ← 1 to n − 1 do

      {Each processor finds candidate for pivot row}
      magnitude ← 0
      for j ← i to n/p do
        if marked[j] = 0 and |a[j][i]| > magnitude then
          magnitude ← |a[j][i]|
          picked ← j
        endif
      endfor
      winner ← id
      {Tournament reduction determines globally best pivot row}
      MAX.TOURNAMENT (id, magnitude, winner)
      if id = winner then marked[picked] ← 1 pivot[picked] ← i
        for j ← i + 1 to n do
          tmp.vector[j] ← a[picked][j]
          tmp.vector[n + 1] ← b[picked]
        endfor
      endif
```

FIGURE 9-16 Part one of a parallel gaussian elimination algorithm for a hypercube multicomputer, assuming a row-oriented decomposition of the coefficient matrix.

```
      {Processor owning pivot row broadcasts it to other processors}
      HYPERCUBE.BROADCAST (id, winner, tmp.vector[(i + 1)..(n + 1)])

      {Processors eliminate column i values in their unmarked rows}
      for j ← 1 to n/p do
        if marked[j] = 0 then
          tmp ← a[j][i] / tmp.vector[i]
          for k ← i + 1 to n do
            a[j][k] ← a[j][k] − tmp.vector[k] × tmp
          endfor
          b[j] ← b[j] − tmp.vector[n + 1] × tmp
        endif
      endfor
    endfor

    {Locate row never used as a pivot row}
    for i ← 1 to n/p do
      if marked[i] = 0 then
        pivot[i] ← n
        break
      endif
    endfor
  endfor
end
```

FIGURE 9-17 Part two of parallel gaussian elimination algorithm for a hypercube multicomputer, assuming a row-oriented decomposition of the coefficient matrix.

can update the values of the umarked rows they control. Pseudocode for this algorithm appears in Figs. 9-16 and 9-17.

We call the processor interaction to determine the pivot row a **tournament**, because we are interested in the identity of the pivot row (the winner) more than the magnitude of the value stored at column i in the pivot row (the score). If we are interested in the identity of the processor with the largest value, we call the interaction a **max-tournament**. If we are interested in the identity of the processor with the smallest value, we call the interaction a **min-tournament**. We can implement a tournament algorithm as a simple variant of the reduction algorithm. Figure 9-18 illustrates max-tournament for a hypercube. At each step of the algorithm, two variables are maintained. Variable *value* contains the largest value encountered so far, and variable *winner* contains the number of the processor submitting that value. Although the pseudocode expresses the exchange of these values as two message-passing steps, a real implementation would undoubtedly combine these values into a single structure that could be passed all at once.

A different distribution of data structures results in a different multicomputer algorithm. For example, if we assign to each processor an *interleaved* group

