CHAPTER 8

Dense Matrix Algorithms

Algorithms involving matrices and vectors are applied in several numerical and non-numerical contexts. This chapter discusses some key algorithms for ***dense*** or ***full matrices*** that have no or few known usable zero entries. We deal specifically with square matrices for pedagogical reasons, but the algorithms in this chapter, wherever applicable, can easily be adapted for rectangular matrices as well.

Due to their regular structure, parallel computations involving matrices and vectors readily lend themselves to data-decomposition (Section 3.2.2). Depending on the computation at hand, the decomposition may be induced by partitioning the input, the output, or the intermediate data. Section 3.4.1 describes in detail the various schemes of partitioning matrices for parallel computation. The algorithms discussed in this chapter use one- and two-dimensional block, cyclic, and block-cyclic partitionings. For the sake of brevity, we will henceforth refer to one- and two-dimensional partitionings as 1-D and 2-D partitionings, respectively.

Another characteristic of most of the algorithms described in this chapter is that they use one task per process. As a result of a one-to-one mapping of tasks to processes, we do not usually refer to the tasks explicitly and decompose or partition the problem directly into processes.

8.1 Matrix-Vector Multiplication

This section addresses the problem of multiplying a dense $n \times n$ matrix A with an $n \times 1$ vector x to yield the $n \times 1$ result vector y. Algorithm 8.1 shows a serial algorithm for this problem. The sequential algorithm requires n^2 multiplications and additions. Assuming

```
1.  procedure MAT_VECT ($A$, $x$, $y$)
2.  begin
3.      for $i := 0$ to $n - 1$ do
4.      begin
5.          $y[i] := 0$;
6.          for $j := 0$ to $n - 1$ do
7.              $y[i] := y[i] + A[i, j] \times x[j]$;
8.      endfor;
9.  end MAT_VECT
```

Algorithm 8.1 A serial algorithm for multiplying an $n \times n$ matrix A with an $n \times 1$ vector x to yield an $n \times 1$ product vector y.

that a multiplication and addition pair takes unit time, the sequential run time is

$$W = n^2. \tag{8.1}$$

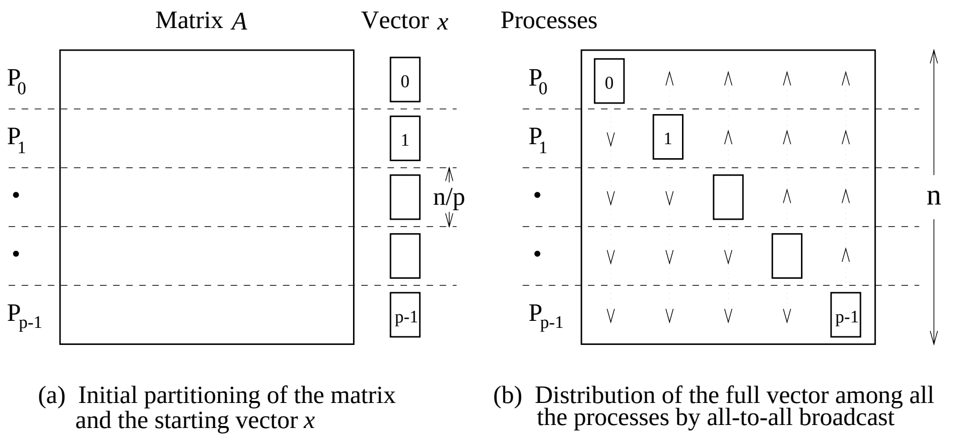
At least three distinct parallel formulations of matrix-vector multiplication are possible, depending on whether rowwise 1-D, columnwise 1-D, or a 2-D partitioning is used.

8.1.1 Rowwise 1-D Partitioning

This section details the parallel algorithm for matrix-vector multiplication using rowwise block 1-D partitioning. The parallel algorithm for columnwise block 1-D partitioning is similar (Problem 8.2) and has a similar expression for parallel run time. Figure 8.1 describes the distribution and movement of data for matrix-vector multiplication with block 1-D partitioning.

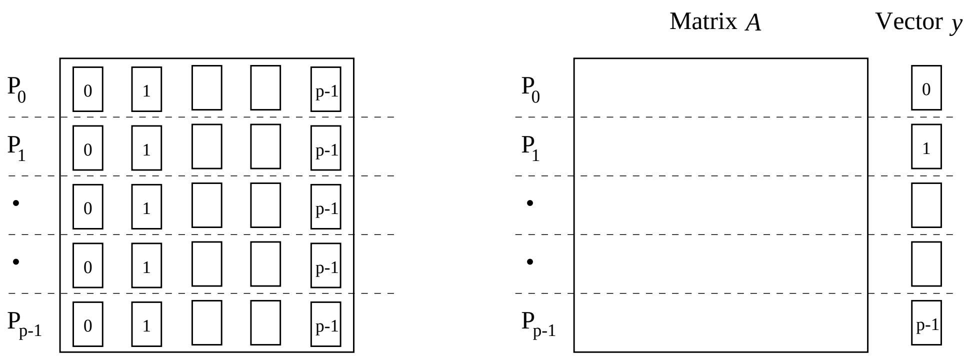
One Row Per Process

First, consider the case in which the $n \times n$ matrix is partitioned among n processes so that each process stores one complete row of the matrix. The $n \times 1$ vector x is distributed such that each process owns one of its elements. The initial distribution of the matrix and the vector for rowwise block 1-D partitioning is shown in Figure 8.1(a). Process P_i initially owns $x[i]$ and $A[i, 0]$, $A[i, 1]$, ..., $A[i, n-1]$ and is responsible for computing $y[i]$. Vector x is multiplied with each row of the matrix (Algorithm 8.1); hence, every process needs the entire vector. Since each process starts with only one element of x, an all-to-all broadcast is required to distribute all the elements to all the processes. Figure 8.1(b) illustrates this communication step. After the vector x is distributed among the processes (Figure 8.1(c)), process P_i computes $y[i] = \Sigma_{j=0}^{n-1}(A[i, j] \times x[j])$ (lines 6 and 7 of Algorithm 8.1). As Figure 8.1(d) shows, the result vector y is stored exactly the way the starting vector x was stored.



(a) Initial partitioning of the matrix and the starting vector x

(b) Distribution of the full vector among all the processes by all-to-all broadcast



(c) Entire vector distributed to each process after the broadcast

(d) Final distribution of the matrix and the result vector y

Figure 8.1 Multiplication of an $n \times n$ matrix with an $n \times 1$ vector using rowwise block 1-D partitioning. For the one-row-per-process case, $p = n$.

Parallel Run Time Starting with one vector element per process, the all-to-all broadcast of the vector elements among n processes requires time $\Theta(n)$ on any architecture (Table 4.1). The multiplication of a single row of A with x is also performed by each process in time $\Theta(n)$. Thus, the entire procedure is completed by n processes in time $\Theta(n)$, resulting in a process-time product of $\Theta(n^2)$. The parallel algorithm is cost-optimal because the complexity of the serial algorithm is $\Theta(n^2)$.

Using Fewer than *n* Processes

Consider the case in which p processes are used such that $p < n$, and the matrix is partitioned among the processes by using block 1-D partitioning. Each process initially stores n/p complete rows of the matrix and a portion of the vector of size n/p. Since the vector

x must be multiplied with each row of the matrix, every process needs the entire vector (that is, all the portions belonging to separate processes). This again requires an all-to-all broadcast as shown in Figure 8.1(b) and (c). The all-to-all broadcast takes place among p processes and involves messages of size n/p. After this communication step, each process multiplies its n/p rows with the vector x to produce n/p elements of the result vector. Figure 8.1(d) shows that the result vector y is distributed in the same format as that of the starting vector x.

Parallel Run Time According to Table 4.1, an all-to-all broadcast of messages of size n/p among p processes takes time $t_s \log p + t_w(n/p)(p-1)$. For large p, this can be approximated by $t_s \log p + t_w n$. After the communication, each process spends time n^2/p multiplying its n/p rows with the vector. Thus, the parallel run time of this procedure is

$$T_P = \frac{n^2}{p} + t_s \log p + t_w n. \tag{8.2}$$

The process-time product for this parallel formulation is $n^2 + t_s p \log p + t_w np$. The algorithm is cost-optimal for $p = O(n)$.

Scalability Analysis We now derive the isoefficiency function for matrix-vector multiplication along the lines of the analysis in Section 5.4.2 by considering the terms of the overhead function one at a time. Consider the parallel run time given by Equation 8.2 for the hypercube architecture. The relation $T_o = pT_P - W$ gives the following expression for the overhead function of matrix-vector multiplication on a hypercube with block 1-D partitioning:

$$T_o = t_s p \log p + t_w np. \tag{8.3}$$

Recall from Chapter 5 that the central relation that determines the isoefficiency function of a parallel algorithm is $W = KT_o$ (Equation 5.14), where $K = E/(1-E)$ and E is the desired efficiency. Rewriting this relation for matrix-vector multiplication, first with only the t_s term of T_o,

$$W = Kt_s p \log p. \tag{8.4}$$

Equation 8.4 gives the isoefficiency term with respect to message startup time. Similarly, for the t_w term of the overhead function,

$$W = Kt_w np.$$

Since $W = n^2$ (Equation 8.1), we derive an expression for W in terms of p, K, and t_w (that is, the isoefficiency function due to t_w) as follows:

$$\begin{aligned} n^2 &= Kt_w np, \\ n &= Kt_w p, \\ n^2 &= K^2 t_w^2 p^2, \\ W &= K^2 t_w^2 p^2. \end{aligned} \tag{8.5}$$

Now consider the degree of concurrency of this parallel algorithm. Using 1-D partitioning, a maximum of n processes can be used to multiply an $n \times n$ matrix with an $n \times 1$ vector. In other words, p is $O(n)$, which yields the following condition:

$$\begin{aligned} n &= \Omega(p), \\ n^2 &= \Omega(p^2), \\ W &= \Omega(p^2). \end{aligned} \tag{8.6}$$

The overall asymptotic isoefficiency function can be determined by comparing Equations 8.4, 8.5, and 8.6. Among the three, Equations 8.5 and 8.6 give the highest asymptotic rate at which the problem size must increase with the number of processes to maintain a fixed efficiency. This rate of $\Theta(p^2)$ is the asymptotic isoefficiency function of the parallel matrix-vector multiplication algorithm with 1-D partitioning.

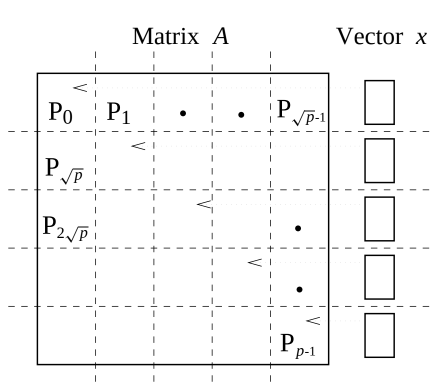
8.1.2 2-D Partitioning

This section discusses parallel matrix-vector multiplication for the case in which the matrix is distributed among the processes using a block 2-D partitioning. Figure 8.2 shows the distribution of the matrix and the distribution and movement of vectors among the processes.

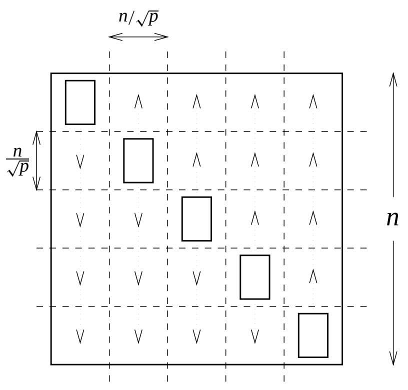
One Element Per Process

We start with the simple case in which an $n \times n$ matrix is partitioned among n^2 processes such that each process owns a single element. The $n \times 1$ vector x is distributed only in the last column of n processes, each of which owns one element of the vector. Since the algorithm multiplies the elements of the vector x with the corresponding elements in each row of the matrix, the vector must be distributed such that the ith element of the vector is available to the ith element of each row of the matrix. The communication steps for this are shown in Figure 8.2(a) and (b). Notice the similarity of Figure 8.2 to Figure 8.1. Before the multiplication, the elements of the matrix and the vector must be in the same relative locations as in Figure 8.1(c). However, the vector communication steps differ between various partitioning strategies. With 1-D partitioning, the elements of the vector cross only the horizontal partition-boundaries (Figure 8.1), but for 2-D partitioning, the vector elements cross both horizontal and vertical partition-boundaries (Figure 8.2).

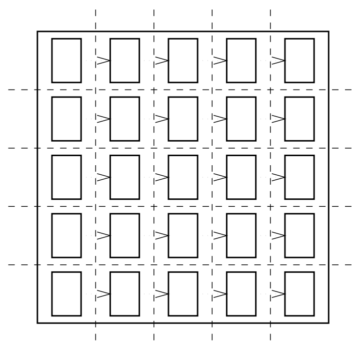
As Figure 8.2(a) shows, the first communication step for the 2-D partitioning aligns the vector x along the principal diagonal of the matrix. Often, the vector is stored along the diagonal instead of the last column, in which case this step is not required. The second step copies the vector elements from each diagonal process to all the processes in the corresponding column. As Figure 8.2(b) shows, this step consists of n simultaneous one-to-all broadcast operations, one in each column of processes. After these two communication steps, each process multiplies its matrix element with the corresponding element of x. To



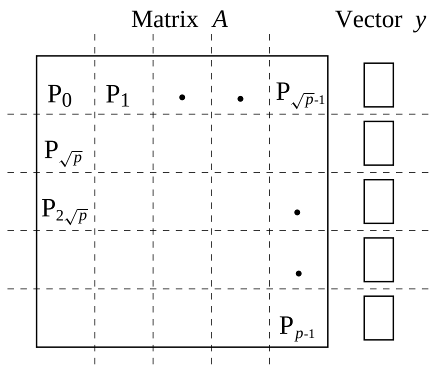
(a) Initial data distribution and communication steps to align the vector along the diagonal



(b) One-to-all broadcast of portions of the vector along the process columns



(c) All-to-one reduction of partial results



(d) Final distribution of the result vector

Figure 8.2 Matrix-vector multiplication with block 2-D partitioning. For the one-element-per-process case, $p = n^2$ if the matrix size is $n \times n$.

obtain the result vector y, the products computed for each row must be added, leaving the sums in the last column of processes. Figure 8.2(c) shows this step, which requires an all-to-one reduction (Section 4.1) in each row with the last process of the row as the destination. The parallel matrix-vector multiplication is complete after the reduction step.

Parallel Run Time Three basic communication operations are used in this algorithm: one-to-one communication to align the vector along the main diagonal, one-to-all broadcast of each vector element among the n processes of each column, and all-to-one reduction in each row. Each of these operations takes time $\Theta(\log n)$. Since each process performs a single multiplication in constant time, the overall parallel run time of this algorithm is $\Theta(n)$. The cost (process-time product) is $\Theta(n^2 \log n)$; hence, the algorithm is not cost-optimal.

Using Fewer than n^2 Processes

A cost-optimal parallel implementation of matrix-vector multiplication with block 2-D partitioning of the matrix can be obtained if the granularity of computation at each process is increased by using fewer than n^2 processes.

Consider a logical two-dimensional mesh of p processes in which each process owns an $(n/\sqrt{p}) \times (n/\sqrt{p})$ block of the matrix. The vector is distributed in portions of $n/\sqrt{p}$ elements in the last process-column only. Figure 8.2 also illustrates the initial data-mapping and the various communication steps for this case. The entire vector must be distributed on each row of processes before the multiplication can be performed. First, the vector is aligned along the main diagonal. For this, each process in the rightmost column sends its $n/\sqrt{p}$ vector elements to the diagonal process in its row. Then a columnwise one-to-all broadcast of these $n/\sqrt{p}$ elements takes place. Each process then performs n^2/p multiplications and locally adds the $n/\sqrt{p}$ sets of products. At the end of this step, as shown in Figure 8.2(c), each process has $n/\sqrt{p}$ partial sums that must be accumulated along each row to obtain the result vector. Hence, the last step of the algorithm is an all-to-one reduction of the $n/\sqrt{p}$ values in each row, with the rightmost process of the row as the destination.

