CHAPTER 11

Solving Sparse Systems of Linear Equations

Solving systems of linear equations is at the core of many problems in engineering and scientific computing. In Chapter 5 we addressed the problem of solving dense systems of linear equations—that is, solving a system of linear equations in which most coefficients are not zero. In this chapter we focus our attention on solving large sparse systems of equations in which a majority of the coefficients are zero. It is important to study sparse systems not only because we encounter them frequently in scientific computing problems, but also because they involve more complex algorithms and data structures than their dense counterparts.

Most scientific computing problems represent a physical system by a mathematical model. To make it suitable for computer solution, the continuous physical domain of the system being modeled is discretized by imposing a grid or a mesh over the domain. Either the grid points or the partitions of the domain dictated by the grid are then regarded as discrete elements. Solving the mathematical model over this discretized domain involves obtaining the values of certain physical quantities at every grid point. For example, Figure 11.1 shows a grid imposed over a sheet of metal insulated on two opposite sides and exposed to temperatures U_0 and U_1 on the other two sides. The steady-state temperature of the entire surface of the sheet is modeled by computing the temperature at each grid point. The same basic approach is used in modeling much more complex systems, such as weather patterns in the atmosphere, ocean currents, and stress on mechanical parts, just to name a few.

Each grid point of a discretized physical domain is simulated based on the influence of the neighboring elements and the surroundings of the domain. For example, in Figure 11.1, the temperature at point 24 is influenced by the temperature at points 16, 17, 18, 23, 32, 31, 30, and 25, and the temperature at point 0 is influenced by the value of U_0 as well as

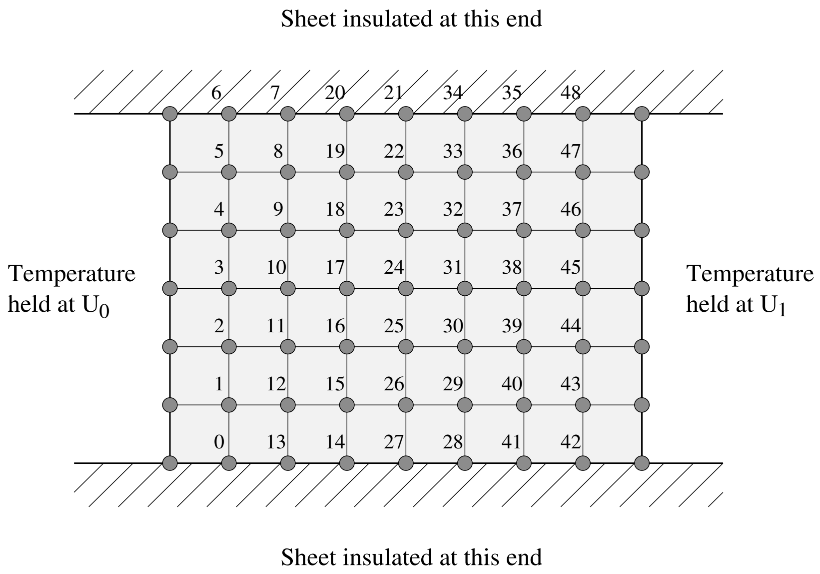


Figure 11.1 Example of a grid imposed over a physical domain consisting of a metal sheet. The temperature of the surface is modeled by computing its value at points 0 through 48.

the temperature at points 1, 12, and 13. Typically, the simulation of a single grid point yields a linear equation that relates the value of a desired physical quantity at the grid point to the values at its neighbors. Since there many grid points in the discretized domain, the task of solving the mathematical model is equivalent to that of solving the set of linear equations associated with all these points. The value of the physical quantity being modeled is represented by a variable at each grid point. The value of a variable in the system of equations depends on only a few other variables—those that correspond to neighboring grid points. As a result, only the coefficients of these variables are nonzero in a typical equation. Most of the coefficients in the system of equations are zero; hence the system is sparse.

As discussed in Section 5.5, a system of n linear equations can be represented in matrix form by $Ax = b$, where A is the $n \times n$ matrix of coefficients, b is an $n \times 1$ vector, and x is the $n \times 1$ solution vector. However, as discussed in Sections 11.3 and 11.5, solving the mathematical model does not always require that the coefficients be explicitly assembled in matrix form. In this chapter, we deal with systems for which, if explicitly assembled, the coefficient matrix A is a sparse matrix; that is, a majority of its elements are zero. More precisely, the matrix A is considered sparse if a computation involving it can utilize the number and location of its nonzero elements to reduce the run time over the same computation on a dense matrix of the same size.

$$\begin{pmatrix} 1 & 0 & 0 & 2 & 0 & 3 \\ 4 & 5 & 0 & 0 & 0 & 0 \\ 0 & 6 & 7 & 0 & 0 & 8 \\ 9 & 0 & 0 & 10 & 11 & 12 \\ 0 & 13 & 0 & 0 & 14 & 0 \\ 0 & 0 & 0 & 0 & 0 & 15 \end{pmatrix}$$

(a) A sparse matrix

VAL

8	6	12	1	15	14	9	2	3	5	13	4	11	7	10

I

2	2	3	0	5	4	3	0	0	1	4	1	3	2	3

J

5	1	5	0	5	4	0	3	5	1	1	0	4	2	3

(b) Storage in coordinate format

Figure 11.2 A 6×6 sparse matrix and its representation in the coordinate storage format.

11.1 Basic Operations

Since this chapter deals primarily with sparse matrices, we first introduce efficient storage schemes for sparse matrices and some simple linear algebra operations using them.

11.1.1 Storage Schemes for Sparse Matrices

It is customary to store an $n \times n$ dense matrix in an $n \times n$ array. However, if the matrix is sparse, storage is wasted because a majority of the elements of the matrix are zero and need not be stored explicitly. For sparse matrices, it is a common practice to store only the nonzero entries and to keep track of their locations in the matrix. A variety of storage schemes are used to store and manipulate sparse matrices. These specialized schemes not only save storage but also yield computational savings. Since the locations of the nonzero elements (and hence, the zero elements) in the matrix are known explicitly, unnecessary multiplications and additions with zero can be avoided. There is no single best data structure for storing sparse matrices. Different data structures are suitable for different operations. Also, some data structures are more suitable for a parallel implementation than others. In the following subsections we briefly describe some common sparse-matrix storage schemes.

Coordinate Format

Given a sparse matrix with q nonzero entries, the ***coordinate format*** stores these entries in a $q \times 1$ array *VAL* in any order. Two additional $q \times 1$ arrays I and J store the i and j coordinates (row and column numbers) of the entries. A 6×6 square matrix and the corresponding coordinate storage format are shown in Figure 11.2. In this figure, as in the remainder of the chapter, we number the rows and columns starting from 0.

Compressed Sparse Row Format

The ***compressed sparse row*** (CSR) format uses the following three arrays to store an $n \times n$ sparse matrix with q nonzero entries:

(1) A $q \times 1$ array *VAL* contains the nonzero elements. These are stored in the order of their rows from 0 to $n - 1$; however, elements of the same row can be stored in any order.
(2) A $q \times 1$ array J that stores the column numbers of each nonzero element.
(3) An $n \times 1$ array I, the i^{th} entry of which points to the first entry of the i^{th} row in *VAL* and J.

Figure 11.3 shows the sparse matrix of Figure 11.2(a) in CSR format. A related scheme is the ***compressed sparse column*** format (CSC), in which the roles of rows and columns are reversed. Another variation of CSR is the ***modified sparse row*** (MSR) format, in which the principal diagonal (which is often fully nonzero) is stored separately and the remaining elements are stored in the regular CSR format.

Diagonal Storage Format

The ***diagonal storage format*** is suited to sparse matrices whose nonzero entries are arranged in a few diagonals. Consider an $n \times n$ matrix consisting of d diagonals with nonzero elements (all other entries are zero). These nonzero diagonals are stored in an $n \times d$ array *VAL*. A $d \times 1$ array *OFFSET* stores the offset of each diagonal with respect to the principal diagonal. The order in which the diagonals are stored is not important. Figure 11.4 shows a sparse matrix stored in this fashion. Since all diagonals other than the principal diagonal have fewer than n elements, there will be unused locations in the array *VAL*. Any zeros within the d diagonals are stored explicitly.

Sometimes all the nonzero diagonals of a sparse matrix form a band around the principal diagonal. In this case, a variation of the diagonal format called ***banded format*** can be used. An $n \times n$ matrix with a band of u diagonals above the principal diagonal and

VAL	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
J	0	3	5	0	1	1	2	5	0	3	4	5	1	4	5

I	0	3	5	8	12	14

Figure 11.3 CSR storage of the 6×6 sparse matrix of Figure 11.2(a).

$$\begin{pmatrix} 1 & 2 & 0 & 0 & 0 & 0 \\ 3 & 4 & 5 & 0 & 0 & 0 \\ 0 & 6 & 7 & 0 & 0 & 0 \\ 8 & 0 & 9 & 10 & 11 & 0 \\ 0 & 13 & 0 & 0 & 14 & 15 \\ 0 & 0 & 16 & 0 & 17 & 18 \end{pmatrix}$$

(a) A sparse matrix

VAL

-	-	1	2
-	3	4	5
-	6	7	0
8	9	10	11
13	0	14	15
16	17	18	-

OFFSET

-3	-1	0	1

(b) Storage in diagonal format

Figure 11.4 A sparse matrix stored in the diagonal format.

l diagonals below it is stored in an $n \times (u + l + 1)$ array. Instead of the *OFFSET* array, banded format uses two parameters to indicate the thickness of the band and its lower or upper limit.

Ellpack-Itpack Format

The ***Ellpack-Itpack format*** is suitable for general sparse matrices in which the maximum number of nonzero elements in any row is not much larger than the average number of nonzero elements per row. In this scheme, an $n \times n$ sparse matrix in which the maximum number of nonzero elements in any row is m, is stored using two $n \times m$ arrays *VAL* and J. Each row of *VAL* contains the nonzero entries of the corresponding row of the sparse matrix, and the array J stores the column numbers of the corresponding entries in *VAL*. Figure 11.5 shows a sparse matrix stored in the Ellpack-Itpack format. All rows of *VAL* and J that have fewer than m nonzero elements in the original matrix have empty spaces. These empty spaces store some sentinel value (-1 in Figure 11.5(b)) that denotes the end of a row.

$$\begin{pmatrix} 1 & 2 & 0 & 0 & 0 & 0 \\ 3 & 4 & 5 & 0 & 0 & 0 \\ 0 & 6 & 7 & 0 & 0 & 0 \\ 8 & 0 & 9 & 10 & 11 & 0 \\ 0 & 13 & 0 & 0 & 14 & 15 \\ 0 & 0 & 16 & 0 & 17 & 18 \end{pmatrix}$$

(a) A sparse matrix

VAL

1	2	-	-
3	4	5	-
6	7	-	-
8	9	10	11
13	14	15	-
16	17	18	-

J

0	1	-1	-
0	1	2	-1
1	2	-1	-
0	2	3	4
1	4	5	-1
2	4	5	-1

(b) Storage in Ellpack-Itpack format

Figure 11.5 A sparse matrix stored in Ellpack-Itpack format.

$$\begin{pmatrix} 1 & 0 & 0 & 2 & 0 & 3 \\ 4 & 5 & 0 & 0 & 0 & 0 \\ 0 & 6 & 7 & 0 & 0 & 8 \\ 9 & 0 & 0 & 10 & 11 & 12 \\ 0 & 13 & 0 & 0 & 14 & 0 \\ 0 & 0 & 0 & 0 & 0 & 15 \end{pmatrix}$$

(a) A sparse matrix

$$\begin{pmatrix} 9 & 0 & 0 & 10 & 11 & 12 \\ 1 & 0 & 0 & 2 & 0 & 3 \\ 0 & 6 & 7 & 0 & 0 & 8 \\ 4 & 5 & 0 & 0 & 0 & 0 \\ 0 & 13 & 0 & 0 & 14 & 0 \\ 0 & 0 & 0 & 0 & 0 & 15 \end{pmatrix}$$

(b) The matrix with reordered rows

VAL	9	1	6	4	13	15	10	2	7	5	14	11	3	8	12
J	0	0	1	0	1	5	3	3	2	1	4	4	5	5	5

I	0	6	11	14

(c) Storage in jagged-diagonal format

Figure 11.6 The jagged-diagonal storage scheme.

Jagged-Diagonal Format

To store a sparse matrix in the ***jagged-diagonal format***, the rows of the matrix are ordered in the decreasing number of nonzero entries. The first nonzero entry of each row is stored in contiguous locations of a $q \times 1$ array *VAL*, where q is the total number of nonzero elements in the sparse matrix. These entries constitute the first jagged diagonal. Then the second nonzero entry of each row is stored in *VAL* (that is, the second jagged diagonal is assembled), and so on. Another $q \times 1$ array J stores the column numbers of the corresponding entries in *VAL*. A third array I of size $m \times 1$ contains pointers to the beginning of each jagged diagonal; m is the maximum number of nonzero entries in any row, which gives the total number of jagged diagonals. Figure 11.6 illustrates the jagged-diagonal storage scheme.

11.1.2 Vector Inner Product

Although not a sparse matrix operation, the inner product of two dense vectors is commonly used in iterative methods for solving systems of linear equations (Section 11.2). The inner product often determines the overall communication complexity and scalability of the

```
procedure INNER_PRODUCT (x, y, a, n)
begin
    a := 0;
    for i := 0 to n − 1 do
        a := a + x[i] × y[i];
end INNER_PRODUCT
```

Program 11.1 An algorithm for computing the inner product of two dense $n \times 1$ vectors x and y.

entire algorithm of which it is a part. As Program 11.1 shows, the inner product is a simple operation in which the corresponding elements of two vectors are multiplied and the resulting products are added together.

If the two $n \times 1$ vectors to be multiplied are uniformly partitioned among p processors, each processor performs n/p multiplications and $(n/p) - 1$ additions. The sums of the n/p products at each processor must be accumulated to obtain the inner product. Assume that the underlying architecture is a hypercube and that it takes time $t_s + t_w \approx t_s$ (assuming t_w to be small compared to t_s) to communicate one word of data between two directly-connected (by bidirectional links) processors. Whether the final inner product must be distributed to all processors (Example 3.7) or is required at only one processor (Example 4.1), the total communication time on a p-processor hypercube is approximately $t_s \log p$. If the underlying architecture is a square mesh with cut-through routing, the communication time is approximately $t_s \log p + 2t_h\sqrt{p}$ (Section 3.2.2).

Recall from Section 3.7.3 that, in addition to the standard data network, some parallel computers have a fast control network that can perform certain global operations in a small, almost constant, time. One such operation is reduction, which starts with a different value on every processor and ends with a single value in each processor that is the result of applying an associative operator (such as logical OR, logical AND, addition, maximum, or minimum) on all the initial values. Section 3.7.3 shows how this operation can be used to accumulate the partial sums and to distribute the value of the inner product to all the processors in the ensemble. As shown in Section 11.2, the presence of a fast reduction operation has a significant effect on the efficiency of iterative algorithms for solving sparse systems of equations.

11.1.3 Sparse Matrix-Vector Multiplication

The multiplication of a sparse matrix with a dense vector is one of the key operations in solving systems of linear equations using iterative methods (Section 11.2). It is, therefore, important to perform this operation efficiently in parallel. The sparse matrices resulting from linear systems of equations often have their nonzero elements distributed according to some pattern. Whenever possible, the parallel implementation of sparse matrix-vector multiplication is tuned according this pattern to attain maximum efficiency. In this sec-

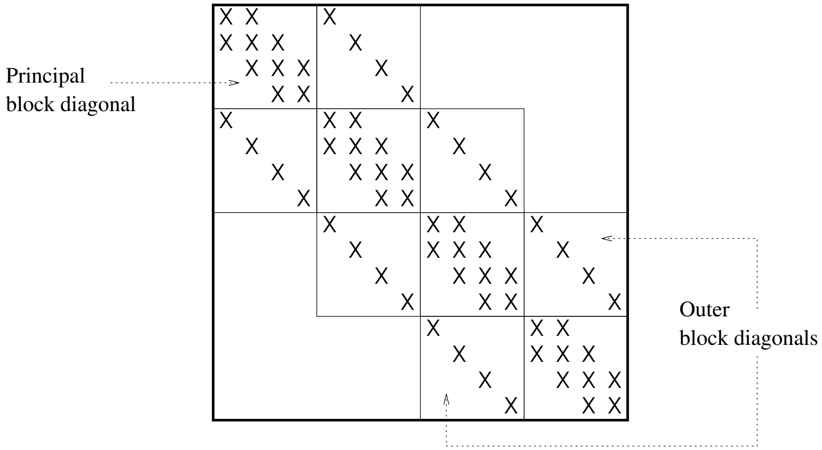


Figure 11.7 A 16×16 block-tridiagonal matrix. The nonzero elements are represented by the symbol $\times$. Zeros are not shown.

tion, we discuss matrix-vector multiplication for three types of sparse matrices that occur commonly in the context of linear systems of equations: (1) matrices in which all nonzero elements are arranged in a few diagonals parallel to and including the principal diagonal; (2) unstructured sparse matrices, in which the location of nonzero elements does not conform to any well-defined structure; and (3) banded sparse matrices, in which the nonzero elements are confined within a band around the principal diagonal; however, inside the band the nonzero elements are distributed in an unstructured manner.

Block-Tridiagonal Matrices

This subsection discusses multiplication of a vector by a sparse matrix that has all its nonzero elements distributed along five diagonals. Furthermore, the diagonals have very specific locations, as illustrated in Figure 11.7 for a 16×16 matrix. One of the five diagonals of the $n \times n$ matrix is the principal diagonal. There are two diagonals immediately adjacent to the principal diagonal on each side. Finally, there are two diagonals at a distance of $\sqrt{n}$ from the principal diagonal on each side. Systems of linear equations with a coefficient matrix of the type shown in Figure 11.7 occasionally arise in scientific computing. Such systems are also pedagogically popular, as they facilitate the exposition of certain key concepts without too many intricacies. Before we discuss matrix-vector multiplication involving this matrix, we will briefly describe how such a matrix originates.

As mentioned earlier, sparse systems of equation often arise from models of physical systems. The ***finite difference method*** is one of the techniques used to obtain an approximate solution to a partial differential equation governing the behavior of a physical system. The finite difference method imposes a regular grid on the physical domain. It

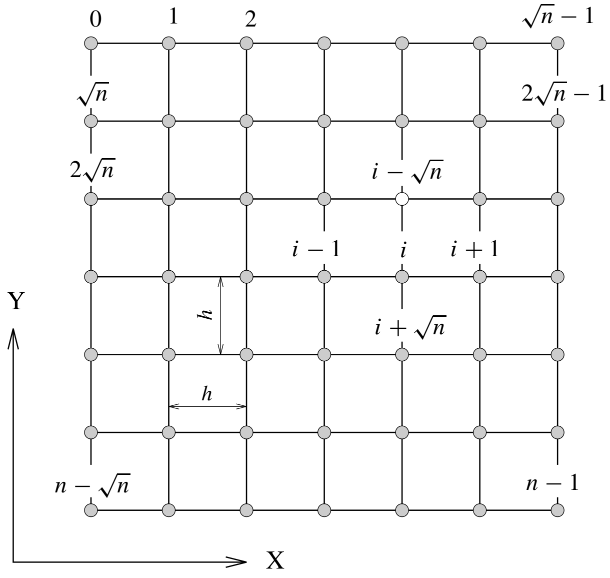


Figure 11.8 A $\sqrt{n} \times \sqrt{n}$ grid with natural ordering of grid points.

then approximates the derivative of an unknown quantity u at a grid point by the ratio of the difference in u at two adjacent grid points to the distance between the grid points. For example, consider a square domain discretized by $\sqrt{n} \times \sqrt{n}$ grid points, as shown in Figure 11.8. Assume that the grid points are numbered in a row-major fashion from left to right and from top to bottom, as shown in the figure. This ordering is called ***natural ordering***. Given a total of n points in the $\sqrt{n} \times \sqrt{n}$ grid, this numbering scheme labels the immediate neighbors of point i on the top, left, right, and bottom points as $i - \sqrt{n}$, $i - 1$, $i + 1$ and $i + \sqrt{n}$, respectively.

Assume that the partial differential equation governing the value of u over the domain is

$$\frac{\delta^2 u}{\delta X^2} + \frac{\delta^2 u}{\delta Y^2} = f. \qquad (11.1)$$

Further assume that the values of u at the n grid points are stored in an $n \times 1$ vector x and that $x[i]$ is the value of u at point i. Let h be the distance between any two neighboring grid points. The finite difference approximation of Equation 11.1 yields the following:

$$\frac{1}{h}\left(\frac{x[i+1]-x[i]}{h} - \frac{x[i]-x[i-1]}{h}\right) + \frac{1}{h}\left(\frac{x[i+\sqrt{n}]-x[i]}{h} - \frac{x[i]-x[i-\sqrt{n}]}{h}\right) = f$$

$$x[i-\sqrt{n}] + x[i-1] - 4x[i] + x[i+1] + x[i+\sqrt{n}] = h^2 f$$

In general, the equation relating the values of the physical quantity at point i to its value at i's neighbors is of the form

$$a_i x[i-\sqrt{n}] + b_i x[i-1] + c_i x[i] + d_i x[i+1] + e_i x[i+\sqrt{n}] = f_i, \qquad (11.2)$$

where a_i, b_i, c_i, d_i, e_i, and f_i are constants. Each point on the grid yields one such equation, and hence, one row in the matrix of coefficients. If the equations are ordered from 0 to $n-1$, and variables are ordered from $x[0]$ to $x[n-1]$ in each equation, the resulting coefficient matrix resembles the one shown in Figure 11.7. In the i^{th} row of the matrix, the four nonzero entries other than the principal diagonal correspond to the four nearest neighbors of the i^{th} point in the grid shown in Figure 11.8. The rows corresponding to the boundary points have fewer nonzero elements because these points have fewer than four neighbors.

The matrix shown in Figure 11.7 is a special case of a ***block-tridiagonal matrix***. A block-tridiagonal matrix consists of three consecutive diagonals composed of matrix blocks along the principal diagonal. We refer to these diagonals of matrix blocks as ***block diagonals***. The block diagonals of the block-tridiagonal matrix we are considering here are composed of blocks of size $\sqrt{n} \times \sqrt{n}$. The blocks of the principal block diagonal are tridiagonal matrices with three consecutive diagonals in the center. The two outer block diagonals are composed of blocks that are simple diagonal matrices with only a nonzero principal diagonal. The principal block diagonal contains $\sqrt{n}$ blocks and each of the outer block diagonals consists of $\sqrt{n} - 1$ blocks.

We will use the type of matrix shown in Figure 11.7 as the model block-tridiagonal matrix in the remainder of this chapter. However, all algorithms using a matrix with this structure are valid for a somewhat more general block-tridiagonal structure. In general, the size of the matrix is $l_1 l_2 \times l_1 l_2$ for some integers l_1 and l_2 (Problem 11.7). It consists of a principal block diagonal composed of $l_2 \times l_2$ tridiagonal matrices, and two adjacent block diagonals on either side composed of $l_2 \times l_2$ diagonal matrices. The principal block diagonal consists of l_1 blocks and the outer block diagonals contain $l_1 - 1$ blocks each. Such a matrix results if the underlying finite difference grid is an $l_1 \times l_2$ rectangle.

Parallel Implementation with Striped Partitioning of the Block-Tridiagonal Matrix Consider the multiplication of an $n \times n$ matrix of the type shown in Figure 11.7 with an $n \times 1$ vector using p processors. Figure 11.9 shows that the matrix and the vector are partitioned among p processors so that every processor gets n/p elements of the vector and each diagonal of the matrix. The diagonal storage scheme is the natural choice for this case. The array *VAL* is distributed among the processors by using block-striped partitioning, and the array *OFFSET* is not required because the offsets $-\sqrt{n}$, -1, 0, 1 and $\sqrt{n}$ are implicit.

Each row of the matrix requires five vector elements for multiplication. The vector element with which the principal diagonal entry of a row is multiplied has the same index as the number of the row, and is available at the same processor as the row. The vector

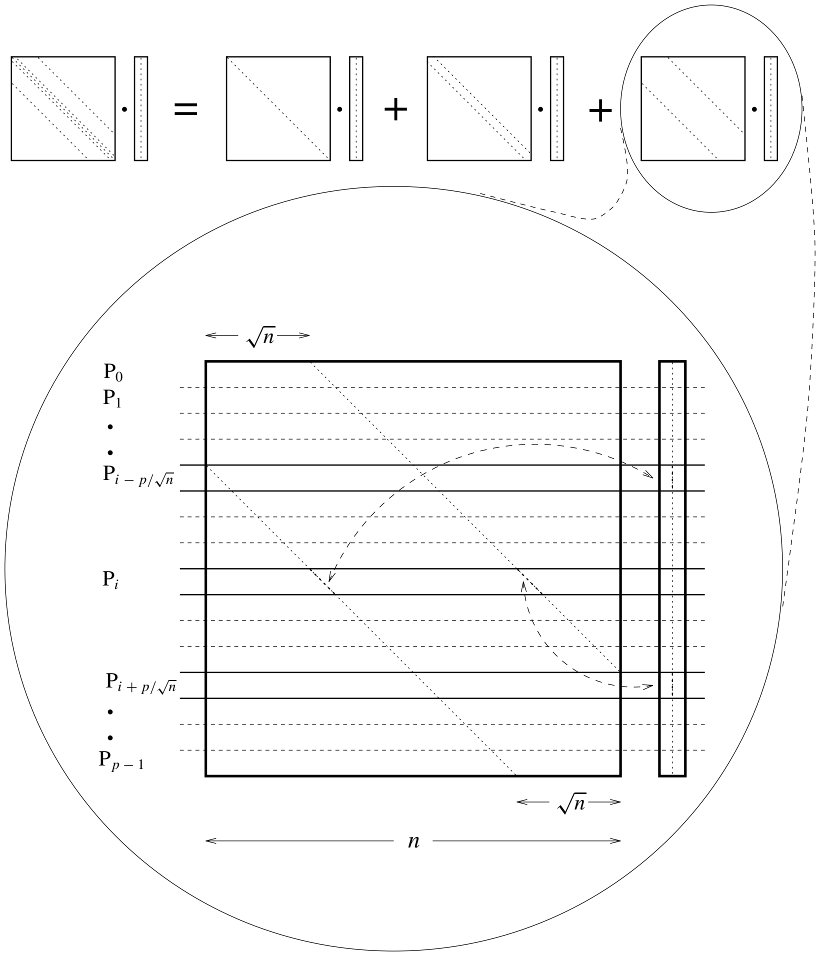


Figure 11.9 Data communication in matrix-vector multiplication with block-striped partitioning of a block-tridiagonal matrix.

elements with which the two inner diagonal entries are multiplied are also available on the same processor, except for the rows that lie on processor boundaries (for example, row number $(n/p) - 1$ is the last row on processor P_0 and needs vector elements $(n/p) - 2$ and n/p; the latter resides on processor P_1). Each processor exchanges its boundary elements with its neighboring processors, and thus, all vector elements required for multiplication with the inner diagonals are now available at each processor. This communication takes $2(t_s + t_w)$ time at each processor.

The entries in each row belonging to the outer diagonals must be multiplied with vector elements whose indices are greater or smaller than the index of the row by $\sqrt{n}$. The communication for this step depends on the number of vector elements stored in each processor. If the number of elements per processor (n/p) is greater than or equal to $\sqrt{n}$ (that is, $p \leq \sqrt{n}$), then the required communication can be accomplished by each pair of neighboring processors exchanging $\sqrt{n}$ vector elements at partition boundaries in time $2(t_s + t_w\sqrt{n})$. This exchange subsumes the exchange of the boundary elements for the multiplication of the inner diagonals. Thus, the total communication time is $2(t_s + t_w\sqrt{n})$.