```
MAX.TOURNAMENT (id, value, winner)
  Reference    id, value, winner

  {This procedure is called from inside a for all statement}
begin
  for i ← 0 to log p − 1 do
    partner ← id ⊗ 2^i
    [partner]tmp.value ⇐ value
    [partner]tmp.winner ⇐ winner
    if tmp.value > value then
      value ← tmp.value
      winner ← tmp.winner
    endif
  endfor
end
```

FIGURE 9-18 Pseudocode implementing the max-tournament procedure, which computes the identity of the processor with the largest value.

of columns of a and a copy of b, then the processors do not need to interact to determine the pivot row. Instead, during iteration i the processor controlling column i examines the elements corresponding to unmarked rows and finds the element with the largest magnitude. No tournament is necessary. This processor then broadcasts elements of column i and the identity of the pivot row to the other processors.

Recall that the grain size of a parallel computation is the amount of work performed per processor interaction. The strategy of maximizing grain size, which we have already discussed in the context of multiprocessors, is also important when designing algorithms for multicomputers. (see Design Strategy 5 in Chap. 7.)

On a multicomputer in which message latency is relatively high, maximizing grain size means minimizing the number of messages sent. In particular, if the messages are small, it makes sense to combine messages heading for the same destination processor in order to reduce message passing overhead.

For example, in the row-oriented version of parallel gaussian elimination, the processor controlling the pivot row must make elements $i + 1$ through n of that row available to all the processors executing the for loop. Rather than broadcast these values one at a time, it makes more sense to broadcast the entire set of $n - i$ row elements.

Likewise, the column-oriented, parallel gaussian-elimination algorithm should be implemented so that during iteration i the processor controlling column i combines the elements of column i and the variable containing the identity of the pivot row into a single message.

Comparing the row-oriented versus the column-oriented implementations of gaussian elimination, we see that there is a communication-computation trade-

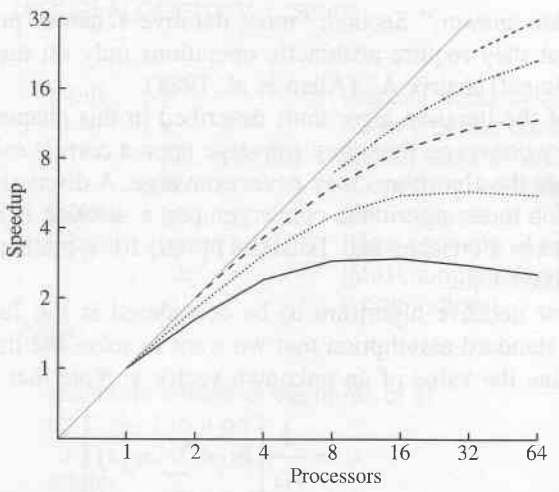


FIGURE 9-19 Scaled speedup achieved by a row-oriented parallel implementation of gaussian elimination on a 64–processor nCUBE 3200, for linear systems of various sizes.

off. In the row-oriented algorithm, processors work together to determine the pivot row. Given a system of size n and p processors, no processor need examine more than n/p values. However, once a processor has determined its local maximum, it must communicate with other processors in order to determine the global maximum. In iteration i of the column-oriented algorithm one processor must perform $n - i$ comparisons, but no communication is required. Both algorithms require a broadcast step after the pivot row has been found. In the case of the row-oriented algorithm a single row is broadcast. In the case of the column-oriented algorithm a single column is broadcast.

For any fixed p, as $n \to \infty$, the time required by the row-oriented algorithm to find the pivot row must be less than that required by the column-oriented algorithm. For this reason, we have chosen to implement the row-oriented algorithm. Figure 9-19 illustrates the scaled speedup achieved by a row-oriented parallel implementation of gaussian elimination on a 64–processor nCUBE 3200, for linear systems of various sizes.

9.5 THE JACOBI ALGORITHM