Parallel Run Time The first step of sending a message of size $n/\sqrt{p}$ from the rightmost process of a row to the diagonal process (Figure 8.2(a)) takes time $t_s + t_w n/\sqrt{p}$. We can perform the columnwise one-to-all broadcast in at most time $(t_s + t_w n/\sqrt{p}) \log(\sqrt{p})$ by using the procedure described in Section 4.1.3. Ignoring the time to perform additions, the final rowwise all-to-one reduction also takes the same amount of time. Assuming that a multiplication and addition pair takes unit time, each process spends approximately n^2/p time in computation. Thus, the parallel run time for this procedure is as follows:

$$\begin{aligned}
T_P &= \overbrace{n^2/p}^{\text{computation}} + \overbrace{t_s + t_w n/\sqrt{p}}^{\text{aligning the vector}} + \\
&\quad \overbrace{(t_s + t_w n/\sqrt{p}) \log(\sqrt{p})}^{\text{columnwise one-to-all broadcast}} + \overbrace{(t_s + t_w n/\sqrt{p}) \log(\sqrt{p})}^{\text{all-to-one reduction}} \\
&\approx \frac{n^2}{p} + t_s \log p + t_w \frac{n}{\sqrt{p}} \log p \qquad (8.7)
\end{aligned}$$

Scalability Analysis By using Equations 8.1 and 8.7, and applying the relation $T_o = pT_p - W$ (Equation 5.1), we get the following expression for the overhead function of this parallel algorithm:

$$T_o = t_s p \log p + t_w n \sqrt{p} \log p. \qquad (8.8)$$

We now perform an approximate isoefficiency analysis along the lines of Section 5.4.2 by considering the terms of the overhead function one at a time (see Problem 8.4 for a more precise isoefficiency analysis). For the t_s term of the overhead function, Equation 5.14 yields

$$W = K t_s p \log p. \tag{8.9}$$

Equation 8.9 gives the isoefficiency term with respect to the message startup time. We can obtain the isoefficiency function due to t_w by balancing the term $t_w n \sqrt{p} \log p$ with the problem size n^2. Using the isoefficiency relation of Equation 5.14, we get the following:

$$\begin{aligned} W = n^2 &= K t_w n \sqrt{p} \log p, \\ n &= K t_w \sqrt{p} \log p, \\ n^2 &= K^2 t_w^2 p \log^2 p, \\ W &= K^2 t_w^2 p \log^2 p. \end{aligned} \tag{8.10}$$

Finally, considering that the degree of concurrency of 2-D partitioning is n^2 (that is, a maximum of n^2 processes can be used), we arrive at the following relation:

$$\begin{aligned} p &= O(n^2), \\ n^2 &= \Omega(p), \\ W &= \Omega(p). \end{aligned} \tag{8.11}$$

Among Equations 8.9, 8.10, and 8.11, the one with the largest right-hand side expression determines the overall isoefficiency function of this parallel algorithm. To simplify the analysis, we ignore the impact of the constants and consider only the asymptotic rate of the growth of problem size that is necessary to maintain constant efficiency. The asymptotic isoefficiency term due to t_w (Equation 8.10) clearly dominates the ones due to t_s (Equation 8.9) and due to concurrency (Equation 8.11). Therefore, the overall asymptotic isoefficiency function is given by $\Theta(p \log^2 p)$.

The isoefficiency function also determines the criterion for cost-optimality (Section 5.4.3). With an isoefficiency function of $\Theta(p \log^2 p)$, the maximum number of processes that can be used cost-optimally for a given problem size W is determined by the following relations:

$$\begin{aligned} p \log^2 p &= O(n^2), \qquad (8.12) \\ \log p + 2 \log\log p &= O(\log n). \end{aligned}$$

Ignoring the lower-order terms,

$$\log p = O(\log n).$$

Substituting $\log n$ for $\log p$ in Equation 8.12,

$$\begin{aligned} p \log^2 n &= O(n^2), \\ p &= O\left(\frac{n^2}{\log^2 n}\right). \end{aligned} \tag{8.13}$$

The right-hand side of Equation 8.13 gives an asymptotic upper bound on the number of processes that can be used cost-optimally for an $n \times n$ matrix-vector multiplication with a 2-D partitioning of the matrix.

Comparison of 1-D and 2-D Partitionings

A comparison of Equations 8.2 and 8.7 shows that matrix-vector multiplication is faster with block 2-D partitioning of the matrix than with block 1-D partitioning for the same number of processes. If the number of processes is greater than n, then the 1-D partitioning cannot be used. However, even if the number of processes is less than or equal to n, the analysis in this section suggests that 2-D partitioning is preferable.

Among the two partitioning schemes, 2-D partitioning has a better (smaller) asymptotic isoefficiency function. Thus, matrix-vector multiplication is more scalable with 2-D partitioning; that is, it can deliver the same efficiency on more processes with 2-D partitioning than with 1-D partitioning.

8.2 Matrix-Matrix Multiplication

This section discusses parallel algorithms for multiplying two $n \times n$ dense, square matrices A and B to yield the product matrix $C = A \times B$. All parallel matrix multiplication algorithms in this chapter are based on the conventional serial algorithm shown in Algorithm 8.2. If we assume that an addition and multiplication pair (line 8) takes unit time, then the sequential run time of this algorithm is n^3. Matrix multiplication algorithms with better asymptotic sequential complexities are available, for example Strassen's algorithm. However, for the sake of simplicity, in this book we assume that the conventional algorithm is the best available serial algorithm. Problem 8.5 explores the performance of parallel matrix multiplication regarding Strassen's method as the base algorithm.

```
1.  procedure MAT_MULT ($A$, $B$, $C$)
2.  begin
3.      for $i := 0$ to $n - 1$ do
4.          for $j := 0$ to $n - 1$ do
5.              begin
6.                  $C[i, j] := 0$;
7.                  for $k := 0$ to $n - 1$ do
8.                      $C[i, j] := C[i, j] + A[i, k] \times B[k, j]$;
9.              endfor;
10. end MAT_MULT
```

Algorithm 8.2 The conventional serial algorithm for multiplication of two $n \times n$ matrices.

```
1.  procedure BLOCK_MAT_MULT ($A$, $B$, $C$)
2.  begin
3.      for $i := 0$ to $q - 1$ do
4.          for $j := 0$ to $q - 1$ do
5.              begin
6.                  Initialize all elements of $C_{i,j}$ to zero;
7.                  for $k := 0$ to $q - 1$ do
8.                      $C_{i,j} := C_{i,j} + A_{i,k} \times B_{k,j}$;
9.              endfor;
10. end BLOCK_MAT_MULT
```

Algorithm 8.3 The block matrix multiplication algorithm for $n \times n$ matrices with a block size of $(n/q) \times (n/q)$.

A concept that is useful in matrix multiplication as well as in a variety of other matrix algorithms is that of block matrix operations. We can often express a matrix computation involving scalar algebraic operations on all its elements in terms of identical matrix algebraic operations on blocks or submatrices of the original matrix. Such algebraic operations on the submatrices are called ***block matrix operations***. For example, an $n \times n$ matrix A can be regarded as a $q \times q$ array of blocks $A_{i,j}$ $(0 \leq i, j < q)$ such that each block is an $(n/q) \times (n/q)$ submatrix. The matrix multiplication algorithm in Algorithm 8.2 can then be rewritten as Algorithm 8.3, in which the multiplication and addition operations on line 8 are matrix multiplication and matrix addition, respectively. Not only are the final results of Algorithm 8.2 and 8.3 identical, but so are the total numbers of scalar additions and multiplications performed by each. Algorithm 8.2 performs n^3 additions and multiplications, and Algorithm 8.3 performs q^3 matrix multiplications, each involving $(n/q) \times (n/q)$ matrices and requiring $(n/q)^3$ additions and multiplications. We can use p processes to implement the block version of matrix multiplication in parallel by choosing $q = \sqrt{p}$ and computing a distinct $C_{i,j}$ block at each process.

In the following sections, we describe a few ways of parallelizing Algorithm 8.3. Each of the following parallel matrix multiplication algorithms uses a block 2-D partitioning of the matrices.

8.2.1 A Simple Parallel Algorithm

Consider two $n \times n$ matrices A and B partitioned into p blocks $A_{i,j}$ and $B_{i,j}$ $(0 \leq i, j < \sqrt{p})$ of size $(n/\sqrt{p}) \times (n/\sqrt{p})$ each. These blocks are mapped onto a $\sqrt{p} \times \sqrt{p}$ logical mesh of processes. The processes are labeled from $P_{0,0}$ to $P_{\sqrt{p}-1,\sqrt{p}-1}$. Process $P_{i,j}$ initially stores $A_{i,j}$ and $B_{i,j}$ and computes block $C_{i,j}$ of the result matrix. Computing submatrix $C_{i,j}$ requires all submatrices $A_{i,k}$ and $B_{k,j}$ for $0 \leq k < \sqrt{p}$. To acquire all the required blocks, an all-to-all broadcast of matrix A's blocks is performed in each row of processes, and an all-to-all broadcast of matrix B's blocks is performed in each column.

After $P_{i,j}$ acquires $A_{i,0}, A_{i,1}, \ldots, A_{i,\sqrt{p}-1}$ and $B_{0,j}, B_{1,j}, \ldots, B_{\sqrt{p}-1,j}$, it performs the submatrix multiplication and addition step of lines 7 and 8 in Algorithm 8.3.

Performance and Scalability Analysis The algorithm requires two all-to-all broadcast steps (each consisting of $\sqrt{p}$ concurrent broadcasts in all rows and columns of the process mesh) among groups of $\sqrt{p}$ processes. The messages consist of submatrices of n^2/p elements. From Table 4.1, the total communication time is $2(t_s \log(\sqrt{p}) + t_w(n^2/p)(\sqrt{p}-1))$. After the communication step, each process computes a submatrix $C_{i,j}$, which requires $\sqrt{p}$ multiplications of $(n/\sqrt{p}) \times (n/\sqrt{p})$ submatrices (lines 7 and 8 of Algorithm 8.3 with $q = \sqrt{p}$). This takes a total of time $\sqrt{p} \times (n/\sqrt{p})^3 = n^3/p$. Thus, the parallel run time is approximately

$$T_P = \frac{n^3}{p} + t_s \log p + 2t_w \frac{n^2}{\sqrt{p}}. \tag{8.14}$$

The process-time product is $n^3 + t_s p \log p + 2t_w n^2 \sqrt{p}$, and the parallel algorithm is cost-optimal for $p = O(n^2)$.

The isoefficiency functions due to t_s and t_w are $t_s p \log p$ and $8(t_w)^3 p^{3/2}$, respectively. Hence, the overall isoefficiency function due to the communication overhead is $\Theta(p^{3/2})$. This algorithm can use a maximum of n^2 processes; hence, $p \leq n^2$ or $n^3 \geq p^{3/2}$. Therefore, the isoefficiency function due to concurrency is also $\Theta(p^{3/2})$.

A notable drawback of this algorithm is its excessive memory requirements. At the end of the communication phase, each process has $\sqrt{p}$ blocks of both matrices A and B. Since each block requires $\Theta(n^2/p)$ memory, each process requires $\Theta(n^2/\sqrt{p})$ memory. The total memory requirement over all the processes is $\Theta(n^2\sqrt{p})$, which is $\sqrt{p}$ times the memory requirement of the sequential algorithm.

8.2.2 Cannon's Algorithm

Cannon's algorithm is a memory-efficient version of the simple algorithm presented in Section 8.2.1. To study this algorithm, we again partition matrices A and B into p square blocks. We label the processes from $P_{0,0}$ to $P_{\sqrt{p}-1,\sqrt{p}-1}$, and initially assign submatrices $A_{i,j}$ and $B_{i,j}$ to process $P_{i,j}$. Although every process in the ith row requires all $\sqrt{p}$ submatrices $A_{i,k}$ $(0 \leq k < \sqrt{p})$, it is possible to schedule the computations of the $\sqrt{p}$ processes of the ith row such that, at any given time, each process is using a different $A_{i,k}$. These blocks can be systematically rotated among the processes after every submatrix multiplication so that every process gets a fresh $A_{i,k}$ after each rotation. If an identical schedule is applied to the columns, then no process holds more than one block of each matrix at any time, and the total memory requirement of the algorithm over all the processes is $\Theta(n^2)$. Cannon's algorithm is based on this idea. The scheduling for the multiplication of submatrices on separate processes in Cannon's algorithm is illustrated in Figure 8.3 for 16 processes.

The first communication step of the algorithm aligns the blocks of A and B in such a way that each process multiplies its local submatrices. As Figure 8.3(a) shows, this align-