If the number of elements per processor is less than $\sqrt{n}$ (that is, $p > \sqrt{n}$), then processor P_j needs portions of the vector located at processors numbered $j \pm p/\sqrt{n}$ to multiply with the matrix elements belonging to the outer diagonals. As a result, each processor exchanges all its n/p elements with processors located at a distance of $p/\sqrt{n}$ from it on either side (Figure 11.9). This is a shift operation (without circulation) of the vector elements in both directions by a distance of $p/\sqrt{n}$ processors. The shift operation is described in Section 3.6. Assume that the underlying architecture is a hypercube with cut-through routing. As discussed in Section 3.6.2, the communication time for each shift is at most $t_s + t_w n/p + t_h \log p$. Thus, the total communication time (for the exchange of boundary elements and for the shifts) when $p > \sqrt{n}$ is approximately $4t_s + 2t_w n/p + 2t_h \log p$.

Except for the first and last $\sqrt{n}$ rows, there are five nonzero entries in each row of the matrix. Assuming that it takes time t_c to perform one multiplication and addition, the computation time is $5t_c n/p$. The overall parallel execution time for matrix-vector multiplication with the block-tridiagonal matrix and its mapping shown in Figure 11.9 is given by the following equations:

Case 1: $\boldsymbol{p \leq \sqrt{n}}$

$$T_P = \overbrace{5t_c n/p}^{\text{computation}} + \overbrace{2(t_s + t_w\sqrt{n})}^{\text{exchange with neighboring processors}} \tag{11.3}$$

Case 2: $\boldsymbol{p > \sqrt{n}}$

$$T_P = \overbrace{5t_c n/p}^{\text{computation}} + \overbrace{2(t_s + t_w)}^{\text{exchange of boundary elements}} + \overbrace{2t_s + 2t_w n/p + 2t_h \log p}^{\text{shift operations}} \tag{11.4}$$

From Equation 11.4 it follows that the isoefficiency function of this parallel implementation of sparse matrix-vector multiplication is $\Theta(p \log p)$. Although this parallel formulation appears quite scalable, there is an upper limit on the efficiency for $p > \sqrt{n}$. The efficiency expression for this case is

$$E = \frac{5t_c}{5t_c + 4t_s p/n + 4t_w + (2t_h p \log p)/n}. \tag{11.5}$$

Equation 11.5 shows that efficiency cannot exceed $5t_c/(4t_w + 5t_c)$. This upper bound on efficiency depends only on the ratio of computation speed to communication bandwidth.

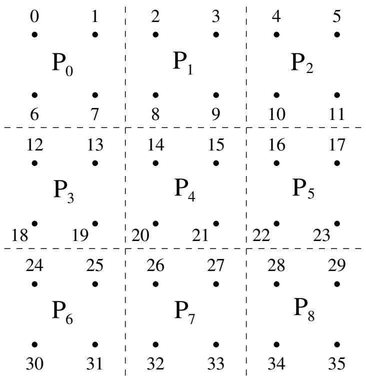


Figure 11.10 Partitioning a 6×6 grid on nine processors.

Therefore, higher efficiency cannot be obtained unless the problem size is increased so that $p \le \sqrt{n}$, in which case the efficiency and isoefficiency function are determined by Equation 11.3. If fewer than $\sqrt{n}$ processors are used, the isoefficiency function due to both concurrency and communication is $\Theta(p^2)$ (Problem 11.12).

If the communication and computation speeds are not balanced, the performance of this parallel formulation of matrix-vector multiplication can be poor. However, we can overcome this upper bound on efficiency by using a better mapping of the matrix onto the processors. We discuss this mapping in the following subsection.

A Faster Parallel Implementation for Matrices Arising from Finite Difference Grids While multiplying the block-tridiagonal matrix of the type shown in Figure 11.7 with a vector, the i^{th} row of the matrix requires an element $x[j]$ of the vector if and only if $A[i, j] \neq 0$. The element $A[i, j]$ is nonzero if and only if points i and j are neighbors in the grid. Thus, the processor storing the i^{th} row of the matrix requires only those elements of the vector whose indices are the same as the indices of the grid points neighboring the i^{th} point.

Now consider the mapping shown in Figure 11.10, which partitions the grid in a block-checkerboard fashion. This partitioning allocates rows of the matrix corresponding to the grid points within a partition to a single processor. The vector is partitioned similarly; the elements with indices corresponding to the grid points in a partition are allocated to a single processor. Using this partitioning, each processor stores $\sqrt{n/p}$ clusters of $\sqrt{n/p}$ matrix rows each (as well as vector elements with the same indices). The starting points of successive clusters are $\sqrt{n}$ rows apart.

To perform matrix-vector multiplication, each processor exchanges the vector elements corresponding to its $\sqrt{n/p}$ boundary points with each of its four neighboring processors. The communication time is $4t_s + 4t_w\sqrt{n/p}$ for both the mesh and hypercube

architectures. The total parallel run time is

$$T_P = 5t_c\frac{n}{p} + 4t_s + 4t_w\sqrt{n/p}. \tag{11.6}$$

The expression for efficiency is

$$E = \frac{5t_c}{5t_c + 4t_s p/n + 4t_w\sqrt{p/n}}. \tag{11.7}$$

A comparison of Equations 11.4 and 11.6 shows that the second data distribution scheme for the block-tridiagonal matrix is strictly superior to the first when $p > \sqrt{n}$. Moreover, in the second scheme, there is no upper bound on efficiency. Thus, efficiency can be increased by increasing the problem size for a given number of processors.

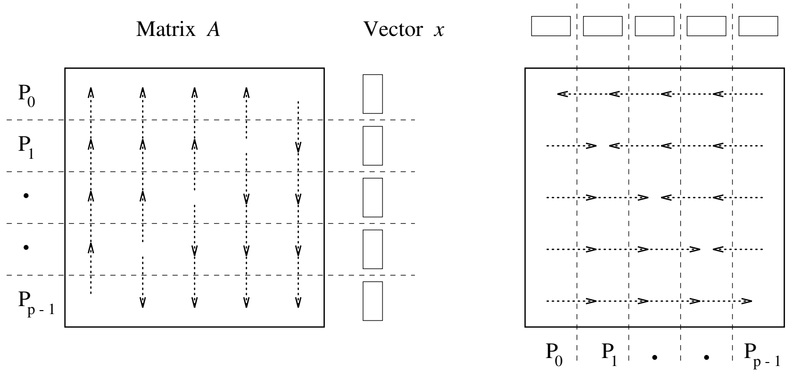
Note that the way the grid points are numbered does not affect the communication overhead in this parallel implementation of matrix-vector multiplication. For a given grid and parallel computer, the communication overhead depends only on the way the grid is partitioned among the processors. If the grid is partitioned as shown in Figure 11.10 and vector elements and matrix rows with identical indices are mapped onto the same processor, then Equation 11.6 holds for any square grid whose points have four neighbors each. Hence, the partitioning illustrated in Figure 11.10 is useful not only for natural ordering and the resulting block-tridiagonal matrix, but also for other ordering schemes such as red-black and multicolored orderings (Section 11.2.2). In general, any $l_1l_2 \times l_1l_2$ matrix arising out of an rectangular $l_1 \times l_2$ finite difference grid can use the partitioning illustrated in Figure 11.10 to minimize communication in matrix-vector multiplication.

Unstructured Sparse Matrices

Consider the multiplication of an $n \times n$ unstructured sparse matrix A with an $n \times 1$ vector x. Assume that the average number of nonzero elements per row in A is m, and hence, the total number of nonzero elements in the entire matrix is mn. Recall that if A is a matrix of coefficients resulting from the model of a physical system, then each row of A contains the coefficients of a linear equation corresponding to one grid point. The number of nonzero coefficients in this equation is equal to the number of neighbors of this grid point. Thus, m is the average number of neighbors that a grid point has. As a result, m is essentially a constant independent of the size of the domain (or the size of the array A) and depends only on the nature of the grid imposed on the domain. The Ellpack-Itpack format is an appropriate storage scheme for A because the number of nonzero elements in different rows of A is not expected to vary over a wide range.

A Simple Parallel Implementation Since the Ellpack-Itpack format is row oriented, we partition the arrays *VAL* and J among p processors such that each processor receives n/p rows, or mn/p nonzero elements of matrix A. The vector x is partitioned uniformly so that each processor initially stores n/p elements.

Recall from Section 5.3 that for dense matrix-vector multiplication, each row of the matrix must be multiplied with the vector. Hence, the vector must be aligned with the rows



(a) All-to-all broadcast with rowwise striping

(b) Multinode accumulation with columnwise striping

Figure 11.11 Data communication in matrix-vector multiplication with block-striped partitioning of an unstructured sparse matrix.

of the matrix in all the processors. Even in the sparse case, if the distribution of nonzero elements is random, then a row can have a nonzero entry in any column. The entire vector must be accessible to each row so that any of its nonzero entries can be multiplied with the corresponding element of the vector. Thus, matrix-vector multiplication requires an all-to-all broadcast among the processors as shown in Figure 11.11(a). The broadcast is followed by the computation phase, in which each processor performs an average of mn/p multiplications and additions. Since each processor is responsible for n/p rows of the matrix, after the computation step, every processor has n/p elements of the result vector, which is distributed among the processors in the same mapping as the starting vector x.

Assuming that the underlying architecture is a hypercube, the all-to-all broadcast of messages containing n/p vector elements among p processors takes $t_s \log p + t_w n$ time. If each multiplication and addition takes time t_c, then the parallel run time is

$$T_P = t_c m \frac{n}{p} + t_s \log p + t_w n. \tag{11.8}$$

Equation 11.8 shows that the communication time, and hence the overall parallel run time, for this implementation of matrix-vector multiplication is $\Theta(n)$. Assuming that m, the average number of nonzero elements per row, is constant, the sequential time complexity of multiplying a sparse $n \times n$ matrix with a vector is also $\Theta(n)$. Thus, this parallel implementation does not lead to any asymptotic reduction in run time. Hence, the parallel implementation is non-cost-optimal and unscalable.

The only way to reduce the parallel run time of this algorithm is to reduce communication time. However, this is not possible if the matrix is partioned into stripes—either along the rows or along the columns. If the vector is distributed among all the processors,

and a storage scheme is used in which a matrix element at a processor can potentially be in any column, then an all-to-all broadcast of the vector elements is unavoidable. This is true for the coordinate format (an entry in the array *VAL* can have any column number), the jagged-diagonal format (for instance, in Figure 11.6(c), the first six elements of *VAL* span columns 0 to 5 while storing a 6×6 array), and any row-based storage scheme such as CSR.

Now consider storing the matrix in compressed sparse column format and partitioning it among the processors such that each processor gets n/p columns. The vector is partitioned uniformly among the processors. As shown in Figure 11.11(b), the vector is already aligned with the rows, and hence, no communication is necessary to perform the multiplication. However, to have the product vector stored in the same format as that of the starting vector, the products of the elements of the i^{th} row with the elements of the vector must be accumulated on the processor that stores the i^{th} column (Problem 5.6). Thus, as shown in Figure 11.11(b), a multinode accumulation operation has to be performed with messages of size n/p. Recall from Chapter 3 that the communication time for this operation is $t_s \log p + t_w n$, which is the same as the communication time for rowwise striping.

A Faster Parallel Formulation for Unstructured Sparse Matrices First consider the parallel formulation independent of the storage scheme. Assume that the $n \times n$ sparse matrix is block-checkerboarded onto a logical $\sqrt{p} \times \sqrt{p}$ mesh of processors embedded in a physical hypercube. Also assume that the vector is partitioned uniformly among the $\sqrt{p}$ processors of the last column. This is the same scenario as in Figure 5.9, except that the matrix is now sparse. Regardless of the type of matrix, communication is the same as in Figure 5.9, and the total communication time on a hypercube with cut-through routing is approximately $t_s \log p + (t_w n \log p)/(\sqrt{p})$ (Problem 5.7).

Assume that nonzero elements are uniformly distributed over the sparse matrix. Checkerboard partitioning divides the matrix into blocks of size $n/\sqrt{p} \times n/\sqrt{p}$. If each row contains an average of m nonzero elements, then the average number of such elements in each block is $m/\sqrt{p} \times n/\sqrt{p}$ (a block has the $(1/p)^{\text{th}}$ portion of n/p rows). Thus, on an average, every processor performs approximately mn/p multiplications and additions. Assuming that it takes time t_c to perform a single addition and multiplication, the average time that a processor spends in computation is $mt_c n/p$. Note that this is only the average computation time per processor, and the actual time varies depending on the number of nonzero elements that fall in the block stored in the processor. For simplicity, we ignore this fact and assume a uniform computation time ($mt_c n/p$) on each processor. The more realistic case, in which the processor containing the maximum number of nonzero elements determines the effective computation time, is discussed in Problem 11.4. Under this uniform workload assumption, the expressions for parallel run time, speedup, and efficiency are as follows:

$$T_P = mt_c \frac{n}{p} + t_s \log p + \frac{3}{2} t_w \frac{n}{\sqrt{p}} \log p \tag{11.9}$$

$$S = \frac{mt_c pn}{mt_c n + t_s p \log p + (3t_w n\sqrt{p}/2) \log p} \tag{11.10}$$

$$E = \frac{mt_c}{mt_c + (t_s p \log p)/n + (3\sqrt{p}\log p)/2} \tag{11.11}$$

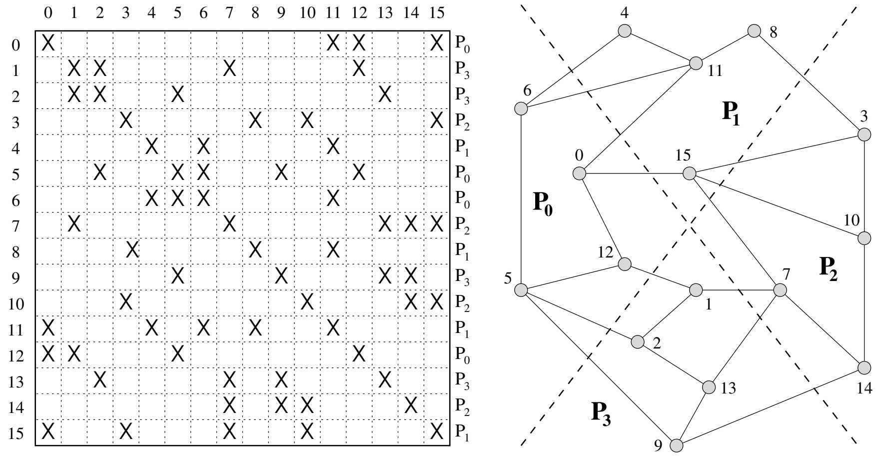
From the preceding equations, we see that even this parallel formulation of matrix-vector multiplication is non-cost-optimal and unscalable, but its parallel run time is $\Theta(\log p) + \Theta((n/\sqrt{p})\log p)$, which is asymptotically smaller than the sequential run time.

A convenient format for storing the sparse matrix for this formulation is to store each block in a separate data structure. For example, if the Ellpack-Itpack format is used, then p separate sets of *VAL* and J arrays need to be maintained—one for each block residing on a separate processor.

A Scalable Parallel Implementation for Unstructured Sparse Matrices We now briefly discuss a scalable formulation of matrix-vector multiplication for a special class of unstructured sparse matrices. Let A be an $n \times n$ unstructured sparse matrix that has a symmetric structure. Let $G(A)$ be a graph with n nodes such that there is an edge between the i^{th} and the j^{th} nodes of $G(A)$ if and only if $A[i, j] \neq 0$ (or $A[j, i] \neq 0$). The matrix A is thus a weighted adjacency matrix of graph $G(A)$ in which each node corresponds to a row of A. A scalable parallel implementation of matrix-vector multiplication exists for a sparse matrix A provided that it is the adjacency matrix of a planar graph $G(A)$. A graph is planar if and only if it can be drawn in a plane such that no edges cross each other. Note that planarity of $G(A)$ is a sufficient, but not a necessary condition for the multiplication of matrix A with a vector to be scalable.

If the graph $G(A)$ is planar, it is possible to partition its nodes (and hence, the rows of A) among processors to yield a scalable parallel formulation for sparse matrix-vector multiplication. The amount of computation that a processors performs is proportional to the total number of nodes in that processor's partition. If $G(A)$ is planar, the total number of words that a processor communicates is proportional to the number of nodes lying along the periphery of that processor's partition. Furthermore, if $G(A)$ is planar, the number of processors with whom a given processor communicates is equal to the number of partitions with whom that processor's partition shares its boundaries. Hence, by reducing the number of partitions (thus, increasing the size of the partitions) it possible to increase the computation to communication ratio of the processors.

Figure 11.12 shows a structurally symmetric randomly sparse matrix and its associated graph. The vector is partitioned among the processors such that its i^{th} element resides on the same processor that stores the i^{th} row of the matrix. Figure 11.12 also shows the partitioning of the graph among processors and the corresponding assignment of the matrix rows to processors. While performing matrix-vector multiplication with this partitioning, the i^{th} row of A requires only those elements of the vector whose indices correspond to the neighbors of the i^{th} node in $G(A)$. The reason is that by the construction of $G(A)$, the i^{th} row has a nonzero element in the j^{th} column if and only if j is connected to i by an edge in $G(A)$. As a result, a processor performs communication for only those rows of A that correspond to the nodes of $G(A)$ lying at the boundary of the processor's partition. If the graph is partitioned properly, the communication cost can be reduced significantly



(a) A 16 x 16 symmetric random sparse matrix (b) The associated graph and its four partitions

Figure 11.12 A 16×16 unstructured sparse matrix with symmetric structure and its associated graph partitioned among four processors.

both in terms of the number of messages and the volume of communication (Problems 11.8 and 11.9).

Partitioning an arbitrary graph $G(A)$ to minimize interprocessor communication is a hard combinatorial problem. However, there are several good heuristics for graph partitioning. These partitioning techniques are described in detail in Section 11.3. Often, the origin of the unstructured sparse matrix A lies in a finite element problem. In such a case, the graph $G(A)$ can be derived from the finite element graph directly.

The technique described here can also be adapted for randomly sparse matrices that are non-symmetric in structure. In such cases, a directed graph results, and the communication takes place in the direction opposite to the direction of an edge crossing a partition boundary. For example, if $A[i, j] \neq 0$, then there is a directed edge from node i to node j in $G(A)$. If nodes i and j belong to different partitions, then the j^{th} element of the vector must be sent to the processor storing the i^{th} row of matrix A.

Banded Unstructured Sparse Matrices

We often encounter linear systems in which the nonzero elements of the sparse matrix of coefficients occur only within a band parallel to the principal diagonal. Even if the nonzero elements are scattered throughout the matrix, it is often possible to restrict them to a band by using certain reordering techniques. In this subsection we discuss matrix-vector multiplication for banded unstructured sparse matrices.

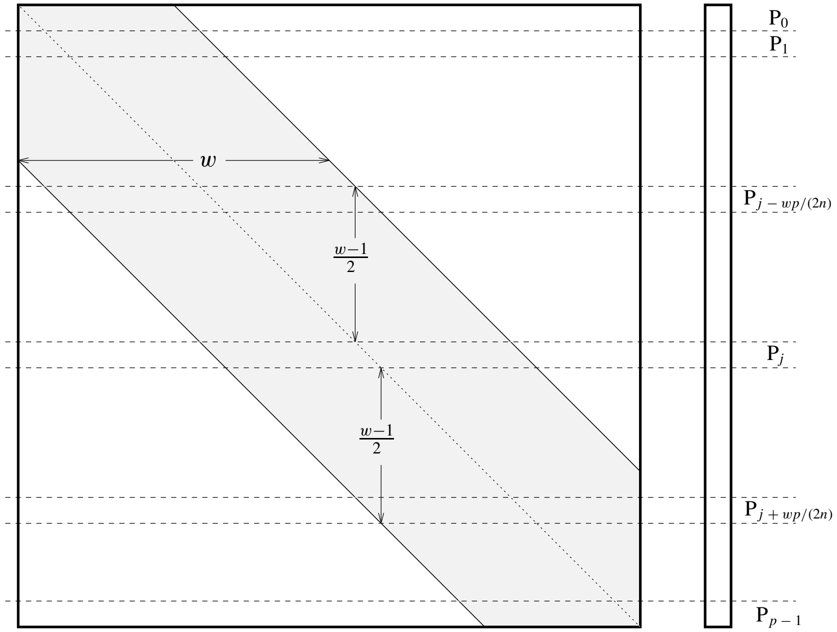


Figure 11.13 Matrix-vector multiplication with a block-striped partitioning of a banded unstructured sparse matrix.

For simplicity of analysis, we assume that the width of the band is w and that it spreads evenly to a width of $(w-1)/2$ on both sides of the principal diagonal. The $n \times n$ matrix is stored in Ellpack-Itpack format, and the average number of nonzero elements per row is m. The matrix and the vector are distributed among the processors as shown in Figure 11.13. Each of the p processors initially stores the nonzero entries of n/p rows of the matrix and n/p elements of the vector. We consider only the case in which neither the width of the band nor the number of processors is trivially small. Hence, we assume that $n/p \ll w$.

Given the distribution of nonzero elements just described, the maximum column index of a nonzero element in the i^{th} row of the matrix is $i + (w-1)/2$, and its minimum column index is $i - (w-1)/2$. Thus, the i^{th} row requires those elements of the vector that have indices between $i - (w-1)/2$ and $i + (w-1)/2$. Furthermore, the indices of the rows that require the i^{th} vector element lie between $i-(w-1)/2$ and $i+(w-1)/2$. Since each processor stores n/p matrix rows, the half band of $(w-1)/2$ rows is spread among $\lceil (w-1)p/2n \rceil$ processors. Hence, a processor needs to send all its n/p vector elements to $\lceil (w-1)p/2n \rceil$ processors on either side. A typical processor P_j communicates with all the processors with labels between $j - \lceil (w-1)p/2n \rceil$ and $j + \lceil (w-1)p/2n \rceil$. From now on, we will assume that $wp/2n$ is a whole number and $\lceil w(p-1)/2n \rceil$ is rounded off to $wp/2n$.

Thus, each processor sends all its vector elements to approximately wp/n processors (see Figure 11.13 for an illustration). Contrast this with the case of an unstructured sparse matrix, in which each processor sends its vector elements to all other processors.

Assuming that the processors are connected in a linear array, communication takes wp/n steps, $wp/2n$ in each direction in the linear array. In the first step, all processors (except those at the ends of the linear array) send vector elements to their neighbors in one direction. In subsequent steps, each processor stores the data received from one neighbor and forwards them to the other neighbor. After performing $wp/2n$ communication steps in one direction, each processor performs another $wp/2n$ similar steps in the other direction. The total communication time is $(t_s + t_w n/p) \times wp/n = t_s wp/n + t_w w$. Since there is an average of m nonzero elements per row of the matrix, and each processor stores the nonzero elements of n/p rows, the average number of scalar multiplication-addition pairs that each processor performs is mn/p. If we assume a uniform workload, the parallel run time is

$$T_P = t_c \frac{mn}{p} + t_s \frac{wp}{n} + t_w w. \tag{11.12}$$

The processor-time product is $t_c mn + t_s wp^2/n + t_w wp$. For cost-optimality, the processor-time product should not exceed the serial time complexity of the algorithm, which is $\Theta(mn)$. Consider the term associated with t_s first. If this term is not to exceed $\Theta(mn)$, then $p^2 w/n = O(mn)$, or $p = O(n\sqrt{m/w})$. Similarly, if the t_w term is not to exceed $\Theta(mn)$, then $wp = O(mn)$, or $p = O(mn/w)$. Since $m < w$, we have $m/w < 1$ and $m/w < \sqrt{m/w}$. Therefore, the overall (most restrictive) condition for cost-optimality is $p = O(mn/w)$.

Thus, matrix-vector multiplication with unstructured sparse matrices is cost-optimal and scalable if the nonzero elements are confined to a band rather than scattered over the entire matrix (Problems 11.10 and 11.11). The number of processors that can be used cost-optimally is directly proportional to the number of nonzero elements in each row of the sparse matrix, and inversely proportional to the width of the band in which the nonzero elements are distributed.

11.2 Iterative Methods for Sparse Linear Systems

Iterative methods are techniques to solve systems of equations of the form $Ax = b$ that generate a sequence of approximations to the solution vector x. In each iteration, the coefficient matrix A is used to perform a matrix-vector multiplication. The number of iterations required to solve a system of equations with a desired precision is usually data dependent; hence, the number of iterations is not known prior to executing the algorithm. Therefore, in this section we analyze the performance and scalability of a single iteration of an iterative method. Iterative methods do not guarantee a solution for all systems of equations. However, when they do yield a solution, they are usually less expensive than direct methods for matrix factorization. In the following section, we study some commonly used iterative methods for solving large sparse systems of linear equations.

11.2.1 Jacobi Iterative Method

The Jacobi iterative method is one of the simplest iterative techniques. The i^{th} equation of a system of linear equations $Ax = b$ is

$$\sum_{j=0}^{n-1} A[i,j]x[j] \quad = \quad b[i]. \tag{11.13}$$

If all the diagonal elements of A are nonzero (or are made nonzero by permuting the rows and columns of A), we can rewrite Equation 11.13 as

$$x[i] \quad = \quad \frac{1}{A[i,i]}\left(b[i] - \sum_{j\neq i} A[i,j]x[j]\right). \tag{11.14}$$

The Jacobi method starts with an initial guess x_0 for the solution vector x. This initial vector x_0 is used in the right-hand side of Equation 11.14 to arrive at the next approximation x_1 to the solution vector. The vector x_1 is then used in the right hand side of Equation 11.14, and the process continues until a close enough approximation to the actual solution is found. A typical iteration step in the Jacobi method is

$$x_k[i] = \frac{1}{A[i,i]}\left(b[i] - \sum_{j\neq i} A[i,j]x_{k-1}[j]\right). \tag{11.15}$$

The process is said to have converged after k iterations of Equation 11.15 if the magnitude of the vector $(b - Ax_k)$ becomes reasonably small. The vector $(b - Ax)$ is zero for the exact solution x. Hence, $(b - Ax_k)$, denoted by r_k, represents the error in the approximation of x and is referred to as the ***residual*** after k iterations. The square root of the inner product $r_k^T r_k$ (that is, $\sqrt{r_k^T r_k}$, which is also called the ***two-norm*** of r_k and is denoted by $\|r_k\|_2$) is commonly used to represent the magnitude of the error at the end of the k^{th} iteration. The procedure terminates when $\|r_k\|_2$ falls below a predetermined threshold, which is usually a very small fraction $\epsilon\|r_0\|_2$ (where, $0 < \epsilon \ll 1$) of the two-norm of the initial residual r_0.

We now express the iteration step of Equation 11.15 in terms of the residual r_k. Equation 11.15 can be rewritten as

$$x_k[i] = \frac{1}{A[i,i]}\left(b[i] - \sum_{j=0}^{n-1} A[i,j]x_{k-1}[j]\right) + x_{k-1}[i]. \tag{11.16}$$

By the definition of the residual, $r_{k-1} = b - Ax_{k-1}$. Therefore, $b[i] - \Sigma_{j=0}^{n-1} A[i,j]x_{k-1}[j]$ in Equation 11.16 can be replaced by $r_{k-1}[i]$. Hence, a Jacobi iteration is given by the following equation:

$$x_k[i] = \frac{r_{k-1}[i]}{A[i,i]} + x_{k-1}[i] \tag{11.17}$$


```
1.  procedure JACOBI_METHOD (A,b,x,ϵ)
2.  begin
3.     k := 0;
4.     Select initial solution vector x_0;
5.     r_0 := b − Ax_0;
6.     while (‖r_k‖_2 > ϵ‖r_0‖_2) do
7.     begin
8.        k := k + 1;
9.        for i := 0 to n − 1 do
10.          x_k[i] := r_{k−1}[i]/A[i, i] + x_{k−1}[i];   /* Equation 11.17 */
11.       r_k := b − Ax_k;
12.    endwhile;
13.    x := x_k;
14. end JACOBI_METHOD
```

Program 11.2 The serial Jacobi iterative method for solving a system of linear equations.

