PRL05 - MNG

Model 2019

Paralelní a distribuované algoritmy

Část 5

Vyhledávání

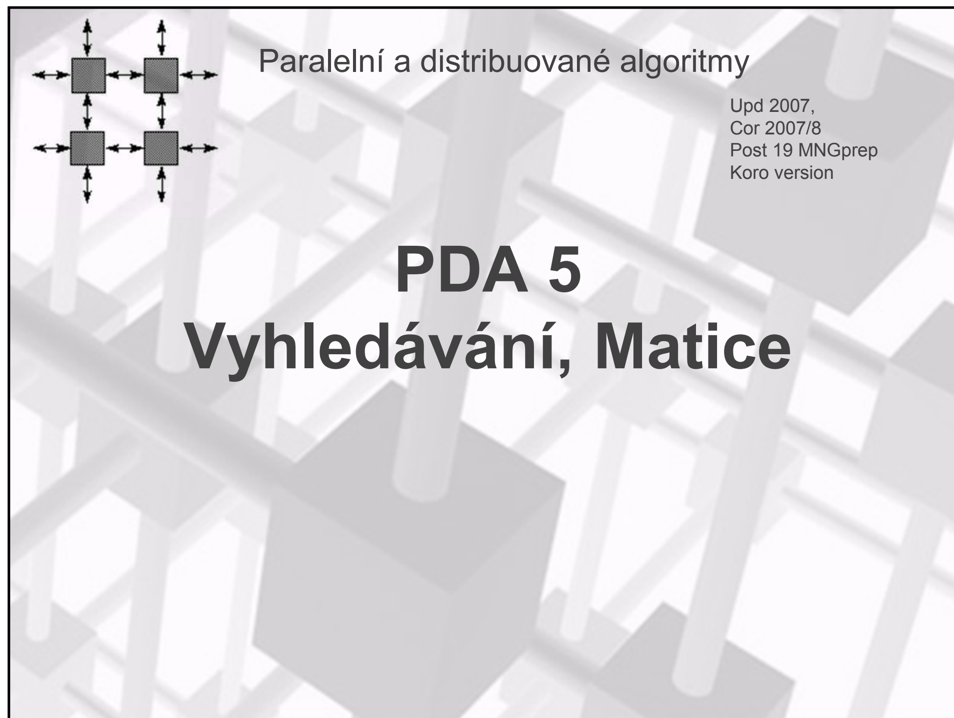
Post 19/20

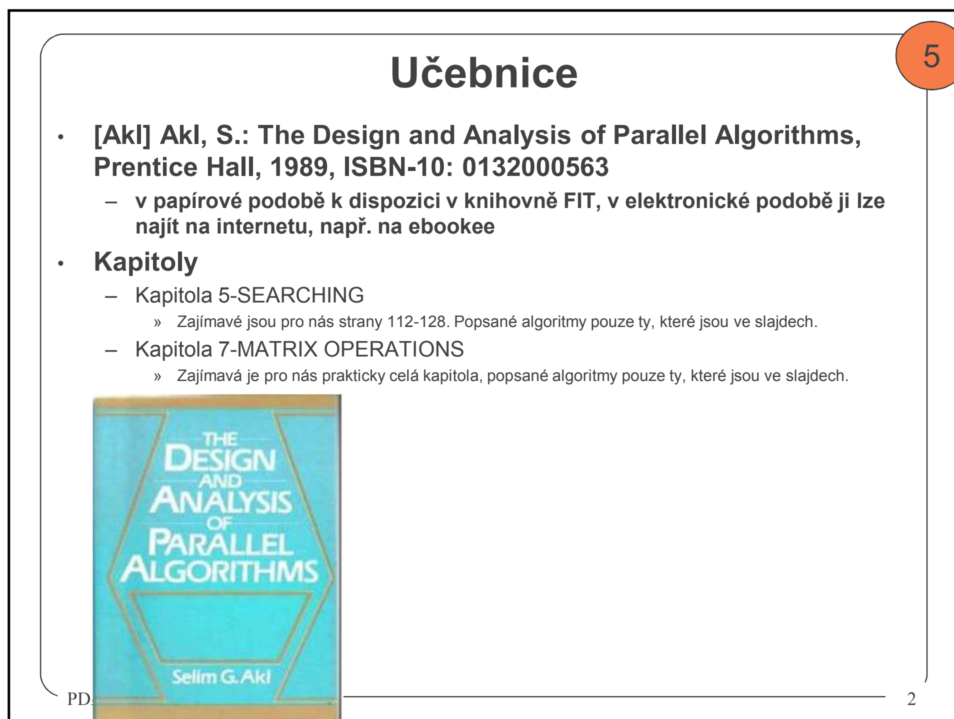
Souhrnné materiály

Ver 0.1

©Petr Hanáček

Paralelní a distribuované algoritmy





©

Veškeré kopírování, jak částečné, tak celého dokumentu je bez předchozího svolení autora zakázáno. Umístění tohoto dokumentu na jakýkoli (i neveřejný) server je zakázáno.