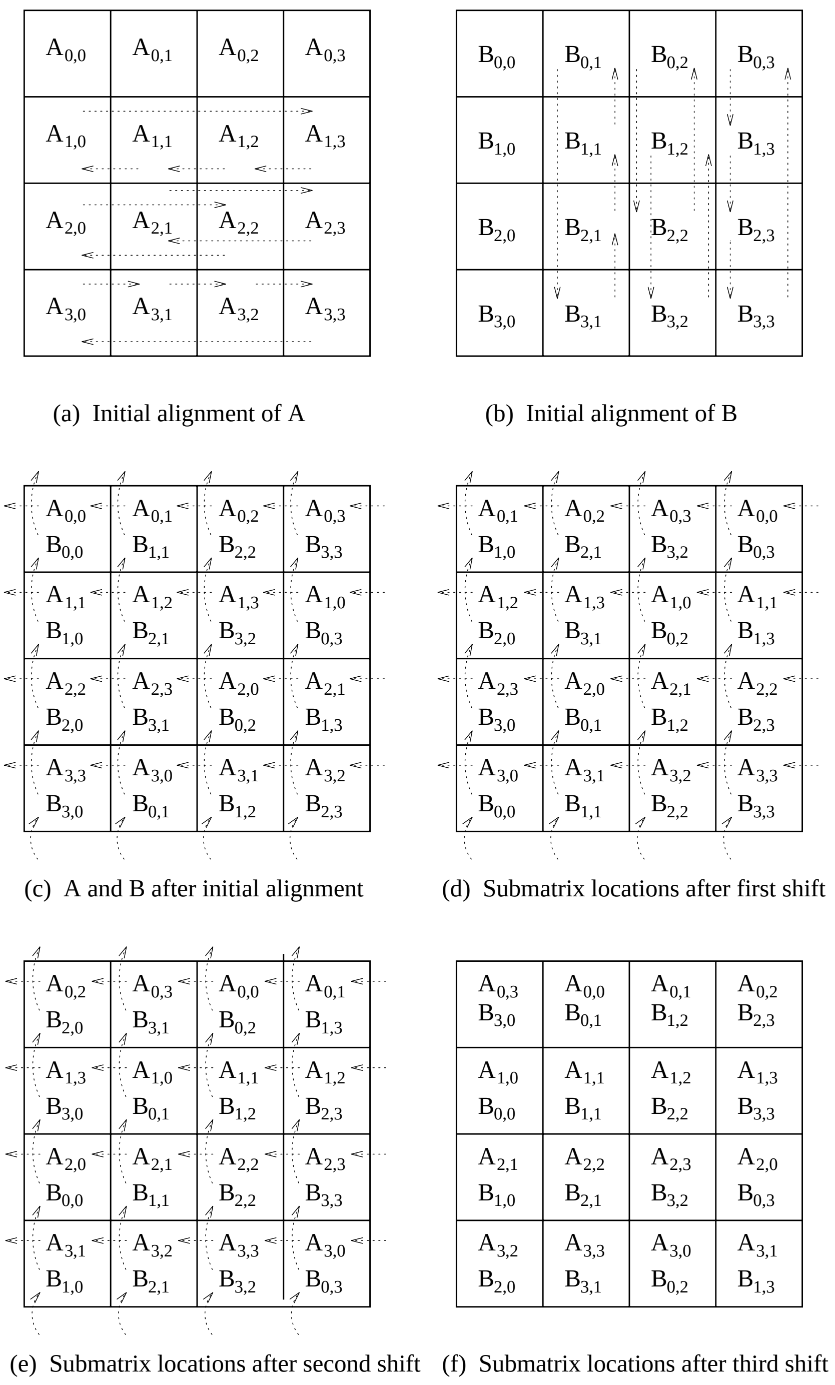

$A_{0,0}$ $A_{0,1}$ $A_{0,2}$ $A_{0,3}$
$A_{1,0}$ $A_{1,1}$ $A_{1,2}$ $A_{1,3}$
$A_{2,0}$ $A_{2,1}$ $A_{2,2}$ $A_{2,3}$
$A_{3,0}$ $A_{3,1}$ $A_{3,2}$ $A_{3,3}$
(a) Initial alignment of A
$B_{0,0}$ $B_{0,1}$ $B_{0,2}$ $B_{0,3}$
$B_{1,0}$ $B_{1,1}$ $B_{1,2}$ $B_{1,3}$
$B_{2,0}$ $B_{2,1}$ $B_{2,2}$ $B_{2,3}$
$B_{3,0}$ $B_{3,1}$ $B_{3,2}$ $B_{3,3}$
(b) Initial alignment of B
$A_{0,0}$ $B_{0,0}$ $A_{0,1}$ $B_{1,1}$ $A_{0,2}$ $B_{2,2}$ $A_{0,3}$ $B_{3,3}$
$A_{1,1}$ $B_{1,0}$ $A_{1,2}$ $B_{2,1}$ $A_{1,3}$ $B_{3,2}$ $A_{1,0}$ $B_{0,3}$
$A_{2,2}$ $B_{2,0}$ $A_{2,3}$ $B_{3,1}$ $A_{2,0}$ $B_{0,2}$ $A_{2,1}$ $B_{1,3}$
$A_{3,3}$ $B_{3,0}$ $A_{3,0}$ $B_{0,1}$ $A_{3,1}$ $B_{1,2}$ $A_{3,2}$ $B_{2,3}$
(c) A and B after initial alignment
$A_{0,1}$ $B_{1,0}$ $A_{0,2}$ $B_{2,1}$ $A_{0,3}$ $B_{3,2}$ $A_{0,0}$ $B_{0,3}$
$A_{1,2}$ $B_{2,0}$ $A_{1,3}$ $B_{3,1}$ $A_{1,0}$ $B_{0,2}$ $A_{1,1}$ $B_{1,3}$
$A_{2,3}$ $B_{3,0}$ $A_{2,0}$ $B_{0,1}$ $A_{2,1}$ $B_{1,2}$ $A_{2,2}$ $B_{2,3}$
$A_{3,0}$ $B_{0,0}$ $A_{3,1}$ $B_{1,1}$ $A_{3,2}$ $B_{2,2}$ $A_{3,3}$ $B_{3,3}$
(d) Submatrix locations after first shift
$A_{0,2}$ $B_{2,0}$ $A_{0,3}$ $B_{3,1}$ $A_{0,0}$ $B_{0,2}$ $A_{0,1}$ $B_{1,3}$
$A_{1,3}$ $B_{3,0}$ $A_{1,0}$ $B_{0,1}$ $A_{1,1}$ $B_{1,2}$ $A_{1,2}$ $B_{2,3}$
$A_{2,0}$ $B_{0,0}$ $A_{2,1}$ $B_{1,1}$ $A_{2,2}$ $B_{2,2}$ $A_{2,3}$ $B_{3,3}$
$A_{3,1}$ $B_{1,0}$ $A_{3,2}$ $B_{2,1}$ $A_{3,3}$ $B_{3,2}$ $A_{3,0}$ $B_{0,3}$
(e) Submatrix locations after second shift
$A_{0,3}$ $B_{3,0}$ $A_{0,0}$ $B_{0,1}$ $A_{0,1}$ $B_{1,2}$ $A_{0,2}$ $B_{2,3}$
$A_{1,0}$ $B_{0,0}$ $A_{1,1}$ $B_{1,1}$ $A_{1,2}$ $B_{2,2}$ $A_{1,3}$ $B_{3,3}$
$A_{2,1}$ $B_{1,0}$ $A_{2,2}$ $B_{2,1}$ $A_{2,3}$ $B_{3,2}$ $A_{2,0}$ $B_{0,3}$
$A_{3,2}$ $B_{2,0}$ $A_{3,3}$ $B_{3,1}$ $A_{3,0}$ $B_{0,2}$ $A_{3,1}$ $B_{1,3}$
(f) Submatrix locations after third shift

Figure 8.3 The communication steps in Cannon's algorithm on 16 processes.

ment is achieved for matrix A by shifting all submatrices $A_{i,j}$ to the left (with wraparound) by i steps. Similarly, as shown in Figure 8.3(b), all submatrices $B_{i,j}$ are shifted up (with wraparound) by j steps. These are circular shift operations (Section 4.6) in each row and column of processes, which leave process $\mathrm{P}_{i,j}$ with submatrices $A_{i,(j+i)\mathrm{mod}\sqrt{p}}$ and $B_{(i+j)\mathrm{mod}\sqrt{p},j}$. Figure 8.3(c) shows the blocks of A and B after the initial alignment, when each process is ready for the first submatrix multiplication. After a submatrix multiplication step, each block of A moves one step left and each block of B moves one step up (again with wraparound), as shown in Figure 8.3(d). A sequence of $\sqrt{p}$ such submatrix multiplications and single-step shifts pairs up each $A_{i,k}$ and $B_{k,j}$ for k $(0 \leq k < \sqrt{p})$ at $\mathrm{P}_{i,j}$. This completes the multiplication of matrices A and B.

Performance Analysis The initial alignment of the two matrices (Figure 8.3(a) and (b)) involves a rowwise and a columnwise circular shift. In any of these shifts, the maximum distance over which a block shifts is $\sqrt{p} - 1$. The two shift operations require a total of time $2(t_s + t_w n^2/p)$ (Table 4.1). Each of the $\sqrt{p}$ single-step shifts in the compute-and-shift phase of the algorithm takes time $t_s + t_w n^2/p$. Thus, the total communication time (for both matrices) during this phase of the algorithm is $2(t_s + t_w n^2/p)\sqrt{p}$. For large enough p on a network with sufficient bandwidth, the communication time for the initial alignment can be disregarded in comparison with the time spent in communication during the compute-and-shift phase.

Each process performs $\sqrt{p}$ multiplications of $(n/\sqrt{p}) \times (n/\sqrt{p})$ submatrices. Assuming that a multiplication and addition pair takes unit time, the total time that each process spends in computation is n^3/p. Thus, the approximate overall parallel run time of this algorithm is

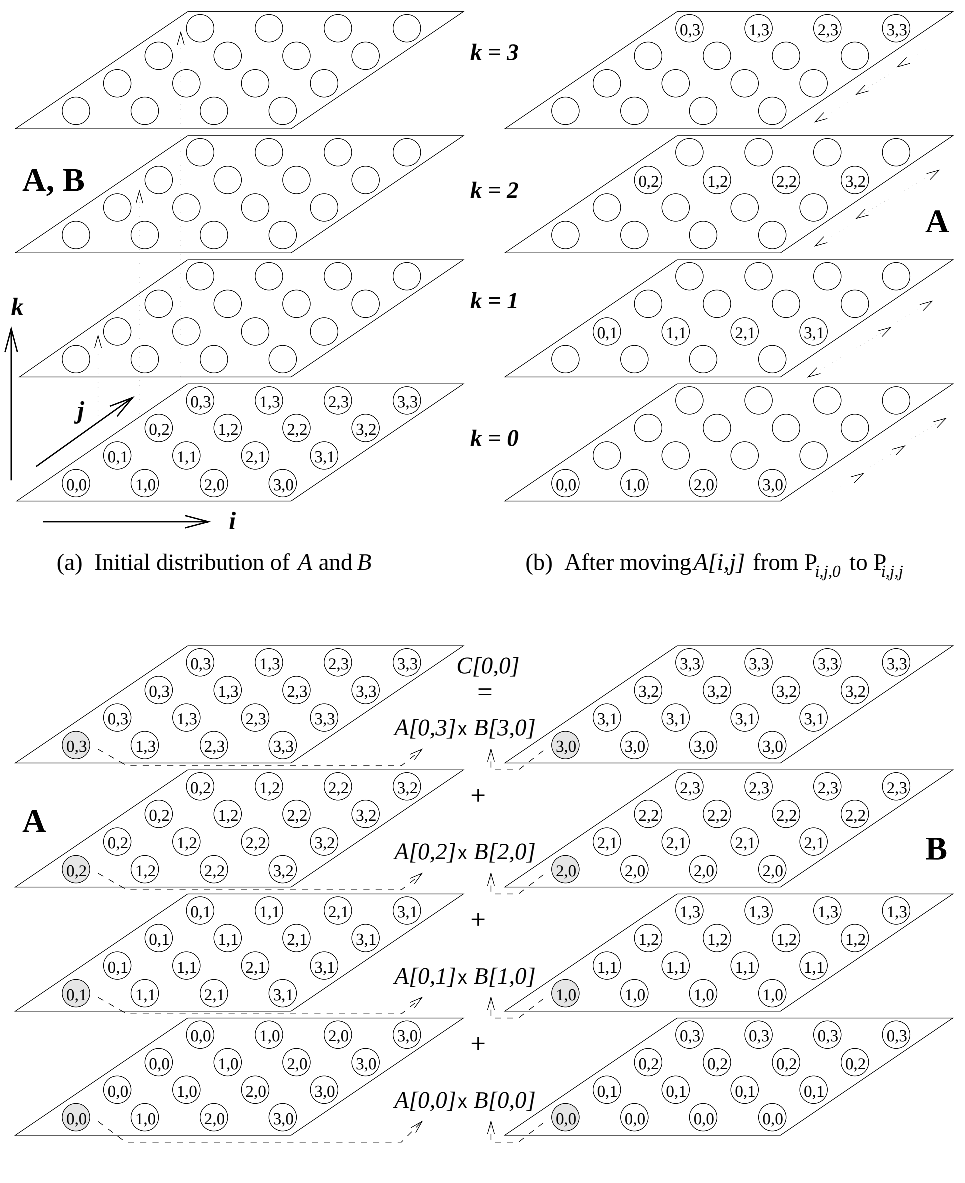
$$T_P = \frac{n^3}{p} + 2\sqrt{p}t_s + 2t_w \frac{n^2}{\sqrt{p}}. \tag{8.15}$$

The cost-optimality condition for Cannon's algorithm is identical to that for the simple algorithm presented in Section 8.2.1. As in the simple algorithm, the isoefficiency function of Cannon's algorithm is $\Theta(p^{3/2})$.

8.2.3 The DNS Algorithm

The matrix multiplication algorithms presented so far use block 2-D partitioning of the input and the output matrices and use a maximum of n^2 processes for $n \times n$ matrices. As a result, these algorithms have a parallel run time of $\Omega(n)$ because there are $\Theta(n^3)$ operations in the serial algorithm. We now present a parallel algorithm based on partitioning intermediate data that can use up to n^3 processes and that performs matrix multiplication in time $\Theta(\log n)$ by using $\Omega(n^3/\log n)$ processes. This algorithm is known as the DNS algorithm because it is due to Dekel, Nassimi, and Sahni.

We first introduce the basic idea, without concern for inter-process communication. Assume that n^3 processes are available for multiplying two $n \times n$ matrices. These processes are arranged in a three-dimensional $n \times n \times n$ logical array. Since the matrix multiplication



(a) Initial distribution of A and B

(b) After moving $A[i,j]$ from $P_{i,j,0}$ to $P_{i,j,j}$

(c) After broadcasting $A[i,j]$ along j axis

(d) Corresponding distribution of B

Figure 8.4 The communication steps in the DNS algorithm while multiplying 4×4 matrices A and B on 64 processes. The shaded processes in part (c) store elements of the first row of A and the shaded processes in part (d) store elements of the first column of B.

algorithm performs n^3 scalar multiplications, each of the n^3 processes is assigned a single scalar multiplication. The processes are labeled according to their location in the array, and the multiplication $A[i, k] \times B[k, j]$ is assigned to process $P_{i,j,k}$ $(0 \leq i, j, k < n)$. After each process performs a single multiplication, the contents of $P_{i,j,0}$, $P_{i,j,1}$, ..., $P_{i,j,n-1}$ are added to obtain $C[i, j]$. The additions for all $C[i, j]$ can be carried out simultaneously in $\log n$ steps each. Thus, it takes one step to multiply and $\log n$ steps to add; that is, it takes time $\Theta(\log n)$ to multiply the $n \times n$ matrices by this algorithm.

We now describe a practical parallel implementation of matrix multiplication based on this idea. As Figure 8.4 shows, the process arrangement can be visualized as n planes of $n \times n$ processes each. Each plane corresponds to a different value of k. Initially, as shown in Figure 8.4(a), the matrices are distributed among the n^2 processes of the plane corresponding to $k = 0$ at the base of the three-dimensional process array. Process $P_{i,j,0}$ initially owns $A[i, j]$ and $B[i, j]$.

The vertical column of processes $P_{i,j,*}$ computes the dot product of row $A[i, *]$ and column $B[*, j]$. Therefore, rows of A and columns of B need to be moved appropriately so that each vertical column of processes $P_{i,j,*}$ has row $A[i, *]$ and column $B[*, j]$. More precisely, process $P_{i,j,k}$ should have $A[i, k]$ and $B[k, j]$.

The communication pattern for distributing the elements of matrix A among the processes is shown in Figure 8.4(a)–(c). First, each column of A moves to a different plane such that the jth column occupies the same position in the plane corresponding to $k = j$ as it initially did in the plane corresponding to $k = 0$. The distribution of A after moving $A[i, j]$ from $P_{i,j,0}$ to $P_{i,j,j}$ is shown in Figure 8.4(b). Now all the columns of A are replicated n times in their respective planes by a parallel one-to-all broadcast along the j axis. The result of this step is shown in Figure 8.4(c), in which the n processes $P_{i,0,j}$, $P_{i,1,j}$, ..., $P_{i,n-1,j}$ receive a copy of $A[i, j]$ from $P_{i,j,j}$. At this point, each vertical column of processes $P_{i,j,*}$ has row $A[i, *]$. More precisely, process $P_{i,j,k}$ has $A[i, k]$.

For matrix B, the communication steps are similar, but the roles of i and j in process subscripts are switched. In the first one-to-one communication step, $B[i, j]$ is moved from $P_{i,j,0}$ to $P_{i,j,i}$. Then it is broadcast from $P_{i,j,i}$ among $P_{0,j,i}$, $P_{1,j,i}$, ..., $P_{n-1,j,i}$. The distribution of B after this one-to-all broadcast along the i axis is shown in Figure 8.4(d). At this point, each vertical column of processes $P_{i,j,*}$ has column $B[*, j]$. Now process $P_{i,j,k}$ has $B[k, j]$, in addition to $A[i, k]$.

After these communication steps, $A[i, k]$ and $B[k, j]$ are multiplied at $P_{i,j,k}$. Now each element $C[i, j]$ of the product matrix is obtained by an all-to-one reduction along the k axis. During this step, process $P_{i,j,0}$ accumulates the results of the multiplication from processes $P_{i,j,1}$, ..., $P_{i,j,n-1}$. Figure 8.4 shows this step for $C[0, 0]$.