The resulting algorithm is given in Program 11.2. The Jacobi algorithm given in Program 11.2 is not guaranteed to converge for all types of matrices. One class of matrices for which it always converges is that of diagonally-dominant matrices. An $n \times n$ matrix A is ***diagonally dominant*** if and only if $|A[i,i]| > \Sigma_{j\neq i}|A[i,j]|, 0 \leq i < n$.

Parallel Implementation

Each iteration of the Jacobi method given in Program 11.2 performs three main computations: the inner product on line 6, the loop of lines 9 and 10, and the matrix-vector multiplication on line 11. If the matrix and the vector are mapped onto the processors of a parallel computer such that $A[i,i]$, $r_k[i]$, and $x_k[i]$ are assigned to the same processor for $0 \leq i < n$, then the loop of lines 9 and 10 does not require any communication. The mappings shown in Figures 11.9–11.13 all satisfy this condition. Sometimes, for the purpose of load balancing, a mapping like the one shown in Figure 11.29(a) (Problem 11.1) may be desirable. In this mapping, each processor performs the same number of scalar multiplications and additions while multiplying the matrix with a vector; however, $A[i,i]$ and $x[i]$ may not be assigned to the same processor. This situation can be remedied by modifying the mapping slightly. The principal diagonal of the $n \times n$ coefficient matrix A is treated as an $n \times 1$ vector and is stored separately from the rest of the matrix. Now the elements of the principal diagonal are mapped onto the same processors as those of the vectors r_k and x_k, and the rest of the matrix is mapped as shown in Figure 11.29(a).

Thus, the loop on lines 9 and 10 can be executed in parallel without any communication. The two steps that require communication in each iteration are the computation of the norm of the residual r_k (line 6), which is a vector inner product, and matrix-vector multiplication (line 11). Vector inner product and sparse matrix-vector multiplication are discussed in Sections 11.1.2 and 11.1.3, respectively. Of these, the operation with the larger communication overhead determines the overall performance and scalability of the Jacobi algorithm on a parallel architecture.

As discussed in Section 11.1.2, if no special hardware is available to add p numbers distributed on p processors, the communication time for an inner-product computation is $\Theta(\log p)$ on a hypercube and $\Theta(\sqrt{p})$ on a square mesh. This translates to a total overhead of $\Theta(p \log p)$ and $\Theta(p^{3/2})$ for each iteration on a hypercube and a mesh, respectively. Therefore, the isoefficiency function of an iteration of the Jacobi method is at least $\Theta(p \log p)$ on a hypercube and $\Theta(p^{3/2})$ on a mesh. As discussed in Section 11.1.3, the communication overhead in parallel sparse matrix-vector multiplication depends on the sparsity pattern on the matrix. If the structure of the sparse matrix is such that the isoefficiency function due to matrix-vector multiplication is less than that due to inner-product computation, then it is possible to reduce the overall isoefficiency function of an iteration of the Jacobi algorithm. Recall from Program 11.2 that the inner-product is computed in each iteration only to test for convergence (line 6). If the convergence test is performed once every $\log p$ iterations on a hypercube, then the total number of iterations may increase by at most $\log p$, but the overhead of each inner-product calculation is amortized over $\log p$ iterations. Thus, the total overhead due to the inner product calculations is reduced to $\Theta(p)$. The isoefficiency function due to the inner-product calculation is then also reduced to $\Theta(p)$. Similar results can be obtained by performing the convergence check once every $\Theta(\sqrt{p})$ iterations on a mesh.

11.2.2 Gauss-Seidel and SOR Methods

As we mentioned earlier, the Jacobi algorithm does not always converge. Even if it does, the rate of its convergence is often very slow. The Gauss-Seidel method improves on the convergence properties of the Jacobi method. However, like the Jacobi method, the Gauss-Seidel method is not always guaranteed to converge.

An iteration of the Jacobi method is based on Equation 11.15. During the k^{th} iteration of Jacobi algorithm to solve an $n \times n$ system, the step of Equation 11.15 is performed to compute each $x_k[i]$ for $0 \le i < n$. The computation of $x_k[i]$ uses the values of $x_{k-1}[0]$, $\ldots$, $x_{k-1}[i-1]$, $x_{k-1}[i+1]$, $\ldots$, $x_{k-1}[n-1]$. Assuming that the $x_k[i]$ values are computed in increasing order of i, the values of $x_k[0], \ldots, x_k[i-1]$ have already been computed before Equation 11.15 is used to compute $x_k[i]$. However, the Jacobi algorithm uses $x_{k-1}[0], \ldots, x_{k-1}[i-1]$ from the previous iteration. The Gauss-Seidel algorithm uses the most recent value of each variable, and as a result, often achieves faster convergence than the Jacobi algorithm. The basic Gauss-Seidel iteration is given by

$$x_k[i] = \frac{1}{A[i,i]}\left(b[i] - \sum_{j=0}^{i-1} x_k[j]A[i,j] - \sum_{j=i+1}^{n-1} x_{k-1}[j]A[i,j]\right). \tag{11.18}$$

Parallel Implementation

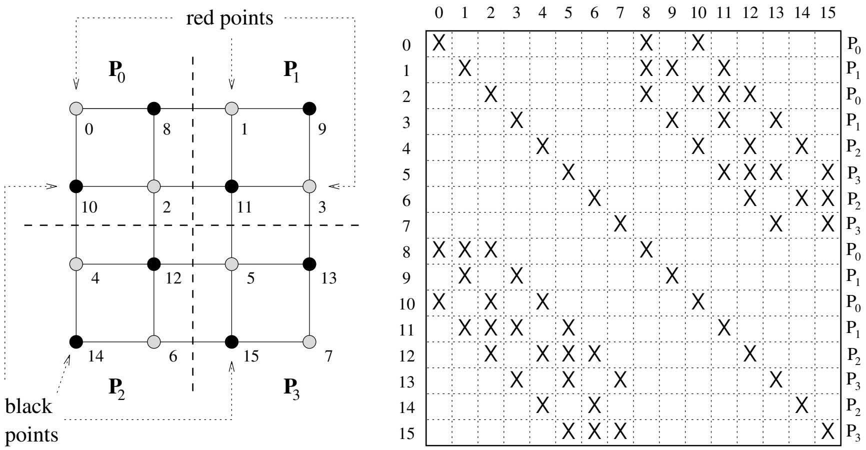
The Gauss-Seidel algorithm performs the basic iteration given by Equation 11.18 until satisfactory convergence is achieved. As in Jacobi method, the test for convergence requires an inner-product computation that involves global communication. However, a convergence check need not be performed after each iteration. Given a parallel architecture, the frequency of the convergence check can be chosen to optimize the overall performance on that architecture (Section 11.2.1). The issues in parallelizing the inner-product computation are discussed in Section 11.1.2. In this section we concentrate on performing the iteration step of Equation 11.18 in parallel.

From a preliminary glance at Equation 11.18 it might appear that computing $x_k[0]$, $x_k[1], \ldots, x_k[n-1]$ in the k^{th} iteration is completely sequential because $x_k[i]$ cannot be computed until $x_k[i-1]$ has been computed for $0 \le i < n$. This is indeed the case if the coefficient matrix A is dense. However, if A is sparse, the computation of $x_k[i]$ need not wait until $x_k[0], \ldots, x_k[i-1]$ have *all* been computed. A majority of elements in the sparse matrix A are zero. If $A[i, j]$ is zero, then $x_k[i]$ on the left-hand side of Equation 11.18 does not depend upon $x_k[j]$. Thus, $x_k[i]$ can be computed as soon as all $x_k[j]$ have been computed such that $j < i$ and $A[i, j] \ne 0$. At any time, all $x_k[i]$ for which this condition is true can be computed in parallel.

Since the computation of $x_k[i]$ depends only on the nonzero elements $A[i, j]$ (with $j < i$) in the coefficient matrix, the degree of parallelism in Gauss-Seidel method is a function of the sparsity pattern of the lower-triangular part of A. For example, consider the block-tridiagonal matrix of the form shown in Figure 11.7. Such a matrix results from a finite difference discretization with a natural ordering of grid points, as shown in Figure 11.8. In the $\sqrt{n} \times \sqrt{n}$ grid in Figure 11.8, except for the points on the left periphery, every point i has point $i-1$ as its neighbor. Therefore, except in the rows corresponding to the grid points on the left periphery, $A[i, i-1]$ is not equal to zero. As a result, for all but $\sqrt{n}$ values of i, the computation of $x_k[i]$ has to wait until $x_k[i-1]$ has been computed. Hence, natural ordering is not suitable for a parallel implementation of Gauss-Seidel algorithm. It can be shown that each Gauss-Seidel iteration on an $n \times n$ block-tridiagonal matrix of the form shown in Figure 11.7 takes at least $\Theta(\sqrt{n})$ time regardless of the number of processors used (Problem 11.13).

The order in which the grid points in a discretized domain are numbered determines the order of the rows and columns in the coefficient matrix, and hence, the location of its nonzero elements. The degree of parallelism in the Gauss-Seidel algorithm depends heavily on this ordering. The rate of convergence of the Gauss-Seidel algorithm for a given grid is also sensitive to this ordering. However, given enough processors, an ordering more amenable to parallelization is likely to yield a better overall performance, unless it results in much worse convergence.

Red-Black Ordering We now introduce a numbering scheme for a finite difference grid so that the resulting coefficient matrix permits a high degree of parallelism in a Gauss-Seidel iteration. We will later extend this scheme to deal with sparse matrices other than those



(a) A 4 × 4 grid with red-black ordering

(b) Corresponding coefficient matrix

Figure 11.14 A partitioning among four processors of a 4×4 finite difference grid with red-black ordering.

resulting from a finite difference discretization. Figure 11.14(a) illustrates this ordering, which is known as ***red-black ordering***. In red-black ordering, alternate grid points in each row and column are colored red, and the remaining points are colored black. For a uniform two-dimensional grid in which each point has a maximum of four neighbors, this ensures that no two directly-connected grid points have the same color. After assigning colors to the grid points, all red points are numbered first in natural order, leaving out the black points. This is followed by numbering all the black points in natural order. If the grid has a total of n points and n is even, the red points are numbered from 0 to $(n/2) - 1$ and the black points are numbered from $n/2$ to $n - 1$.

Figure 11.14(b) shows the sparse matrix resulting from the 4×4 grid of Figure 11.14(a). In the coefficient matrix, the first $n/2$ rows correspond to the red points, and the last $n/2$ rows correspond to the black points in the grid. Since red points have only black neighbors and vice versa, the first $n/2$ rows have nondiagonal nonzero elements in only the last $n/2$ columns, and the last $n/2$ rows have nondiagonal nonzero elements in only the first $n/2$ columns.

Consider a parallel implementation of the Gauss-Seidel method on a two-dimensional mesh of processors. The grid is partitioned among the processors of the mesh in a block-checkerboard fashion. Figure 11.14 shows the allocation of grid points and the rows of the coefficient matrix among four processors. With red-black ordering, each iteration of the Gauss-Seidel algorithm is performed in two phases. In the k^{th} iteration, first, $x_k[0], x_k[1], \ldots, x_k[n/2-1]$ are computed in parallel. Each of these variables corresponds to red points, and uses values of the variables corresponding to its black neighbors from the previous iteration. To perform this computation, each processor sends the variables corresponding to the black points lying at each of its four partition boundaries to the respective neighboring processors. A typical processor has four boundaries with $\sqrt{n/p}$ points on each boundary. Half of these points are red and the other half are black. Therefore, the communication time of the first phase is $4 \times (t_s + t_w\sqrt{n/p} \times 1/2)$, which is equal to $4t_s + 2t_w\sqrt{n/p}$. In the second phase, $x_k[n/2], x_k[n/2+1], \ldots, x_k[n-1]$ are computed in parallel. Each of these variables use the values of the variables corresponding to the red neighbors that were computed in the first phase of the k^{th} iteration. This requires an exchange of all the $\sqrt{n}/(2\sqrt{p})$ variables corresponding to the red points at each of the four partition boundaries. As in the first phase, the communication in the second phase takes $4t_s + 2t_w\sqrt{n/p}$ time.

Each evaluation of Equation 11.18 for the coefficient matrix resulting from a grid of the form shown in Figure 11.14(a) requires at most four multiplications, four subtractions, and one division. The number of multiplications and subtractions is at most four because there are at most four nondiagonal nonzero elements in each row of A—one corresponding to each of the four neighbors of a point in the grid. These operations are performed once for each variable in every iteration. Assuming that this constant amount of computation per grid point (or per row of the coefficient matrix) takes time t_c, the total execution time per iteration is

$$T_P = t_c n/p + 8t_s + 4t_w\sqrt{n/p}. \tag{11.19}$$

Equation 11.19 does not include the time spent in testing for convergence, which depends on how and with what frequency the convergence test is performed.

Multicolored Ordering for General Matrices Recall from Figure 11.12 that a matrix A can be regarded as the adjacency matrix of a graph $G(A)$. We now extend the idea behind red-black ordering to devise an ordering scheme for sparse matrices that arise from finite element problems (Section 11.3).

Multicolored ordering is an ordering scheme in which the nodes of graph $G(A)$ associated with a matrix A are colored such that no two neighboring nodes have the same color. Although coloring the graph in such a way is a combinatorial problem of exponential complexity, usually simple heuristics are sufficient to color most graphs arising out of practical problems by using a small number of colors. The nodes of each color are assigned labels one after the other, and all nodes of the same color have consecutive labels. The system of equations is rewritten such that the i^{th} equation and the variable x_i correspond to the node labeled i in $G(A)$. Figure 11.15 shows the multicolored ordering of a graph with four colors. The coefficient matrix of the system of linear equations resulting from the grid of Figure 11.15 is of size 32×32. An iteration of parallel Gauss-Seidel algorithm to solve this system is performed in four phases. In the first phase of the k^{th} iteration, $x_k[0], \ldots, x_k[7]$ are computed in parallel using $x_{k-1}[8], \ldots, x_{k-1}[31]$. In the second phase, $x_k[8], \ldots, x_k[15]$ are computed in parallel using $x_k[0], \ldots, x_k[7]$ and $x_{k-1}[16], \ldots, x_{k-1}[31]$. In the third phase, $x_k[16], \ldots, x_k[23]$ are computed in par-

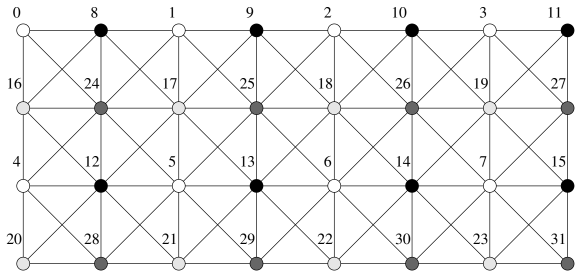


Figure 11.15 Multicolored ordering of a finite element graph using four colors.

allel using $x_k[0], \ldots, x_k[15]$ and $x_{k-1}[24], \ldots, x_{k-1}[31]$, and finally, in the fourth phase, $x_k[24], \ldots, x_k[31]$ are computed in parallel using $x_k[0], \ldots, x_k[24]$.

In general, if the coefficient matrix is ordered using multicolored ordering, the number of phases in a Gauss-Seidel iteration is equal to the number of colors used in the ordering. In any iteration, all variables corresponding to the grid points of the same color can be updated in parallel.

The SOR Method

Often, we can obtain a significant improvement in convergence speed by modifying the iteration step of Equation 11.18. The ***successive overrelaxation (SOR)*** method is such an extension of the Gauss-Seidel method. The SOR method computes $x_k[i]$ in the k^{th} iteration as a weighted average of $x_{k-1}[i]$ and the $x_k[i]$ given by Equation 11.18. In the SOR algorithm, a parameter ω (typically, $0 < \omega \leq 2$) is appropriately chosen, and the following iteration step is used:

$$x_k[i] = (i - \omega)x_{k-1}[i] + \frac{\omega}{A[i,i]}\left(b[i] - \sum_{j=0}^{i-1} x_k[j]A[i,j] - \sum_{j=i+1}^{n-1} x_{k-1}[j]A[i,j]\right) \tag{11.20}$$

The parallelization issues and the communication costs in a parallel implementation of the SOR algorithm are the same as those for the Gauss-Seidel algorithm.

★ 11.2.3 The Conjugate Gradient Method

The ***conjugate gradient (CG) method*** is one of the most powerful and widely used iterative methods for solving large sparse systems of linear equations of the form $Ax = b$, where A is a symmetric positive definite matrix. A real $n \times n$ matrix A is ***positive definite*** if $x^T Ax > 0$ for any $n \times 1$ real, nonzero vector x.

The CG method belongs to a class of iterative methods known as ***minimization methods***. For a symmetric positive definite matrix A, the unique x that minimizes the

quadratic function $q(x) = (1/2)x^T Ax - x^T b$ is the solution to the system $Ax = b$. The reason is that the gradient of $q(x)$ is $Ax - b$, which is zero when $q(x)$ is minimum. An iteration of a minimization method is of the form

$$x_k = x_{k-1} + \alpha_k p_k, \tag{11.21}$$

where α_k is a scalar step size and p_k is the direction vector. For a given x_{k-1} and p_k, the scalar α_k is chosen to minimize $q(x_k)$; that is, α_k is the value of α for which $q(x_{k-1} + \alpha p_k)$ is minimum. The function $q(x_{k-1} + \alpha p_k)$ is quadratic in α, and its minimization leads to the condition

$$\alpha_k = \frac{p_k^T r_{k-1}}{p_k^T A p_k}, \tag{11.22}$$

where $r_{k-1} = b - Ax_{k-1}$ is the residual vector after $k-1$ iterations. The residual need not be computed explicitly in each iteration because it can be computed incrementally by using its value from the previous iteration. In the k^{th} iteration, the residual r_k can be expressed as follows:

$$\begin{aligned} r_k &= b - Ax_k \\ &= b - A(x_{k-1} + \alpha_k p_k) \\ &= b - Ax_{k-1} - \alpha_k A p_k \\ &= r_{k-1} - \alpha_k A p_k \end{aligned} \tag{11.23}$$

Thus, the only matrix-vector product computed in each iteration is Ap_k, which is already required to compute α_k (Equation 11.22).

If A is a symmetric positive definite matrix and $p_1, p_2, \ldots, p_n$ are direction vectors that are conjugate with respect to A (that is, $p_i^T A p_j = 0$ for all $0 < i, j \leq n, i \neq j$), then x_k in Equation 11.21 converges to the solution of $Ax = b$ in at most n iterations, assuming no rounding errors. In practice, however, the number of iterations that yields an acceptable approximation to the solution is much smaller than n. In the CG algorithm, the set of A-conjugate direction vectors is chosen as follows:

$$\begin{aligned} p_1 &= r_0 = b \\ p_{k+1} &= r_k + \frac{\|r_k\|_2^2 p_k}{\|r_{k-1}\|_2^2} \end{aligned} \tag{11.24}$$

With the preceding choice of direction vectors, the ratio $(p_k^T r_{k-1})/(p_k^T A p_k)$ is equal to $\|r_{k-1}\|_2^2/(p_k^T A p_k)$. Thus, Equation 11.22 can be rewritten as follows:

$$\alpha_k = \frac{\|r_{k-1}\|_2^2}{p_k^T A p_k} \tag{11.25}$$

Equations 11.21, 11.23, 11.24, and 11.25 lead to the conjugate gradient algorithm given in Program 11.3. The algorithm terminates when the two-norm of the current residual falls below a predetermined fraction of the two-norm of the initial residual r_0.

procedure CG (A, b, x, ϵ)
begin
  $k := 0$; $x_0 := 0$; $r_0 := b$; $\rho_0 := \|r_0\|_2^2$;
  while $(\sqrt{\rho_i} > \epsilon \|r_0\|_2)$ **do**
  begin
    if $(k = 0)$ **then** $p_1 := r_0$
    else $p_{k+1} := r_k + \rho_k p_k / \rho_{k-1}$; /* Equation 11.24 */
    $k := k + 1$;
    $w_k := A p_k$;
    $\alpha_k := \rho_{k-1} / p_k^T w_k$; /* Equation 11.25 */
    $x_k := x_{k-1} + \alpha_k p_k$; /* Equation 11.21 */
    $r_k := r_{k-1} - \alpha_k w_k$; /* Equation 11.23 */
    $\rho_k := \|r_k\|_2^2$;
  endwhile;
  $x := x_k$;
end CG

Program 11.3 The conjugate gradient (CG) algorithm.

The Preconditioned Conjugate Gradient Algorithm

If the coefficient matrix A has l distinct eigenvalues, the conjugate gradient algorithm given in Program 11.3 converges to the solution of the system $Ax = b$ in at most l iterations (assuming no rounding errors). Therefore, if A has many distinct eigenvalues that vary widely in magnitude, the CG algorithm may require a large number of iterations to converge to an acceptable approximation to the solution. The speed of convergence of the CG algorithm can be increased by preconditioning A with the congruence transformation $\tilde{A} = RAR^T$, where R is a nonsingular matrix. R is chosen such that $\tilde{A}$ has fewer distinct eigenvalues than A. The CG algorithm is then used to solve $\tilde{A}\tilde{x} = \tilde{b}$, where $\tilde{x} = (R^T)^{-1}x$ and $\tilde{b} = Rb$. The resulting algorithm is called the ***preconditioned conjugate gradient (PCG) algorithm***.

There are certain problems with applying the CG algorithm directly to the system $\tilde{A}\tilde{x} = \tilde{b}$. Unless R is a diagonal matrix, the sparsity pattern of A is not preserved in $\tilde{A}$. Moreover, the matrix multiplications involved in computing $\tilde{A}$ can be expensive. Fortunately, it is possible to formulate the PCG algorithm so that the explicit computation of $\tilde{A}$ is avoided. A practical PCG algorithm works with the original matrix A; however, it maintains the same convergence rate as that for the system $\tilde{A}\tilde{x} = \tilde{b}$. Such a practical PCG algorithm is given in Program 11.4. The matrix M in the program is referred to as the ***preconditioner*** matrix and is given by $M = (R^T R)^{-1}$. If M is an identity matrix, then the PCG algorithm reduces to the unpreconditioned algorithm of Program 11.3.

In practical implementations of the PCG algorithm, the preconditioner M is directly chosen as a symmetric positive definite matrix, and computing it by using $M = (R^T R)^{-1}$

1. **procedure** PCG (A, b, M, x, ϵ)
2. **begin**
3. $k := 0$; $x_0 := 0$; $r_0 := b$; $\rho_0 := \|r_0\|_2^2$;
4.   **while** $(\sqrt{\rho_i} > \epsilon \|r_0\|_2)$ **do**
5.   **begin**
6.   Solve the system $M z_k = r_k$;
7. $\gamma_k := r_k^T z_k$;
8.   **if** $(k = 0)$ **then** $p_1 := z_0$
9.   **else** $p_{k+1} := z_k + \gamma_k p_k / \gamma_{k-1}$;
10. $k := k + 1$;
11. $w_k := A p_k$;
12. $\alpha_k := \gamma_{k-1} / p_k^T w_k$;
13. $x_k := x_{k-1} + \alpha_k p_k$;
14. $r_k := r_{k-1} - \alpha_k w_k$;
15. $\rho_k := \|r_k\|_2^2$;
16.   **endwhile**;
17. $x := x_k$;
18. **end** PCG

Program 11.4 The preconditioned conjugate gradient (PCG) algorithm.

is not required. For obvious reasons, M is chosen such that solving the system $M z_k = r_k$ on line 6 in each iteration of Program 11.4 is not too costly.

Parallel Implementations of the PCG Algorithm

As Program 11.4 shows, the PCG algorithm involves the following four types of computations in each iteration:

(1) **SAXPY operations:** The operations on lines 9, 13, and 14 of Program 11.4 are known as ***simple ax plus y (SAXPY)*** operations, where a is a scalar, and x and y are vectors. Each of these operations can be performed sequentially in time $\Theta(n)$, regardless of the preconditioner and the type of coefficient matrix. If all vectors are distributed identically among the processors, these steps require no communication in a parallel implementation. The reason is that the vector elements with the same indices are involved in a given arithmetic operation, and thus are locally available on each processor. Using p processors, each of these steps is performed in time $\Theta(n/p)$ on any architecture.

(2) **Vector inner products:** Lines 7, 12, and 15 in Program 11.4 involve vector inner-product computation. In a serial implementation, each of these steps is performed in $\Theta(n)$ time. As discussed in Section 11.1.2, a parallel implementation with p processors takes $\Theta(n/p) + t_s \log p$ time on the hypercube architecture and

$\Theta(n/p) + t_s \log p + 2t_h\sqrt{p}$ time on a mesh. If the parallel computer supports fast reduction operations, the communication time for the inner-product calculations can be ignored.

(3) **Matrix-vector multiplication:** The computation and the communication cost of the matrix-vector multiplication step of line 11 depends on the structure of the sparse matrix A. We study parallel implementations of the PCG algorithm for two cases—one in which A is a block-tridiagonal matrix of the type shown in Figure 11.7, and the other in which it is a banded unstructured sparse matrix.

(4) **Solving the system $Mz_k = r_k$:** The PCG algorithm solves a system of linear equations $Mz_k = r_k$ in each iteration (line 6). The preconditioner M is chosen so that solving the system $Mz_k = r_k$ is inexpensive compared to solving the original system of equations $Ax = b$. Nevertheless, preconditioning increases the amount of computation in each iteration. For good preconditioners, however, the increase is compensated by a reduction in the number of iterations required to achieve acceptable convergence.

The computation and the communication requirements of this step depend on the type of preconditioner used. In this chapter we study parallel implementations of the PCG algorithm for two types of preconditioning methods: (1) diagonal preconditioning, in which the preconditioner matrix M has nonzero elements only along the principal diagonal, and (2) incomplete Cholesky (IC) preconditioning, in which M is based on an incomplete Cholesky factorization of A. There are several variants of the IC preconditioner. We describe a few variants for which solving the system $Mz_k = r_k$ is an easily parallelizable operation. A PCG algorithm using IC preconditioning is also referred to as an ICCG algorithm.

Note that, among the four types of computations that we just described, an unpreconditioned conjugate gradient algorithm performs only the first three. In the remainder of this section, we consider parallel implementations of the PCG algorithm for different combinations of preconditioners and coefficient matrix types. As we will see, if M is a diagonal preconditioner, then solving the system $Mz_k = r_k$ does not require any interprocessor communication. Hence, the communication time in a CG iteration with diagonal preconditioning is the same as that in an iteration of the unpreconditioned algorithm.

A Diagonal Preconditioner and a Matrix Resulting from a Finite Difference Discretization Assume that the $n \times n$ coefficient matrix A arises from a $\sqrt{n} \times \sqrt{n}$ finite difference grid, and that the grid points (and hence, the matrix rows) are partitioned among the processors as shown in Figure 11.10. Let the preconditioner matrix M be a simple diagonal matrix with nonzero elements only along its principal diagonal, which is usually derived from (or is the same as) the principal diagonal of A. Solving the system $Mz_k = r_k$ on line 6 of Program 11.4 is equivalent to dividing each element of r by the diagonal entry of the corresponding row of M. No communication is required because elements with identical indices reside on the same processor. Hence, in a p-processor implementation, each processor performs this operation in $\Theta(n/p)$ time. The matrix-

vector multiplication operation requires $\Theta(n/p)$ time for computation and $4t_s + 4t_w\sqrt{n/p}$ time for communication (Equation 11.6). Each of the three inner products takes $\Theta(n/p)$ for computation. Additionally, it takes approximately $t_s \log p$ time for communication on a hypercube and $t_s \log p + 2t_h\sqrt{p}$ time on a mesh with cut-through routing (Section 11.1.2).