5. VYHLEDÁVÁNÍ

- Máme sekvenci $X = \{x_1, x_2, ..., x_n\}$ a prvek $\underline{x}$
- Máme za úkol zjistit, zda $x = x_k$ a případně zjistit $\underline{k}$
- Optimální sekvenční algoritmus
  a) X není seřazená - sekvenční vyhledávání
  t(n)=O(n) c(n)=O(n)
  (je třeba prozkoumat $\underline{n}$ prvků)
  b) X je seřazená - binární vyhledávání
  t(n)=O(log n) c(n)=O(log n)
  (pro rozlišení mezi $\underline{n}$ prvky je třeba prozkoumat log n prvků)

5.1 N-ary Search

- Vyhledává v seřazené posloupnosti
- Princip:
  - Při binárním vyhledávání zjistíme v každém kroku ve které polovině se prvek nachází za pomocí jednoho procesoru.
  - S použitím N procesorů lze provést N+1 ární vyhledávání - v jednom kroku zjistíme, ve které z N+1 části se může prvek nacházet
- Počet kroků je $g = \lceil \log(n+1)/\log(N+1) \rceil$

©

Veškeré kopírování, jak částečné, tak celého dokumentu je bez předchozího svolení autora zakázáno. Umístění tohoto dokumentu na jakýkoli (i neveřejný) server je zakázáno.

- Analýza
- Je třeba CREW PRAM
  - $t(n) = O(\log(n+1)/\log(N+1)) = O(\log_{N+1}(n+1))$
  - $c(n) = O(N.\log_{N+1}(n+1))$ →což není optimální

5.2 Unsorted Search

- vyhledává prvek v neseřazené posloupnosti
- model je PRAM s N procesory

```
Algoritmus
procedura SEARCH(S, x, k) { posloupnost, hledaný prvek, výsledek}
1. for i = 1 to N do in parallel
        read x
   endfor
2. for i = 1 to N do in parallel
        Si = {s(i-1).(n/N)+1, s(i-1).(n/N)+2, ..., si.(n/N)}
        SEQUENTIAL_SEARCH (Si, x, ki)
          - paralelní Search volá sekvenční SEQUENTAL Search
   endfor
3. for i = 1 to N do in parallel
        if ki > 0 then k = ki endif
   endfor
```


© Veškeré kopírování, jak částečné, tak celého dokumentu je bez předchozího svolení autora zakázáno. Umístění tohoto dokumentu na jakýkoli (i neveřejný) server je zakázáno.

Analýza

- **EREW**
  - 1. krok = O(log n) 2. krok = O(n/N) 3.krok = O(log N)
  - t(n) = O(log N + n/N) c(n) = O(N.log N + n)
- **CREW**
  - 1. krok = O(1) 2. krok = O(n/N) 3.krok = O(log N)
  - t(n) = O(log N + n/N) c(n) = O(N.log N + n)
- **CRCW**
  - 1. krok = O(1) 2. krok = O(n/N) 3.krok = O(1)
  - t(n) = O(n/N) c(n) = O(n) → což je optimální