The DNS algorithm has three main communication steps: (1) moving the columns of A and the rows of B to their respective planes, (2) performing one-to-all broadcast along the j axis for A and along the i axis for B, and (3) all-to-one reduction along the k axis. All these operations are performed within groups of n processes and take time $\Theta(\log n)$. Thus, the parallel run time for multiplying two $n \times n$ matrices using the DNS algorithm on n^3 processes is $\Theta(\log n)$.

DNS Algorithm with Fewer than n^3 Processes

The DNS algorithm is not cost-optimal for n^3 processes, since its process-time product of $\Theta(n^3 \log n)$ exceeds the $\Theta(n^3)$ sequential complexity of matrix multiplication. We now present a cost-optimal version of this algorithm that uses fewer than n^3 processes. Another variant of the DNS algorithm that uses fewer than n^3 processes is described in Problem 8.6.

Assume that the number of processes p is equal to q^3 for some $q < n$. To implement the DNS algorithm, the two matrices are partitioned into blocks of size $(n/q) \times (n/q)$. Each matrix can thus be regarded as a $q \times q$ two-dimensional square array of blocks. The implementation of this algorithm on q^3 processes is very similar to that on n^3 processes. The only difference is that now we operate on blocks rather than on individual elements. Since $1 \le q \le n$, the number of processes can vary between 1 and n^3.

Performance Analysis The first one-to-one communication step is performed for both A and B, and takes time $t_s + t_w(n/q)^2$ for each matrix. The second step of one-to-all broadcast is also performed for both matrices and takes time $t_s \log q + t_w(n/q)^2 \log q$ for each matrix. The final all-to-one reduction is performed only once (for matrix C) and takes time $t_s \log q + t_w(n/q)^2 \log q$. The multiplication of $(n/q) \times (n/q)$ submatrices by each process takes time $(n/q)^3$. We can ignore the communication time for the first one-to-one communication step because it is much smaller than the communication time of one-to-all broadcasts and all-to-one reduction. We can also ignore the computation time for addition in the final reduction phase because it is of a smaller order of magnitude than the computation time for multiplying the submatrices. With these assumptions, we get the following approximate expression for the parallel run time of the DNS algorithm:

$$T_P \approx \left(\frac{n}{q}\right)^3 + 3t_s \log q + 3t_w \left(\frac{n}{q}\right)^2 \log q$$

Since $q = p^{1/3}$, we get

$$T_P = \frac{n^3}{p} + t_s \log p + t_w \frac{n^2}{p^{2/3}} \log p. \tag{8.16}$$

The total cost of this parallel algorithm is $n^3 + t_s p \log p + t_w n^2 p^{1/3} \log p$. The isoefficiency function is $\Theta(p(\log p)^3)$. The algorithm is cost-optimal for $n^3 = \Omega(p(\log p)^3)$, or $p = O(n^3/(\log n)^3)$.

8.3 Solving a System of Linear Equations

This section discusses the problem of solving a system of linear equations of the form

$$\begin{array}{ccccccccc}
a_{0,0}x_0 & + & a_{0,1}x_1 & + & \cdots & + & a_{0,n-1}x_{n-1} & = & b_0, \\
a_{1,0}x_0 & + & a_{1,1}x_1 & + & \cdots & + & a_{1,n-1}x_{n-1} & = & b_1,
\end{array}$$

$$\begin{array}{ccccccccc} \vdots & & \vdots & & & & \vdots & & \vdots \\ a_{n-1,0}x_0 & + & a_{n-1,1}x_1 & + & \cdots & + & a_{n-1,n-1}x_{n-1} & = & b_{n-1}. \end{array}$$

In matrix notation, this system is written as $Ax = b$. Here A is a dense $n \times n$ matrix of coefficients such that $A[i, j] = a_{i,j}$, b is an $n \times 1$ vector $[b_0, b_1, \ldots, b_{n-1}]^T$, and x is the desired solution vector $[x_0, x_1, \ldots, x_{n-1}]^T$. We will make all subsequent references to $a_{i,j}$ by $A[i, j]$ and x_i by $x[i]$.

A system of equations $Ax = b$ is usually solved in two stages. First, through a series of algebraic manipulations, the original system of equations is reduced to an upper-triangular system of the form

$$\begin{array}{ccccccccccc} x_0 & + & u_{0,1}x_1 & + & u_{0,2}x_2 & + & \cdots & + & u_{0,n-1}x_{n-1} & = & y_0, \\ & & x_1 & + & u_{1,2}x_2 & + & \cdots & + & u_{1,n-1}x_{n-1} & = & y_1, \\ & & & & & & & & \vdots & & \vdots \\ & & & & & & & & x_{n-1} & = & y_{n-1}. \end{array}$$

We write this as $Ux = y$, where U is a unit upper-triangular matrix – one in which all subdiagonal entries are zero and all principal diagonal entries are equal to one. Formally, $U[i, j] = 0$ if $i > j$, otherwise $U[i, j] = u_{i,j}$. Furthermore, $U[i, i] = 1$ for $0 \le i < n$. In the second stage of solving a system of linear equations, the upper-triangular system is solved for the variables in reverse order from $x[n-1]$ to $x[0]$ by a procedure known as ***back-substitution*** (Section 8.3.3).

We discuss parallel formulations of the classical Gaussian elimination method for upper-triangularization in Sections 8.3.1 and 8.3.2. In Section 8.3.1, we describe a straightforward Gaussian elimination algorithm assuming that the coefficient matrix is nonsingular, and its rows and columns are permuted in a way that the algorithm is numerically stable. Section 8.3.2 discusses the case in which a numerically stable solution of the system of equations requires permuting the columns of the matrix during the execution of the Gaussian elimination algorithm.

Although we discuss Gaussian elimination in the context of upper-triangularization, a similar procedure can be used to factorize matrix A as the product of a lower-triangular matrix L and a unit upper-triangular matrix U so that $A = L \times U$. This factorization is commonly referred to as ***LU factorization***. Performing LU factorization (rather than upper-triangularization) is particularly useful if multiple systems of equations with the same left-hand side Ax need to be solved. Algorithm 3.3 gives a procedure for column-oriented LU factorization.

8.3.1 A Simple Gaussian Elimination Algorithm

The serial Gaussian elimination algorithm has three nested loops. Several variations of the algorithm exist, depending on the order in which the loops are arranged. Algorithm 8.4

```
1.  procedure GAUSSIAN_ELIMINATION ($A, b, y$)
2.  begin
3.      for $k := 0$ to $n - 1$ do                /* Outer loop */
4.      begin
5.          for $j := k + 1$ to $n - 1$ do
6.              $A[k, j] := A[k, j]/A[k, k]$;   /* Division step */
7.          $y[k] := b[k]/A[k, k]$;
8.          $A[k, k] := 1$;
9.          for $i := k + 1$ to $n - 1$ do
10.         begin
11.             for $j := k + 1$ to $n - 1$ do
12.                 $A[i, j] := A[i, j] - A[i, k] \times A[k, j]$; /* Elimination step */
13.             $b[i] := b[i] - A[i, k] \times y[k]$;
14.             $A[i, k] := 0$;
15.         endfor;          /* Line 9 */
16.     endfor;              /* Line 3 */
17. end GAUSSIAN_ELIMINATION
```

Algorithm 8.4 A serial Gaussian elimination algorithm that converts the system of linear equations $Ax = b$ to a unit upper-triangular system $Ux = y$. The matrix U occupies the upper-triangular locations of A. This algorithm assumes that $A[k, k] \neq 0$ when it is used as a divisor on lines 6 and 7.

shows one variation of Gaussian elimination, which we will adopt for parallel implementation in the remainder of this section. This program converts a system of linear equations $Ax = b$ to a unit upper-triangular system $Ux = y$. We assume that the matrix U shares storage with A and overwrites the upper-triangular portion of A. The element $A[k, j]$ computed on line 6 of Algorithm 8.4 is actually $U[k, j]$. Similarly, the element $A[k, k]$ equated to 1 on line 8 is $U[k, k]$. Algorithm 8.4 assumes that $A[k, k] \neq 0$ when it is used as a divisor on lines 6 and 7.

In this section, we will concentrate only on the operations on matrix A in Algorithm 8.4. The operations on vector b on lines 7 and 13 of the program are straightforward to implement. Hence, in the rest of the section, we will ignore these steps. If the steps on lines 7, 8, 13, and 14 are not performed, then Algorithm 8.4 leads to the LU factorization of A as a product $L \times U$. After the termination of the procedure, L is stored in the lower-triangular part of A, and U occupies the locations above the principal diagonal.

For k varying from 0 to $n - 1$, the Gaussian elimination procedure systematically eliminates variable $x[k]$ from equations $k + 1$ to $n - 1$ so that the matrix of coefficients becomes upper-triangular. As shown in Algorithm 8.4, in the kth iteration of the outer loop (starting on line 3), an appropriate multiple of the kth equation is subtracted from each of the equations $k + 1$ to $n - 1$ (loop starting on line 9). The multiples of the kth equation (or the kth row of matrix A) are chosen such that the kth coefficient becomes zero in equations $k + 1$

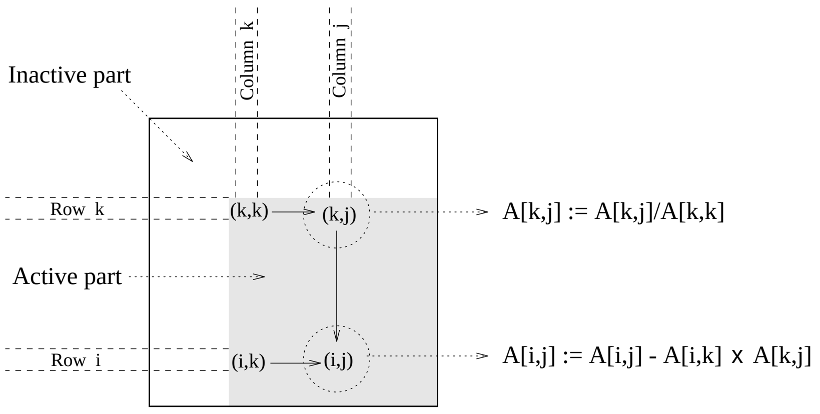


Figure 8.5 A typical computation in Gaussian elimination.

to $n - 1$ eliminating $x[k]$ from these equations. A typical computation of the Gaussian elimination procedure in the kth iteration of the outer loop is shown in Figure 8.5. The kth iteration of the outer loop does not involve any computation on rows 1 to $k - 1$ or columns 1 to $k - 1$. Thus, at this stage, only the lower-right $(n - k) \times (n - k)$ submatrix of A (the shaded portion in Figure 8.5) is computationally active.

Gaussian elimination involves approximately $n^2/2$ divisions (line 6) and approximately $(n^3/3) - (n^2/2)$ subtractions and multiplications (line 12). In this section, we assume that each scalar arithmetic operation takes unit time. With this assumption, the sequential run time of the procedure is approximately $2n^3/3$ (for large n); that is,

$$W = \frac{2}{3}n^3. \tag{8.17}$$

Parallel Implementation with 1-D Partitioning

We now consider a parallel implementation of Algorithm 8.4, in which the coefficient matrix is rowwise 1-D partitioned among the processes. A parallel implementation of this algorithm with columnwise 1-D partitioning is very similar, and its details can be worked out based on the implementation using rowwise 1-D partitioning (Problems 8.8 and 8.9).

We first consider the case in which one row is assigned to each process, and the $n \times n$ coefficient matrix A is partitioned along the rows among n processes labeled from P_0 to P_{n-1}. In this mapping, process P_i initially stores elements $A[i, j]$ for $0 \leq j < n$. Figure 8.6 illustrates this mapping of the matrix onto the processes for $n = 8$. The figure also illustrates the computation and communication that take place in the iteration of the outer loop when $k = 3$.

Algorithm 8.4 and Figure 8.5 show that $A[k, k + 1]$, $A[k, k + 2], \ldots, A[k, n - 1]$ are divided by $A[k, k]$ (line 6) at the beginning of the kth iteration. All matrix elements participating in this operation (shown by the shaded portion of the matrix in Figure 8.6(a))

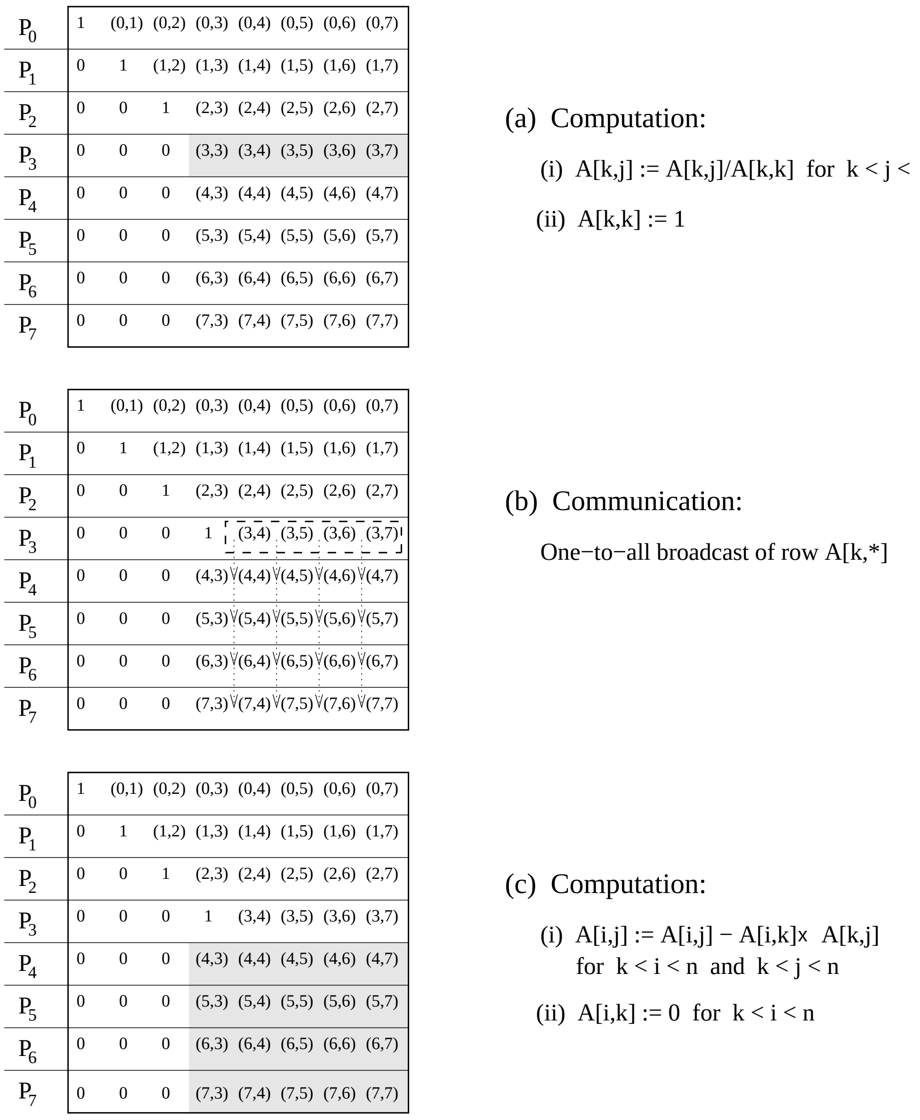

(a) Computation:
(i) A[k,j] := A[k,j]/A[k,k] for k < j <
(ii) A[k,k] := 1
(b) Communication:
One−to−all broadcast of row A[k,*]
(c) Computation:
(i) A[i,j] := A[i,j] − A[i,k]× A[k,j] for k < i < n and k < j < n
(ii) A[i,k] := 0 for k < i < n

Figure 8.6 Gaussian elimination steps during the iteration corresponding to $k = 3$ for an 8×8 matrix partitioned rowwise among eight processes.

belong to the same process. So this step does not require any communication. In the second computation step of the algorithm (the elimination step of line 12), the modified (after division) elements of the kth row are used by all other rows of the active part of the matrix. As Figure 8.6(b) shows, this requires a one-to-all broadcast of the active part of the kth row to the processes storing rows $k + 1$ to $n - 1$. Finally, the computation $A[i, j] := A[i, j] - A[i, k] \times A[k, j]$ takes place in the remaining active portion of the matrix, which is shown shaded in Figure 8.6(c).