The total time spent in performing all the computation in each iteration is $\Theta(n/p)$. If $t_c^{'}$ is the constant associated with the computation, then the parallel run time for a single iteration of the PCG algorithm on hypercube and mesh architectures is given by the following equations:

Hypercube:

$$T_P = \overbrace{t_c^{'}\frac{n}{p}}^{\text{computation}} + \overbrace{3t_s \log p}^{\text{inner products}} + \overbrace{4t_s + 4t_w\sqrt{n/p}}^{\text{matrix-vector multiplication}} \tag{11.26}$$

Mesh with cut-through routing:

$$T_P = \overbrace{t_c^{'}\frac{n}{p}}^{\text{computation}} + \overbrace{3t_s \log p + 6t_h\sqrt{p}}^{\text{inner products}} + \overbrace{4t_s + 4t_w\sqrt{n/p}}^{\text{matrix-vector multiplication}} \tag{11.27}$$

The isoefficiency functions of this implementation of the PCG algorithm for the hypercube and mesh architectures can be derived using the expressions in Equations 11.26 and 11.27, respectively. Since the total useful computation performed in each iteration is $\Theta(n)$, the isoefficiency function for the hypercube is $\Theta(p \log p)$, and that for the mesh is $\Theta(p\sqrt{p})$ for an iteration of the algorithm. Both isoefficiency functions result from the communication due to vector inner-product computations. If the algorithm is executed on a machine with a fast built-in reduction operation, this overhead can be ignored. In that case, the only time spent in communication in each iteration is $4t_s + 4t_w\sqrt{n/p}$ on both the mesh and hypercube architectures. The resulting expression for parallel run time is

$$T_P = t_c^{'}\frac{n}{p} + 4t_s + 4t_w\sqrt{n/p}. \tag{11.28}$$

An isoefficiency function of $\Theta(p)$ follows from Equation 11.28, which means that the parallel system is ideally scalable (Problem 11.17). Thus, for this algorithm, the availability of a fast reduction operation proves to be very useful.

An IC Preconditioner and a Matrix Obtained from Red-Black Ordering Program 5.6 gives a row-oriented Cholesky factorization algorithm for dense matrices. If the same algorithm is used to factorize a sparse matrix A as the product $L \times L^T$, where L is a lower-triangular matrix, then the factors L and L^T are much less sparse than A. A ***no-fill incomplete Cholesky factorization*** is a procedure that performs the computation of line 10 of Program 5.6 only if $A[i, j]$ is nonzero. Replacing line 10 of Program 5.6 by

if $A[i, j] \neq 0$ **then** $A[i, j] := A[i, j] - A[k, i] \times A[k, j]$;

and executing the resulting algorithm on a sparse matrix A yields a sparse upper-triangular matrix $L^{'T}$. The locations of the nonzero elements in $L^{'T}$ coincide exactly with the locations of the nonzero elements in the upper-triangular portion of A. Since A is symmetric, the locations of nonzero elements in $L^{'}$ coincide with those in the lower-triangular part of A.

We use the matrix $M = L^{'}L^{'T}$ as the preconditioner on line 6 of the PCG algorithm given in Program 11.4. With this choice of M, the system $Mz_k = r_k$ is solved by solving the following two triangular systems:

(1) solve $L^{'}u = r_k$
(2) solve $L^{'T}z_k = u$

We can adapt the back-substitution algorithm described in Section 5.5.3 to solve both of the preceding triangular systems (for the lower-triangular system, the outer loop of the back-substitution algorithm in Program 5.5 must be reversed). In the worst case, this process can be almost completely sequential; that is, it may require the variables of the triangular system to be solved one after the other. Therefore, it is important that the coefficient matrix A, and hence the triangular matrices $L^{'}$ and $L^{'T}$, is ordered so that the solution to $Mz_k = r_k$ in each iteration of the PCG algorithm can be parallelized effectively.

Consider an $n \times n$ matrix A of the form shown in Figure 11.14(b), which results from a red-black ordering of the points in a $\sqrt{n} \times \sqrt{n}$ finite difference grid. The matrices $L^{'}$ and $L^{'T}$ have nonzero elements in the same locations as the nonzero elements in A's lower- and upper-triangular parts, respectively. While solving $L^{'}u = r_k$, first, $u[0], u[1], \ldots, u[(n/2) - 1]$ are computed in parallel. The absence of nondiagonal nonzeros in the upper half of $L^{'}$ permits the values of these variables to be computed in parallel. Next, the values of these variables are substituted in the lower half of the system $L^{'}u = r_k$ and the remaining variables $u[n/2], u[(n/2) + 1], \ldots, u[n - 1]$ are computed in parallel. Similarly, the upper-triangular system $L^{'T}z_k = u$ is solved in two phases—each computing half of the elements of z_k in parallel. Thus, the entire system $Mz_k = r_k$ is solved in four phases.

It is interesting to observe that there is a close similarity between performing a Gauss-Seidel iteration (Equation 11.18) on a matrix A and solving triangular systems with the same sparsity pattern as the corresponding triangular halves of A. If the coefficient matrix is derived from a red-black ordering of the points of a finite difference grid, then just like a Gauss-Seidel iteration, a triangular system is solved in two phases. In general, an ordering scheme that allows parallelization of a Gauss-Seidel iteration also allows parallelization of the solution of $Mz_k = r_k$ in the PCG algorithm.

Assume that the $\sqrt{n} \times \sqrt{n}$ finite difference grid from which the coefficient matrix A is derived is mapped onto a p-processor mesh by using a block-checkerboard partitioning as shown in Figure 11.10. A processor that stores the information related to the i^{th} point in the grid also stores the i^{th} rows of the factors $L^{'}$ and $L^{'T}$ of the preconditioner matrix M. With this mapping of matrix rows onto the processors, the communication and computation times for solving $Mz_k = r_k$ in a PCG iteration are identical to the communication and computation times for performing the step given by Equation 11.18 in a Gauss-Seidel

iteration (Problem 11.18). The other operations (that is, SAXPY, inner products, and matrix-vector multiplication) in an iteration of the parallel PCG algorithm take the same amount of time as in the case of diagonal preconditioning discussed earlier.

A Truncated IC Preconditioner and a Block-Tridiagonal Matrix It is usually observed that natural ordering of grid points leads to faster convergence of the conjugate gradient algorithm than do the red-black or multicolored orderings. In the IC preconditioning technique previously described, the step of solving the system $Mz_k = r_k$ is parallelizable to only a limited extent for block-tridiagonal matrices (Problems 11.13 and 11.18), which arise from the natural ordering of points in a finite difference grid. We now describe another variant of the IC preconditioner that permits a highly parallel solution to the system $Mz_k = r_k$ for a block-tridiagonal matrix of coefficients.

If A is a symmetric positive definite matrix, it can be expressed as

$$A = D + L + L^T,$$

where D is the diagonal matrix consisting of the diagonal entries of A, and L is the strictly lower-triangular matrix consisting of the two lower diagonals of A. The preconditioner matrix M is chosen as

$$M = (I + L\tilde{D}^{-1})\tilde{D}(I + \tilde{D}^{-1}L^T), \tag{11.29}$$

where the diagonal matrix $\tilde{D}$ is chosen such that the principal diagonals of A and M are the same. Hence, the relationship between D and $\tilde{D}$ is as follows:

$$\begin{aligned} D &= \text{diag}(M) \\ &= \text{diag}((I + L\tilde{D}^{-1})\tilde{D}(I + \tilde{D}^{-1}L^T)) \\ &= \text{diag}(\tilde{D} + L^T + L + L\tilde{D}^{-1}L^T) \\ &= \tilde{D} + \text{diag}(L\tilde{D}^{-1}L^T) \end{aligned} \tag{11.30}$$

Since D and L are known from A, the diagonal $\tilde{D}$ can be determined using Equation 11.30 (Problem 11.16). This $\tilde{D}$, substituted in Equation 11.29, determines the preconditioner M. However, the matrix M is not assembled explicitly. Only the triangular matrix $L\tilde{D}^{-1}$ needs to computed.

Let us refer to $L\tilde{D}^{-1}$ by the strictly lower-triangular matrix $-L^{'}$. From Equation 11.29, the matrix M can be expressed as $M = (I - L^{'})\tilde{D}(I - L^{'T})$. Now the system $Mz_k = r_k$ is solved by the following steps:

(1) solve $(I - L^{'})u = r_k$
(2) solve $\tilde{D}v = u$
(3) solve $(I - L^{'T})z_k = v$

Since $L^{'}$ is a strictly lower-triangular $n \times n$ matrix, $L^{'i} = 0$ for $i \geq n$. Therefore, $(I - L^{'})^{-1}$ can be expressed as $(I + L^{'} + L^{'2} + \cdots + L^{'n-1})$. A similar expansion also holds for $L^{'T}$. These series can be truncated to τ powers of $L^{'}$ and $L^{'T}$ because M is a diagonally-dominant matrix, and the contribution of $L^{'\tau}$ and $(L^{'T})^{\tau}$ to $(I - L^{'})^{-1}$ and $(I - L^{'T})^{-1}$, respectively, becomes smaller as τ increases. If $\tilde{L} = (I + L^{'} + \cdots + L^{'\tau})$, then solving the system $Mz_k = r_k$ is equivalent to performing the following matrix-vector multiplications:

$$\begin{array}{ll} (1) & u \approx \tilde{L} r_k \\ (2) & v \approx \tilde{D}^{-1} u \\ (3) & z_k \approx \tilde{L}^T v \end{array}$$

Usually τ = 2, 3, or 4 is chosen, depending on the degree of diagonal dominance of M and the degree of precision desired. We discuss each of these three cases of practical importance separately.

Assume that, in the parallel formulation of this algorithm, the communication overhead due to vector inner-product computations is eliminated by special hardware that provides a fast reduction operation. Hence, communication overhead is incurred only in matrix-vector multiplication on line 11, and in solving the system $Mz_k = r_k$ on line 6 of Program 11.4. This assumption allows us to concentrate on the impact of varying τ on the computation and communication requirements of the algorithm. We assume that the components of the preconditioner (such as $\tilde{D}$ and $\tilde{L}$) are precomputed. Therefore, the rest of the analysis in this section pertains only to the computation and communication that is performed in each iteration in a parallel implementation of the PCG algorithm.

If $\tau = 2$ is used, then $\tilde{L} = I + L^{'} + L^{'2}$. The matrix $\tilde{L}$ has six diagonals—the principal diagonal and diagonals with offsets 1, 2, $\sqrt{n}$, $\sqrt{n}+1$, and $2\sqrt{n}$ in the lower-triangular part. Similarly, $\tilde{L}^T$ has six diagonals in the upper triangular part. Let the time taken by an addition and a multiplication be t_c. Step (2) of the process of solving the system $Mz_k = r_k$ is similar to that for the diagonal preconditioner. Besides this step, the truncated IC preconditioner requires the multiplication of a vector with 12 diagonals—six in step (1), and six in step (3). As a result, in addition to the $t_c n/p$ computation time in each iteration (as for the diagonal preconditioner), each processor spends an extra $12 t_c n/p$ time solving the system $Mz_k = r_k$.

Recall from the discussion of matrix-vector multiplication for the block-tridiagonal matrix (Section 11.1.3) that to multiply a vector with the diagonals at offsets 1 and $\sqrt{n}$ in the upper- and lower-triangular parts of the coefficient matrix, a grid point requires information from all four of its neighboring points. As a result, each processor exchanges the vector elements for its $\sqrt{n/p}$ boundary points with each of its four neighbors. If the vector is multiplied by the diagonals with offsets 1, 2, $\sqrt{n}$, $\sqrt{n}+1$, and $2\sqrt{n}$ in each half of the matrix, then each grid point needs information corresponding to the 12 neighboring points shown in Figure 11.16. Thus, matrix-vector multiplication involving the matrix $\tilde{L}$ requires each processor to exchange the vector elements for two layers of boundary points (that is, $2\sqrt{n/p}$ points) with each of its four neighbors. The total communication time for steps (1) and (3) is thus $4t_s + 8t_w\sqrt{n/p}$. Accounting for the $4t_s + 4t_w\sqrt{n/p}$ communication time for matrix-vector multiplication in line 11 of Program 11.4, the total communication time in each iteration is $8t_s + 12t_w\sqrt{n/p}$. The overall parallel run time of an iteration is given

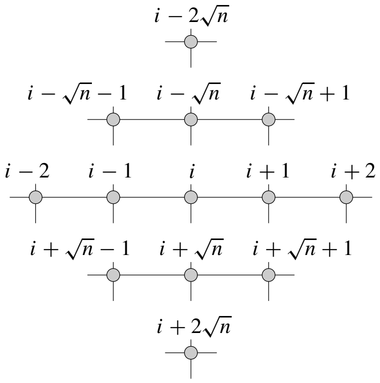


Figure 11.16 The 12 grid points from which point i receives vector elements.

by the following equations:

$$\begin{aligned} T_P &= \overbrace{12 t_c n/p}^{\text{matrix-vector multiplications for solving } Mz_k = r_k} \\ &+ \overbrace{t_c^{'} n/p}^{\text{other computations}} \\ &+ \overbrace{4t_s + 8t_w\sqrt{n/p}}^{\text{communication in solving } Mz_k = r_k} \\ &+ \overbrace{4t_s + 4t_w\sqrt{n/p}}^{\text{communication in matrix-vector multiplication}} \end{aligned}$$

$$T_P = (t_c^{'} + 12t_c)\frac{n}{p} + 8t_s + 12t_w\sqrt{\frac{n}{p}} \tag{11.31}$$

If $\tau = 3$ is chosen, then $\tilde{L} = I + L^{'} + L^{'2} + L^{'3}$, which can be rewritten as $(I + L^{'})(I + L^{'2})$. Here there are two methods to perform steps (1) and (3) of solving $Mz_k = r_k$. The first method constructs the matrices $I + L^{'} + L^{'2} + L^{'3}$ $I + L^{'T} + (L^{'T})^2 + (L^{'T})^3$ explicitly. Each of these matrices contains ten diagonals with offsets 0, 1, 2, 3, $\sqrt{n}$, $\sqrt{n}+1$, $\sqrt{n}+2$, $2\sqrt{n}$, $2\sqrt{n}+1$, and $3\sqrt{n}$. The total computation time for steps (1) and (3) is $20t_c n/p$ for each processor. Extending the case of $\tau = 2$ to $\tau = 3$, each processor exchanges the vector elements corresponding to the three layers of boundary points (that is, $3\sqrt{n/p}$ points) with each of its four neighbors before the two matrix-vector multiplications. This requires a total communication time of $4t_s + 12\sqrt{n/p}$. Thus, the parallel run time of an iteration is given by the following equations:

$$\begin{aligned} T_P &= \overbrace{20t_c n/p}^{\text{matrix-vector multiplications for solving } Mz_k = r_k} \\ &+ \overbrace{t_c' n/p}^{\text{other computations}} \\ &+ \overbrace{4t_s + 12t_w\sqrt{n/p}}^{\text{communication in solving } Mz_k = r_k} \\ &+ \overbrace{4t_s + 4t_w\sqrt{n/p}}^{\text{communication in matrix-vector multiplication}} \end{aligned}$$

$$T_P = (t_c' + 20t_c)\frac{n}{p} + 8t_s + 16t_w\sqrt{\frac{n}{p}} \tag{11.32}$$

The second method for $\tau = 3$ uses the fact that $\tilde{L} = (I + L')(I + L'^2)$. This method performs step (1) in two stages. The first stage involves the multiplication of the vector r with $(I + L'^2)$, which has four diagonals at offsets of 0, 2, $\sqrt{n} + 1$, and $2\sqrt{n}$. The product is then multiplied with $(I + L')$, which has three diagonals with offsets 0, 1, and $\sqrt{n}$. Step (3) is performed similarly. Thus, the total computation time is $(2 \times 4 + 2 \times 3)t_c n/p$ = $14t_c n/p$ per processor, per iteration. This time is less than the computation time for the case in which $\tilde{L} = I + L' + L'^2 + L'^3$ is used in unfactorized form. However, the communication time is now higher than in the previous case. Each processor exchanges vector elements for two layers of boundary points (that is, $2\sqrt{n/p}$ points) with each of its four neighbors. These vector elements are required for matrix-vector multiplication involving $(I + L'^2)$ and $(I + (L'^T)^2)$. In addition to this, each processor needs to exchange vector elements corresponding to its $\sqrt{n/p}$ boundary points with its four neighbors for multiplication with $(I + L')$ and $(I + L'^T)$. Thus, the total communication time (including that of the matrix-vector multiplication in line 11) per iteration is $12t_s + 16t_w\sqrt{n/p}$. The parallel run time for each iteration is as follows:

$$\begin{aligned} T_P &= \overbrace{14t_c n/p}^{\text{matrix-vector multiplications for solving } Mz_k = r_k} \\ &+ \overbrace{t_c' n/p}^{\text{other computations}} \\ &+ \overbrace{8t_s + 12t_w\sqrt{n/p}}^{\text{communication in solving } Mz_k = r_k} \\ &+ \overbrace{4t_s + 4t_w\sqrt{n/p}}^{\text{communication in matrix-vector multiplication}} \end{aligned}$$

$$T_P = (t_c' + 14t_c)\frac{n}{p} + 12t_s + 16t_w\sqrt{\frac{n}{p}} \tag{11.33}$$

A comparison of Equations 11.32 and 11.33 shows that, although the term associated with t_c is smaller in Equation 11.33, the term associated with t_s is smaller in Equation 11.32. Hence, the choice of the method to be used in practice depends on the relative values of the machine-dependent constants t_c and t_s and on the values of n and p. Note that, whenever τ is odd, the series $(I + L' + \cdots + L'^{\tau})$ can be factorized as shown here for $\tau = 3$.

An analysis similar to that for $\tau = 2$ and $\tau = 3$ shows that, for $\tau = 4$, the computation time for solving the system $Mz_k = r_k$ is $30t_c n/p$. The total communication time per iteration is $8t_s + 20t_w\sqrt{n/p}$. For general τ, if $\tilde{L}$ is not factorized, the number of diagonals in $\tilde{L}$ is $(\tau + 1)(\tau + 2)/2$. These diagonals are distributed in $\tau + 1$ clusters at distances of $\sqrt{n}$ from each other. The first cluster, which includes the principal diagonal, has $\tau + 1$ diagonals, and then the number of diagonals in each cluster decreases by one. The last cluster has only one diagonal at a distance of $\tau\sqrt{n}$ from the principal diagonal. For $\sqrt{n/p} > \tau$, solving the system $Mz_k = r_k$ requires each processor to exchange vector elements corresponding to τ layers of boundary points (that is, $\tau\sqrt{n/p}$ points) with each of its four neighboring processors. The expression for parallel run time for the general case is given by the following equations:

$$\begin{aligned} T_P &= \overbrace{(\tau + 1)(\tau + 2)t_c n/p}^{\text{matrix-vector multiplications for solving } Mz_k = r_k} \\ &+ \overbrace{t_c' n/p}^{\text{other computations}} \\ &+ \overbrace{4t_s + 4\tau t_w\sqrt{n/p}}^{\text{communication in solving } Mz_k = r_k} \\ &+ \overbrace{4t_s + 4t_w\sqrt{n/p}}^{\text{communication in matrix-vector multiplication}} \end{aligned}$$

$$T_P = (t_c' + (\tau^2 + 3\tau + 2)t_c)\frac{n}{p} + 8t_s + 4(\tau + 1)t_w\sqrt{\frac{n}{p}} \tag{11.34}$$

Disregarding the communication in computing the inner products, a comparison of Equations 11.26 and 11.34 shows that the use of a truncated IC preconditioner involves more computation per iteration of the PCG algorithm over a simple diagonal preconditioner. On the other hand, an IC preconditioner significantly reduces the number of iterations required to achieve a given level of precision over a diagonal preconditioner. The truncated IC preconditioning also results in an efficiency higher than that in the case of diagonal preconditioning (Problem 11.19), assuming that the efficiency of a parallel implementation is computed with respect to an identical algorithm running on a single processor. The overall performance of the PCG algorithm is governed by the amount of computation per iteration, the number of iterations, and the efficiency of the parallel implementation. Therefore, IC preconditioning may yield a better overall parallel run time by virtue of a better efficiency than diagonal preconditioning, even if the latter is faster in a serial implementation. A comparison of truncated IC preconditioners with different values of

τ presents similar tradeoffs. Equation 11.34 shows that the amount of computation in each iteration is proportional to τ^2, and the volume of communication in each iteration is proportional to τ. Therefore, for the same values of n and p, both parallel run time and efficiency increase as τ increases. For a given problem, different values of τ may lead to optimal implementations for different values of p; a higher p favors a higher τ (Problem 11.19).

A Diagonal Preconditioner and a Banded Unstructured Sparse Matrix Consider a symmetric positive definite matrix of coefficients in which the nonzero elements are uniformly distributed in a band of width w along the principal diagonal. If there is an average of m nonzero elements per row, then as shown in Section 11.1.3, the parallel run time for matrix-vector multiplication is $t_c mn/p + t_s wp/n + t_w w$ (Equation 11.12). When we disregard the communication cost of vector inner product computation, the parallel run time of an iteration of the PCG algorithm using the diagonal preconditioner is

$$T_P = (t_c^{'} + t_c m)\frac{n}{p} + t_s \frac{wp}{n} + t_w w. \tag{11.35}$$

A Truncated IC Preconditioner and a Banded Unstructured Sparse Matrix Consider the use of the IC preconditioner for banded sparse matrices. The preconditioner matrix M is of the form $(I - L^{'})\tilde{D}(I - L^{'T})$, where $\tilde{D}$ is a diagonal matrix and $L^{'}$ is a strictly lower-triangular sparse matrix whose nonzero elements are located in exactly the same positions as in the lower-triangular part of the coefficient matrix A. The system $Mz_k = r_k$ is solved in the same manner as in the case of the block-tridiagonal matrix discussed earlier. Consider the general case in which $\tilde{L} = (I + L^{'} + \cdots + L^{'\tau})$ is used as an approximation of $(I - L^{'})^{-1}$. If $L^{'}$ has a bandwidth of $w/2$, then the bandwidth of $L^{'\tau}$ has an upper bound of $\tau w/2$ (Problem 11.20). As a result, $\tilde{L}$ also has a bandwidth of less than $\tau w/2$. The same holds true for $\tilde{L}^T$ as well. According to Equation 11.12, the total communication time to multiply both $\tilde{L}$ and $\tilde{L}^T$ with the vectors is, at most, $t_s \tau wp/n + t_w \tau w$.

The matrices $L^{'}$ and $L^{'T}$ consist of bands of width approximately $w/2$ along the principal diagonal in the lower and the upper-triangular halves, respectively. Since the sparsity pattern of $L^{'}$ and $L^{'T}$ in their respective halves is identical to that of A, they have an average of approximately $m/2$ nonzero elements per row (A has an average of m nonzero elements per row). If A is large and $m \ll w$, then the average number of nonzero elements per row in $L^{'\tau}$ and $(L^{'T})^{\tau}$ is $(m/2)^{\tau}$ (Problem 11.21). Since $1 + a + a^2 + \cdots + a^{\tau} < a^{\tau+1}$ for $a > 1$, the number of nonzero elements in $\tilde{L}$ and $\tilde{L}^T$ each has upper bound of $(m/2)^{\tau+1}$. Hence, the computation time per processor for solving the system $Mz_k = r_k$ on a p-processor ensemble is $2t_c(n/p)(m/2)^{\tau+1}$ in each iteration. The total time spent in solving $Mz_k = r_k$ in each iteration is $2t_c(n/p)(m/2)^{\tau+1} + t_s \tau wp/n + t_w \tau w$.

From Equation 11.12, the time required to perform matrix-vector multiplication is $t_c mn/p + t_s wp/n + t_w w$. Thus, the overall parallel run time per iteration of the PCG algorithm with a banded unstructured sparse matrix and truncated IC preconditioner is

$$T_P = \left(t_c^{'} + t_c(2(\frac{m}{2})^{\tau+1} + m)\right)\frac{n}{p} + t_s(\tau + 1)\frac{wp}{n} + t_w(\tau + 1)w. \tag{11.36}$$

11.3 Finite Element Method

The ***finite element method*** (FEM) is an active application area of massively parallel computing. FEM is a computational tool for deriving approximate numerical solutions to partial differential equations over a discretized domain.

To introduce the finite element method for solving differential equations, we use the simple example of modeling the steady-state temperature at various points on a metal sheet. As Figure 11.17 shows, the domain is a two-dimensional rectangular sheet of metal on which a regular grid of square elements is imposed. The domain is bounded by the coordinates (0, 0), (1, 0), (1, 1), and (0, 1). As the figure shows, the grid divides the domain into small areas called ***elements***. There are 48 elements in the discretized domain shown in Figure 11.17. The internal nodes or grid points at which the coefficients must be determined are labeled 0 through 48. Unlike the finite difference grid shown in Figure 11.8, a grid point exchanges information with all the other grid points with which it shares an element. Hence, each point has nine neighbors (including itself), and each row of the resulting sparse matrix of coefficients has nine nonzero entries.

The steady-state temperature u at any point (X, Y) on the metal sheet is governed by the Laplace equation

$$\frac{\delta^2 u}{\delta X^2} + \frac{\delta^2 u}{\delta Y^2} = 0. \tag{11.37}$$

The values of the physical quantity being modeled (in this case, temperature) at the boundary of the physical domain are governed by what are referred to as ***boundary conditions***. Since the sheet is insulated at the top and bottom, the following boundary conditions result:

$$\frac{\delta u}{\delta Y} = 0, \quad Y = 0, \quad 0 \le X \le 1 \tag{11.38}$$

$$\frac{\delta u}{\delta Y} = 0, \quad Y = 1, \quad 0 \le X \le 1 \tag{11.39}$$

Furthermore, assume that the temperatures at the other two ends of the sheet are U_0 and U_1, respectively. The corresponding boundary conditions are as follows:

$$u = U_0, \quad X = 0, \quad 0 \le Y \le 1 \tag{11.40}$$

$$u = U_1, \quad X = 1, \quad 0 \le Y \le 1 \tag{11.41}$$

The boundary conditions involving derivatives of the solution, such as those given by Equations 11.38 and 11.39, are referred to as ***Neumann boundary conditions***. Boundary

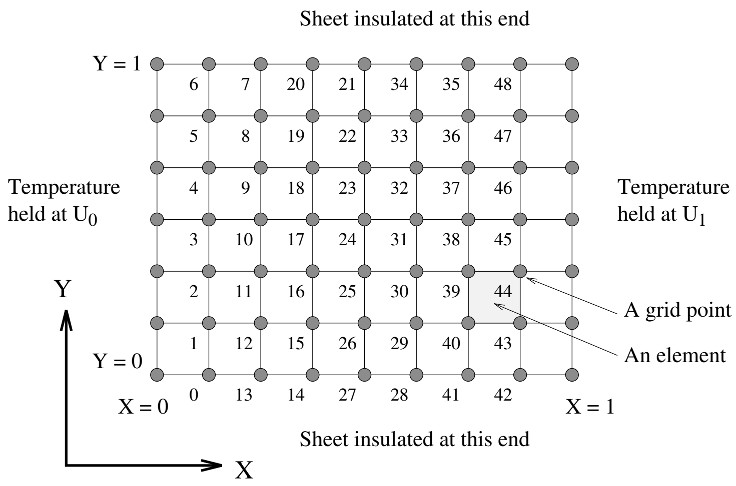


Figure 11.17 A grid dividing a sheet of metal into a finite element mesh of 48 elements. The internal nodes or grid points at which the coefficients must be determined are labeled 0 through 48. The nodes at the periphery are not labeled because the temperature at these points is determined by boundary conditions.

conditions involving only the solution, such as those given by Equations 11.40 and 11.41, are known as ***Dirichlet boundary conditions***.

The temperature on the surface of the sheet is a continuous function of X and Y governed by Equation 11.37. The FEM derives an approximation of this function by dividing the domain into elements. The value of the function is typically expressed as a simple polynomial that is a linear combination of a set of functions of X and Y called ***basis functions***. The coefficients of the basis functions at each node are derived from a system of linear equations. This system arises from the minimization of error between the approximate and exact solutions of the partial differential equation. Thus, the FEM transforms the Laplace equation into a set of linear equations of the form $Ax = b$. In the context of FEM, the coefficient matrix A is called the ***stiffness matrix*** and b the ***force vector***. This system is solved for the vector x, which gives the value of the coefficients of the basis functions at different node points in the discretized domain. The stiffness matrix A for the FEM can be derived by computing a set of definite integrals over the elements of the finite element graph. If nodes i and j in the finite element mesh share elements, then $A[i, j]$ is given by the summation of the integrals calculated over all the elements shared by points i and j. Thus, the only nonzero entries in the matrix are $A[i, j]$ such that grid points i and j share an element.