PDA 7

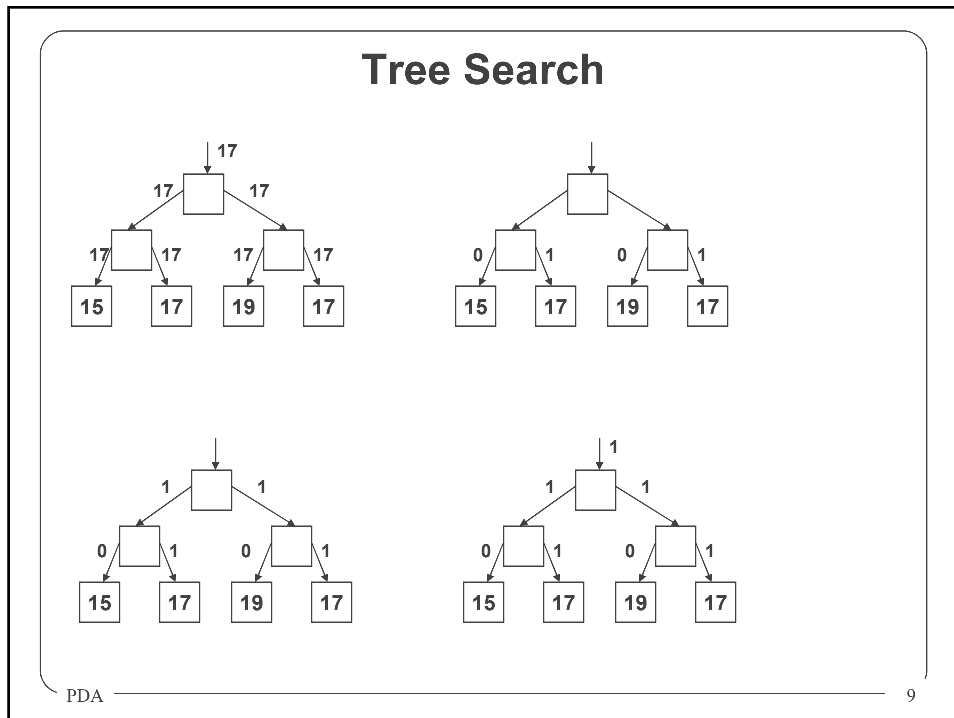
5.3 Tree Search

- Vyhledávání v neseřazené posloupnosti
- Stromová architektura s 2n-1 procesory

- Algoritmus
  - 1. Kořen načte hledanou hodnotu x a předá ji synům ... až se dostane ke všem listům
  - 2. Listy obsahují seznam prvků, ve kterých se vyhledává (každý list jeden). Všechny listy paralelně porovnávají x a x_i, výsledek je 0 nebo 1.
  - 3. Hodnoty všech listů se předají kořenu
    - každý ne list spočte logické or svých synů a výsledek zašle otci. Kořen dostane 0 - nenalezeno, 1- nalezeno
- Analýza
  - Krok (1) má složitost O(log n), krok (2) má konstantní složitost, krok (3) má O(log n).
  - t(n) = O(log n) p(n) = 2.n-1
  - c(n) = t(n).p(n) = O(n.log n) → což není optimální

PDA 8

© Veškeré kopírování, jak částečné, tak celého dokumentu je bez předchozího svolení autora zakázáno. Umístění tohoto dokumentu na jakýkoli (i neveřejný) server je zakázáno.

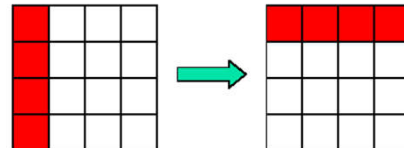


©
Veškeré kopírování, jak částečné, tak celého dokumentu je bez předchozího svolení autora zakázáno. Umístění tohoto dokumentu na jakýkoli (i neveřejný) server je zakázáno.

6. TRANSPOZICE

- Matici n x n s prvky a_{ij} převést na matici s prvky a_{ji}
- Sekvenční řešení:

```
procedure TRANSPOSE(A)
  for i=2 to n do
    for j=1 to i-1 do
      aij ⇔ aji
    endfor
  endfor
```

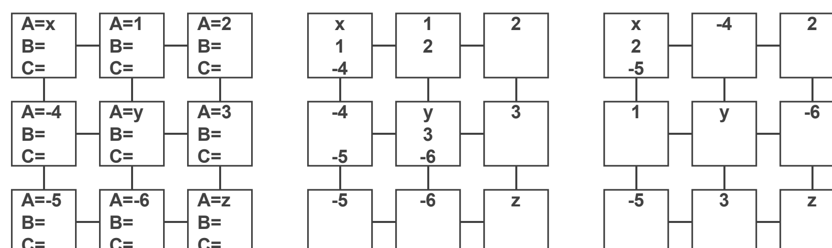


- Složitost je $O(n^2)$.
- **n je zde počet řádků/sloupců matice**

PDA 11

6.1 Mesh transpose

- Mřížka n x n procesorů
- Každý procesor má 3 registry
  - A - obsahuje a_{ij}, a_{ji} po ukončení
  - B - hodnoty od pravého (horního) souseda
  - C - hodnoty od levého (dolního) souseda