The computation step corresponding to Figure 8.6(a) in the kth iteration requires $n - k - 1$ divisions at process P_k. Similarly, the computation step of Figure 8.6(c) involves $n - k - 1$ multiplications and subtractions in the kth iteration at all processes P_i, such that $k < i < n$. Assuming a single arithmetic operation takes unit time, the total time spent in computation in the kth iteration is $3(n - k - 1)$. Note that when P_k is performing the divisions, the remaining $p - 1$ processes are idle, and while processes $\mathrm{P}_{k+1}, \ldots, \mathrm{P}_{n-1}$ are performing the elimination step, processes $\mathrm{P}_0, \ldots, \mathrm{P}_k$ are idle. Thus, the total time spent during the computation steps shown in Figures 8.6(a) and (c) in this parallel implementation of Gaussian elimination is $3\Sigma_{k=0}^{n-1}(n - k - 1)$, which is equal to $3n(n - 1)/2$.

The communication step of Figure 8.6(b) takes time $(t_s + t_w(n-k-1)) \log n$ (Table 4.1). Hence, the total communication time over all iterations is $\Sigma_{k=0}^{n-1}(t_s + t_w(n - k - 1)) \log n$, which is equal to $t_s n \log n + t_w(n(n - 1)/2) \log n$. The overall parallel run time of this algorithm is

$$T_P = \frac{3}{2}n(n - 1) + t_s n \log n + \frac{1}{2}t_w n(n - 1) \log n. \qquad (8.18)$$

Since the number of processes is n, the cost, or the process-time product, is $\Theta(n^3 \log n)$ due to the term associated with t_w in Equation 8.18. This cost is asymptotically higher than the sequential run time of this algorithm (Equation 8.17). Hence, this parallel implementation is not cost-optimal.

Pipelined Communication and Computation We now present a parallel implementation of Gaussian elimination that is cost-optimal on n processes.

In the parallel Gaussian elimination algorithm just presented, the n iterations of the outer loop of Algorithm 8.4 execute sequentially. At any given time, all processes work on the same iteration. The $(k + 1)$th iteration starts only after all the computation and communication for the kth iteration is complete. The performance of the algorithm can be improved substantially if the processes work asynchronously; that is, no process waits for the others to finish an iteration before starting the next one. We call this the ***asynchronous*** or ***pipelined*** version of Gaussian elimination. Figure 8.7 illustrates the pipelined Algorithm 8.4 for a 5×5 matrix partitioned along the rows onto a logical linear array of five processes.

During the kth iteration of Algorithm 8.4, process P_k broadcasts part of the kth row of the matrix to processes $\mathrm{P}_{k+1}, \ldots, \mathrm{P}_{n-1}$ (Figure 8.6(b)). Assuming that the processes form a logical linear array, and P_{k+1} is the first process to receive the kth row from process P_k. Then process P_{k+1} must forward this data to P_{k+2}. However, after forwarding the kth row

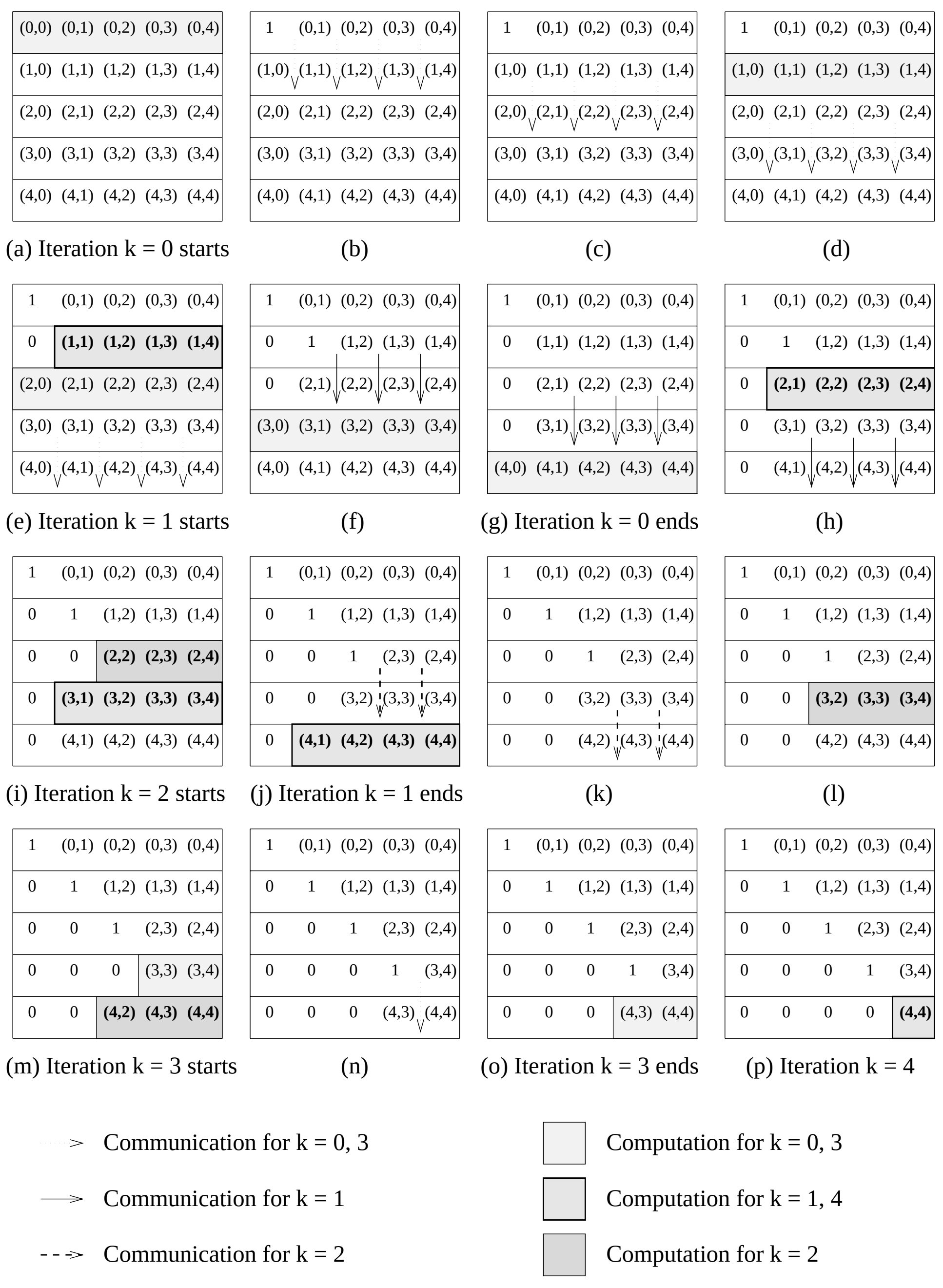


Figure 8.7 Pipelined Gaussian elimination on a 5 $\times$ 5 matrix partitioned with one row per process.

to P_{k+2}, process P_{k+1} need not wait to perform the elimination step (line 12) until all the processes up to P_{n-1} have received the kth row. Similarly, P_{k+2} can start its computation as soon as it has forwarded the kth row to P_{k+3}, and so on. Meanwhile, after completing the computation for the kth iteration, P_{k+1} can perform the division step (line 6), and start the broadcast of the $(k+1)$th row by sending it to P_{k+2}.

In pipelined Gaussian elimination, each process independently performs the following sequence of actions repeatedly until all n iterations are complete. For the sake of simplicity, we assume that steps (1) and (2) take the same amount of time (this assumption does not affect the analysis):

1. If a process has any data destined for other processes, it sends those data to the appropriate process.
2. If the process can perform some computation using the data it has, it does so.
3. Otherwise, the process waits to receive data to be used for one of the above actions.

Figure 8.7 shows the 16 steps in the pipelined parallel execution of Gaussian elimination for a 5×5 matrix partitioned along the rows among five processes. As Figure 8.7(a) shows, the first step is to perform the division on row 0 at process P_0. The modified row 0 is then sent to P_1 (Figure 8.7(b)), which forwards it to P_2 (Figure 8.7(c)). Now P_1 is free to perform the elimination step using row 0 (Figure 8.7(d)). In the next step (Figure 8.7(e)), P_2 performs the elimination step using row 0. In the same step, P_1, having finished its computation for iteration 0, starts the division step of iteration 1. At any given time, different stages of the same iteration can be active on different processes. For instance, in Figure 8.7(h), process P_2 performs the elimination step of iteration 1 while processes P_3 and P_4 are engaged in communication for the same iteration. Furthermore, more than one iteration may be active simultaneously on different processes. For instance, in Figure 8.7(i), process P_2 is performing the division step of iteration 2 while process P_3 is performing the elimination step of iteration 1.

We now show that, unlike the synchronous algorithm in which all processes work on the same iteration at a time, the pipelined or the asynchronous version of Gaussian elimination is cost-optimal. As Figure 8.7 shows, the initiation of consecutive iterations of the outer loop of Algorithm 8.4 is separated by a constant number of steps. A total of n such iterations are initiated. The last iteration modifies only the bottom-right corner element of the coefficient matrix; hence, it completes in a constant time after its initiation. Thus, the total number of steps in the entire pipelined procedure is $\Theta(n)$ (Problem 8.7). In any step, either $O(n)$ elements are communicated between directly-connected processes, or a division step is performed on $O(n)$ elements of a row, or an elimination step is performed on $O(n)$ elements of a row. Each of these operations take $O(n)$ time. Hence, the entire procedure consists of $\Theta(n)$ steps of $O(n)$ complexity each, and its parallel run time is $O(n^2)$. Since n processes are used, the cost is $O(n^3)$, which is of the same order as the sequential complexity of Gaussian elimination. Hence, the pipelined version of parallel Gaussian elimination with 1-D partitioning of the coefficient matrix is cost-optimal.

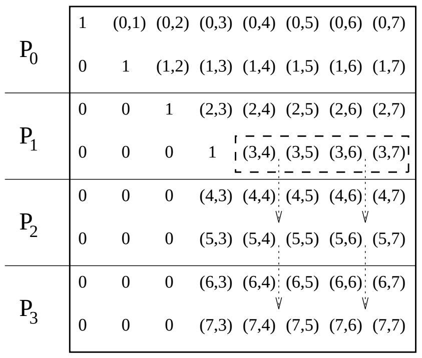


Figure 8.8 The communication in the Gaussian elimination iteration corresponding to $k = 3$ for an 8×8 matrix distributed among four processes using block 1-D partitioning.

Block 1-D Partitioning with Fewer than n Processes The preceding pipelined implementation of parallel Gaussian elimination can be easily adapted for the case in which $n > p$. Consider an $n \times n$ matrix partitIoned among p processes ($p < n$) such that each process is assigned n/p contiguous rows of the matrix. Figure 8.8 illustrates the communication steps in a typical iteration of Gaussian elimination with such a mapping. As the figure shows, the kth iteration of the algorithm requires that the active part of the kth row be sent to the processes storing rows $k+1, k+2, \ldots, n-1$.

Figure 8.9(a) shows that, with block 1-D partitioning, a process with all rows belonging to the active part of the matrix performs $(n-k-1)n/p$ multiplications and subtractions during the elimination step of the kth iteration. Note that in the last $(n/p) - 1$ iterations, no process has all active rows, but we ignore this anomaly. If the pipelined version of

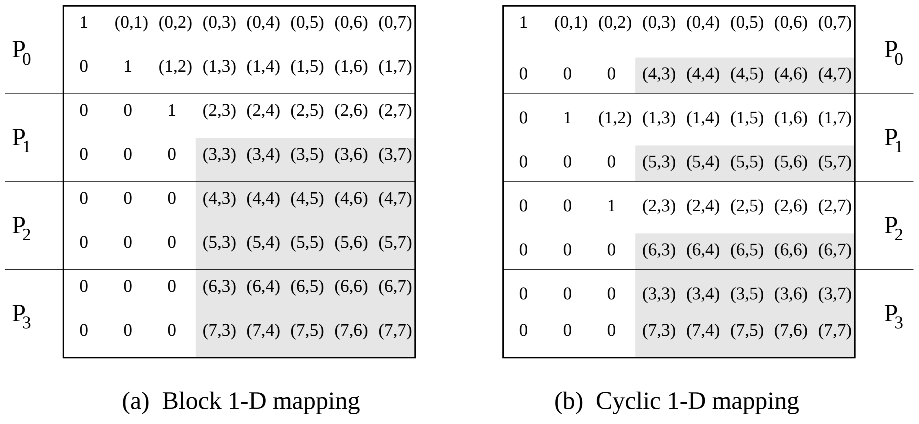


Figure 8.9 Computation load on different processes in block and cyclic 1-D partitioning of an 8×8 matrix on four processes during the Gaussian elimination iteration corresponding to $k = 3$.

the algorithm is used, then the number of arithmetic operations on a maximally-loaded process in the kth iteration $(2(n-k-1)n/p)$ is much higher than the number of words communicated $(n-k-1)$ by a process in the same iteration. Thus, for sufficiently large values of n with respect to p, computation dominates communication in each iteration. Assuming that each scalar multiplication and subtraction pair takes unit time, the total parallel run time of this algorithm (ignoring communication overhead) is $2(n/p)\Sigma_{k=0}^{n-1}(n-k-1)$, which is approximately equal to n^3/p.

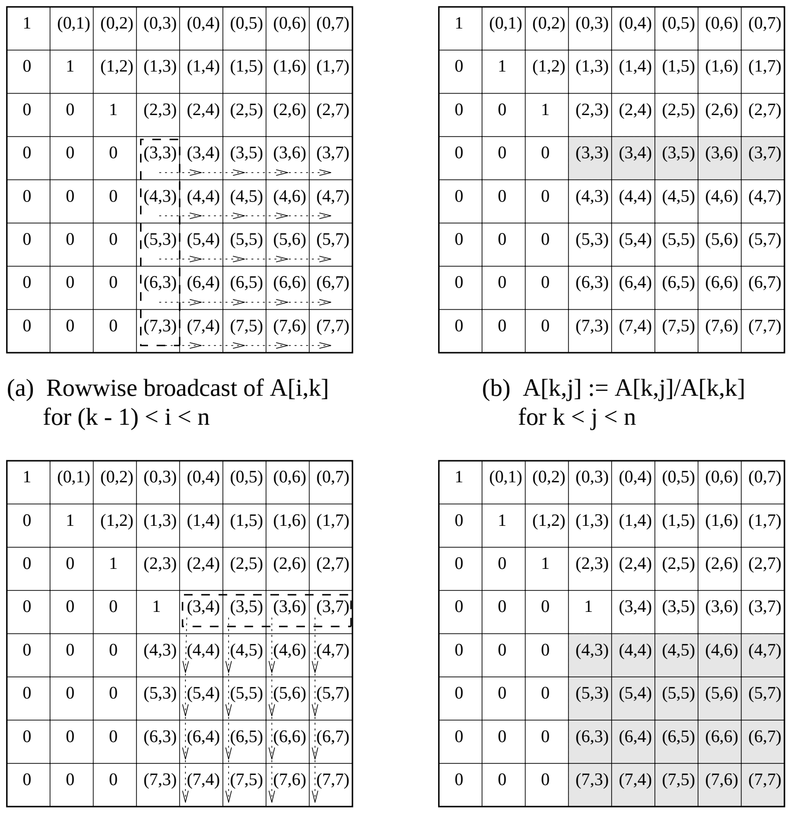
The process-time product of this algorithm is n^3, even if the communication costs are ignored. Thus, the cost of the parallel algorithm is higher than the sequential run time (Equation 8.17) by a factor of 3/2. This inefficiency of Gaussian elimination with block 1-D partitioning is due to process idling resulting from an uneven load distribution. As Figure 8.9(a) shows for an 8×8 matrix and four processes, during the iteration corresponding to $k = 3$ (in the outer loop of Algorithm 8.4), one process is completely idle, one is partially loaded, and only two processes are fully active. By the time half of the iterations of the outer loop are over, only half the processes are active. The remaining idle processes make the parallel algorithm costlier than the sequential algorithm.