The following are some important properties of the stiffness matrix and the force vector:

(1) For most applications, the finite element graph is not a regular structure as shown in Figure 11.17, but is highly irregular. Thus, the stiffness matrix is usually an unstructured sparse matrix.
(2) Computing the stiffness matrix and force vector is relatively inexpensive compared to the overall solution of the linear system. Furthermore, computing individual entries of A requires only computations local to the element. Consequently, this computation is trivial to parallelize.
(3) The resulting system of linear equations is large and sparse, and hence solving it is the most computationally expensive phase of the FEM. It is this phase for which efficient parallel solutions are critical.

Both iterative and direct solvers are used in finite element computations. Iterative solvers are often less expensive in terms of memory and time. However, in some cases, iterative solvers are not guaranteed to converge to a solution, necessitating direct solvers. In this section we assume that an iterative method like the unpreconditioned conjugate gradient method (Program 11.3) is used, which performs SAXPY operations, vector inner-product computations, and a matrix-vector multiplication in each iteration.

The SAXPY operations do not involve any communication overhead. The communication time per iteration for vector inner products depends only on the number of processors in use. This time is $\Theta(\log p)$ on a hypercube and $\Theta(\sqrt{p})$ on a mesh. In the presence of a fast, hardware-supported reduction operation, we can assume that it is a small constant. The communication requirements of matrix-vector multiplication are critically dependent on the spatial decomposition of the domain and the assignment of its partitions to the processors. If the domain is partitioned among processors, then information corresponding to the elements at the partition boundaries is exchanged among neighboring processors during matrix-vector multiplication (Figure 11.12). In each iteration, the time spent in computation by a processor is proportional to the number of elements assigned to it. If a partition shares boundaries with α other partitions and has β boundary elements, the per-iteration communication time of the processor holding this partition is proportional to $\alpha t_s + \beta t_w$.

The principal issues in efficient parallel implementations of FEM are minimizing load imbalance among processors and maximizing the ratio of computation to communication on the processors. The former is achieved by assigning a nearly equal number of elements to each partition. The latter requires that the number of elements along a partition boundary be small compared to the total number of elements within the partition.

In the remainder of this section, we discuss some commonly used techniques for partitioning the finite element domain among processors. Although we chose examples with quadrilateral elements, triangular elements are also commonly used.

Partitioning Methods for Finite Element Graphs

The communication pattern, and hence the overall efficiency, of a parallel implementation of an FEM computation is a function of the partitioning of the domain among processors.

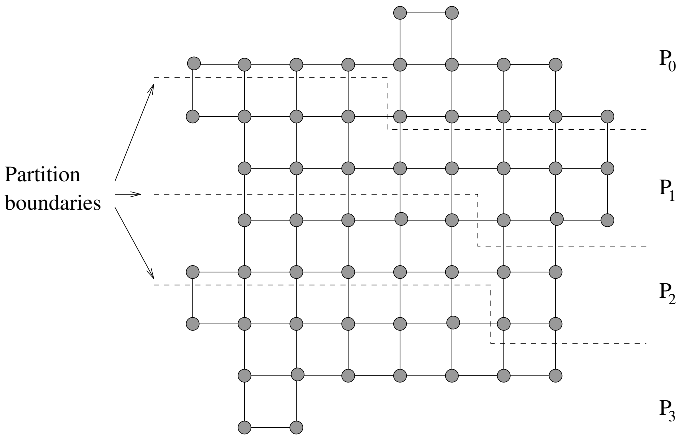


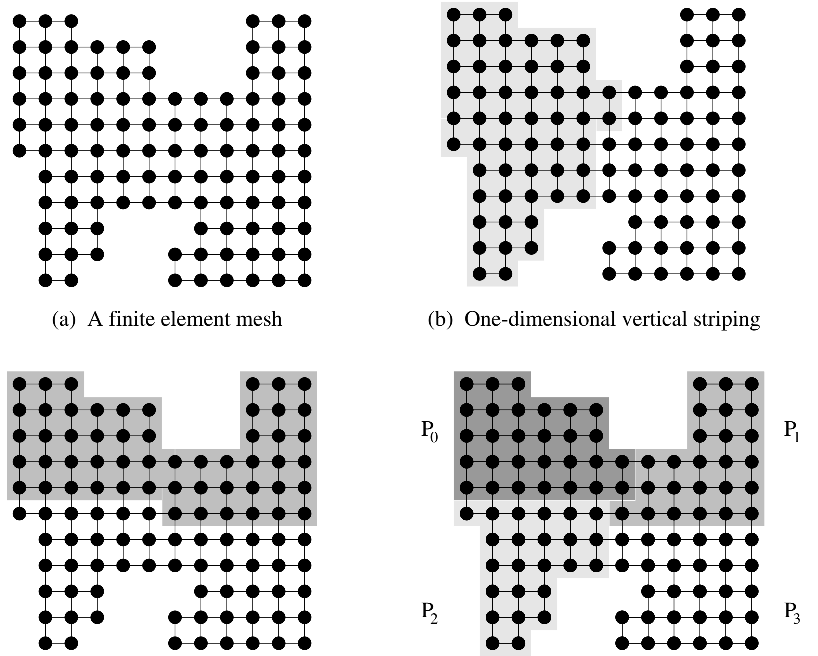
Figure 11.18 One-dimensional striped partitioning of a finite element mesh among four processors.

Deriving an optimal partitioning that balances the load and minimizes communication and idling costs is an NP-hard problem. Therefore, several heuristic schemes have been devised to derive reasonable partitions for finite element meshes in polynomial time.

One-Dimensional Striped Partitioning for Mesh Graphs A finite element graph is called a ***mesh graph*** or a finite element mesh if it is composed of quadrilateral elements and it can be embedded into a uniform two-dimensional grid such that each element boundary maps onto exactly one edge in the grid. For example, the finite element graph shown in Figure 11.18 is a mesh graph. A one-dimensional ***striped partitioning*** divides a finite element mesh into p stripes such that each stripe runs the length (or width) of the mesh. If the mesh has n nodes, striped partitioning assigns either $\lceil n/p \rceil$ or $\lfloor n/p \rfloor$ nodes to each processor. Figure 11.18 illustrates a one-dimensional striped partitioning of a finite element mesh among four processors.

One-dimensional striped partitioning generates stripes that adjoin only one stripe on either side. Consequently, even on weak architectures such as a linear array, the partitioning yields a nearest-neighbor mapping. To enforce the nearest neighbor constraint, a sufficient condition is that n/p (the number of elements assigned to a processor) is greater than the smaller dimension of the mesh. Striped partitioning affords good load balance and locality of communication. However, it may communicate large amounts of data among processors.

Two-Dimensional Striped Partitioning for Mesh Graphs The two-dimensional striped partitioning scheme maps a finite element mesh onto a two-dimensional mesh of processors. Figure 11.19 illustrates this process for a 2×2 mesh of processors. As shown in the figure, the two-dimensional striped partitioning uses two orthogonal one-dimensional



(a) A finite element mesh

(b) One-dimensional vertical striping

(c) One-dimensional horizontal striping

(d) Overlapping two one-dimensional partitions

Figure 11.19 Two-dimensional striping of a mesh graph for a 2×2 mesh of processors.

striped partitions. First, the finite element graph is partitioned into vertical stripes (Figure 11.19(b)). Then the graph is striped horizontally (Figure 11.19(c)). Finally, the two partitionings are overlapped (Figure 11.19(d)), and the resulting partitions (whose number is equal to the product of vertical and horizontal partitions) are assigned to processors.

As Figure 11.19 illustrates, two-dimensional striping may not yield partitions with identical numbers of nodes. Consequently, the partitioning phase must be followed by another phase that balances the load between partitions. This load-balancing phase is called ***boundary refinement***. Boundary refinement balances the load by transferring nodes from heavily loaded to lightly loaded processors. At the same time, it maintains the nearest-neighbor relationships between partitions.

Assume that the total number of nodes (grid points) in the finite element mesh is n and that the number of nodes assigned to processor i in the initial partitioning is n_i. Also assume that there are p processors and that n is divisible by p. Boundary refinement uses a $p \times p$ ***load transfer matrix***. Entry (i, j) of this matrix specifies the number of nodes that must be transferred from the partition assigned to processor i to that assigned to processor j. For a perfect load balance, the net change in the load (the number of nodes) of processor

i should be $n_i - n/p$ (this can be positive or negative). This load is transferred from one of the four neighbors of processor i, which are referred to by $u(i)$ (up), $d(i)$ (down), $l(i)$ (left), and $r(i)$ (right). We ignore minor boundary overlaps between diagonally located processors (for example, P_1 and P_2 in Figure 11.19). Let U_i, D_i, L_i, and R_i represent the number of elements transferred to processor i from processors $u(i)$, $d(i)$, $l(i)$, and $r(i)$, respectively (these numbers can be negative). The entries in the load transfer matrix must satisfy the following conditions:

$$\begin{aligned} L_i + R_i + D_i + U_i &= n_i - n/p \\ L_i &= -R_{l(i)} \\ U_i &= -D_{u(i)} \end{aligned}$$

In addition to these, there are conditions corresponding to the partitions that lie on the boundary of the domain. The solution of the system of equations arising from the complete set of conditions yields the load transfer matrix. Note that the number of variables in this system of equations is $\Theta(p)$, whereas the number of variables in the original system of equations being solved by the FEM is n. In practice, n is much greater than p.

Having constructed the load transfer matrix, the load transfer (or boundary refinement) procedure proceeds iteratively. In each step, nodes on the boundary of partitions between two processors are identified and placed in a queue. The required number of nodes are transferred from this queue. If the number of nodes that need to be transferred is more than the number of nodes in the queue, the process is repeated.

Striped Partitioning Schemes for Generalized Graphs In the preceding sections, we studied striped partitioning for mesh graphs. A finite element graph that is not a mesh graph is referred to as a generalized graph. The elements of a ***generalized graph*** may not be squares or rectangles of a uniform size. Figure 11.20 shows a generalized finite element graph. The striping process described for mesh graphs does not extend naturally to generalized graphs, in which the elements are not organized into explicit rows and columns. For a generalized graph, a process called ***levelization*** organizes the graph into stripes.

Levelization begins by identifying a peripheral node or a set of connected peripheral nodes. A peripheral node is characterized by a peripheral edge, which belongs to a single element, unlike a non-peripheral edge, which belongs to two elements. Connected peripheral nodes on one boundary of the domain are assigned the label 1. All unlabeled nodes that share an element with a node labeled 1 are labeled 2. This process continues, and all unlabeled nodes sharing an element with a node labeled l are labeled $l+1$. Figure 11.20 illustrates a one-dimensional levelization of a finite element graph.

Striped partitioning of generalized graphs counts off nodes at a given level and proceeds to the next higher level when all the nodes at a level are exhausted. Let n be the total number of nodes and p be the number of processors. Let m be the total number of levels (labeled from 1 to m) and r_i be the sum of number of nodes in the two contiguous levels i and $i+1$. If $r = \max\{r_1, r_2, \ldots, r_{m-1}\}$, then a sufficient condition to ensure nearest-neighbor communication is that $n/p > r$.

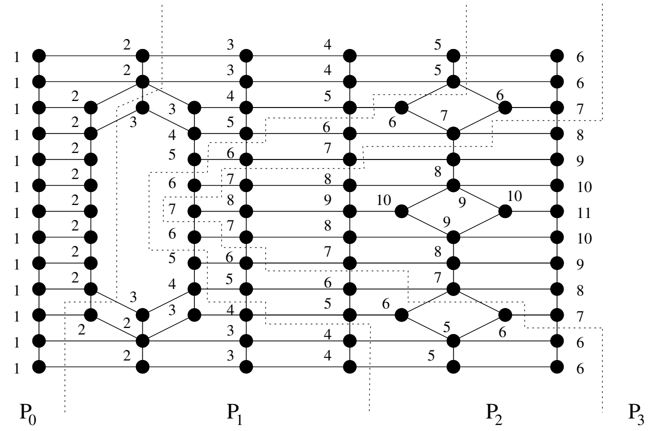


Figure 11.20 The levelization process for a generalized finite element graph. The levels partition the graph into four one-dimensional stripes that are assigned to four processors.

Two-dimensional striping partitions the graph similarly. Again, nodes are counted by levels, rather than along dimensions as in the case of mesh graphs.

Scattered Decomposition ***Scattered decomposition*** (also referred to as ***modular mapping***) is an extensively applied technique for decomposing highly irregular domains. This method balances the load by partitioning the domain into a large number r of rectangular clusters such that $r \gg p$. Each processor handles a disjoint set of r/p such clusters. For irregular problems, increasing r (and consequently decreasing the area of each partition) yields a better load balance. However, load balance is achieved at the cost of increased communication overhead between partitions that are not on the same processor.

Recursive Bisection Techniques ***Recursive bisection*** techniques partition the domain by recursively subdividing it into two parts at each step. For $p = 2^k$ processors, recursively subdividing the domain k times yields p partitions. Bisection techniques differ in the manner in which the domain is subdivided. These techniques are based on the assumption that it is possible to derive near-optimal partitions by subdividing the domain into two parts, maintaining optimality at each step.

In this section, we discuss three recursive bisection techniques based on different criteria for subdividing the domain.

(1) **Recursive Coordinate Bisection:** ***Recursive coordinate bisection*** is the most intuitive of the recursive bisection techniques. The domain is subdivided at each step, based on the physical coordinates of the nodes. Consider a finite element graph with the set of nodes $\{v_0, v_1, \ldots, v_{n-1}\}$. Assume that the spatial coordinates

of the nodes are available. Recursive coordinate bisection subdivides the domain into two parts along the longer dimension (assume that this is along the X axis). Nodes in the set $\{v_0, v_1, \ldots, v_{n-1}\}$ are sorted by their X coordinate. The first half of the sorted list of nodes is assigned to the first subdivision and the other half to the second subdivision. Each subdivision is then recursively divided, and the process continues until it generates p partitions.

This algorithm has two drawbacks. First, the partitions often have many edges of the grid that cross partition boundaries. Second, the partitions may be disconnected. Both properties are highly undesirable. They occur because recursive coordinate bisection uses no connectivity information. However, some recent coordinate bisection schemes do overcome these limitations (Section 11.6).

(2) **Recursive Graph Bisection:** Since recursive coordinate bisection ignores connectivity information, it is unable to minimize the number of grid edges crossing partition boundaries. ***Recursive graph bisection*** remedies this problem by using graph distance rather than coordinate distance to partition the domain.

Let $d_{i,j}$ be the number of edges on the shortest path from node v_i to node v_j. The recursive graph bisection technique first determines the two nodes in the graph that are farthest in terms of graph distance. Two nodes v_i and v_j are farthest if $d_{i,j} \geq d_{p,q}$ for every pair of nodes (v_p, v_q) in the graph. These nodes are called the ***extremities*** of the graph. All the other nodes are organized according to their distance from nodes v_i and v_j. A node is assigned to the partition containing the closer extremity.

It is computationally expensive to find the exact extremities of a graph. However, algorithms are available that yield good approximations of the extremities. An algorithm known as the ***reverse Cuthill-McKee*** algorithm is commonly used for this purpose. This algorithm determines the approximate extremities and uses one of them as the root for establishing a level structure. This process is identical to levelization for striped partitioning. Vertices are counted off as they are organized into the level structure, and the partitioning is complete when half of the nodes have been assigned. Note that this partitioning strategy ensures that at least one of the two partitions is connected.

(3) **Recursive Spectral Bisection:** Given a graph G, ***recursive spectral bisection*** uses the discrete ***Laplacian*** L_G of the graph to divide the domain into two parts. The matrix L_G is equal to $A - D$, where A is the adjacency matrix of graph G and D is a diagonal matrix in which element $D[i,i]$ is the degree $g(v_i)$ of node v_i. Therefore,

$$L_G[i,j] = \begin{cases} -g(v_i) & \text{if } i = j, \\ 1 & \text{if edge } (v_i, v_j) \text{ exists in the graph,} \\ 0 & \text{otherwise.} \end{cases} \tag{11.42}$$

The discrete Laplacian L_G is a negative semidefinite matrix. Furthermore, its largest eigenvalue is zero and the corresponding eigenvector consists of all ones.

Assuming that the graph is connected, the magnitude of the second largest eigenvalue gives a measure of the connectivity of the graph. The eigenvector corresponding to this eigenvalue, when associated with the nodes, gives a measure of the distances between the nodes. Consequently, it can be used to divide the domain into two parts. This vector is referred to as the ***Fiedler vector***. The partitioning is done by sorting nodes according to their weights in the eigenvector and dividing the sorted list of nodes into two equal parts.

The computationally intensive part of this algorithm is the computation of the Fiedler vector. One common algorithm used for computing this is the ***Lanczos algorithm***. The details of this algorithm are beyond the scope of this book. Readers are referred to Section 11.6 for related bibliographic remarks.

We have seen that recursive coordinate bisection may create partitions with shapes that have poor communication properties. Recursive graph partitioning yields more compact partitions but they may be disconnected. Recursive spectral bisection, on the other hand, yields connected partitions that are well balanced. Although a comprehensive performance analysis does not exist for these schemes, some experimental results tend to favor recursive spectral bisection over the other two schemes on a variety of problems.

11.4 Direct Methods for Sparse Linear Systems

Despite their high computational cost, direct methods are useful for solving sparse linear systems because they are general and robust. Although there is substantial parallelism inherent in sparse direct methods, only limited success has been achieved to date in developing efficient general-purpose parallel formulations for them. The reasons for this are twofold. First, the amount of computation relative to the size of the system to be solved is very small. For example, Gaussian elimination involving a dense $n \times n$ matrix has a sequential time complexity of $\Theta(n^3)$. In contrast to a dense matrix, consider the block-tridiagonal matrix of coefficients arising from a natural ordering of points on a $\sqrt{n} \times \sqrt{n}$ regular grid (Figure 11.7). Since the outermost diagonals are at a distance of $\sqrt{n}$ rows or columns from the principal diagonal, the two inner loops of Gaussian elimination are executed only $\sqrt{n}$ times, resulting in a sequential complexity of $\Theta(n^2)$. If the nested-dissection ordering described in Section 11.4.1 is used instead of a natural ordering, this complexity can be further reduced to $\Theta(n^{3/2})$. Since there are few computations in the overall problem, poor efficiencies result because even a modest amount of communication can create a serious imbalance in the relative amounts of time that processors spend in communication and computation.

The second reason for the inefficiency of parallel sparse direct solvers is that most attempts made to date to implement sparse direct methods on parallel computers are based on good serial formulations. The goals of a serial formulation, such as minimizing memory use and operation count, may be inappropriate in a parallel setting. Besides, these goals may seriously conflict with the goals of a parallel formulation, such as maximizing the number of independent tasks, minimizing communication, and balancing load among processors.

Developing efficient general-purpose parallel formulations of direct methods for unstructured or random sparse matrices is currently an active area of research. Although all of these methods are based on Gaussian elimination (for general matrices) and Cholesky factorization (for symmetric positive definite matrices), their parallel formulations can be quite complicated. In this section we only outline some general techniques used in parallel sparse direct solvers. We assume row-oriented Gaussian elimination (Program 5.4) and row-oriented Cholesky factorization (Program 5.6) as the base algorithms to adapt for sparse linear systems. Parallel implementations with column-oriented versions are very similar, and may be numerically superior for the reasons discussed in Section 5.5.4.

The process of obtaining a direct solution to a general sparse system of linear equations of the form $Ax = b$ consists of four distinct phases: ordering, symbolic factorization, numerical factorization, and solving a triangular system. In the following subsections, we discuss each of these phases.

11.4.1 Ordering

Ordering is an important phase of solving a sparse linear system because it determines the overall efficiency of the remaining steps. The aim of ***ordering*** is to generate a permutation of the original coefficient matrix so that the permuted matrix leads to a faster and more stable solution. The numerical stability of the solution is increased by ensuring that the diagonal elements or pivots are large compared to the remaining elements of their respective rows. The ordering criteria for obtaining a faster parallel solution are more complex.

During factorization, when a row of a sparse matrix A is subtracted from another row, some of the zeros in the latter row may become nonzero. When the k^{th} row is the pivot, then a zero in position $A[i, j]$ becomes nonzero for all $i, j > k$ such that $A[i, k] \neq 0$ and $A[k, j] \neq 0$. In this case, the nonzero element at $A[i, j]$ is said to be generated as a result of ***fill-in***. For example, consider the sparse matrix shown in Figure 11.21(a), in which the nonzero elements are denoted by the symbol $\times$. In the first step of factorization, a multiple of row 0 is subtracted from rows 1, 4, and 7. This step changes the zeros at locations $A[1, 4]$, $A[1, 7]$, $A[4, 1]$, and $A[7, 1]$ to nonzero values. Each such fill-in is denoted by the symbol $\square$. Now a multiple of row 1 is subtracted from rows 2, 4, and 7, introducing nonzero elements in positions $A[2, 4]$, $A[2, 7]$, $A[4, 2]$, and $A[7, 2]$. Figure 11.21(b) shows all the fill-in resulting from the complete factorization of the matrix.

It is possible to reorder the rows and columns of the matrix shown in Figure 11.21(a) so that factorization does not generate any nonzero elements in the positions occupied by zeros in the unfactorized matrix. Figure 11.21(c) shows one such permutation of this matrix. In this figure, the label of each row is of the form i (j). The label denotes that the i^{th} row and column of the reordered matrix correspond to the j^{th} row and column of the original matrix. Factorization of the reordered matrix does not result in any fill-in. In general, reordering may not completely eliminate fill-in, but in most cases it can significantly reduce it.

	0	1	2	3	4	5	6	7
0	X	X			X			X
1	X	X	X					
2		X	X	X				
3			X	X				
4	X				X	X		X
5					X	X	X	
6						X	X	
7	X				X			X

(a) A random sparse matrix

	0	1	2	3	4	5	6	7
0	X	X			X			X
1	X	X	X		□			□
2		X	X	X	□			□
3			X	X	□			□
4	X	□	□	□	X	X		X
5					X	X	X	□
6						X	X	□
7	X	□	□	□	X	□	□	X

(b) Fill-in on factorization

	(3)	(6)	(5)	(4)	(7)	(0)	(1)	(2)
0 (3)	X							X
1 (6)		X	X					
2 (5)		X	X	X				
3 (4)			X	X	X	X		
4 (7)				X	X	X		
5 (0)				X	X	X	X	
6 (1)						X	X	X
7 (2)	X						X	X

(c) A no-fill-in reordering

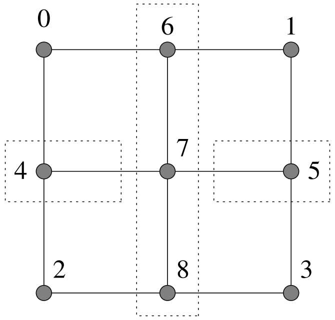
Figure 11.21 Fill-in during the factorization of a sparse matrix and its reordering to eliminate fill-in. The nonzero elements in the original sparse matrix are denoted by $\times$, and the nonzero elements introduced due to fill-in are denoted by $\square$. In part (c), the old row and column numbers before reordering are shown in parentheses.

In addition to providing numerical stability and reducing fill-in, another goal of ordering in a parallel sparse direct solver is to increase the number of independent tasks. For example, for the matrix shown in Figure 11.21(c), a multiple of row 0 can be subtracted from row 7 in parallel with the subtraction of a multiple of row 1 from row 2. In other words, both rows 0 and 1 can be used as pivots simultaneously. This kind of parallelism in Gaussian elimination is available only for sparse matrices. During the factorization of the original matrix in Figure 11.21(a), all pivots from 0 to 7 must be used sequentially. Thus, reordering the coefficient matrix can not only reduce the fill-in, but can also increase the parallelism in the factorization process.

Ordering the rows and columns of a matrix to minimize fill-in is a very expensive combinatorial problem with an exponential complexity. Therefore, heuristics are used for ordering. For a given sparse matrix, the best heuristic for reducing fill-in is not necessarily the one that results in maximum parallelism or minimum interprocessor communication. On current and future generation of message-passing parallel computers, the availability of memory may not be a major concern. Hence, reducing fill-in to restrict memory use is relatively less important in the parallel context.

In addition to increasing the memory requirement, fill-in also increases computation. However, even some increase in computation can be compensated, provided that the ordering scheme increases the number of parallel tasks and/or lowers communication during the factorization phase. This is because two important causes of inefficiency of parallel sparse direct methods are a low computation-to-communication ratio and insufficient parallelism. Hence, a parallel sparse direct solver may benefit from an ordering that reduces communication and increases parallelism, even at the cost of increasing the fill-in to some extent.

Although several good heuristics for ordering are known, in this section we describe two schemes that show promise for adaptation to parallel sparse direct solvers.



(a) A 3 $\times$ 3 finite difference grid with nested dissection numbering

	0	1	2	3	4	5	6	7	8
0	X				X		X		
1		X				X	X		
2			X		X				X
3				X		X			X
4	X		X		X		□	X	□
5		X		X		X	□	X	□
6	X	X			□	□	X	X	□
7					X	X	X	X	X
8			X	X	□	□	□	X	X

(b) The resulting sparse matrix with fill-in on factorization

Figure 11.22 The nested-dissection ordering of a 3×3 finite difference grid.

Nested-Dissection Ordering

Nested-dissection ordering is best explained in terms of finite difference grids, as shown in Figure 11.22. Part (a) of this figure shows a small square grid of points. To obtain a nested-dissection ordering of the coefficient matrix resulting from this grid, we assign numbers to the grid points by following the nested-dissection algorithm. First, a set of points is chosen whose removal divides the domain into two disconnected subdomains of equal (or almost equal) size. The points that are chosen are numbered after all the points in both subdomains have been numbered. The points in the subdomains are recursively numbered by using the same strategy. After all grid points are numbered, the ordered matrix of coefficients is obtained by making the i^{th} point correspond to the i^{th} variable and the i^{th} equation. The sparse matrix resulting from the nested-dissection ordering of the points in the grid of Figure 11.22(a) is shown in Figure 11.22(b).

Nested dissection can even be used for irregular grids and for sparse matrices that do not arise from finite element problems. An n-node graph can be constructed from the $n \times n$ matrix of coefficients A by placing an edge between points i and j if and only if $A[i, j] \neq 0$ or $A[j, i] \neq 0$, as shown in Figure 11.12. The nodes of the graph are renumbered as previously described. The matrix is then reordered according to the new numbering, subject to numerical stability.

Minimum-Degree Ordering

After k steps of Gaussian elimination, let c_i be the number of nonzero elements in the i^{th} column of the $(n - k) \times (n - k)$ active matrix, and let r_i be the number of nonzero elements in its i^{th} row. We define a cost function $C(i)$ as the product $(c_i - 1) \times (r_i - 1)$. In ***minimum-degree ordering***, The i^{th} column is chosen as the pivot in the $(k + 1)^{\text{st}}$ iteration (that is, the i^{th} row and column become the $(k + 1)^{\text{st}}$ row and column) if $C(i)$ is minimum and $A[i, i]$ satisfies some numerical stability criterion.

The sparsity pattern of the matrix resulting from minimum-degree ordering is sensitive to tie-breaking between candidate pivots whose cost is the same. For example, the matrix shown in Figure 11.22(b) happens to satisfy the minimum-degree criterion, although other minimum-degree permutations are possible. Different permutations of a matrix satisfying the minimum-degree criterion may result in different degrees of fill-in and parallelism during the factorization phase (Problems 11.23 and 11.29).

11.4.2 Symbolic Factorization

The symbolic factorization phase determines the structure of the triangular matrices that would result from factorizing the ordered coefficient matrix. ***Symbolic factorization*** sets up the data structures for storing the resulting matrices, and allocates an appropriate amount of memory for these data structures. The information on the structure of the factors is also used by the algorithms for numerical factorization, which is the next phase of solving the system.