PDA 12

©

Veškeré kopírování, jak částečné, tak celého dokumentu je bez předchozího svolení autora zakázáno. Umístění tohoto dokumentu na jakýkoli (i neveřejný) server je zakázáno.

x	-4 -5	2
1 2	y	-6
-5	3	z

x	-4	-5
1	y	-6
2	3	z

- Analýza
  - Nejdelší cesta prvku je 2(n-1) kroků, tj.
  - $t(n) = O(n)$
  - $p(n) = n^2$ a cena $c(n) = O(n^3)$ → není optimální

EREW transpose

-

P21

P32

P31

© Veškeré kopírování, jak částečné, tak celého dokumentu je bez předchozího svolení autora zakázáno. Umístění tohoto dokumentu na jakýkoli (i neveřejný) server je zakázáno.

```
procedure EREW TRANSPOSE(A)
for i=2 to n do in parallel
  for j=1 to n-1 do in parallel
      a_ij ⇔ a_ji
  endfor
endfor
```

- Analýza
  - $t(n) = O(1)$
  - $p(n) = (n^2-n)/2 = O(n^2)$
  - $c(n) = O(n^2)$ → což je optimální

PDA 15

7. NÁSOBENÍ MATIC

- Součin matic A (m x n) a B (n x k) je matice C (m x k) kde:

$$c_{ij} = \sum_{s=1}^{n} a_{is} * b_{sj} \qquad 1<=i<=m \quad 1<=j<=k$$

Složitost je O (n^3)

Složitost optimálního algoritmu není známa, je O (n^x), kde 2<x<3

Žádný algoritmus nemá lepší složitost než O (n^2)

```
procedure MATRIX MULT(A, B, C)
for i=1 to m do
  for j=1 to k do
    c_ij = 0
    for s=1 to n do
        c_ij=c_ij + (a_is * b_sj)
    endfor
  endfor
endfor
```

PDA 16

© Veškeré kopírování, jak částečné, tak celého dokumentu je bez předchozího svolení autora zakázáno. Umístění tohoto dokumentu na jakýkoli (i neveřejný) server je zakázáno.

Násobení matic – sdílená paměť

- Nerealistická varianta

```
procedure MATRIX MULT(A, B, C)
for i=1 to m do in parallel
  for j=1 to k do  in parallel
    cij = 0
    for s=1 to n do  in parallel
         cij=cij + (ais * bsj)
    endfor
  endfor
endfor
```

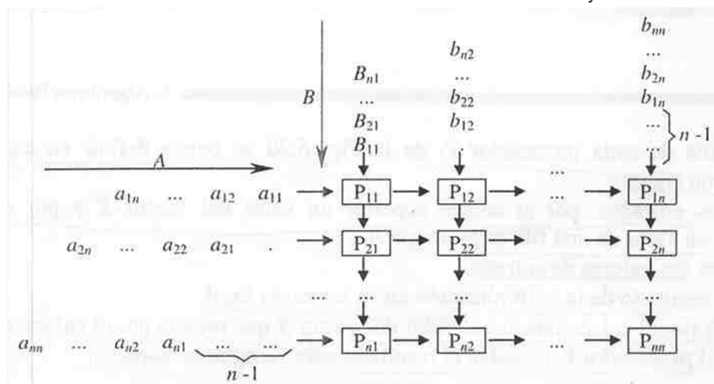
- Realistická varianta

```
procedure MATRIX MULT(A, B, C)
for i=1 to m do in parallel
  for j=1 to k do  in parallel
    cij = 0
    for s=1 to n do
         cij=cij + (ais * bsj)
    endfor
  endfor
endfor
```

PDA 17

7.1 Mesh multiplication

- Mřížka n x k procesorů
- Prvky matic A a B se přivádějí do procesorů 1. řádku a 1 sloupce
- Každý procesor P(i,j) obsahuje prvek c_{ij}



PDA 18

© Veškeré kopírování, jak částečné, tak celého dokumentu je bez předchozího svolení autora zakázáno. Umístění tohoto dokumentu na jakýkoli (i neveřejný) server je zakázáno.

```
Algoritmus
procedure MESH MULT(A, B, C)
for i=1 to m do in parallel
  for j=1 to k do in parallel
    cij = 0
    while P(i,j) receives inputs a,b do
         cij = cij + (a * b)
         if i<m then send b to P(i+1, j)
         if j<k then send a to P(i, j+1)
    endwhile
  endfor
endfor
```