This problem can be alleviated if the matrix is partitioned among the processes using cyclic 1-D mapping as shown in Figure 8.9(b). With the cyclic 1-D partitioning, the difference between the computational loads of a maximally loaded process and the least loaded process in any iteration is of at most one row (that is, $O(n)$ arithmetic operations). Since there are n iterations, the cumulative overhead due to process idling is only $O(n^2p)$ with a cyclic mapping, compared to $\Theta(n^3)$ with a block mapping (Problem 8.12).

Parallel Implementation with 2-D Partitioning

We now describe a parallel implementation of Algorithm 8.4 in which the $n \times n$ matrix A is mapped onto an $n \times n$ mesh of processes such that process $P_{i,j}$ initially stores $A[i, j]$. The communication and computation steps in the iteration of the outer loop corresponding to $k = 3$ are illustrated in Figure 8.10 for $n = 8$. Algorithm 8.4 and Figures 8.5 and 8.10 show that in the kth iteration of the outer loop, $A[k, k]$ is required by processes $P_{k,k+1}$, $P_{k,k+2}, \ldots, P_{k,n-1}$ to divide $A[k, k+1], A[k, k+2], \ldots, A[k, n-1]$, respectively. After the division on line 6, the modified elements of the kth row are used to perform the elimination step by all the other rows in the active part of the matrix. The modified (after the division on line 6) elements of the kth row are used by all other rows of the active part of the matrix. Similarly, the elements of the kth column are used by all other columns of the active part of the matrix for the elimination step. As Figure 8.10 shows, the communication in the kth iteration requires a one-to-all broadcast of $A[i, k]$ along the ith row (Figure 8.10(a)) for $k \le i < n$, and a one-to-all broadcast of $A[k, j]$ along the jth column (Figure 8.10(c)) for $k < j < n$. Just like the 1-D partitioning case, a non-cost-optimal parallel formulation results if these broadcasts are performed synchronously on all processes (Problem 8.11).

Pipelined Communication and Computation Based on our experience with Gaussian elimination using 1-D partitioning of the coefficient matrix, we develop a pipelined



(a) Rowwise broadcast of A[i,k] for (k - 1) < i < n

(b) A[k,j] := A[k,j]/A[k,k] for k < j < n

(c) Columnwise broadcast of A[k,j] for k < j < n

(d) A[i,j] := A[i,j]-A[i,k] x A[k,j] for k < i < n and k < j < n

Figure 8.10 Various steps in the Gaussian elimination iteration corresponding to k = 3 for an 8 $\times$ 8 matrix on 64 processes arranged in a logical two-dimensional mesh.

version of the algorithm using 2-D partitioning.

As Figure 8.10 shows, in the kth iteration of the outer loop (lines 3–16 of Algorithm 8.4), $A[k, k]$ is sent to the right from $P_{k,k}$ to $P_{k,k+1}$ to $P_{k,k+2}$, and so on, until it reaches $P_{k,n-1}$. Process $P_{k,k+1}$ performs the division $A[k, k + 1]/A[k, k]$ as soon as it receives $A[k, k]$ from $P_{k,k}$. It does not have to wait for $A[k, k]$ to reach all the way up to $P_{k,n-1}$ before performing its local computation. Similarly, any subsequent process $P_{k,j}$ of the kth row can perform its division as soon as it receives $A[k, k]$. After performing the division, $A[k, j]$ is ready to be communicated downward in the jth column. As $A[k, j]$ moves down, each process it passes is free to use it for computation. Processes in the jth column need not wait until $A[k, j]$ reaches the last process of the column. Thus, $P_{i,j}$ performs the elimination step $A[i, j] := A[i, j] - A[i, k] \times A[k, j]$ as soon as $A[i, k]$ and $A[k, j]$ are available. Since some processes perform the computation for a given iteration earlier than other processes, they start working on subsequent iterations sooner.

The communication and computation can be pipelined in several ways. We present one such scheme in Figure 8.11. In Figure 8.11(a), the iteration of the outer loop for $k = 0$ starts at process $P_{0,0}$, when $P_{0,0}$ sends $A[0, 0]$ to $P_{0,1}$. Upon receiving $A[0, 0]$, $P_{0,1}$ computes $A[0, 1] := A[0, 1]/A[0, 0]$ (Figure 8.11(b)). Now $P_{0,1}$ forwards $A[0, 0]$ to $P_{0,2}$ and also sends the updated $A[0, 1]$ down to $P_{1,1}$ (Figure 8.11(c)). At the same time, $P_{1,0}$ sends $A[1, 0]$ to $P_{1,1}$. Having received $A[0, 1]$ and $A[1, 0]$, $P_{1,1}$ performs the elimination step $A[1, 1] := A[1, 1] - A[1, 0] \times A[0, 1]$, and having received $A[0, 0]$, $P_{0,2}$ performs the division step $A[0, 2] := A[0, 2]/A[0, 0]$ (Figure 8.11(d)). After this computation step, another set of processes (that is, processes $P_{0,2}$, $P_{1,1}$, and $P_{2,0}$) is ready to initiate communication (Figure 8.11(e)).

All processes performing communication or computation during a particular iteration lie along a diagonal in the bottom-left to top-right direction (for example, $P_{0,2}$, $P_{1,1}$, and $P_{2,0}$ performing communication in Figure 8.11(e) and $P_{0,3}$, $P_{1,2}$, and $P_{2,1}$ performing computation in Figure 8.11(f)). As the parallel algorithm progresses, this diagonal moves toward the bottom-right corner of the logical 2-D mesh. Thus, the computation and communication for each iteration moves through the mesh from top-left to bottom-right as a "front." After the front corresponding to a certain iteration passes through a process, the process is free to perform subsequent iterations. For instance, in Figure 8.11(g), after the front for $k = 0$ has passed $P_{1,1}$, it initiates the iteration for $k = 1$ by sending $A[1, 1]$ to $P_{1,2}$. This initiates a front for $k = 1$, which closely follows the front for $k = 0$. Similarly, a third front for $k = 2$ starts at $P_{2,2}$ (Figure 8.11(m)). Thus, multiple fronts that correspond to different iterations are active simultaneously.

Every step of an iteration, such as division, elimination, or transmitting a value to a neighboring process, is a constant-time operation. Therefore, a front moves a single step closer to the bottom-right corner of the matrix in constant time (equivalent to two steps of Figure 8.11). The front for $k = 0$ takes time $\Theta(n)$ to reach $P_{n-1,n-1}$ after its initiation at $P_{0,0}$. The algorithm initiates n fronts for the n iterations of the outer loop. Each front lags behind the previous one by a single step. Thus, the last front passes the bottom-right corner of the matrix $\Theta(n)$ steps after the first one. The total time elapsed between the first front starting at $P_{0,0}$ and the last one finishing is $\Theta(n)$. The procedure is complete after the last front passes the bottom-right corner of the matrix; hence, the total parallel run time is $\Theta(n)$. Since n^2 process are used, the cost of the pipelined version of Gaussian elimination is $\Theta(n^3)$, which is the same as the sequential run time of the algorithm. Hence, the pipelined version of Gaussian elimination with 2-D partitioning is cost-optimal.

2-D Partitioning with Fewer than n^2 Processes Consider the case in which p processes are used so that $p < n^2$ and the matrix is mapped onto a $\sqrt{p} \times \sqrt{p}$ mesh by using block 2-D partitioning. Figure 8.12 illustrates that a typical parallel Gaussian iteration involves a rowwise and a columnwise communication of $n/\sqrt{p}$ values. Figure 8.13(a) illustrates the load distribution in block 2-D mapping for $n = 8$ and $p = 16$.

Figures 8.12 and 8.13(a) show that a process containing a completely active part of the

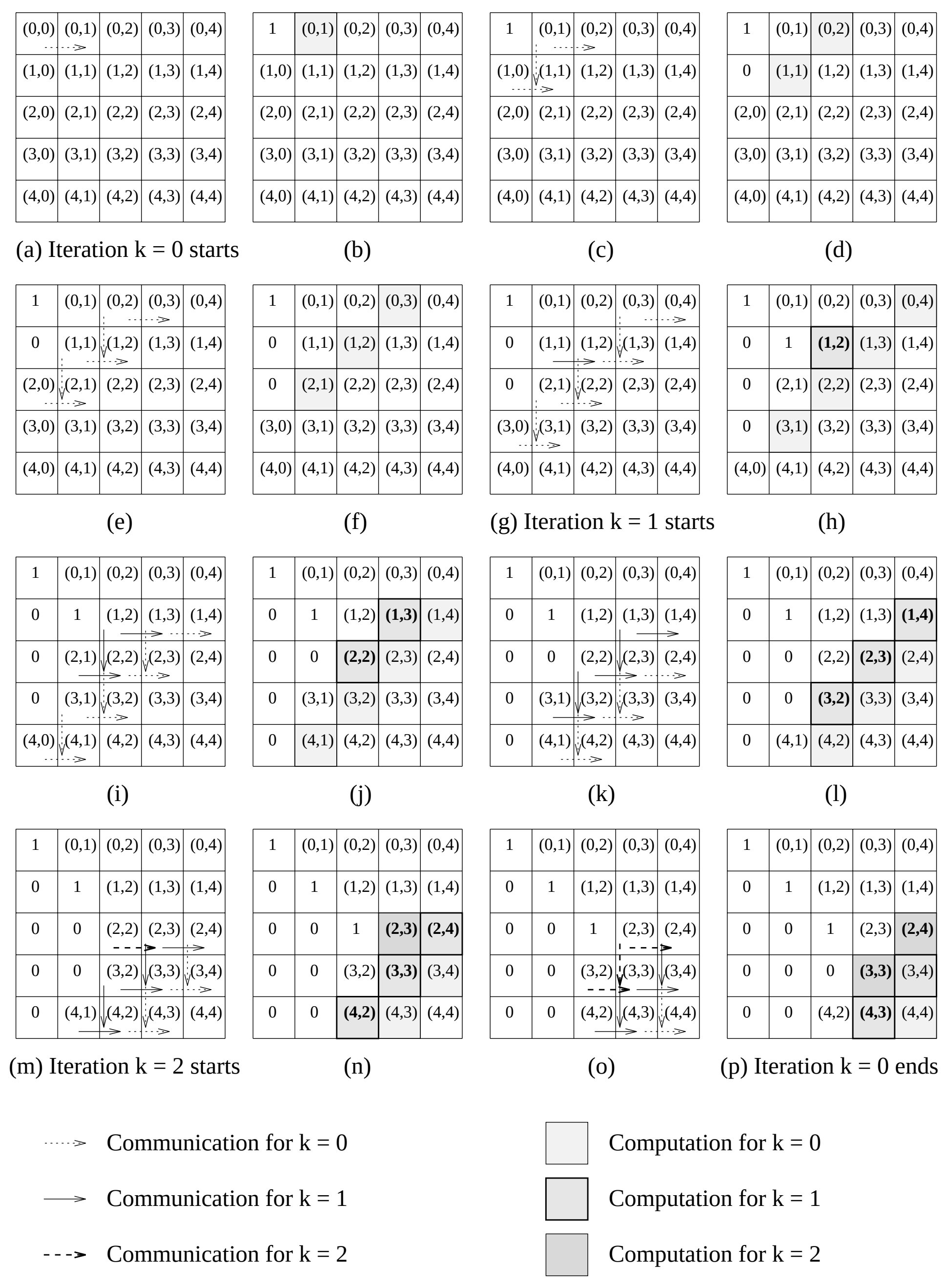
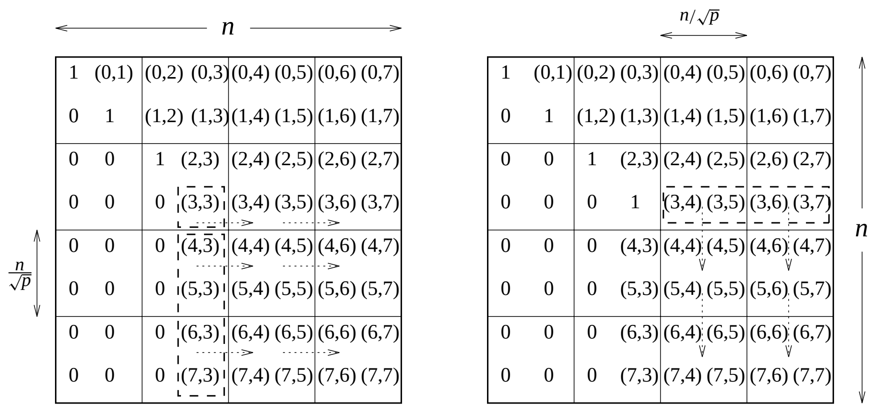


Figure 8.11 Pipelined Gaussian elimination for a 5 $\times$ 5 matrix with 25 processes.



(a) Rowwise broadcast of A[i,k] for i = k to (n - 1)

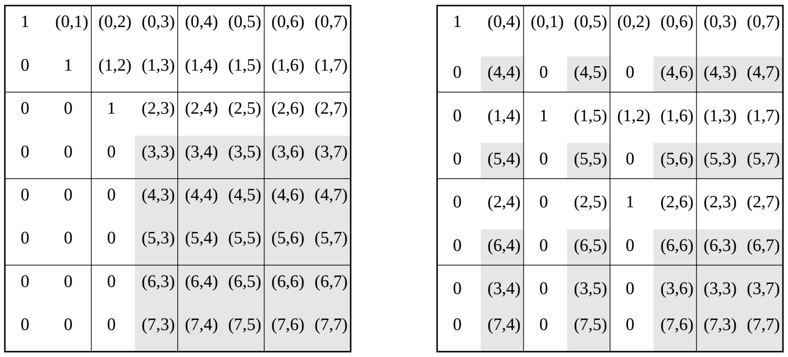
(b) Columnwise broadcast of A[k,j] for j = (k + 1) to (n - 1)

Figure 8.12 The communication steps in the Gaussian elimination iteration corresponding to k = 3 for an 8 $\times$ 8 matrix on 16 processes of a two-dimensional mesh.

matrix performs n^2/p multiplications and subtractions, and communicates $n/\sqrt{p}$ words along its row and its column (ignoring the fact that in the last $(n/\sqrt{p}) - 1$ iterations, the active part of the matrix becomes smaller than the size of a block, and no process contains a completely active part of the matrix). If the pipelined version of the algorithm is used, the number of arithmetic operations per process ($2n^2/p$) is an order of magnitude higher than the number of words communicated per process ($n/\sqrt{p}$) in each iteration. Thus, for sufficiently large values of n^2 with respect to p, the communication in each iteration is dominated by computation. Ignoring the communication cost and assuming that each scalar arithmetic operation takes unit time, the total parallel run time of this algorithm is $(2n^2/p) \times n$, which is equal to $2n^3/p$. The process-time product is $2n^3$, which is three times the cost of the serial algorithm (Equation 8.17). As a result, there is an upper bound of $1/3$ on the efficiency of the parallel algorithm.

As in the case of a block 1-D mapping, the inefficiency of Gaussian elimination with a block 2-D partitioning of the matrix is due to process idling resulting from an uneven load distribution. Figure 8.13(a) shows the active part of an 8 $\times$ 8 matrix of coefficients in the iteration of the outer loop for $k = 3$ when the matrix is block 2-D partitioned among 16 processes. As shown in the figure, seven out of 16 processes are fully idle, five are partially loaded, and only four are fully active. By the time half of the iterations of the outer loop have been completed, only one-fourth of the processes are active. The remaining idle processes make the parallel algorithm much costlier than the sequential algorithm.