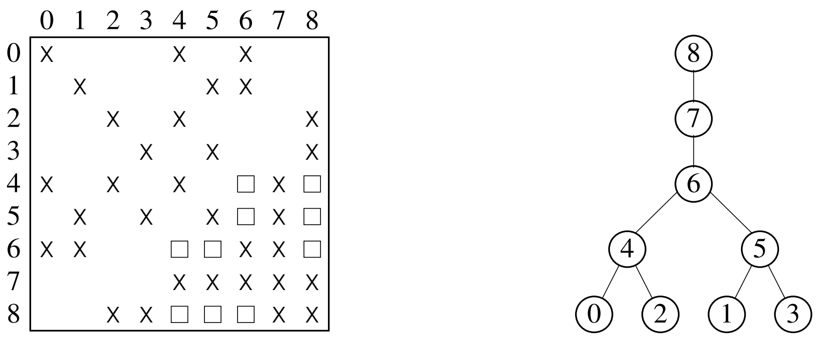
Symbolic factorization is quite complicated if numerical pivoting is required. In such cases, it is usually merged with the next phase, which is numerical factorization. If numerical pivoting is not required, then determining the sparsity pattern of the factors is straightforward. In this case, symbolic factorization simply determines the fill-in caused by the elimination of each row of the matrix in sequence. When the k^{th} row of the coefficient matrix A is used as the pivot, a fill-in is created for every $A[i, j] \neq 0$ if $i, j > k$, $A[i, k] \neq 0$, and $A[k, j] \neq 0$.

For performing symbolic factorization on matrices that do not require numerical pivoting, serial algorithms are available whose run time is proportional to the number of nonzero elements in the matrix (Section 11.6). Due to the availability of very fast serial algorithms, and the high data-distribution cost involved in parallelizing them, implementations of parallel symbolic factorization on message-passing computers tend to be inefficient. Moreover, symbolic factorization is often performed once and then several systems with the same sparsity pattern are solved, amortizing the cost of symbolic factorization over all the systems.

11.4.3 Numerical Factorization

Numerical factorization refers to performing arithmetic operations on the coefficient matrix A to produce a lower-triangular matrix L and an upper triangular matrix U. Usually, the basic algorithm used for numerical factorization is either Gaussian elimination (for general systems) or Cholesky factorization (for symmetric positive definite systems). In this section we choose the row-oriented versions of these algorithms given in Programs 3.1 and 5.6, respectively.

In Gaussian elimination for dense matrices, pivots are chosen sequentially because a pivot modifies all the rows of the unfactorized part of the matrix. A row can be chosen



(a) (b)

Figure 11.23 A sparse matrix and the corresponding elimination tree.

as a pivot row only after it has been modified by the previous pivot. In the sparse case, however, two or more rows can be completely independent. In a sparse matrix A, row k directly modifies row i if $i > k$ and $A[i,k] \neq 0$. Row k indirectly modifies row i if a row j modified (directly or indirectly) by row k modifies row i. Rows k and i are independent if row k does not modify row i directly or indirectly.

The mutually independent rows of a sparse matrix can be used as pivots in any order, or even simultaneously in a parallel implementation. This is a very important source of parallelism in sparse matrix factorization. For example, in Figure 11.22(b), rows 0, 1, 2, or 3 can be used as pivots in any order, or in parallel. The remaining four rows are modified by one or more of these rows. For example, row 4 cannot be used as a pivot until rows 0 and 2 have been used as pivots to modify row 4. Similarly, row 5 can be used as a pivot only after it has been modified by rows 1 and 3. Thus, a partial ordering exists among the rows of the sparse matrix that determines when a certain row can be used as a pivot.

A useful concept that yields this partial order and helps abstract this form of parallelism in sparse matrix factorization is that of ***elimination trees***. Assume that the matrix of coefficients A has been ordered such that the numerical stability criteria are satisfied and choosing pivots in the order 0 to $n-1$ yields a stable solution. The elimination tree corresponding to this matrix has one node for each row. Node j is the parent of node i if $j > i$ and j is smallest among all k such that $A[k,i]$ is nonzero in the $(n-i) \times (n-i)$ active (unfactorized) part of the matrix remaining after the first $(i-1)$ rows have been used as pivots. For instance, the sparse matrix of Figure 11.22(b) is reproduced in Figure 11.23(a), and the corresponding elimination tree is shown in Figure 11.23(b). In a parallel implementation, rows whose entire set of descendants has been eliminated can be eliminated in parallel.

Since each level of an elimination tree must be processed one after the other from the leaves to the root, the parallel run time is a function of the height of the tree. The average

number of independent tasks, and hence the amount of parallelism, is a function of the width of the elimination tree. Thus, short and bushy elimination trees are more suitable for parallel sparse direct solvers than tall and lean trees.

The ordering of rows and columns of the sparse matrix plays an important role in determining the structure of the resulting elimination tree. Usually, nested-dissection ordering results in short, well-balanced trees, and minimum-degree ordering results in tall, unbalanced trees. However, it is observed in practice that minimum-degree ordering generates less fill-in than does the nested-dissection ordering.

Serial Sparse Factorization Algorithms

In the row-oriented serial Gaussian elimination algorithm for dense matrices (Program 5.4), during the k^{th} iteration, the k^{th} row of the matrix is used to update all remaining $n-k-1$ rows. However, in the sparse case, the k^{th} row is used to update only those rows in the active part of A that have a nonzero element in the k^{th} column. Therefore, the serial algorithm for the sparse case is more complex since it must keep track of the sparsity pattern of the active part of the matrix to determine the rows that require modification by a pivot. To describe the sparse algorithm, we introduce the following two operations:

(1) *divide*(k), which divides every nonzero element of the k^{th} row in the upper-triangular part of the matrix by $A[k,k]$; and
(2) *modify*$(i, A[k,*])$, which subtracts a multiple of the sparse vector $A[k,*]$ from the i^{th} row of the matrix A. This multiple is the product of $A[i,k]$ and the k^{th} row of A.

The *divide* operation corresponds to lines 5 and 6 of the dense algorithm given in Program 5.4, and the *modify* operation corresponds to lines 11 and 12. To keep track of the rows that each pivot should modify, we define a data structure S_i as the set of all the rows of A with indices smaller than i that modify the i^{th} row during the factorization of A.

Program 11.5 outlines a serial sparse Gaussian elimination algorithm. This algorithm is fairly straightforward. It assumes that the numerical stability criteria have been taken into account in the ordering phase and that pivots can be chosen in order from 0 to $n-1$. When the k^{th} row is chosen as the pivot, all the *modify* (modify_gauss) operations are performed on this row (line 7). Then the pivot row is ready for the *divide* (divide_gauss) operation (line 8). The *divide* operation is followed by incorporating the pivot row (that is, the k^{th} row) in S_i for all i such that the i^{th} row needs to be modified by the k^{th} row (line 9). Program 11.5 generates the data structures S_i during the course of factorization. Often, the data structures S_i are generated during the symbolic factorization phase. Note that, in practice, the steps on lines 6 and 7 do not require that the condition $j \in S_k$ be checked serially for all j from 0 to $k-1$. The data structures of the sparse storage scheme being used maintain the relevant rows in the correct order. Similarly, all $A[i,k]$ $(k < i < n)$ on line 9 and all $A[k,j]$ $(k < j < n)$ on lines 15 and 21 in Program 11.5 are not explicitly checked for nonzero entries. If that was the case, then each execution of lines 9, 15 and

```
1.  procedure SERIAL_SPARSE_GAUSS (A)
2.  begin
3.      for i := 0 to n − 1 do S_i := ∅;
4.      for k := 0 to n − 1 do
5.      begin
6.          for j := 0 to k − 1 do
7.              if j ∈ S_k then modify_gauss (k, A[j, ∗]);
8.          divide_gauss (k);
9.          for all i such that ((i > k) and (A[i, k] ≠ 0)) do S_i := S_i ∪ {k};
10.     endfor;
11. end SERIAL_SPARSE_GAUSS
12.
13. procedure modify_gauss (i, A[k, ∗])
14. begin
15.     for all j such that ((j > k) and (A[k, j] ≠ 0)) do
16.         A[i, j] := A[i, j] − A[i, k] × A[k, j];
17. end modify_gauss
18.
19. procedure divide_gauss (k)
20. begin
21.     for all j such that ((j > k) and (A[k, j] ≠ 0)) do
22.         A[k, j] := A[k, j]/A[k, k];
23. end divide_gauss
```

Program 11.5 A serial sparse Gaussian elimination algorithm and the corresponding *modify* (modify_gauss) and *divide* (divide_gauss) operations.

21 would take $\Theta(n)$ time—the same as that for a dense matrix. The use of sparse storage schemes described in Section 11.1.1 helps keep track of the nonzero entries in a row or column of A. Thus, these operations can be performed in $\Theta(m)$ time, where m is the number of nonzero entries in the k^{th} column (line 9) or the k^{th} row (lines 15 and 21) of the active part of the matrix during the k^{th} iteration of the outer loop of Gaussian elimination.

An algorithm similar to procedure SERIAL_SPARSE_GAUSS in Program 11.5 can be used to perform Cholesky factorization on a sparse symmetric positive definite matrix A. The *modify* and *divide* operations to be used in the case of Cholesky factorization are given by procedures modify_chol and divide_chol in Program 11.6. Note that, in the case of Cholesky factorization, the rows in S_k can be chosen in any order to modify the k^{th} row. This is in contrast to Gaussian elimination, in which the rows in S_k must be applied in an increasing order while modifying the k^{th} row.


```
procedure modify_chol (i, Vector)
begin
    for all j such that ((j ≥ i) and (Vector[j] ≠ 0)) do
        A[i, j] := A[i, j] − Vector[j];
end modify_chol

procedure divide_chol (k)
begin
    A[k, k] := √A[k, k];
    for all j such that ((j > k) and (A[k, j] ≠ 0)) do
        A[k, j] := A[k, j]/A[k, k];
end divide_chol
```

Program 11.6 The *modify* (modify_chol) and *divide* (divide_chol) operations for use with a sparse row-oriented Cholesky factorization.

A Parallel Implementation of Sparse Gaussian Elimination

There are three levels of parallelism available in sparse factorization:

(1) ***Fine-grain*** parallelism at the level of individual scalar floating-point operations.
(2) ***Medium-grain*** parallelism at the level of performing floating-point operations over nonzero elements of entire rows or columns of the coefficient matrix (such as *divide* and *modify* operations).
(3) ***Coarse-grain*** parallelism at the level of updating groups of rows or columns that can be solved independently of other such groups. If the factorization process is viewed as a collection of subtasks whose partial ordering is defined by an elimination tree, then coarse-grain parallelism refers to processing entire subtrees of the elimination tree.

Fine-grain parallelism is not suitable for message-passing computers, even with state-of-the-art hardware technology. We first describe a parallel implementation of sparse Gaussian elimination that exploits only medium-grain parallelism. In the next two subsections, we will describe parallel algorithms for sparse matrix factorization that exploit both medium- and coarse-grain parallelism.

Program 11.7 shows a parallel sparse Gaussian elimination algorithm for a message-passing parallel computer, in which each processor stores one row of the coefficient matrix. This algorithm uses n processors to factorize the $n \times n$ matrix A whose k^{th} row is initially assigned to the k^{th} processor. Thus, the k^{th} processor performs all the *modify*$(k, A[i, *])$ operations (for any i such that the i^{th} row modifies the k^{th} row; that is, $i \in S_k$) and the *divide*(k) operation. In the loop that starts at line 3, a processor receives the information

```
1.  procedure PARALLEL_SPARSE_GAUSS (my_id, A[my_id, *])
2.  begin
3.      for i := 0 to my_id − 1 do
4.          if i ∈ S_my_id then
5.          begin
6.              receive (A[i, *]) from the processor labeled i;
7.              modify_gauss (my_id, A[i, *]);
8.          endif;                    /* Line 4 */
9.      divide_gauss (my_id);
10.     for all j such that ((j > my_id) and (A[j, my_id] ≠ 0)) do
11.         send (A[my_id, *]) to the processor labeled j;
12. end PARALLEL_SPARSE_GAUSS
```

Program 11.7 A parallel sparse Gaussian elimination algorithm for the case in which each processor stores one row of the matrix of coefficients. Data is mapped such that the processor labeled my_id stores row number my_id. The algorithm assumes that S_{my_id} has been generated during symbolic factorization, and is available before numerical factorization starts.

required to *modify* the row it stores. Note that a processor is blocked on the **receive** on line 6 until the *divide*(i) operation has been performed at processor i. When the entire subtree rooted at node k of the elimination tree has been processed and the *modify*$(k, A[i, *])$ operations have been performed for all $i \in S_k$, the *divide*(k) operation is performed. After performing the *divide* operation, a processor sends its row to all the processors that need this row to modify the rows assigned to them.

Program 11.7 is of academic interest only, as it would be too inefficient on any practical parallel computer. In practice, the number of processors used is much less than n, the dimension of the coefficient matrix A. An algorithm similar to Program 11.7 can be used to perform sparse Cholesky factorization as well. Unlike Gaussian elimination, for Cholesky factorization the modifying rows need not be received in order; a processor labeled k receives a sparse vector of the form $A[i, *]$ as soon as it arrives, and performs the *modify*$(k, A[i, *])$ operation. Finally, when *modify*$(k, A[i, *])$ has been performed for all $i \in S_k$, processor k performs *divide*(k).

Parallel Fan-Out Algorithm

We now describe a parallel algorithm (Program 11.8), known as the ***fan-out*** algorithm, for sparse matrix factorization. This algorithm uses fewer than n processors for an $n \times n$ matrix, and each processor stores more than one row of the sparse matrix of coefficients A. The set of rows belonging to the i^{th} processor is stored in *List*(i). The algorithm performs a *divide* operation (line 10) on any of its rows after the subtree rooted at the node corresponding to that row has been processed and all the *modify* operations on that row have


```
1.  procedure PARALLEL_FAN_OUT (my_id, List(my_id))
2.  begin
3.      while (List(my_id) ≠ ∅) do
4.      begin
5.          if (∃i ∈ List(my_id) such that
                vectors A[k, *] have been received for all k ∈ S_i) then
6.          begin
7.              for k := 0 to i − 1 do
8.                  if k ∈ S_i then modify_gauss (i, A[k, *]);
9.              List(my_id) := List(my_id) − {i};
10.             divide_gauss (i);
11.             for all j such that ((j > i) and(A[j, i] ≠ 0)) do
12.                 send (j, A[i, *]) to processor storing the j^th row;
13.         endif;                    /* Line 5 */
14.         if (there is an incoming message) then
15.             receive and store the message;
16.     endwhile;                     /* Line 3 */
17. end PARALLEL_FAN_OUT
```

Program 11.8 A parallel fan-out algorithm for Gaussian elimination. Processor my_id stores the set of rows assigned to it in a list called *List*(my_id). The algorithm assumes that S_{my_id} has been generated during symbolic factorization, and is available before numerical factorization starts.

been performed. The algorithm then sends the divided row to the processors responsible for the rows that must be modified by the divided row.

Program 11.8 can be easily modified for sparse Cholesky factorization. In Cholesky factorization, the *modify* operations (line 8) can be performed in any order. Hence, it is not necessary to store a message. As soon as a message is received, the corresponding *modify* operation can be performed.

Note that, in the fan-out algorithm, the processor storing the i^{th} row receives a message for every *modify*$(i, A[k, *])$ operation if the i^{th} and k^{th} rows reside on different processors. The total number of messages sent can be reduced by concatenating all outgoing messages from a processor that modify the same row. If $k \in S_i \cap$ *List*(my_id) (that is, $A[k, *]$ belongs to those rows in S_i that reside on processor my_id), then a single message containing all such rows $A[k, *]$ is sent from processor my_id to the processor storing the i^{th} row. This message is sent after the *divide*(k) operations for all $k \in S_i \cap$ *List*(my_id) have been performed on processor my_id. This strategy does not reduce the total volume of communication in sparse Gaussian elimination, but it reduces the overhead due to message startup time, which may otherwise dominate the communication time.

Parallel Fan-In Algorithm for Cholesky Factorization

This subsection describes a parallel algorithm for the row-oriented Cholesky factorization of sparse symmetric positive definite matrices. This algorithm, known as the ***fan-in*** algorithm, is an improvement over the fan-out algorithm. Recall from Program 11.5 that the *modify*$(i, A[k, *])$ operation in Gaussian elimination subtracts a multiple of a part of the k^{th} row of the coefficient matrix from the corresponding part of the i^{th} row. The factor by which the modifying row (that is, the k^{th} row) is multiplied is the element $A[i, k]$ of the row to be modified (that is, the i^{th} row). Unlike Gaussian elimination, in a row-oriented Cholesky factorization algorithm (Program 5.6), the factor by which the modifying row (that is, the k^{th} row) is multiplied is the element $A[k, i]$ of the modifying row itself. This means that the multiple of the k^{th} row that needs to be subtracted from the i^{th} row can be computed at the processor that stores the k^{th} row.

Recall that the number of messages passed in the fan-out algorithm can be reduced by concatenating all the outgoing messages from a processor that modify the same row. In the case of Cholesky factorization, the appropriate multiples of these outgoing rows can be added together and sent to the destination processor as a single vector, which is then subtracted from the row to be modified. Thus, not only the number of messages, but also the total volume of communication, is reduced. The fan-in algorithm is based on this strategy.

Program 11.9 gives the fan-in algorithm for the row-oriented Cholesky factorization of an $n \times n$ sparse symmetric positive definite matrix A. A significant difference between Programs 11.8 and 11.9 is that, after performing the *divide*(i) operation, a processor does not send the i^{th} row to all the processors storing the rows that must be modified by the i^{th} row. Instead, each processor stores sparse vectors $Update_j$ for $0 \le j < n$. Initially, all the elements in these vectors are zeros. If the i^{th} row modifies the j^{th} row, then after *divide*(i) has been performed, the appropriate multiple of the former is computed by multiplying it with $A[i, j]$. The product $A[i, *] \times A[i, j]$ is then added to $Update_j$. After the *divide* operation has been performed on all rows stored at processor my_id that modify the j^{th} row, processor my_id sends $Update_j$ to the processor storing the j^{th} row. Thus, instead of sending one message for each component of $Update_j$, each processor with a nonzero $Update_j$ sends only one message to the processor storing the j^{th} row. The processor storing the j^{th} row, upon receiving an $Update_j$, subtracts it from the j^{th} row.

Mapping Matrix Rows onto Processors

As mentioned earlier, fine-grain parallelism in sparse matrix factorization is not suitable for message-passing computers. However, exploiting only coarse-grain parallelism is unlikely to yield highly scalable parallel formulations. The reason is that, despite reducing communication costs, using only coarse-grain parallelism limits the degree of concurrency, which decreases further as the computation progresses toward the root of the elimination tree.


```
1.  procedure PARALLEL_FAN_IN (my_id, List(my_id))
2.  begin
3.     for i := 0 to n − 1 do Update^i := 0;
4.     while (List(my_id) ≠ ∅) do
5.     begin
6.        if (∃i ∈ List(my_id) such that
                divide(j) has been performed for all j ∈ S_i) then
7.        begin
8.           while (messages of the form (i, Vector) have not been received
                  from all processors that store rows belonging to S_i) do
9.           begin
10.             receive (i, Vector);
11.             modify (i, Vector);
12.          endwhile;          /* Line 8 */
13.          List(my_id) := List(my_id) − {i};
14.          divide(i);
15.          for all j such that ((j > i) and(A[j, i] ≠ 0)) do
16.          begin
17.             Update_j := Update_j + A[i, j] × A[i, *];
18.             if (divide(k) has been performed
                   for all k ∈ (S_j ∩ List(my_id))) then
19.                send (j, Update_j) to the processor storing the j^th row;
20.          endfor;            /* Line 15 */
21.       endif;                /* Line 6 */
22.    endwhile;                /* Line 4 */
23. end PARALLEL_FAN_IN
```

Program 11.9 A parallel fan-in algorithm for sparse direct row-oriented Cholesky factorization of symmetric positive definite matrices. The algorithm assumes that S_{my_id} has been generated during symbolic factorization, and is available before numerical factorization starts.

An ideal parallel formulation exploits both coarse-grain and medium-grain parallelism. As the computation progresses up the elimination tree, the availability of coarse-grain parallelism diminishes because the number of independent subtrees of the elimination tree decreases. However, medium-grain parallelism becomes more viable as the fill-in in the unfactorized part of the coefficient matrix increases, and hence, a typical *divide* or *modify* operation encounters more nonzero elements in a row. Therefore, it is advantageous to shift the emphasis systematically from coarse-grain to medium-grain parallelism as the computation progresses. Figure 11.24 shows a distribution of the rows of the matrix among the processors based on this approach.

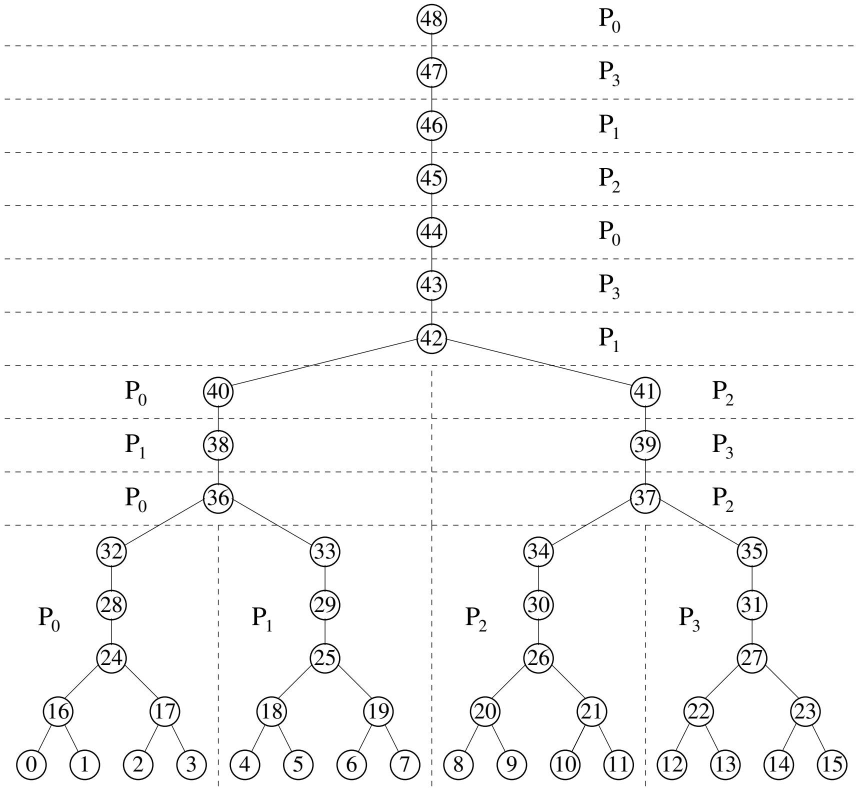


Figure 11.24 An example of a good strategy for using four processors to partition the elimination tree that corresponds to a matrix of coefficients resulting from the nested-dissection numbering of a 7×7 finite difference grid in which each processor has four neighbors.

Figure 11.24 shows the elimination tree corresponding to a matrix resulting from a nested-dissection numbering of a 7×7 finite difference grid of the form shown in Figure 11.8 (Problem 11.28). The lower, wider part of the elimination tree is partitioned vertically. In this part, independent subtrees are assigned to individual processors that perform complete updates on their respective sets of sparse rows. This assignment of subtrees to processors exploits coarse-grain parallelism. The upper, narrower part of the elimination tree is partitioned horizontally among the processors so that medium-grain parallelism is exploited. In this part of the elimination tree, individual rows are assigned to processors, and each processor performs *modify* and *divide* on its respective rows.

The partitioning strategy illustrated in Figure 11.24 keeps communication low in the initial stage of parallel factorization because the modifying and the modified rows mostly

belong to the same processor. In the later part, the communication per processor increases but is balanced by a corresponding increase in the amount of computation that a *modify* or a *divide* operation requires. At the same time, processors do not starve due to the narrowing of the elimination tree. Such an approach is likely to yield better parallel formulations of sparse factorization because it limits the communication cost while providing a high degree of concurrency. Both of these factors are important for achieving good scalability.

11.4.4 Solving a Triangular System

Like the ordering phase, solving a triangular system requires much less computation than the factorization phase. Furthermore, often this step has only a limited amount of parallelism. Still, it is desirable to perform this step in parallel for a number of reasons. First, gathering the triangular factors at a single processor after the parallel factorization phase entails substantial communication overhead. Second, the amount of memory available on a single processor may be insufficient to accommodate the entire problem. Third, with a relatively more efficient parallel factorization method, this step may dominate the overall run time. Therefore, to prevent this step from becoming a bottleneck, whatever gain in run time is achieved by parallelizing it should be exploited. Even if a parallel implementation of this step is inefficient and provides only a moderate speedup, it will increase the efficiency of the entire process of solving the sparse linear system.

The overall approach for solving a sparse triangular system in parallel is straightforward. First, all the equations with only one variable are solved. The values of the solved variables are then substituted concurrently into all the equations in which these variables are used. This step results in a fresh set of equations with only one unsolved variable each. All the equations in this set are now solved in parallel. These steps are repeated until the entire system is solved.

11.5 Multigrid Methods

Multigrid methods are iterative algorithms for solving partial differential equations by using multiple grids of varying degrees of fineness over the same domain. An approximate solution obtained by a coarse discretization is used as the initial approximation for obtaining a more precise solution by a finer discretization, and so on.

Consider a domain D and a sequence of successively finer discretizations G_0, G_1, ..., G_m. Discretization G_0 is the coarsest, and G_m is the finest. Figure 11.25 illustrates such discretizations for a square domain with $m = 3$. As the figure shows, the grid points in G_i are a subset of the grid points in G_{i+1}. The linear system of equations arising from discretization G_0 is the smallest and, consequently, the easiest to solve. After a solution or an approximation to the solution of this system has been obtained, the values of the physical quantity being modeled (say, u) at the grid points in $G_1 - G_0$ are approximated by interpolation from the values of u at the grid points in G_0. The values of u at the grid points in G_1 thus obtained serve as the initial approximation for an iterative method to solve the

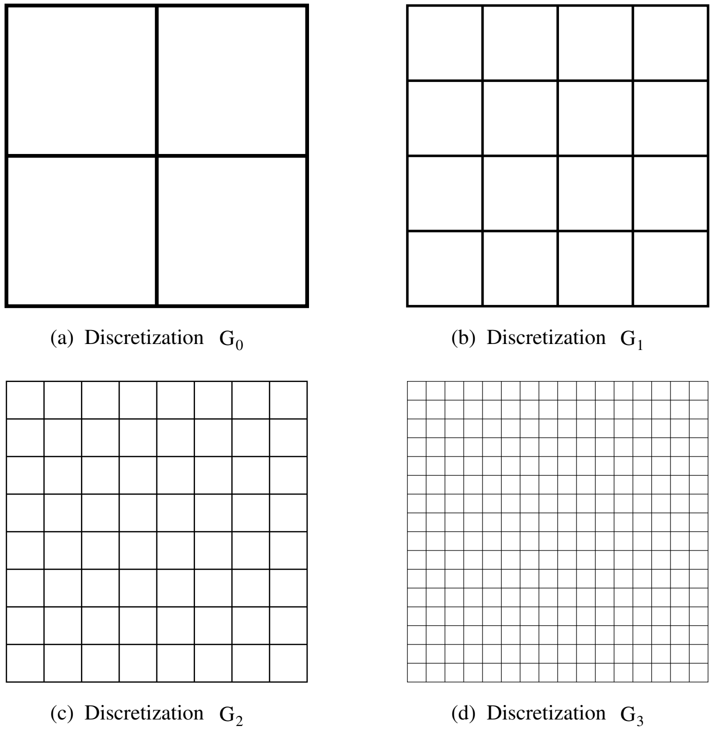


Figure 11.25 Successively finer discretizations of a domain.

system of linear equations arising out of G_1. This procedure is continued for successively finer discretizations.