- Analýza
- Prvky a_{m1} a b_{1k} potřebují m+k+n-2 kroků, aby se dostaly k poslednímu procesoru P(m, k)
- $t(n) = O(n)$ $p(n)=O(n^2)$
- $c(n) = O(n^3)$ → což není optimální

7.1.1 Násobení matice vektorem

- Součin matice A (m x n) a vektoru U (n) je vektor V (m) kde:

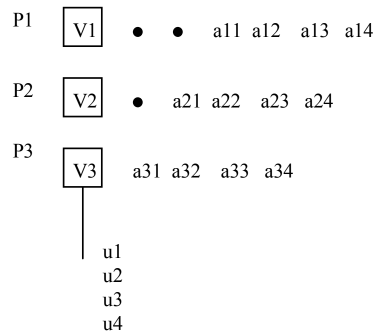
$$v_i = \sum_{j=1}^{n} a_{ij} * u_j \qquad 1 \le i \le m$$

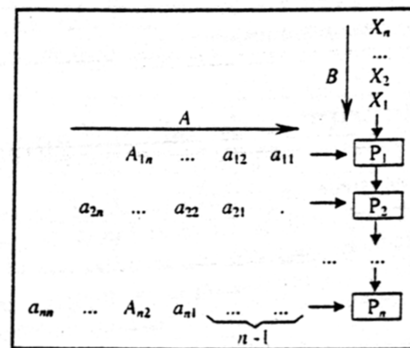
© Veškeré kopírování, jak částečné, tak celého dokumentu je bez předchozího svolení autora zakázáno. Umístění tohoto dokumentu na jakýkoli (i neveřejný) server je zakázáno.

10

7.1.2 Linear array multiplication

- Lineární pole m procesorů, každý obsahuje jeden prvek v_i





- Algoritmus

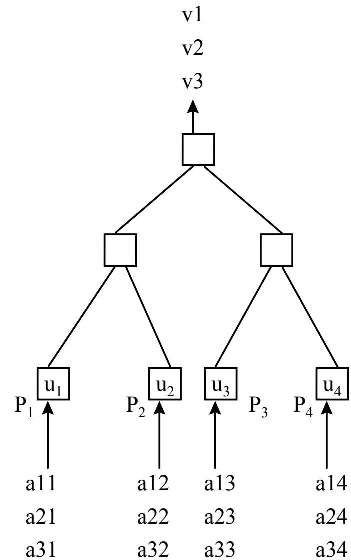
```
procedure LINEAR MV MULT
for i=1 to m do in parallel
  vi = 0
  while Pi receives inputs a and u do
    vi = vi + (a * u)
    if i>1 then send u to Pi-1
  endwhile
endfor
```

- Analýza
  - $t(n) = m+n-1 = t(n)=O(n)$ $p(n)=O(n)$
  - $c(n)=O(n^2)$ → což je optimální

© Veškeré kopírování, jak částečné, tak celého dokumentu je bez předchozího svolení autora zakázáno. Umístění tohoto dokumentu na jakýkoli (i neveřejný) server je zakázáno.

7.2 Tree MV multiplication

- Čas m+n-1 předchozího algoritmu je možno zlepšit na m-1+log n při zdvojnásobení počtu procesorů
- Architektura má n listových procesorů $P_1 \ldots P_n$ a n-1 nelistových procesorů
- Listové procesory násobí, nelistové sčítají




```
Algoritmus
procedure TREE MV MULT(A, U V)
do steps 1 and 2 in parallel
(1) for i=1 to n do in parallel
        for j=1 to m do
                send ui*dij to parent
        endfor
    endfor
(2) for i=n+1 to 2n-1 do in parallel
        while Pi receives two inputs do
                compute the sum
                if i<2n-1       then send result to parent
                                else write result
                endif
        endwhile
    endfor
```

- Analýza
  - t(n)=m-1+ log n = O(n)
  - $c(n)=O(n^2)$ → což je optimální

© Veškeré kopírování, jak částečné, tak celého dokumentu je bez předchozího svolení autora zakázáno. Umístění tohoto dokumentu na jakýkoli (i neveřejný) server je zakázáno.