In the remaining sections of this chapter we will examine iterative methods for solving systems of linear equations. Iterative algorithms are frequently used to solve the large, sparse linear systems generated when working with partial differential equations, using discrete methods. Iterative methods have two advantages over direct methods. First, "while these methods do not formally yield [a solution] in a finite number of steps, one can terminate such a procedure after a finite number of iterations when it has produced a sufficiently good

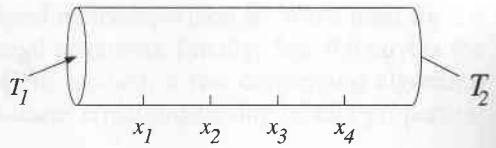


FIGURE 9-2 An insulated rod with fixed end temperatures T_1 and T_2. We want to know the temperatures at points x_1, x_2, x_3, x_4, which divide the bar into five segments of equal length.

rod of conducting material (Fig. 9-2). We assume the rod has a uniform cross section, so that all points in a cross section have equal temperature. This enables us to describe temperature as a function of distance from one end of the rod. We assume the temperatures T_1 and T_2 at the ends of the rod are fixed through exposure to a constant heat source. Finally, we assume that the rod is swathed in insulating material. In other words, no heat escapes from the sides of the rod—all heat transfer is at the ends.

Finding the steady-state temperature at four evenly spaced points x_1, x_2, x_3, and x_4 can be expressed as a system of linear equations:

$$\begin{aligned} x_1 - 0.5x_2 &= 0.5T_1 \\ -0.5x_1 + x_2 - 0.5x_3 &= 0 \\ -0.5x_2 + x_3 - 0.5x_4 &= 0 \\ -0.5x_3 + x_4 &= 0.5T_2 \end{aligned} \tag{9.4}$$

Here are three more definitions that we will use later in this chapter.

Definition 9.6. An $n \times n$ matrix $\mathbf{A}$ is **diagonally dominant** if $|a_{i,i}| > \Sigma_{j \neq i} |a_{i,j}|$ for $1 \leq i \leq n$.

Definition 9.7. An $n \times n$ matrix $\mathbf{A}$ is **symmetric** if $a_{ij} = a_{ji}$ for $1 \leq i, j \leq n$.

Definition 9.8. An $n \times n$ matrix $\mathbf{A}$ is **positive definite** if it is symmetric, diagonally dominant, and $a_{ii} > 0$ for $1 \leq i \leq n$.

9.2 BACK SUBSTITUTION

In this section we describe the parallelization of an algorithm used to solve the linear system $\mathbf{Ax} = \mathbf{b}$ where $\mathbf{A}$ is upper triangular.

Given a system of linear equations $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is an upper triangular $n \times n$ matrix, the back substitution algorithm solves the linear system in time $\Theta(n^2)$. Let's view the algorithm by using a simple example. Suppose we want to solve the system

$$\begin{aligned} 1x_1 + 1x_2 - 1x_3 + 4x_4 &= 8 \\ -2x_2 - 3x_3 + 1x_4 &= 5 \\ 2x_3 - 3x_4 &= 0 \\ 2x_4 &= 4 \end{aligned} \tag{9.5}$$

We can solve the last equation directly, since it has only a single unknown. After we have determined that $x_4 = 2$, we can simplify the other equations by

removing their x_4 terms and adjusting the value of their **b** terms:

$$\begin{aligned} 1x_1 + 1x_2 - 1x_3 &= 0 \\ -2x_2 - 3x_3 &= 3 \\ 2x_3 &= 6 \\ 2x_4 &= 4 \end{aligned} \tag{9.6}$$

Now the third equation has only a single unknown, and a simple division yields $x_3 = 3$. Again, we use this information to simplify the two equations above it:

$$\begin{aligned} 1x_1 + 1x_2 &= 3 \\ -2x_2 &= 12 \\ 2x_3 &= 6 \\ 2x_4 &= 4 \end{aligned} \tag{9.7}$$

We have simplified the second equation to contain only a single unknown, and dividing b_2 by a_{22} yields $x_2 = -6$. After subtracting $x_2 \times a_{12}$ from b_1 we have

$$\begin{aligned} 1x_1 &= 9 \\ -2x_2 &= 12 \\ 2x_3 &= 6 \\ 2x_4 &= 4 \end{aligned} \tag{9.8}$$

and it is easy to see that $x_1 = 9$.

A sequential algorithm to perform back substitution appears in Fig. 9-3. The time complexity of this algorithm is $\Theta(n^2)$.

How amenable to parallelization is the back substitution algorithm? It is often difficult to determine the inherent parallelism of an algorithm from a simple examination of the code. Sometimes construction of a task graph can make the parallelism (or lack of parallelism) apparent. The task graph has these properties:

```
BACK.SUBSTITUTION (SISD):

Global  n              {Size of system}
        a[1...n][1...n] {Elements of A}
        b[1...n]       {Elements of b}
        x[1...n]       {Elements of x}
        i              {Column index}
        j              {Row index}
1. begin
2.    for i ← n downto 1 do
3.       x[i] ← b[i] / a[i][i]
4.       for j ← 1 to i – 1 do
5.          b[j] ← b[j] – x[i] × a[j][i]
6.          a[j][i] ← 0      {This line is optional}
7.       endfor
8.    endfor
9. end
```

FIGURE 9-3 Sequential back substitution algorithm. Given an upper triangular system of size n, the algorithm has time complexity $\Theta(n^2)$.

1. It contains one vertex for every time a variable is assigned a value.
2. It contains one vertex for every variable that is accessed but never assigned a value.
3. It contains one edge for every use-def dependency between a variable referenced on the right-hand side of an assignment statement and a variable assigned a value on the left-hand side. The edge is directed toward the variable assigned a value.

We have used these rules to construct a task graph for the back substitution algorithm applied to an upper triangular system of size 4 (see Fig. 9-4). Note that there are multiple vertices corresponding to each variable $b[i]$, since the algorithm repeatedly updates these variables.

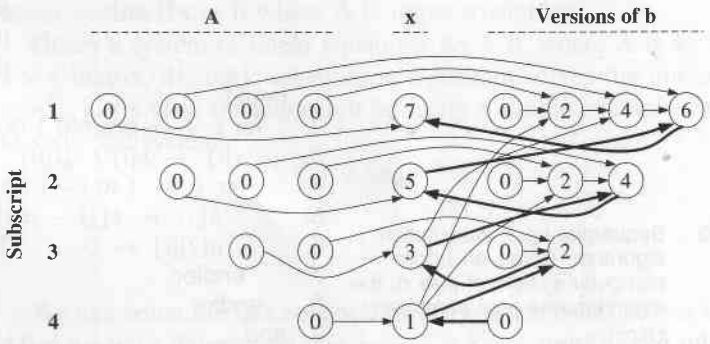
Once the task graph has been constructed, we label each vertex according to the following rules:

1. If no edges enter a vertex, the vertex has label 0.
2. If the vertex has at least one incoming edge, its label is equal to 1 plus the maximum label of any vertex associated with an incoming edge.

The labels inside the vertices in Fig. 9-4 have been assigned using these rules. The labels represent the depth of each part of the computation in the task graph. We have used bold arcs to indicate one of the critical paths of the task graph. It is evident from the critical path that the elements of x must be computed one at a time.

With this knowledge, it is clear our only alternative is to parallelize the inner for loop. The parallel algorithm, designed for a UMA multiprocessor, appears in Fig. 9-5. The grain size is small; even in the first iteration there are only $n - 1$ multiplications and $n - 1$ subtractions. As the algorithm progresses, the

FIGURE 9-4 Task graph for the sequential back substitution algorithm solving an upper triangular system of size 4. Elements of vector b are updated, so the graph shows one vertex for each value of each element. The label inside each vertex indicates the depth of the task in the graph. A critical path is highlighted.



BACK.SUBSTITUTION (UMA MULTIPROCESSOR):

```
Global   n              {Size of system}
         p              {Number of processes}
         a[1...n][1...n] {Elements of A}
         b[1...n]       {Elements of b}
         x[1...n]       {Elements of x}
         i              {Column index}
Local    j              {Process identifier}
         k              {Row index}