The process that we just described is only a part of the multigrid method. In this method, which has several variations, information is exchanged bidirectionally between grids of varying granularity. The information from coarser discretizations is used to derive approximate starting points for finer grids. Information is also projected onto the coarser grids from the finer grids. In general, iterations over a certain discretization G_i are used to refine the solution at its grid points. This improved solution is either projected onto the next coarser grid G_{i-1} or interpolated onto the next finer grid G_{i+1}. Different variations of the multigrid method use different cycles of interpolation and projection, some of which are shown in Figure 11.26.

In summary, a typical multigrid algorithm involves three types of computations:

(1) ***Interpolation*** of values of variables in discretization G_i to approximate the values of variables in the next finer discretization G_{i+1}.

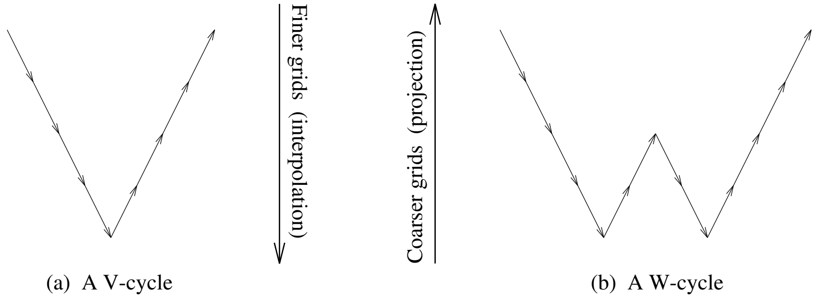


Figure 11.26 Some typical cycles of interpolation and projection used in multigrid methods.

(2) ***Projection*** of values of variables in discretization G_i onto the variables in the next coarser discretization G_{i-1}. Since G_{i-1} is a subset of G_i, the variables in G_{i-1} can simply inherit the values from the corresponding variables in G_i during projection. This process is known as ***injection***. Otherwise, a variable in G_{i-1} can be assigned a value based on the weighted average of the values of a cluster of variables around it in G_i.

(3) ***Relaxation***, which is the process of applying an iterative method to refine the approximate solution at any level of discretization. Typically, Jacobi, damped or weighted Jacobi, or Gauss-Seidel methods are used to perform relaxation in multigrid algorithms.

The advantages of using the multigrid technique are manifold. Usually, iterative methods converge faster on coarse grids than on fine grids. Furthermore, since the number of variables in the systems corresponding to coarse grids is very small, it is often possible to obtain exact solutions for these systems by using direct methods. Therefore, it is possible to obtain good starting points for iterations on fine grids at a relatively low computational cost. Iterative methods also converge faster with good initial approximations. In addition, the convergence rate of an iterative method is usually higher during the initial iterations. By iterating only a few times at each step of a cycle, we always work in the region of a high convergence rate. Thus, multigrid methods arrive at an acceptable solution to a system of linear equations arising out of a fine grid at a much faster rate than a typical iterative method applied directly on the same grid. Multigrid methods can be used for finite difference, finite element, or finite volume problems.

Parallel Implementation

We now consider a parallel implementation of a simple multigrid computation on a mesh-connected computer with cut-through routing. Since a mesh can be embedded into a hypercube, adapting this implementation for a hypercube is straightforward (Problem 11.31).

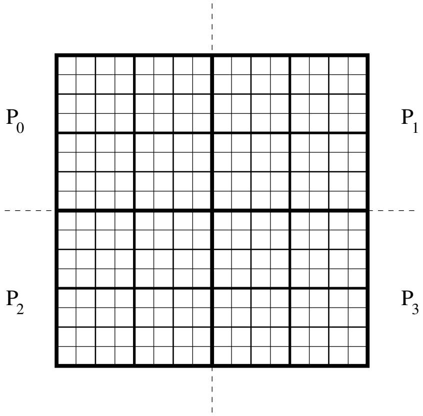


Figure 11.27 The domain of Figure 11.25 mapped onto four processors.

We assume that the domain is partitioned among the processors in a block-checkerboard fashion. Parallel implementations with the domain stripe-partitioned in one dimension are less efficient and less scalable (Problem 11.33).

First, we describe a parallel implementation of the multigrid technique for the simple case, in which the number of processors is less than or equal to the number of elements in the coarsest discretization of the domain. In this case there is no processor idling, and the only overhead is due to communication. Figure 11.27 shows such a partitioning of the domain and its discretizations G_0, G_1, G_2, and G_3, shown in Figure 11.25. Assume that $m+1$ is the total number of discretizations used ($G_0, G_1, \ldots, G_m$), the number of elements in the finest discretization G_m is n, and the number of elements in every successively coarser discretization reduces by a factor of c (in Figure 11.27, $m = 3$, $n = 256$, and $c = 4$). Hence, the number of elements in the coarsest discretization G_0 is n/c^m (for the case under consideration, $p \le n/c^m$). Also assume that the total number of interpolations, projections, and relaxation iterations is the same for each discretization. Let this number be η.

While working with discretization G_i, the amount of computation that a processor performs during an interpolation, projection, or relaxation iteration is proportional to the number of elements in a partition in G_i. Let the combined constant of proportionality for all three types of computations be t_c. Since the number of elements per partition in G_m is n/p, the total time t_{comp} spent in computation by each processor during the execution of the entire multigrid procedure is given by the following equation:

$$t_{comp} = \eta t_c \sum_{i=0}^{m} \frac{n}{pc^i}$$

$$
\begin{aligned}
&= \eta t_c \frac{n(c - 1/c^m)}{p(c-1)} \\
&\approx \frac{t_c \eta c n}{p(c-1)} \qquad (11.43)
\end{aligned}
$$

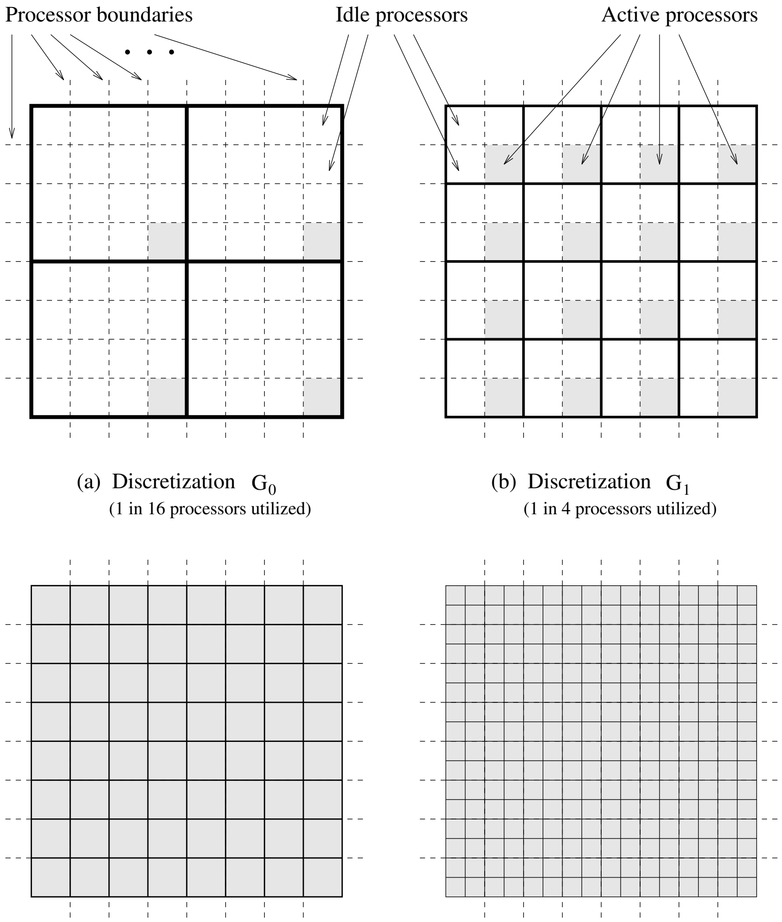
During the interpolation phases, a grid point of the finer grid requires values corresponding to the points of the coarse grid around it. Thus, an exchange of the values corresponding to the coarse-grid points lying at partition boundaries among neighboring processors suffices to perform interpolation in parallel. Projection using injection requires no communication. If the method of weighted averages is used for projection, then this phase requires an exchange of the values corresponding to the points on the finer grid points lying at partition boundaries among neighboring processors. For the partitioning illustrated in Figure 11.27, the iterative method used in the relaxation phases also requires nearest-neighbor communication of values corresponding to the points along partition boundaries to compute the matrix-vector product in each iteration (Section 11.1.3 and Figure 11.10). We disregard the communication penalty in computing vector inner products during relaxation. This assumption is valid if either hardware-supported fast reduction operations render the cost of computing a global sum insignificant, or a relaxation method like the Jacobi method is used so that an inner product is not computed in every iteration (Section 11.2.1). Thus, in each interpolation and projection step, as well as in each relaxation iteration, a processor exchanges four messages—one with each neighbor. The size of each message is proportional to the number of grid points along an edge of the square partition assigned to the processor. The number of such boundary elements in discretization G_i is equal to the square root of the number of elements in each partition in G_i. Hence, the total time t_{comm} spent in communication by each processor during the entire multigrid procedure is given by the following equation:

$$
\begin{aligned}
t_{comm} &= 4\eta \sum_{i=0}^{m} \left(t_s + t_w \sqrt{\frac{n}{pc^i}} \right) \\
&= 4(m+1)t_s\eta + 4t_w\eta \frac{\sqrt{c} - 1/c^{m/2}}{\sqrt{c}-1} \sqrt{\frac{n}{p}} \\
&\approx 4(m+1)t_s\eta + 4t_w\eta \frac{\sqrt{cn}}{(\sqrt{c}-1)\sqrt{p}} \qquad (11.44)
\end{aligned}
$$

From Equations 11.43 and 11.44, the total parallel run time $t_{comp} + t_{comm}$ is

$$T_P = \eta \left(t_c \frac{cn}{p(c-1)} + 4(m+1)t_s + 4t_w \frac{\sqrt{cn}}{(\sqrt{c}-1)\sqrt{p}} \right). \qquad (11.45)$$

We now consider the case in which the number of processors is greater than the number of elements in the coarsest discretization. Assume that $p = n/c^r$, where $0 \le r < m$; that is, the number of processors is equal to the number of elements in G_{m-r}. This case is illustrated in Figure 11.28 for $n = 256$, $m = 3$, $r = 1$, $c = 4$, and $p = 64$. As shown in the figure, some processors remain idle during the computations corresponding



(a) Discretization G_0 (1 in 16 processors utilized)

(b) Discretization G_1 (1 in 4 processors utilized)

(c) Discretization G_2 (all processors utilized)

(d) Discretization G_3 (all processors utilized)

Figure 11.28 The domain of Figure 11.25 partitioned among 64 processors. In discretizations G_0 and G_1 there are multiple processors for a single element. Therefore, some processors remain idle and only the shaded processors are active. In discretization G_2, one element is assigned to each processor, and in G_3, four elements are assigned to each processor.

to $G_0, G_1, \ldots, G_{m-r-1}$. During the interpolations, projections, and relaxation iterations corresponding to these discretizations, the active processors perform only one computation and one communication in every iteration. However, on a two-dimensional mesh, the active processors are no longer directly-connected (Figures 11.28(a) and (b)). In the remaining discretizations $G_{m-r}, \ldots, G_m$, the computation per processor is proportional to the number of elements in each partition, and the volume of communication per processor is proportional to the square root of the number of elements in each partition. The overall parallel run time on a mesh with cut-through routing is given by the following equation:

$$
\begin{aligned}
T_P \quad = \quad & \overbrace{\eta\left((m-r)t_c + t_c\sum_{i=0}^{r}\frac{n}{pc^i}\right)}^{\text{computation or idling}} \\
& + \overbrace{\eta\left(4(m+1)t_s + 4(m-r)t_w + 4t_w\sum_{i=0}^{r}\sqrt{\frac{n}{pc^i}} + 4t_h\sum_{i=1}^{m-r}\sqrt{c^i}\right)}^{\text{communication}}
\end{aligned}
$$

In practice, the number of processors p is much smaller than the number of elements n in the finest discretization. Hence, $(m-r) \ll \sqrt{n/p}$, and the expression for parallel run time can be approximated by

$$
\begin{aligned}
T_P \quad \approx \quad & \eta\left((m-r)t_c + t_c\frac{cn}{p(c-1)}\right) \\
& + \eta\left(4(m+1)t_s + 4t_w\frac{\sqrt{cn}}{(\sqrt{c}-1)\sqrt{p}} + 4t_h\frac{\sqrt{c}}{\sqrt{c}-1}(c^{(m-r)/2}-1)\right).
\end{aligned}
\tag{11.46}
$$

The isoefficiency function of a multigrid computation on a mesh with cut-through routing is $\Theta(p)$ in the absence of overhead due to inner-product computations (Problem 11.30). In other words, the parallel system is ideally scalable and requires that n be proportional to p for cost optimality or for maintaining constant efficiency. However, the constant of proportionality depends on the values of m, r, and c (besides the hardware-related constants). Even if the number of processors is greater than the number of elements in the coarsest discretization, the isoefficiency function remains linear in p (Problem 11.32). The idling time of processors during the computation of coarse discretizations is only a small fraction of the total parallel run time. The reason is that the complexity of the interpolation, projection, and relaxation steps increases rapidly as the grid is made finer. Therefore, a large fraction of the run time is spent working with fine discretizations, which do not involve any idling.

11.6 Bibliographic Remarks

Due to the importance of sparse matrices in scientific and engineering applications, the amount of literature on parallel algorithms for sparse matrix computations is immense.

There are a number of good references for storage schemes and basic operations on sparse matrices [DER90, FWPS92, GL81, Pet91, Saa90, Wij89]. Ferng et al. [FPW93] discuss implementations of some basic sparse matrix-vector computations on SIMD computers. Parallel sparse matrix-vector multiplication is discussed by several authors [BEP93, CS93a, CS93b, Ham92].

Iterative methods for solving large sparse systems are very popular on parallel computers because they can be parallelized more easily than direct methods. Parallel iterative methods in general are discussed by Bertsekas and Tsitsiklis [BT89], Dongarra et al. [DDSvdV91], Golub and Ortega [GO93], and Petiton [Pet91]. Among iterative methods, parallelization of the conjugate gradient (CG) algorithm (and its variants) has received the most attention [And88, AOES88, DM91, GKS92, HS92, JP92, KC91, KS84a, LR88, MG87, PWD91, SS85, vdV82, vdV87a]. The description of the serial unpreconditioned and preconditioned CG algorithms in this chapter is based on the description by Golub and Van Loan [GL89b] and Golub and Ortega [GO93]. The truncated incomplete Cholesky preconditioner for block-tridiagonal matrices described in this chapter was first used by van der Vorst [vdV82] and subsequently studied by Kamath and Sameh [KS84b], and Gupta, Kumar, and Sameh [GKS92]. Variations of the CG algorithm for reducing the overall communication cost of synchronization and vector inner-product computation have been presented by some authors [AAIT89, CG89, DER93, DR92]. Parallel relaxation methods such as Jacobi, Gauss-Seidel, and SOR are described by Adams and Ortega [AO82], Bertsekas and Tsitsiklis [BT89], and Golub and Ortega [GO93]. In addition to the red-black and multicolored orderings for Gauss-Seidel and SOR methods presented in this chapter, Golub and Ortega [GO93] describe a ***diagonal ordering*** that facilitates pipelined implementations of these methods. In this chapter, we have not covered a class of iterative solvers known as ***projection methods***. Parallel projection methods have been discussed by Bramley and Sameh [BS89] and Kamath and Weeratunga [KW91]. Parallel implementations of the finite element method (FEM) based on unpreconditioned CG methods without explicitly assembling the stiffness matrix are discussed in detail by Fox et al. [FJL$^+$88]. The performance of these techniques depends on the partitioning of the domain. A number of heuristic approaches have been presented to derive reasonable suboptimal partitions [AOES88, CR92, MO87, PCF$^+$91, SE87].

A number of techniques have been developed for partitioning finite element graphs. Striped partitioning is described by Morrison and Otto [MO87] and Schwan et al. [SBB$^+$87]. The use of scattered decomposition in FEM is described by Fox et al. [FJL$^+$88], Morrison and Otto [MO87], and Williams [Wil87]. A detailed discussion on scattered decomposition along with several interesting analytical results is presented by Nicol and Saltz [NS90]. Berger and Bokhari [BB87] describe the use of binary decomposition. Sadayappan and Ercal [SE87] proposed the two-dimensional decomposition scheme with boundary refinement. Chung and Ranka [CR92] describe the two-way striping and greedy assignment schemes for partitioning FEM graphs and give the details of the load balancing algorithm for use in conjunction with two-dimensional striped partitioning. Heath and Raghavan [HR92] give a fully parallel algorithm for computing graph separators based on coordinate bisection. This scheme uses connectivity information to limit cross edges. Raghavan [Rag93b] extends this scheme to the three-dimensional case. Our discussion of recursive bisection is based on its description by Simon [Sim91]. Pothen et al. [PSL90] discuss the use of eigenvectors of the adjacency matrix to partition finite element graphs. A discussion of the Lanczos algorithm for computing the Fiedler vector can be found in the book by Parlett [Par80]. Pothen et al. [PSWB92] describe a new partitioning scheme for FEM graphs called ***spectral nested dissection***. The scalability of some of these partitioning techniques has been analyzed by Grama and Kumar [GK92].

Various ordering schemes for sparse matrices, and their suitability for parallel factorization have been studied by a number of researchers [GL89a, JK82, Liu89a, Liu89b, LL87, LPP89, PSWB92]. The work in developing efficient parallel ordering algorithms is fairly rudimentary to date, and only a few references are available on this topic [Con90, HR92, Liu85, Pet84].

Parallel symbolic factorization is treated by Alaghband [Ala89], Gilbert and Hafsteinsson [GH90], George et al. [GHLN87], Heath et al. [HNP91], Heath and Raghavan [HR93], and Zmijewski and Gilbert [ZG88].

The most computationally expensive phase of obtaining a direct solution to a sparse system of linear equations is numerical factorization. As a result, parallel numerical factorization has received much attention [AEL90a, AEL$^+$90b, AJ85, BDK$^+$89, CGLN84, Con86, DDSvdV91, GHLN89, GHLN88, GLN89, GN89, GS92, HNP91, HR93, Leu89, Rag93a, SR89, Zmi87]. A class of algorithms called ***multifrontal methods*** [DR83, Liu90a] is becoming increasingly popular for solving sparse linear systems on parallel computers. Multifrontal methods are generalizations of frontal methods, which keep a relatively small portion of the matrix in main memory at a time and use a full matrix representation for this active portion of the matrix. Multifrontal methods use multiple active portions, and this is the basic source of parallelism in a multifrontal algorithm. Parallel formulations of multifrontal methods have been described by Duff [Duf86], Geist [Gei87], Lucas [Luc87], Pothen and Sun [PS91], and Pozo and Smith [PS93]. Ashcraft et al. [AELS90] compare the fan-out, fan-in, and multifrontal approaches for sparse numerical factorization. The discussion on fine, medium, and coarse levels of granularity in sparse numerical factorization is due to Liu [Liu86]. Liu [Liu90b] discusses the role of elimination trees in sparse factorization in detail.

Solving triangular systems involves very few computations compared to factorization. Moreover, the process has limited parallelism. Therefore, the prospects for developing efficient parallel implementations of this phase are bleak. Solving sparse triangular systems of linear equations is discussed by Alvarado, Pothen, and Schreiber [APS92], Anderson and Saad [AS89], and Alvarado and Pothen [AS93].

In Section 11.4, we concentrated mainly on direct methods for solving sparse linear systems involving unstructured sparse matrices of coefficients. There are systems of practical importance in which the matrix of coefficients has a special structure. Notable among such systems are tridiagonal, block-tridiagonal, and banded systems. Parallel algorithms for solving tridiagonal systems have been described by Stone [Sto73,

(a) A sparse matrix in coordinate format

	P_0					P_1					P_2				
VAL	8	6	12	1	15	14	9	2	3	5	13	4	11	7	10
I	2	2	3	0	5	4	3	0	0	1	4	1	3	2	3
J	5	1	5	0	5	4	0	3	5	1	1	0	4	2	3

(b) The matrix in Ellpack-Itpack format

	VAL				J			
P_0	1	2	3	-	1	2	3	-1
	4	5	-	-	4	5	-1	-
P_1	6	7	8	-	6	7	8	-1
	9	10	11	12	9	10	11	12
P_2	13	14	-	-	13	14	-1	-
	15	-	-	-	15	-1	-	-

Figure 11.29 A 6×6 sparse matrix in the coordinate format and its scattered form distributed among three processors.

Sto75], van der Vorst [vdV87b], and Wang [Wan81]. Parallel banded systems are discussed by Cleary [Cle89], Dongarra and Johnsson [DJ87], Johnsson [Joh85], Lawrie and Sameh [LS84], and Meier [Mei85].

Despite the best research efforts, there are still gaps in the current understanding of parallel sparse factorization. Most of the research up to the time of this writing has been empirical, and very few efforts to theoretically analyze the scalability and available parallelism of sparse direct methods have been made [Sch92, Wor91].

Multigrid techniques are gaining some popularity for solving linear systems. A few texts, such as those by Briggs [Bri87] and Hackbrush [Hac85], provide excellent discussions on multigrid methods. Bertsekas and Tsitsiklis [BT89], Fox et al. [FJL+88], and Golub and Ortega [GO93] present parallelization techniques for the multigrid method. Chan and Saad [CS86], and Chan and Tuminaro [CT87] describe parallel implementations of multigrid algorithms on hypercubes. Chan and Schreiber [CS85] also address some issues in parallel multigrid algorithms.

Problems

11.1 Consider an unstructured sparse matrix stored in the coordinate format and partitioned uniformly among p processors as shown in Figure 11.29(a). Assuming that the underlying architecture is a hypercube with cut-though routing, give an expression for the time spent in communication to have the original matrix uniformly distributed among the processors in Ellpack-Itpack format as shown in Figure 11.29(b). Assume that the size of the matrix is $n \times n$, the total number of nonzero elements is q, and the number of processors is p. Describe algorithms for conversion between the two forms shown in Figures 11.29(a) and (b).

Hint: Use all-to-all personalized communication.

11.2 Assume that an $n \times n$ sparse matrix A with a total of q nonzero elements is mapped onto p processors as shown in Figure 11.29(a), so that each processor is assigned q/p elements. Assume that an $n \times 1$ vector x is uniformly distributed among the processors so that each processor is assigned n/p of its elements. Describe an algorithm to compute the matrix-vector product Ax on a hypercube with cut-through routing. What is the parallel run time?
Hint: Two communication operations are involved—all-to-all broadcast and all-to-all personalized communication—both with messages of size at most n/p.

11.3 Refer to Problem 11.2. What is the parallel run time if the matrix A is mapped onto the processors as shown in Figure 11.29(b), so that each processor is responsible for n/p rows of the matrix? Compare this time with that obtained in Problem 11.2. What could be a possible advantage of using the coordinate format over the Ellpack-Itpack format?

11.4 Derive an expression for the parallel execution time of matrix-vector multiplication involving an $n \times n$ unstructured sparse matrix uniformly block-checkerboarded on a p-processor hypercube. Use the expected value of the maximum computation time as the effective computation time of all the processors. This time accounts for the idle time of all the processors other than the one that has the maximum amount of work. Assume a random distribution of nonzero elements among the rows of the matrix such that the average number of zeros per row is m.
Hint: **[KW85]** If there are r independent tasks with a mean completion time of μ and a standard deviation of σ, and if they are assigned to p processors such that each processor gets r/p tasks, then the expected completion time for the processor with the maximum load is $r\mu/p + \sigma\sqrt{2(r/p)\log p}$, provided that r is large compared to $p \log p$. In the problem at hand, $r = n^2$. The values of μ and σ can be computed as follows: Each of the elements of the $n \times n$ matrix can be considered equivalent to a task that takes zero time if the element is zero and t_c time if the element is nonzero. Since mn/n^2 ($= m/n$) of the elements are nonzero, the expected value of the completion time of any task is

$$\mu = t_c m/n.$$

The standard deviation is

$$\sigma = \sqrt{\frac{\Sigma_{i=1}^{n^2}(t_c x_i - \mu)^2}{n^2}},$$

where $x_i = 0$ if the i^{th} (out of n^2) element of the matrix is zero, and $x_i = 1$ if the i^{th} element is nonzero. Since the number of nonzero elements is mn and the number of zeros is $n^2 - mn$, we have

$$\sigma = t_c\sqrt{\frac{mn(1-\frac{m}{n})^2 + (n^2 - mn)(\frac{m}{n})^2}{n^2}}$$

$$= t_c \sqrt{\frac{m(1 + (\frac{m}{n})^2 - 2\frac{m}{n}) + (n - m)(\frac{m}{n})^2}{n}}$$

$$= t_c \sqrt{\frac{m}{n} - (\frac{m}{n})^2}.$$

11.5 Repeat Problem 11.4 for the case in which the matrix is mapped onto a p-processor hypercube by using block-striped partitioning.

11.6 Assuming that $q = mn$, compare the parallel execution times obtained in Problems 11.2 and 11.5. Use the comparison to determine the situations in which the coordinate format (Figure 11.29(a)) is preferable over the Ellpack-Itpack format (Figure 11.29(b)) and vice versa.

11.7 Rewrite Equation 11.2 for an $l_1 \times l_2$ finite difference grid. Describe the structure of the block-tridiagonal coefficient matrix corresponding to the system of equations resulting from this finite difference grid.

11.8 Consider the multiplication of the matrix shown in Figure 11.12(a) with a 16×1 vector using four processors. For the mapping of rows onto processors as shown in the figure, what is the parallel run time of an optimal algorithm in terms of t_c (time to perform one multiplication and one addition), t_s, and t_w? How does this time compare with the run times of the other two mappings given in Section 11.1.3 (Equations 11.8 and 11.9).

11.9 Consider a parallel implementation of unstructured sparse matrix-vector multiplication based on partitioning the graph associated with the matrix as shown in Figure 11.12. Assume that (1) the $n \times n$ matrix has an average of m nonzero elements in each row and that the graph associated with the matrix is a planar graph partitioned uniformly among p processors; (2) in a typical partition of the graph, the number of nodes lying along the partition boundary (that is, the nodes with incident edges that cross the partition boundary) is of the order of the square root of the total number of nodes in the partition; and (3) each partition shares its boundaries with at most c other partitions, where c is a small constant. Derive an expression for the parallel run time of matrix-vector multiplication in order terms. Is the algorithm scalable? If so, derive an expression for the isoefficiency function in order terms.

11.10 Derive an expression for the isoefficiency function for multiplying a banded sparse $n \times n$ matrix with an $n \times 1$ vector by using the mapping shown in Figure 11.13. Assume that a row of the matrix has an average of m nonzero elements distributed uniformly in a band of width w around the principal diagonal of the matrix.

11.11 Derive an expression for the parallel run time on p processors for multiplying a banded unstructured sparse $n \times n$ matrix with an $n \times 1$ vector such that a row of the matrix has an average of m nonzero elements distributed uniformly in a band of width w around the principal diagonal of the matrix. Assume that $m = \alpha n^s$ ($\alpha > 0, 0 \leq s \leq 1$) and $w = \beta n^t$ ($\beta > 0, 0 \leq t \leq 1$). Derive expressions in terms of p, E, t_c, t_s, and t_w for the rate at which n has to increase with p to maintain the efficiency fixed at a value E for the following sets of values for α, β, s, and t:

(a) $\alpha = 5.0, s = 0.0, \beta = 1.0, t = 0.5$
(b) $\alpha = 0.001, s = 0.5, \beta = 1.0, t = 0.5$
(c) $\alpha = 0.001, s = 0.5, \beta = 1.0, t = 1.0$
(d) $\alpha = 0.001, s = 1.0, \beta = 1.0, t = 1.0$
(e) $\alpha = 0.001, s = 1.0, \beta = 1.0, t = 0.5$

11.12 Derive an expression for the isoefficiency function for the multiplication of an $n \times n$ block-tridiagonal matrix of the form shown in Figure 11.7 with an $n \times 1$ vector on a p-processor hypercube with the data mapping shown in Figure 11.9. Treat the cases $p > \sqrt{n}$ and $p \leq \sqrt{n}$ separately.