©

Veškeré kopírování, jak částečné, tak celého dokumentu je bez předchozího svolení autora zakázáno. Umístění tohoto dokumentu na jakýkoli (i neveřejný) server je zakázáno.

5

Searching

5.1 INTRODUCTION

Searching is one of the most fundamental operations in the field of computing. It is used in any application where we need to find out whether an element belongs to a list or, more generally, retrieve from a file information associated with that element. In its most basic form the ***searching problem*** is stated as follows: Given a sequence $S = \{s_1, s_2, \ldots, s_n\}$ of integers and an integer x, it is required to determine whether $x = s_k$ for some s_k in S.

In sequential computing, the problem is solved by scanning the sequence S and comparing x with its successive elements until either an integer equal to x is found or the sequence is exhausted without success. This is given in what follows as procedure SEQUENTIAL SEARCH. As soon as an s_k in S is found such that $x = s_k$, the procedure returns k; otherwise 0 is returned.

```
procedure SEQUENTIAL SEARCH (S, x, k)

Step 1: (1.1) i ← 1
        (1.2) k ← 0.
Step 2: while (i ≤ n and k = 0) do
           if s_i = x then k ← i end if
           i ← i + 1
        end while. □
```

In the worst case, the procedure takes $O(n)$ time. This is clearly optimal since every element of S must be examined (when x is not in S) before declaring failure. Alternatively, if S is sorted in nondecreasing order, then procedure BINARY SEARCH of section 3.3.2 can return the index of an element of S equal to x (or 0 if no such element exists) in $O(\log n)$ time. Again, this is optimal since this many bits are needed to distinguish among the n elements of S.

In this chapter we discuss parallel searching algorithms. We begin by considering the case where S is sorted in nondecreasing order and show how searching can be performed on the SM SIMD model. As it turns out, our EREW searching algorithm is

no faster than procedure BINARY SEARCH. On the other hand, the CREW algorithm matches a lower bound on the number of parallel steps required to search a sorted sequence, assuming that all the elements of S are distinct. When this assumption is removed, a CRCW algorithm is needed to achieve the best possible speedup. We then turn to the more general case where the elements of S are in random order. Here, although the SM SIMD algorithms are faster than procedure SEQUENTIAL SEARCH, the same speedup can be achieved on a weaker model, namely, a tree-connected SIMD computer. Finally, we present a parallel search algorithm for a mesh-connected SIMD computer that, under some assumptions about signal propagation time along wires, is superior to the tree algorithm.

5.2 SEARCHING A SORTED SEQUENCE

We assume throughout this section that the sequence $S = \{s_1, s_2, \ldots, s_n\}$ is sorted in nondecreasing order, that is, $s_1 \leqslant s_2 \leqslant \cdots \leqslant s_n$. Typically, a file with n records is available, which is sorted on the s field of each record. This file is to be searched using s as the *key;* that is, given an integer x, a record is sought whose s field equals x. If such a record is found, then the information stored in the other fields may now be retrieved. The format of a record is illustrated in Fig. 5.1. Note that if the values of the s fields are not unique and all records whose s fields equal a given x are needed, then the search algorithm is continued until the file is exhausted. For simplicity we begin by assuming that the s_i are distinct; this assumption is later removed.

5.2.1 EREW Searching

Assume that an N-processor EREW SM SIMD computer is available to search S for a given element x, where $1 < N \leqslant n$. To begin, the value of x must be made known to all processors. This can be done using procedure BROADCAST in $O(\log N)$ time. The sequence S is then subdivided into N subsequences of length n/N each, and processor P_i is assigned $\{s_{(i-1)(n/N)+1}, s_{(i-1)(n/N)+2}, \ldots, s_{i(n/N)}\}$. All processors now perform procedure BINARY SEARCH on their assigned subsequences. This requires $O(\log(n/N))$ in the worst case. Since the elements of S are all distinct, at most one processor finds an s_k equal to x and returns k. The total time required by this EREW searching algorithm is therefore $O(\log N) + O(\log(n/N))$, which is $O(\log n)$. Since this is precisely the time required by procedure BINARY SEARCH (running on a single processor!), no speedup is achieved by this approach.

Figure 5.1 Format of record in file to be searched.

5.2.2 CREW Searching

Again, assume that an N-processor CREW SM SIMD computer is available to search S for a given element **x**, where $1 < N \leqslant n$. The same algorithm described for the EREW computer can be used here except that in this case all processors can read x simultaneously in constant time and then proceed to perform procedure BINARY SEARCH on their assigned subsequences. This requires $O(\log(n/N))$ time in the worst case, which is faster than procedure BINARY SEARCH applied sequentially to the entire sequence.