This problem can be alleviated if the matrix is partitioned in a 2-D cyclic fashion as shown in Figure 8.13(b). With the cyclic 2-D partitioning, the maximum difference in computational load between any two processes in any iteration is that of one row and one column update. For example, in Figure 8.13(b), n^2/p matrix elements are active in the



(a) Block-checkerboard mapping (b) Cyclic-checkerboard mapping

Figure 8.13 Computational load on different processes in block and cyclic 2-D mappings of an 8 $\times$ 8 matrix onto 16 processes during the Gaussian elimination iteration corresponding to k = 3.

bottom-right process, and $(n-1)^2/p$ elements are active in the top-left process. The difference in workload between any two processes is at most $\Theta(n/\sqrt{p})$ in any iteration, which contributes $\Theta(n\sqrt{p})$ to the overhead function. Since there are n iterations, the cumulative overhead due to process idling is only $\Theta(n^2\sqrt{p})$ with cyclic mapping in contrast to $\Theta(n^3)$ with block mapping (Problem 8.12). In practical parallel implementations of Gaussian elimination and LU factorization, a block-cyclic mapping is used to reduce the overhead due to message startup time associated with a pure cyclic mapping and to obtain better serial CPU utilization by performing block-matrix operations (Problem 8.15).

From the discussion in this section, we conclude that pipelined parallel Gaussian elimination for an $n \times n$ matrix takes time $\Theta(n^3/p)$ on p processes with both 1-D and 2-D partitioning schemes. 2-D partitioning can use more processes ($O(n^2)$) than 1-D partitioning ($O(n)$) for an $n \times n$ coefficient matrix. Hence, an implementation with 2-D partitioning is more scalable.

8.3.2 Gaussian Elimination with Partial Pivoting

The Gaussian elimination algorithm in Algorithm 8.4 fails if any diagonal entry $A[k,k]$ of the matrix of coefficients is close or equal to zero. To avoid this problem and to ensure the numerical stability of the algorithm, a technique called ***partial pivoting*** is used. At the beginning of the outer loop in the kth iteration, this method selects a column i (called the ***pivot*** column) such that $A[k,i]$ is the largest in magnitude among all $A[k,j]$ such that $k \le j < n$. It then exchanges the kth and the ith columns before starting the iteration. These columns can either be exchanged explicitly by physically moving them into each other's locations, or they can be exchanged implicitly by simply maintaining an $n \times 1$ permutation vector to keep track of the new indices of the columns of A. If partial pivoting is performed with an implicit exchange of column indices, then the factors L and U are not exactly triangular matrices, but columnwise permutations of triangular matrices.

Assuming that columns are exchanged explicitly, the value of $A[k, k]$ used as the divisor on line 6 of Algorithm 8.4 (after exchanging columns k and i) is greater than or equal to any $A[k, j]$ that it divides in the kth iteration. Partial pivoting in Algorithm 8.4 results in a unit upper-triangular matrix in which all elements above the principal diagonal have an absolute value of less than one.

1-D Partitioning

Performing partial pivoting is straightforward with rowwise partitioning as discussed in Section 8.3.1. Before performing the divide operation in the kth iteration, the process storing the kth row makes a comparison pass over the active portion of this row, and selects the element with the largest absolute value as the divisor. This element determines the pivot column, and all processes must know the index of this column. This information can be passed on to the rest of the processes along with the modified (after the division) elements of the kth row. The combined pivot-search and division step takes time $\Theta(n - k - 1)$ in the kth iteration, as in case of Gaussian elimination without pivoting. Thus, partial pivoting has no significant effect on the performance of Algorithm 8.4 if the coefficient matrix is partitioned along the rows.

Now consider a columnwise 1-D partitioning of the coefficient matrix. In the absence of pivoting, parallel implementations of Gaussian elimination with rowwise and columnwise 1-D partitioning are almost identical (Problem 8.9). However, the two are significantly different if partial pivoting is performed.

The first difference is that, unlike rowwise partitioning, the pivot search is distributed in columnwise partitioning. If the matrix size is $n \times n$ and the number of processes is p, then the pivot search in columnwise partitioning involves two steps. During pivot search for the kth iteration, first each process determines the maximum of the n/p (or fewer) elements of the kth row that it stores. The next step is to find the maximum of the resulting p (or fewer) values, and to distribute the maximum among all processes. Each pivot search takes time $\Theta(n/p) + \Theta(\log p)$. For sufficiently large values of n with respect to p, this is less than the time $\Theta(n)$ it takes to perform a pivot search with rowwise partitioning. This seems to suggest that a columnwise partitioning is better for partial pivoting that a rowwise partitioning. However, the following factors favor rowwise partitioning.

Figure 8.7 shows how communication and computation "fronts" move from top to bottom in the pipelined version of Gaussian elimination with rowwise 1-D partitioning. Similarly, the communication and computation fronts move from left to right in case of columnwise 1-D partitioning. This means that the $(k + 1)$th row is not ready for pivot search for the $(k + 1)$th iteration (that is, it is not fully updated) until the front corresponding to the kth iteration reaches the rightmost process. As a result, the $(k + 1)$th iteration cannot start until the entire kth iteration is complete. This effectively eliminates pipelining, and we are therefore forced to use the synchronous version with poor efficiency.

While performing partial pivoting, columns of the coefficient matrix may or may not be explicitly exchanged. In either case, the performance of Algorithm 8.4 is adversely affected with columnwise 1-D partitioning. Recall that cyclic or block-cyclic mappings result in a better load balance in Gaussian elimination than a block mapping. A cyclic mapping ensures that the active portion of the matrix is almost uniformly distributed among the processes at every stage of Gaussian elimination. If pivot columns are not exchanged explicitly, then this condition may cease to hold. After a pivot column is used, it no longer stays in the active portion of the matrix. As a result of pivoting without explicit exchange, columns are arbitrarily removed from the different processes' active portions of the matrix. This randomness may disturb the uniform distribution of the active portion. On the other hand, if columns belonging to different processes are exchanged explicitly, then this exchange requires communication between the processes. A rowwise 1-D partitioning neither requires communication for exchanging columns, nor does it lose the load-balance if columns are not exchanged explicitly.

2-D Partitioning

In the case of 2-D partitioning of the coefficient matrix, partial pivoting seriously restricts pipelining, although it does not completely eliminate it. Recall that in the pipelined version of Gaussian elimination with 2-D partitioning, fronts corresponding to various iterations move from top-left to bottom-right. The pivot search for the $(k+1)$th iteration can commence as soon as the front corresponding to the kth iteration has moved past the diagonal of the active matrix joining its top-right and bottom-left corners.

Thus, partial pivoting may lead to considerable performance degradation in parallel Gaussian elimination with 2-D partitioning. If numerical considerations allow, it may be possible to reduce the performance loss due to partial pivoting. We can restrict the search for the pivot in the kth iteration to a band of q columns (instead of all $n-k$ columns). In this case, the ith column is selected as the pivot in the kth iteration if $A[k,i]$ is the largest element in a band of q elements of the active part of the ith row. This restricted partial pivoting not only reduces the communication cost, but also permits limited pipelining. By restricting the number of columns for pivot search to q, an iteration can start as soon as the previous iteration has updated the first $q+1$ columns.

Another way to get around the loss of pipelining due to partial pivoting in Gaussian elimination with 2-D partitioning is to use fast algorithms for one-to-all broadcast, such as those described in Section 4.7.1. With 2-D partitioning of the $n \times n$ coefficient matrix on p processes, a process spends time $\Theta(n/\sqrt{p})$ in communication in each iteration of the pipelined version of Gaussian elimination. Disregarding the message startup time t_s, a non-pipelined version that performs explicit one-to-all broadcasts using the algorithm of Section 4.1 spends time $\Theta((n/\sqrt{p})\log p)$ communicating in each iteration. This communication time is higher than that of the pipelined version. The one-to-all broadcast algorithms described in Section 4.7.1 take time $\Theta(n/\sqrt{p})$ in each iteration (disregarding the startup time). This time is asymptotically equal to the per-iteration communication time of the pipelined algorithm. Hence, using a smart algorithm to perform one-to-all broadcast, even non-pipelined parallel Gaussian elimination can attain performance comparable to that of the pipelined algorithm. However, the one-to-all broadcast algorithms described in Section 4.7.1 split a message into smaller parts and route them separately. For these algorithms to be effective, the sizes of the messages should be large enough; that is, n should be large compared to p.

Although pipelining and pivoting do not go together in Gaussian elimination with 2-D partitioning, the discussion of 2-D partitioning in this section is still useful. With some modification, it applies to the Cholesky factorization algorithm (Algorithm 8.6 in Problem 8.16), which does not require pivoting. Cholesky factorization applies only to symmetric, positive definite matrices. A real $n \times n$ matrix A is ***positive definite*** if $x^T Ax > 0$ for any $n \times 1$ nonzero, real vector x. The communication pattern in Cholesky factorization is quite similar to that of Gaussian elimination (Problem 8.16), except that, due to symmetric lower and upper-triangular halves in the matrix, Cholesky factorization uses only one triangular half of the matrix.

8.3.3 Solving a Triangular System: Back-Substitution

We now briefly discuss the second stage of solving a system of linear equations. After the full matrix A has been reduced to an upper-triangular matrix U with ones along the principal diagonal, we perform back-substitution to determine the vector x. A sequential back-substitution algorithm for solving an upper-triangular system of equations $Ux = y$ is shown in Algorithm 8.5.

Starting with the last equation, each iteration of the main loop (lines 3–8) of Algorithm 8.5 computes the values of a variable and substitutes the variable's value back into the remaining equations. The program performs approximately $n^2/2$ multiplications and subtractions. Note that the number of arithmetic operations in back-substitution is less than that in Gaussian elimination by a factor of $\Theta(n)$. Hence, if back-substitution is used in conjunction with Gaussian elimination, it is best to use the matrix partitioning scheme that is the most efficient for parallel Gaussian elimination.

```
1.   procedure BACK_SUBSTITUTION ($U, x, y$)
2.   begin
3.      for $k := n - 1$ downto 0 do   /* Main loop */
4.         begin
5.            $x[k] := y[k]$;
6.            for $i := k - 1$ downto 0 do
7.               $y[i] := y[i] - x[k] \times U[i, k]$;
8.         endfor;
9.   end BACK_SUBSTITUTION
```

Algorithm 8.5 A serial algorithm for back-substitution. U is an upper-triangular matrix with all entries of the principal diagonal equal to one, and all subdiagonal entries equal to zero.

Consider a rowwise block 1-D mapping of the $n \times n$ matrix U onto p processes. Let the vector y be distributed uniformly among all the processes. The value of the variable solved in a typical iteration of the main loop (line 3) must be sent to all the processes with equations involving that variable. This communication can be pipelined (Problem 8.22). If so, the time to perform the computations of an iteration dominates the time that a process spends in communication in an iteration. In every iteration of a pipelined implementation, a process receives (or generates) the value of a variable and sends that value to another process. Using the value of the variable solved in the current iteration, a process also performs up to n/p multiplications and subtractions (lines 6 and 7). Hence, each step of a pipelined implementation requires a constant amount of time for communication and time $\Theta(n/p)$ for computation. The algorithm terminates in $\Theta(n)$ steps (Problem 8.22), and the parallel run time of the entire algorithm is $\Theta(n^2/p)$.

If the matrix is partitioned by using 2-D partitioning on a $\sqrt{p} \times \sqrt{p}$ logical mesh of processes, and the elements of the vector are distributed along one of the columns of the process mesh, then only the $\sqrt{p}$ processes containing the vector perform any computation. Using pipelining to communicate the appropriate elements of U to the process containing the corresponding elements of y for the substitution step (line 7), the algorithm can be executed in time $\Theta(n^2/\sqrt{p})$ (Problem 8.22). Thus, the cost of parallel back-substitution with 2-D mapping is $\Theta(n^2\sqrt{p})$. The algorithm is not cost-optimal because its sequential cost is only $\Theta(n^2)$. However, the entire process of solving the linear system, including upper-triangularization using Gaussian elimination, is still cost-optimal for $\sqrt{p} = O(n)$ because the sequential complexity of the entire process is $\Theta(n^3)$.

8.3.4 Numerical Considerations in Solving Systems of Linear Equations

A system of linear equations of the form $Ax = b$ can be solved by using a factorization algorithm to express A as the product of a lower-triangular matrix L, and a unit upper-triangular matrix U. The system of equations is then rewritten as $LUx = b$, and is solved in two steps. First, the lower-triangular system $Ly = b$ is solved for y. Second, the upper-triangular system $Ux = y$ is solved for x.

The Gaussian elimination algorithm given in Algorithm 8.4 effectively factorizes A into L and U. However, it also solves the lower-triangular system $Ly = b$ on the fly by means of steps on lines 7 and 13. Algorithm 8.4 gives what is called a ***row-oriented*** Gaussian elimination algorithm. In this algorithm, multiples of rows are subtracted from other rows. If partial pivoting, as described in Section 8.3.2, is incorporated into this algorithm, then the resulting upper-triangular matrix U has all its elements less than or equal to one in magnitude. The lower-triangular matrix L, whether implicit or explicit, may have elements with larger numerical values. While solving the system $Ax = b$, the triangular system $Ly = b$ is solved first. If L contains large elements, then rounding errors can occur while solving for y due to the finite precision of floating-point numbers in the computer. These errors in y are propagated through the solution of $Ux = y$.

An alternate form of Gaussian elimination is the ***column-oriented*** form that can be obtained from Algorithm 8.4 by reversing the roles of rows and columns. In the column-oriented algorithm, multiples of columns are subtracted from other columns, pivot search is also performed along the columns, and numerical stability is guaranteed by row interchanges, if needed. All elements of the lower-triangular matrix L generated by the column-oriented algorithm have a magnitude less than or equal to one. This minimizes numerical error while solving $Ly = b$, and results in a significantly smaller error in the overall solution than the row-oriented algorithm. Algorithm 3.3 gives a procedure for column-oriented LU factorization.

From a practical point of view, the column-oriented Gaussian elimination algorithm is more useful than the row-oriented algorithm. We have chosen to present the row-oriented algorithm in detail in this chapter because it is more intuitive. It is easy to see that the system of linear equations resulting from the subtraction of a multiple of an equation from other equations is equivalent to the original system. The entire discussion on the row-oriented algorithm of Algorithm 8.4 presented in this section applies to the column-oriented algorithm with the roles of rows and columns reversed. For example, columnwise 1-D partitioning is more suitable than rowwise 1-D partitioning for the column-oriented algorithm with partial pivoting.

8.4 Bibliographic Remarks

Matrix transposition with 1-D partitioning is essentially an all-to-all personalized communication problem [Ede89]. Hence, all the references in Chapter 4 for all-to-all personalized communication apply directly to matrix transposition. The recursive transposition algorithm, popularly known as RTA, was first reported by Eklundh [Ekl72]. Its adaptations for hypercubes have been described by Bertsekas and Tsitsiklis [BT97], Fox and Furmanski [FF86], Johnsson [Joh87], and McBryan and Van de Velde [MdV87] for one-port communication on each process. Johnsson [Joh87] also discusses parallel RTA for hypercubes that permit simultaneous communication on all channels. Further improvements on the hypercube RTA have been suggested by Ho and Raghunath [HR91], Johnsson and Ho [JH88], Johnsson [Joh90], and Stout and Wagar [SW87].