begin
  for i ← n downto 2 do
    x[i] ← b[i] / a[i][i]
    forall P_j where 1 ≤ j ≤ p do
      for k ← j to i − 1 step p do
        b[k] ← b[k] − x[i] × a[k][i]
        a[k][i] ← 0      {This line is optional}
      endfor
    endforall
  endfor
end
```

FIGURE 9-5 Parallel version of back substitution algorithm suitable for implementation on a UMA multiprocessor.

number decreases linearly to 1 multiplication and 1 subtraction. For this reason we cannot expect this algorithm to achieve high speedup.

Figure 9-6 illustrates the speedup achieved by our parallel back substitution algorithm on a lightly loaded Sequent Symmetry UMA multiprocessor. Note the Amdahl effect: speedup on any fixed number of processors increases with the problem size.

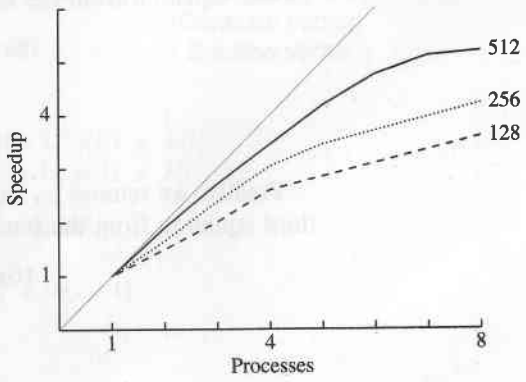
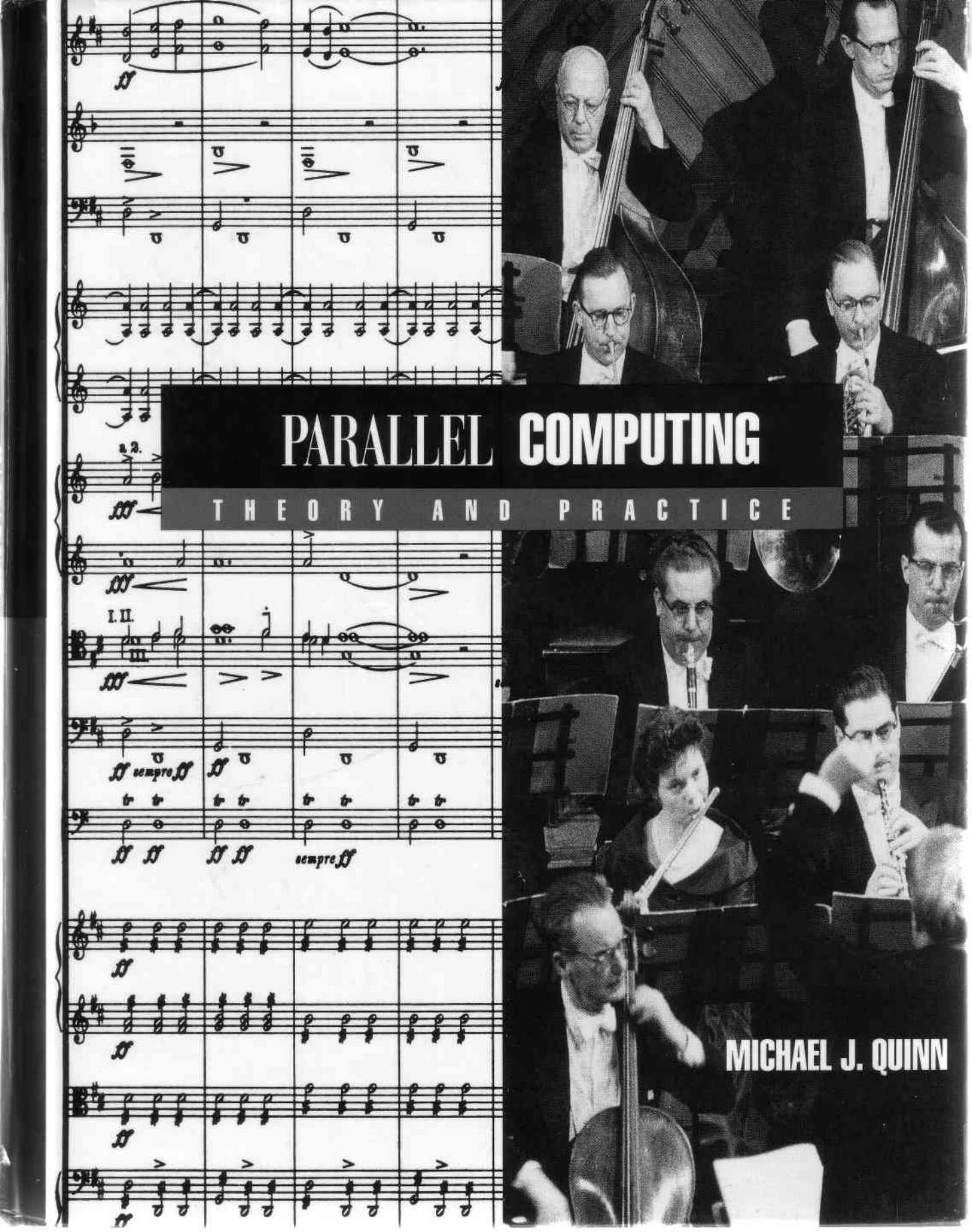


FIGURE 9-6 Speedup achieved on Sequent Symmetry by the parallel back substitution algorithm solving triangular systems of various sizes.



PARALLEL COMPUTING

THEORY AND PRACTICE

MICHAEL J. QUINN