11.13 Show that a Gauss-Seidel iteration requires at least $2\sqrt{n} - 1$ sequential steps for a block-tridiagonal matrix derived from a $\sqrt{n} \times \sqrt{n}$ finite difference grid of the form shown in Figure 11.8. Show that, in the k^{th} iteration, the computation of Equation 11.18 can be used to compute $x_k[i]$ for at most $\sqrt{n}$ values of i in parallel.

11.14 Give optimal partitionings of the grid shown in Figure 11.15 for the Gauss-Seidel algorithm on a four- and a 16-processor mesh with store-and-forward routing. Do any of the mappings change if the architecture is a hypercube or a mesh with cut-through routing?

11.15 Consider an $\sqrt{n} \times \sqrt{n}$ finite element graph of the type shown in Figure 11.15. Given an $\sqrt{p} \times \sqrt{p}$ mesh of processors, give an optimal partitioning for Gauss-Seidel algorithm. Disregarding the time spent in testing for convergence, what is the parallel run time of each iteration?

11.16 **[KS84b]** Consider the relationship $D = \tilde{D} + \text{diag}(L\tilde{D}^{-1}L^T)$, where the diagonal matrix D and the strictly lower-triangular matrix L are known. Derive a recurrence relation to determine the nonzero entries $\tilde{D}[i, i]$ of the unknown diagonal matrix $\tilde{D}$.

11.17 Derive the isoefficiency functions for an iteration of the PCG algorithm with a diagonal preconditioner for the hypercube and mesh architectures. Take the expressions for parallel run times for these architectures from Equations 11.26 and 11.27, respectively. Do the asymptotic isoefficiency functions change if the overhead due to vector inner-product computation is ignored (that is, if the expression for the parallel run time is taken from Equation 11.28)?

11.18 Show that, for an $n \times n$ matrix A derived from a finite difference grid partitioned among the processors as shown in Figure 11.10, the time to perform the step of Equation 11.18 in parallel is exactly equal to the parallel run time for solving $Mz_k = r_k$, where M is the preconditioner matrix derived from a no-fill incomplete Cholesky factorization of A.

11.19 Assume that $n = 65536$, $t_c^{'} = 10$, $t_c = t_w = 1$, and $t_s = 40$. Disregarding the $3t_s \log p$ term in Equation 11.26, plot T_P versus p curves for Equations 11.26, 11.31, 11.32, and 11.33. How does the value of τ effect the parallel run time, speedup, and isoefficiency function of an iteration of the PCG algorithm with a truncated IC preconditioner?

11.20 Show that a banded unstructured sparse matrix of bandwidth $w_1 + w_2 - 1$ results from the multiplication of two $n \times n$ banded unstructured sparse matrices with their nonzero elements distributed within bands of width w_1 and w_2 along their respective principal diagonals.

11.21 In Problem 11.20, assume that the average number of nonzero elements per row in the two matrices to be multiplied is m_1 and m_2, respectively. Show that the average number of nonzero elements per row in the product matrix is approximately $m_1 m_2$. Assume that n is large, $m_1 \ll w_1$, and $m_2 \ll w_2$.

11.22 Show that the sparse matrix of Figure 11.22(b) satisfies the criterion of minimum-degree ordering.

11.23 Reorder the sparse matrix shown in Figure 11.22(b) according to a different minimum-degree tie-breaking criterion. If, at any stage, there are multiple rows with the same cost, choose the one with the highest index. Which matrix has a higher fill-in, the one shown in Figure 11.22(b), or the one derived in this problem?

11.24 Plot the sparsity pattern of the coefficient matrix resulting from the nested-dissection ordering of a 7×7 finite difference grid of the form shown in Figure 11.8.

11.25 Reorder the sparse matrix of Problem 11.24 using minimum-degree ordering. To break ties, choose a row with the smallest index.

11.26 Reorder the sparse matrix of Problem 11.24 using a natural ordering and a red-black ordering of grid points.

11.27 Plot the locations of fill-in upon factorization in all the four sparse matrices in Problems 11.24–11.26. Which of these leads to maximum fill-in?

11.28 Draw the elimination trees for the four sparse matrices of Problems 11.24–11.26. Which of these results in maximum parallelism?

11.29 Reorder the sparse matrix of Problem 11.24 using minimum-degree ordering. To break ties, choose the row with the highest index. Plot the locations of fill-in upon factorization of the resulting matrix. Also draw the corresponding elimination tree. Does the tie-breaking strategy of minimum-degree ordering affect fill-in? Does it affect the degree of parallelism in numerical factorization?

11.30 Derive an expression for the isoefficiency function of the multigrid computation described in Section 11.5 for a mesh-connected parallel computer with cut-through routing. Assume that there is no overhead due to global sum computations and that the number of processors is equal to the number of elements in the coarsest discretization. What is the isoefficiency function in the absence of this assumption?

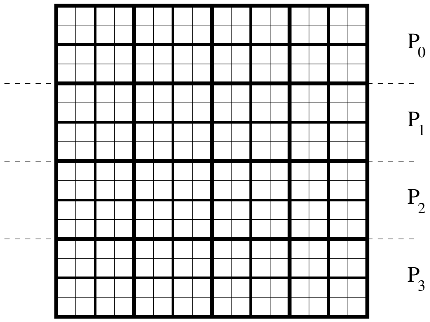


Figure 11.30 A domain with three levels of discretization stripe-partitioned among four processors of a linear array.

11.31 Derive an expression for the parallel run time and the isoefficiency function for performing the multigrid computation described in Section 11.5 on a hypercube. Assume that there is no overhead due to global sum computations, and the number of processors is equal to the number of elements in the coarsest discretization.

11.32 Show that the isoefficiency function of a multigrid computation on a mesh with cut-through routing in the absence of any overhead due to inner-product computations is linear in p, even if the number of processors is greater than the number of elements in the coarsest discretization.
Hint: Show that the total overhead due to processor idling is $\Theta(p)$.

11.33 Consider the multigrid algorithm for a square domain with discretizations G_0, G_1, ..., G_m. The finest discretization G_m has n elements, and the number of elements reduces by a factor of four in each successively coarser discretization. The domain is partitioned into stripes and distributed on a p-processor linear array as illustrated in Figure 11.30 for $m = 2$, $n = 256$, and $p = 4$. Derive expressions for the parallel run time and the isoefficiency function for this parallel implementation of the multigrid method under the assumptions of Section 11.5. For simplicity, assume that the number of processors is less than or equal to $\sqrt{n}/2^m$, so that there is no idling and so that communication always takes place among directly-connected processors in the linear array.

References

[AAIT89] D. Amitai, A. Averbuch, S. Itzikowitz, and E. Turkel. Asynchronous numerical solution of PDEs on parallel computers. In *Proceedings of the Fourth SIAM Conference on Parallel Processing for Scientific Computing*, 131–136, 1989.

[AEL90a] C. Ashcraft, S. C. Eisenstat, and J. W.-H. Liu. A fan-in algorithm for distributed sparse numerical factorization. *SIAM Journal on Scientific and Statistical Computing*, 11:593–599, 1990.

[AEL+90b] C. Ashcraft, S. C. Eisenstat, J. W.-H. Liu, B. W. Peyton, and A. H. Sherman. A compute-ahead implementation of the fan-in sparse distributed factorization scheme. Technical Report ORNL/TM-11496, Oak Ridge National Laboratory, Oak Ridge, TN, 1990.

[AELS90] C. Ashcraft, S. C. Eisenstat, J. W.-H. Liu, and A. H. Sherman. A comparison of three column based distributed sparse factorization schemes. Technical Report YALEU/DCS/RR-810, Yale University, New Haven, CT, 1990. Also appears in *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing*, 1991.

[AJ85] G. Alaghband and H. Jordan. Multiprocessor sparse L/U decomposition with controlled fill-in. Technical Report 85-48, ICASE, NASA Langley Research Center, Hampton, VA, 1985.

[Ala89] G. Alaghband. Parallel pivoting combined with parallel reduction and fill-in control. *Parallel Computing*, 11:201–221, 1989.

[And88] E. Anderson. Parallel implementation of preconditioned conjugate gradient methods for solving sparse systems of linear equations. Technical Report 805, CSRD, University of Illinois at Urbana, Urbana, IL, 1988.

[AO82] L. M. Adams and J. M. Ortega. A multi-color SOR method for parallel computation. In *Proceedings of the 1982 International Conference on Parallel Processing*, 53–56, 1982.

[AOES88] C. Aykanat, F. Ozguner, F. Ercal, and P. Sadayappan. Iterative algorithms for solution of large sparse systems of linear equations on hypercubes. *IEEE Transactions on Computers*, 37(12):1554–1567, 1988.

[APS92] F. L. Alvarado, A. Pothen, and R. Schreiber. Highly parallel sparse triangular solution. Technical Report RIACS TR 92.11, NASA Ames Research Center, Moffet Field, CA, May 1992. Also appears in J. A. George, John R. Gilbert, and J. W.-H. Liu, editors, *Sparse Matrix Computations: Graph Theory Issues and Algorithms* (An IMA Workshop Volume). Springer-Verlag, New York, NY, 1992.

[AS89] E. Anderson and Y. Saad. Solving sparse triangular linear systems on parallel computers. *International Journal of High Speed Computing*, 1:73–96, 1989.

[AS93] F. L. Alvarado and R. Schreiber. Optimal parallel solution of sparse triangular systems. *SIAM Journal on Scientific and Statistical Computing*, 14:446–460, 1993.

[BB87] M. J. Berger and S. H. Bokhari. Partitioning strategy for nonuniform problems on multiprocessors. *IEEE Transactions on Computers*, C-36(5):570–580, 1987.

[BDK+89] J. Browne, J. J. Dongarra, A. H. Karp, K. Kennedy, and D. J. Kuck. 1988 Gordon Bell prize (special report). *IEEE Software*, May 1989.

[BEP93] F. Bodin, J. Erthel, and T. Priol. Parallel sparse matrix by vector multiplication using a shared virtual memory environment. In *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, 421–428, 1993.

[Bri87] W. L. Briggs. *A Multigrid Tutorial*. SIAM, Philadelphia, PA, 1987.

[BS89] R. Bramley and A. H. Sameh. Parallel row projection algorithms for nonsymmetric systems. In *Proceedings of the Fourth SIAM Conference on Parallel Processing for Scientific Computing*, 60–62, 1989.

[BT89] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall, Englewood Cliffs, NJ, 1989.

[CG89] A. T. Chronopoulos and C. W. Gear. On the efficient implementation of preconditioned s-step conjugate gradient methods on multiprocessors with memory hierarchy. *Parallel Computing*, 11:37–53, 1989.

[CGLN84] E. Chu, J. A. George, J. W.-H. Liu, and E. G.-Y. Ng. Users guide for SPARSPAK–A: Waterloo sparse linear equations package. Technical Report CS-84-36, University of Waterloo, Waterloo, IA, 1984.

[Cle89] A. Cleary. *Algorithms for solving narrowly banded linear systems on parallel computers by direct methods*. Ph.D. thesis, Applied Mathematics, University of Virginia, Charlottesville, VA, 1989.

[Con86] J. M. Conroy. Parallel direct solution of sparse linear system of equations. Technical Report TR1714, University of Maryland, College Park, MD, 1986.

[Con90] J. M. Conroy. Parallel nested dissection. *Parallel Computing*, 16:139–156, 1990.

[CR92] Y.-C. Chung and S. Ranka. Mapping finite element graphs on hypercubes. *Journal of Supercomputing*, 6:257–282, 1992.

[CS85] T. F. Chan and R. Schreiber. Parallel networks for multigrid algorithms: Architecture and complexity. *SIAM Journal on Scientific and Statistical Computing*, 6:698–711, 1985.

[CS86] T. F. Chan and Y. Saad. Multigrid algorithms on the hypercube multiprocessor. *IEEE Transactions on Computers*, C-35:969–977, 1986.

[CS93a] R. Cook and J. Sadecki. Sparse matrix vector multiplication. Technical report, Center for Mathematical Software Research, University of Liverpool, UK, 1993.

[CS93b] R. Cook and J. Sadecki. Sparse matrix vector product on distributed-memory MIMD architectures. In *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, 429–436, 1993.

[CT87] T. F. Chan and R. S. Tuminaro. Implementation of multigrid algorithms on hypercubes. In M. T. Heath, editor, *Hypercube Multiprocessors 1987*. SIAM, Philadelphia, PA, 1987.

[DDSvdV91] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers*. SIAM, Philadelphia, PA, 1991.

[DER90] I. S. Duff, M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, Oxford, UK, 1990.

[DER93] E. D'Azevedo, V. Eijkhout, and C. H. Romine. A matrix framework for conjugate gradient methods and some variants of CG with less synchronization overhead. In *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, 644–646, 1993.

[DJ87] J. J. Dongarra and S. L. Johnsson. Solving banded systems on a parallel processor. *Parallel Computing*, 5:219–246, 1987.

[DM91] E. M. Daoudi and P. Manneback. Parallel ICCG algorithm on distributed-memory architecture. In *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing*, 78–83, 1991.

[DR83] I. S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Transactions on Mathematical Software*, 9:302–325, 1983.

[DR92] E. D'Azevedo and C. H. Romine. Reducing communication costs in the conjugate gradient algorithm on distributed-memory multiprocessors. Technical Report ORNL/TM-12192, Oak Ridge National Laboratory, Oak Ridge, TN, 1992.

[Duf86] I. S. Duff. Parallel implementation of multifrontal schemes. *Parallel Computing*, 3:193–204, 1986.

[FJL+88] G. C. Fox, M. Johnson, G. Lyzenga, S. W. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors: Volume 1*. Prentice-Hall, Englewood Cliffs, NJ, 1988.

[FPW93] W. Ferng, S. G. Petiton, and K. Wu. Basic sparse matrix computations on data parallel computers. In *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, 462–466, 1993.

[FWPS92] W. Ferng, K. Wu, S. G. Petiton, and Y. Saad. Basic sparse matrix computations on massively parallel computers. Technical Report 92-084, Army High Performance Computing Research Center, University of Minnesota, Minneapolis, MN, 1992.

[Gei87] G. A. Geist. Solving finite element problems with parallel multifrontal schemes. In M. T. Heath, editor, *Hypercube Multiprocessors, 1987*, 656–661. SIAM, Philadelphia, PA, 1987.

[GH90] J. R. Gilbert and H. Hafsteinsson. Parallel symbolic factorization of sparse linear systems. *Parallel Computing*, 14:151–162, 1990.

[GHLN87] J. A. George, M. T. Heath, J. W.-H. Liu, and E. G.-Y. Ng. Symbolic Cholesky factorization on a local memory multiprocessor. *Parallel Computing*, 5:85–95, 1987.

[GHLN88] J. A. George, M. T. Heath, J. W.-H. Liu, and E. G.-Y. Ng. Sparse Cholesky factorization on a local memory multiprocessor. *SIAM Journal on Scientific and Statistical Computing*, 9:327–340, 1988.

[GHLN89] J. A. George, M. T. Heath, J. W.-H. Liu, and E. G.-Y. Ng. Solution of sparse positive definite systems on a hypercube. *Journal of Computational and Applied Mathematics*, 27:129–156, 1989. Also available as Technical Report ORNL/TM-10865, Oak Ridge National Laboratory, Oak Ridge, TN, 1988.

[GK92] A. Grama and V. Kumar. Scalability analysis of partitioning strategies for finite element graphs. In *Supercomputing '92 Proceedings*, 83–92, 1992.

[GKS92] A. Gupta, V. Kumar, and A. H. Sameh. Performance and scalability of preconditioned conjugate gradient methods on parallel computers. Technical Report TR 92-64, Department of Computer Science, University of Minnesota, Minneapolis, MN, 1992. A short version appears in *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pages 664–674, 1993.

[GL81] J. A. George and J. W.-H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[GL89a] J. A. George and J. W.-H. Liu. The evolution of the minimum degree ordering algorithm. *SIAM Review*, 31(1):1–19, March 1989.

[GL89b] G. H. Golub and C. V. Loan. *Matrix Computations: Second Edition*. The Johns Hopkins University Press, Baltimore, MD, 1989.

[GLN89] J. A. George, J. W.-H. Liu, and E. G.-Y. Ng. Communication results for parallel sparse Cholesky factorization on a hypercube. *Parallel Computing*, 10(3):287–298, May 1989.

[GN89] G. A. Geist and E. G.-Y. Ng. Task scheduling for parallel sparse Cholesky factorization. *International Journal of Parallel Programming*, 18(4):291–314, 1989.

[GO93] G. H. Golub and J. M. Ortega. *Scientific Computing: An Introduction with Parallel Computing*. Academic Press, Boston, MA, 1993.

[GS92] J. R. Gilbert and R. Schreiber. Highly parallel sparse Cholesky factorization. *SIAM Journal on Scientific and Statistical Computing*, 13:1151–1172, 1992.

[Hac85] W. Hackbrush. *Multigrid Methods with Applications*. Springer-Verlag, New York, NY, 1985.

[Ham92] S. W. Hammond. Mapping unstructured grid computations to massively parallel computers. Technical Report RIACS TR 92.14, NASA Ames Research Center, Moffet Field, CA, 1992.

[HNP91] M. T. Heath, E. G.-Y. Ng, and B. W. Peyton. Parallel algorithms for sparse linear systems. *SIAM Review*, 33:420–460, 1991. Also appears in K. A. Gallivan et al. *Parallel Algorithms for Matrix Computations*. SIAM, Philadelphia, PA, 1990.

[HR92] M. T. Heath and P. Raghavan. A Cartesian nested dissection algorithm. Technical Report 92-1772, Department of Computer Science, University of Illinois, Urbana, IL, 1992. To appear in *SIAM Journal on Matrix Analysis and Applications*, 1994.

[HR93] M. T. Heath and P. Raghavan. Distributed solution of sparse linear systems. Technical Report 93-1793, Department of Computer Science, University of Illinois, Urbana, IL, 1993.

[HS92] S. W. Hammond and R. Schreiber. Efficient ICCG on a shared-memory multiprocessor. *International Journal of High Speed Computing*, 4(1):1–22, March 1992.

[JK82] J. Jess and H. Kees. A data structure for parallel L/U decomposition. *IEEE Transactions on Computers*, C-31:231–239, 1982.

[Joh85] S. L. Johnsson. Solving narrow banded systems on ensemble architectures. *ACM Transactions on Mathematical Software*, 11:271–288, 1985.

[JP92] M. T. Jones and P. E. Plassmann. Scalable iterative solution of sparse linear systems. Technical report, Argonne National Laboratory, Argonne, IL, 1992.

[KC91] S. K. Kim and A. T. Chronopoulos. A class of Lanczos-like algorithms implemented on parallel computers. *Parallel Computing*, 17:763–777, 1991.

[KS84a] C. Kamath and A. H. Sameh. The preconditioned conjugate gradient algorithm on a multiprocessor. In R. Vichnevetsky and R. S. Stepleman, editors, *Advances in Computer Methods for Partial Differential Equations*. IMACS, 1984.

[KS84b] C. Kamath and A. H. Sameh. The preconditioned conjugate gradient algorithm on a multiprocessor. Technical Report ANL/MCS-TM-28, Argonne National Labs, Mathematics and Computer Science Division, Argonne, IL, 1984.

[KW85] C. P. Kruskal and A. Weiss. Allocating independent subtasks on parallel processors. *IEEE Transactions on Software Engineering*, SE-11(10):1001–1016, October 1985.

[KW91] C. Kamath and S. Weeratunga. Projection methods on a distributed-memory MIMD multiprocessor. In *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing*, 92–97, 1991.

[Leu89] M. R. Leuze. Independent set orderings for parallel matrix factorization by Gaussian elimination. *Parallel Computing*, 10:177–191, 1989.

[Liu85] J. W.-H. Liu. Modification of the minimum degree algorithm by multiple elimination. *ACM Transactions on Mathematical Software*, 11:141–153, 1985.

[Liu86] J. W.-H. Liu. Computational models and task scheduling for parallel sparse Cholesky factorization. *Parallel Computing*, 3:327–342, 1986.

[Liu89a] J. W.-H. Liu. A linear reordering algorithm for parallel pivoting of chordal graphs. *SIAM Journal on Discrete Mathematics*, 2:100–107, 1989.

[Liu89b] J. W.-H. Liu. Reordering sparse matrices for parallel elimination. *Parallel Computing*, 11:73–91, 1989.

[Liu90a] J. W.-H. Liu. The multifrontal method for sparse matrix solution: Theory and practice. Technical Report CS-90-04, York University, Ontario, Canada, 1990. Also appears in *SIAM Review*, 34:82–109, 1992.

[Liu90b] J. W.-H. Liu. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications*, 11:134–172, 1990.

[LL87] C. E. Leiserson and T. G. Lewis. Orderings for parallel sparse symmetric factorization. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, 27–32, 1987.

[LPP89] T. G. Lewis, B. W. Peyton, and A. Pothen. A fast algorithm for reordering sparse matrices for parallel factorization. *SIAM Journal on Scientific and Statistical Computing*, 10:1146–1173, 1989.

[LR88] R. W. Leland and J. S. Rolett. Evaluation of parallel conjugate gradient algorithm. In K. W. Morton and M. J. Baines, editors, *Numerical Methods in Fluid Dynamics III*, 478–483. Oxford University Press, Oxford, UK, 1988.

[LS84] D. H. Lawrie and A. H. Sameh. The computation and communication complexity of a parallel banded system solver. *ACM Transactions on Mathematical Software*, 10:185–195, 1984.

[Luc87] R. Lucas. *Solving planar systems of equations on distributed-memory multiprocessors*. Ph.D. thesis, Department of Electrical Engineering, Stanford University, Palo Alto, CA, 1987. Also see *IEEE Transactions on Computer Aided Design*, 6:981–991, 1987.

[Mei85] U. Meier. A parallel partition method for solving banded systems of linear equations. *Parallel Computing*, 2:33–43, 1985.

[MG87] R. Melhem and D. B. Gannon. Toward efficient implementation of preconditioned conjugate gradient methods on vector supercomputers. *International Journal of Supercomputing Applications*, I(1):70–97, 1987.

[MO87] R. Morrison and S. W. Otto. The scattered decomposition for finite elements. *Journal of Scientific Computing*, 2(1), March 1987.

[NS90] D. M. Nicol and J. H. Saltz. An analysis of scatter decomposition. *IEEE Transactions on Computers*, 39:1337–1345, November 1990.

[Par80] B. N. Parlett. *The Symmetric Eigenvalue Problem*. Prentice-Hall, Englewood Cliffs, NJ, 1980.

[PCF+91] J. W. Parker, T. Cwik, R. Ferraro, P. Liewer, P. Lyster, and J. Patterson. Helmholtz finite elements performance on Mark III and Intel iPSC/860 hypercubes. In *The Sixth Distributed Memory Computing Conference Proceedings*, 1991.

[Pet84] F. Peters. Parallel pivoting algorithms for sparse symmetric matrices. *Parallel Computing*, 1:99–110, 1984.

[Pet91] S. G. Petiton. Massively parallel sparse matrix computation for iterative methods. Technical Report YALEU/DCS/878, Yale University, Department of Computer Science, New Haven, CT, 1991.

[PS91] A. Pothen and C. Sun. Distributed multifrontal factorization using clique trees. In *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing*, 34–40, 1991.

[PS93] R. Pozo and S. L. Smith. Performance evaluation of the parallel multifrontal method in a distributed-memory environment. In *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, 453–456, 1993.

[PSL90] A. Pothen, H. D. Simon, and K.-P. Liou. Partioning sparce matrices with eigenvectors of graphs. *SIAM Journal of Mathematical Analysis and Applications*, 11(3):430–452, 1990.

[PSWB92] A. Pothen, H. D. Simon, L. Wang, and S. T. Bernard. Towards a fast implementation of spectral nested dissection. In *Supercomputing '92 Proceedings*, 42–51, 1992.

[PWD91] S. G. Petiton and C. Weill-Duflos. Very sparse preconditioned conjugate gradient on massively parallel architectures. In *Proceedings of the 13th World Congress on Computation and Applied Mathematics*, 1991.

[Rag93a] P. Raghavan. Distributed sparse Gaussian elimination and orthogonal factorization. Technical Report 93-1818, Department of Computer Science, University of Illinois, Urbana, IL, 1993.

[Rag93b] P. Raghavan. Line and plane separators. Technical Report 93-1794, Department of Computer Science, University of Illinois, Urbana, IL, 1993.

[Saa90] Y. Saad. SPARSKIT: A basic tool kit for sparse matrix computations. Technical Report 90-20, Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffet Field, CA, 1990.

[SBB+87] K. Schwan, W. Bo, N. Bauman, P. Sadayappan, and F. Ercal. Mapping parallel applications to a hypercube. In M. T. Heath, editor, *Hypercube Multiprocessors 1987*, 141–151. SIAM, Philadelphia, PA, 1987.

[Sch92] R. Schreiber. Scalability of sparse direct solvers. Technical Report RIACS TR 92.13, NASA Ames Research Center, Moffet Field, CA, May 1992. Also appears in J. A. George, John R. Gilbert, and J. W.-H. Liu, editors, *Sparse Matrix Computations: Graph Theory Issues and Algorithms* (An IMA Workshop Volume). Springer-Verlag, New York, NY, 1992.

[SE87] P. Sadayappan and F. Ercal. Mapping of finite element graphs onto processor meshes. *IEEE Transactions on Computers*, C-36:1408–1424, 1987.

[Sim91] H. D. Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, 2(2/3):135–148, 1991.

[SR89] P. Sadayappan and S. K. Rao. Communication reduction for distributed sparse matrix factorization on a processors mesh. In *Supercomputing '89 Proceedings*, 371–379, 1989.

[SS85] Y. Saad and M. H. Schultz. Parallel implementations of preconditioned conjugate gradient methods. Technical Report YALEU/DCS/RR-425, Yale University, Department of Computer Science, New Haven, CT, 1985.

[Sto73] H. S. Stone. An efficient parallel algorithm for the solution of a tridiagonal linear system of equations. *Journal of the ACM*, 20:27–38, 1973.

[Sto75] H. S. Stone. Parallel tridiagonal equation solvers. *ACM Transactions on Mathematical Software*, 1:289–307, 1975.

[vdV82] H. A. van der Vorst. A vectorizable variant of some ICCG methods. *SIAM Journal on Scientific and Statistical Computing*, III(3):350–356, 1982.

[vdV87a] H. A. van der Vorst. Large tridiagonal and block tridiagonal linear systems on vector and parallel computers. *Parallel Computing*, 5:45–54, 1987.

[vdV87b] H. A. van der Vorst. Large tridiagonal and block tridiagonal linear systems on vector and parallel computers. *Parallel Computing*, 5:45–54, 1987.

[Wan81] H. H. Wang. A parallel method for tridiagonal equations. *ACM Transactions on Mathematical Software*, 7:170–183, 1981.

[Wij89] H. A. G. Wijshoff. Implementing sparse BLAS primitives on concurrent/vector processors: a case study. Technical Report 843, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, 1989.

[Wil87] W. I. Williams. Load balancing on hypercubes: A preliminary look. In M. T. Heath, editor, *Hypercube Multiprocessors 1987*, 108–113. SIAM, Philadelphia, PA, 1987.

[Wor91] P. H. Worley. Limits on parallelism in the numerical solution of linear PDEs. *SIAM Journal on Scientific and Statistical Computing*, 12:1–35, January 1991.

[ZG88] E. Zmijewski and J. R. Gilbert. A parallel algorithm for sparse symbolic factorization on a multiprocessor. *Parallel Computing*, 7:199–210, 1988.

[Zmi87] E. Zmijewski. *Sparse Cholesky factorization on a multiprocessor*. Ph.D. thesis, Department of Computer Science, Cornell University, Ithaca, NY, 1987.