A number of sources of parallel dense linear algebra algorithms, including those for matrix-vector multiplication and matrix multiplication, are available [CAHH91, GPS90, GL96a, Joh87, Mod88, OS85]. Since dense matrix multiplication is highly computationally intensive, there has been a great deal of interest in developing parallel formulations of this algorithm and in testing its performance on various parallel architectures [Akl89, Ber89, CAHH91, Can69, Cha79, CS88, DNS81, dV89, FJL$^+$88, FOH87, GK91, GL96a, Hip89, HJE91, Joh87, PV80, Tic88]. Some of the early parallel formulations of matrix multiplication were developed by Cannon [Can69], Dekel, Nassimi, and Sahni [DNS81], and Fox *et al.* [FOH87]. Variants and improvements of these algorithms have been presented by Berntsen [Ber89], and by Ho, Johnsson, and Edelman [HJE91]. In particular, Berntsen [Ber89] presents an algorithm that has strictly smaller communication overhead than Cannon's algorithm, but has a smaller degree of concurrency. Ho, Johnsson, and Edelman [HJE91] present another variant of Cannon's algorithm for a hypercube that permits communication on all channels simultaneously. This algorithm, while reducing communication, also reduces the degree of concurrency. Gupta and Kumar [GK91] present a detailed scalability analysis of several matrix multiplication algorithms. They present an analysis to determine the best algorithm to multiply two $n \times n$ matrices on a p-process hypercube for different ranges of n, p and the hardware-related constants. They also show that the improvements suggested by Berntsen and Ho et al. do not improve the overall scalability of matrix multiplication on a hypercube.

Parallel algorithms for LU factorization and solving dense systems of linear equations have been discussed by several researchers [Ber84, BT97, CG87, Cha87, Dav86, DHvdV93, FJL+88, Gei85, GH86, GPS90, GR88, Joh87, LD90, Lei92, Mod88, Mol86, OR88, Ort88, OS86, PR85, Rob90, Saa86, Vav89]. Geist and Heath [GH85, GH86], and Heath [Hea85] specifically concentrate on parallel dense Cholesky factorization. Parallel algorithms for solving triangular systems have also been studied in detail [EHHR88, HR88, LC88, LC89, RO88, Rom87]. Demmel, Heath, and van der Vorst [DHvdV93] present a comprehensive survey of parallel matrix computations considering numerical implications in detail.

A portable software implementation of all matrix and vector operations discussed in this chapter, and many more, is available as PBLAS (parallel basic linear algebra subroutines) [C+95]. The ScaLAPACK library [B+97] uses PBLAS to implement a variety of linear algebra routines of practical importance, including procedures for various methods of matrix factorizations and solving systems of linear equations.

Problems

8.1 Consider the two algorithms for all-to-all personalized communication in Section 4.5.3. Which method would you use on a 64-node parallel computer with $\Theta(p)$ bisection width for transposing a 1024×1024 matrix with the 1-D partitioning if $t_s = 100\mu$s and $t_w = 1\mu$s? Why?

8.2 Describe a parallel formulation of matrix-vector multiplication in which the matrix is 1-D block-partitioned along the columns and the vector is equally partitioned among all the processes. Show that the parallel run time is the same as in case of rowwise 1-D block partitioning.

Hint: The basic communication operation used in the case of columnwise 1-D partitioning is all-to-all reduction, as opposed to all-to-all broadcast in the case of rowwise 1-D partitioning. Problem 4.8 describes all-to-all reduction.

8.3 Section 8.1.2 describes and analyzes matrix-vector multiplication with 2-D partitioning. If $n \gg \sqrt{p}$, then suggest ways of improving the parallel run time to $n^2/p + 2t_s \log p + 2t_w(n/\sqrt{p})$. Is the improved method more scalable than the one used in Section 8.1.2?

8.4 The overhead function for multiplying an $n \times n$ 2-D partitioned matrix with an $n \times 1$ vector using p processes is $t_s p \log p + t_w n\sqrt{p} \log p$ (Equation 8.8). Substituting this expression in Equation 5.14 yields a quadratic equation in n. Using this equation, determine the precise isoefficiency function for the parallel algorithm and compare it with Equations 8.9 and 8.10. Does this comparison alter the conclusion that the term associated with t_w is responsible for the overall isoefficiency function of this parallel algorithm?

8.5 Strassen's method [AHU74, CLR90] for matrix multiplication is an algorithm based on the divide-and-conquer technique. The sequential complexity of multiplying two $n \times n$ matrices using Strassen's algorithm is $\Theta(n^{2.81})$. Consider the simple matrix multiplication algorithm (Section 8.2.1) for multiplying two $n \times n$ matrices using p processes. Assume that the $n/\sqrt{p} \times n/\sqrt{p}$ submatrices are multiplied using Strassen's algorithm at each process. Derive an expression for the parallel run time of this algorithm. Is the parallel algorithm cost-optimal?

8.6 **(DNS algorithm with fewer than n^3 processes [DNS81])** Section 8.2.3 describes a parallel formulation of the DNS algorithm that uses fewer than n^3 processes. Another variation of this algorithm works with $p = n^2 q$ processes, where $1 \leq q \leq n$. Here the process arrangement is regarded as a $q \times q \times q$ logical three-dimensional array of "superprocesses," in which each superprocess is an $(n/q) \times (n/q)$ mesh of processes. This variant can be viewed as identical to the block variant described in Section 8.2.3, except that the role of each process is now assumed by an $(n/q) \times (n/q)$ logical mesh of processes. This means that each block multiplication of $(n/q) \times (n/q)$ submatrices is performed in parallel by $(n/q)^2$ processes rather than by a single process. Any of the algorithms described in Sections 8.2.1 or 8.2.2 can be used to perform this multiplication.

Derive an expression for the parallel run time for this variant of the DNS algorithm in terms of n, p, t_s, and t_w. Compare the expression with Equation 8.16. Discuss the relative merits and drawbacks of the two variations of the DNS algorithm for fewer than n^3 processes.

8.7 Figure 8.7 shows that the pipelined version of Gaussian elimination requires 16 steps for a 5×5 matrix partitioned rowwise on five processes. Show that, in general, the algorithm illustrated in this figure completes in $4(n-1)$ steps for an $n \times n$ matrix partitioned rowwise with one row assigned to each process.

8.8 Describe in detail a parallel implementation of the Gaussian elimination algorithm of Algorithm 8.4 without pivoting if the $n \times n$ coefficient matrix is partitioned columnwise among p processes. Consider both pipelined and non-pipelined implementations. Also consider the cases $p = n$ and $p < n$.

Hint: The parallel implementation of Gaussian elimination described in Section 8.3.1 shows horizontal and vertical communication on a logical two-dimensional mesh of processes (Figure 8.12). A rowwise partitioning requires only the vertical part of this communication. Similarly, columnwise partitioning performs only the horizontal part of this communication.

8.9 Derive expressions for the parallel run times of all the implementations in Problem 8.8. Is the run time of any of these parallel implementations significantly different from the corresponding implementation with rowwise 1-D partitioning?

8.10 Rework Problem 8.9 with partial pivoting. In which implementations are the parallel run times significantly different for rowwise and columnwise partitioning?

8.11 Show that Gaussian elimination on an $n \times n$ matrix 2-D partitioned on an $n \times n$ logical mesh of processes is not cost-optimal if the $2n$ one-to-all broadcasts are performed synchronously.

8.12 Show that the cumulative idle time over all the processes in the Gaussian elimination algorithm is $\Theta(n^3)$ for a block mapping, whether the $n \times n$ matrix is partitioned along one or both dimensions. Show that this idle time is reduced to $\Theta(n^2 p)$ for cyclic 1-D mapping and $\Theta(n^2\sqrt{p})$ for cyclic 2-D mapping.

8.13 Prove that the isoefficiency function of the asynchronous version of the Gaussian elimination with 2-D mapping is $\Theta(p^{3/2})$ if pivoting is not performed.

8.14 Derive precise expressions for the parallel run time of Gaussian elimination with and without partial pivoting if the $n \times n$ matrix of coefficients is partitioned among p processes of a logical square two-dimensional mesh in the following formats:
(a) Rowwise block 1-D partitioning.
(b) Rowwise cyclic 1-D partitioning.
(c) Columnwise block 1-D partitioning.
(d) Columnwise cyclic 1-D partitioning.

8.15 Rewrite Algorithm 8.4 in terms of block matrix operations as discussed at the beginning of Section 8.2. Consider Gaussian elimination of an $n \times n$ matrix partitioned into a $q \times q$ array of submatrices, where each submatrix is of size of $n/q \times n/q$. This array of blocks is mapped onto a logical $\sqrt{p} \times \sqrt{p}$ mesh of processes in a cyclic manner, resulting in a 2-D block cyclic mapping of the original matrix onto the mesh. Assume that $n > q > \sqrt{p}$ and that n is divisible by q, which in turn is divisible by $\sqrt{p}$. Derive expressions for the parallel run time for both synchronous and pipelined versions of Gaussian elimination.
Hint: The division step $A[k, j] := A[k, j]/A[k, k]$ is replaced by submatrix operation $A_{k,j} := A_{k,k}^{-1} A_{k,j}$, where $A_{k,k}$ is the lower triangular part of the kth diagonal submatrix.

8.16 **(Cholesky factorization)** Algorithm 8.6 describes a row-oriented version of the Cholesky factorization algorithm for factorizing a symmetric positive definite matrix into the form $A = U^T U$. Cholesky factorization does not require pivoting. Describe a pipelined parallel formulation of this algorithm that uses 2-D partitioning of the matrix on a square mesh of processes. Draw a picture similar to Figure 8.11.

```
1.  procedure CHOLESKY ($A$)
2.  begin
3.      for $k := 0$ to $n - 1$ do
4.          begin
5.              $A[k, k] := \sqrt{A[k, k]}$;
6.              for $j := k + 1$ to $n - 1$ do
7.                  $A[k, j] := A[k, j]/A[k, k]$;
8.              for $i := k + 1$ to $n - 1$ do
9.                  for $j := i$ to $n - 1$ do
10.                     $A[i, j] := A[i, j] - A[k, i] \times A[k, j]$;
11.         endfor;        /* Line 3 */
12. end CHOLESKY
```

Algorithm 8.6 A row-oriented Cholesky factorization algorithm.

8.17 **(Scaled speedup)** Scaled speedup (Section 5.7) is defined as the speedup obtained when the problem size is increased linearly with the number of processes; that is, if W is chosen as a base problem size for a single process, then

$$\textit{Scaled speedup} = \frac{Wp}{T_P(Wp, p)}. \tag{8.19}$$

For the simple matrix multiplication algorithm described in Section 8.2.1, plot the standard and scaled speedup curves for the base problem of multiplying 16×16 matrices. Use p = 1, 4, 16, 64, and 256. Assume that $t_s = 10$ and $t_w = 1$ in Equation 8.14.

8.18 Plot a third speedup curve for Problem 8.17, in which the problem size is scaled up according to the isoefficiency function, which is $\Theta(p^{3/2})$. Use the same values of t_s and t_w.
Hint: The scaled speedup under this method of scaling is

$$\textit{Isoefficient scaled speedup} = \frac{Wp^{3/2}}{T_P(Wp^{3/2}, p)}.$$

8.19 Plot the efficiency curves for the simple matrix multiplication algorithm corresponding to the standard speedup curve (Problem 8.17), the scaled speedup curve (Problem 8.17), and the speedup curve when the problem size is increased according to the isoefficiency function (Problem 8.18).

8.20 A drawback of increasing the number of processes without increasing the total workload is that the speedup does not increase linearly with the number of processes, and the efficiency drops monotonically. Based on your experience with Problems 8.17 and 8.19, discuss whether using scaled speedup instead of standard speedup solves the problem in general. What can you say about the isoefficiency function of a parallel algorithm whose scaled speedup curve matches the speedup curve determined by increasing the problem size according to the isoefficiency function?

8.21 **(Time-constrained scaling)** Assume that $t_s = 10$ and $t_w = 1$ in the expression of parallel execution time (Equation 8.14) of the matrix-multiplication algorithm discussed in Section 8.2.1. For p = 1, 4, 16, 64, 256, 1024, and 4096, what is the largest problem that can be solved if the total run time is not to exceed 512 time units? In general, is it possible to solve an arbitrarily large problem in a fixed amount of time, provided that an unlimited number of processes is available? Give a brief explanation.

8.22 Describe a pipelined algorithm for performing back-substitution to solve a triangular system of equations of the form $Ux = y$, where the $n \times n$ unit upper-triangular matrix U is 2-D partitioned onto an $n \times n$ mesh of processes. Give an expression for the parallel run time of the algorithm. Modify the algorithm to work on fewer than n^2 processes, and derive an expression for the parallel execution time of the modified algorithm.

8.23 Consider the parallel algorithm given in Algorithm 8.7 for multiplying two $n \times n$ matrices A and B to obtain the product matrix C. Assume that it takes time t_{local} for a memory read or write operation on a matrix element and time t_c to add and multiply two numbers. Determine the parallel run time for this algorithm on an n^2-processor CREW PRAM. Is this parallel algorithm cost-optimal?

8.24 Assuming that concurrent read accesses to a memory location are serialized on an EREW PRAM, derive the parallel run time of the algorithm given in Algorithm 8.7 on an n^2-processor EREW PRAM. Is this algorithm cost-optimal on an EREW PRAM?

8.25 Consider a shared-address-space parallel computer with n^2 processors. Assume

```
1.  procedure MAT_MULT_CREW_PRAM ($A$, $B$, $C$, $n$)
2.  begin
3.      Organize the $n^2$ processes into a logical mesh of $n \times n$;
4.      for each process $P_{i,j}$ do
5.      begin
6.          $C[i, j] := 0$;
7.          for $k := 0$ to $n - 1$ do
8.              $C[i, j] := C[i, j] + A[i, k] \times B[k, j]$;
9.      endfor;
10. end MAT_MULT_CREW_PRAM
```

Algorithm 8.7 An algorithm for multiplying two $n \times n$ matrices A and B on a CREW PRAM, yielding matrix $C = A \times B$.

```
1.  procedure MAT_MULT_EREW_PRAM ($A$, $B$, $C$, $n$)
2.  begin
3.      Organize the $n^2$ processes into a logical mesh of $n \times n$;
4.      for each process $P_{i,j}$ do
5.      begin
6.          $C[i, j] := 0$;
7.          for $k := 0$ to $n - 1$ do
8.              $C[i, j] := C[i, j]+$
                        $A[i, (i + j + k) \bmod n] \times B[(i + j + k) \bmod n, j]$;
9.      endfor;
10. end MAT_MULT_EREW_PRAM
```

Algorithm 8.8 An algorithm for multiplying two $n \times n$ matrices A and B on an EREW PRAM, yielding matrix $C = A \times B$.

that each processor has some local memory, and $A[i, j]$ and $B[i, j]$ are stored in the local memory of processor $P_{i,j}$. Furthermore, processor $P_{i,j}$ computes $C[i, j]$ in its local memory. Assume that it takes time $t_{local} + t_w$ to perform a read or write operation on nonlocal memory and time t_{local} on local memory. Derive an expression for the parallel run time of the algorithm in Algorithm 8.7 on this parallel computer.

8.26 Algorithm 8.7 can be modified so that the parallel run time on an EREW PRAM is less than that in Problem 8.24. The modified program is shown in Algorithm 8.8. What is the parallel run time of Algorithm 8.8 on an EREW PRAM and a shared-address-space parallel computer with memory access times as described in Problems 8.24 and 8.25? Is the algorithm cost-optimal on these architectures?

8.27 Consider an implementation of Algorithm 8.8 on a shared-address-space parallel computer with fewer than n^2 (say, p) processors and with memory access times as described in Problem 8.25. What is the parallel runtime?

8.28 Consider the implementation of the parallel matrix multiplication algorithm presented in Section 8.2.1 on a shared-address-space computer with memory access times as given in Problem 8.25. In this algorithm, each processor first receives all the data it needs into its local memory, and then performs the computation. Derive the parallel run time of this algorithm. Compare the performance of this algorithm with that in Problem 8.27.

8.29 Use the results of Problems 8.23–8.28 to comment on the viability of the PRAM model as a platform for parallel algorithm design. Also comment on the relevance of the message-passing model for shared-address-space computers.