It is possible, however, to do even better. The idea is to use a parallel version of the binary search approach. Recall that during each iteration of procedure BINARY SEARCH the middle element s_m of the sequence searched is probed and tested for equality with the input x. If $s_m > x$, then all the elements larger than s_m are discarded; otherwise all the elements smaller than s_m are discarded. Thus, the next iteration is applied to a sequence half as long as previously. The procedure terminates when the probed element equals x or when all elements have been discarded. In the parallel version, there are N processors and hence an $(N + 1)$-ary search can be used. At each stage, the sequence is split into $N + 1$ subsequences of equal length and the N processors simultaneously probe the elements at the boundary between successive subsequences. This is illustrated in Fig. 5.2. Every processor compares the element s of S it probes with **x**:

1. If $s > x$, then if an element equal to **x** is in the sequence at all, it must precede s; consequently, s and all the elements that follow it (i.e., to its right in Fig. 5.2) are removed from consideration.
2. The opposite takes place if $s < x$.

Thus each processor splits the sequence into two parts: those elements to be discarded as they definitely do not contain an element equal to x and those that might and are hence kept. This narrows down the search to the intersection of all the parts to be kept, that is, the subsequence between two elements probed in this stage. This subsequence, shown hachured in Fig. 5.2, is searched in the next stage by the same process. This continues until either an element equal to **x** is found or all the elements of S are discarded. Since every stage is applied to a sequence whose length is $1/(N + 1)$ the length of the sequence searched during the previous stage less 1, $O(\log_{N+1}(n + 1))$ stages are needed. We now develop the algorithm formally and then show that this is precisely the number of steps it requires in the worst case.

Let g be the smallest integer such that $n \leqslant (N + 1)^g - 1$, that is, $g = \lceil \log(n + 1)/\log(N + 1) \rceil$. It is possible to prove by induction that g stages are sufficient to search a sequence of length n for an element equal to an input **x**. Indeed, the statement is true for $g = 0$. Assume it is true for $(N + 1)^{g-1} - 1$. Now, to search a sequence of length $(N + 1)^g - 1$, processor P_i, $i = 1, 2, \ldots, N$, compares x to s_j where $j = i(N + 1)^{g-1}$, as shown in Fig. 5.3. Following this comparison, only a subsequence of length $(N + 1)^{g-1} - 1$ needs to be searched, thus proving our claim. This subsequence, shown hachured in Fig. 5.3, can be determined as follows. Each

processor P_i uses a variable c_i that takes the value left or ***right*** according to whether the part of the sequence P_i decides to keep is to the left or right of the element it compared to x during this stage. Initially, the value of each c_i is irrelevant and can be assigned arbitrarily. Two constants c_0 = right and c_{N+1} = left are also used. Following the comparison between x and an element s_{j_i} of S, P_i assigns a value to c_i (unless $s_{j_i} = x$, in which case the value of c_i is again irrelevant). If $c_i \neq c_{i-1}$ for some i, $1 \leqslant i \leqslant N$, then the sequence to be searched next runs from s_q to s_r, where $q = (i-1)(N+1)^{g-1} + 1$ and $r = i(N+1)^{g-1} - 1$. Precisely one processor updates q and r in the shared memory, and all remaining processors can simultaneoiusly read the updated values in constant time. The algorithm is given in what follows as procedure **CREW SEARCH.** The procedure takes S and x as input: If $x = s_k$ for some k, then k is returned; otherwise a 0 is returned.

procedure CREW SEARCH (S, x, k)

```
Step 1: {Initialize indices of sequence to be searched}
        (1.1) q ← 1
        (1.2) r ← n.
Step 2: {Initialize results and maximum number of stages}
        (2.1) k ← 0
        (2.2) g ← ⌈log(n + 1)/log(N + 1)⌉.
Step 3: while (q ≤ r and k = 0) do
        (3.1) j_0 ← q − 1
        (3.2) for i = 1 to N do in parallel
                (i) j_i ← (q − 1) + i(N + 1)^(g−1)
                      {P_i compares x to s_j and determines the part of the sequence to be kept}
               (ii) if j_i ≤ r
                      then if s_(j_i) = x
                             then k ← j_i
                             else if s_(j_i) > x
                                    then c_i ← left
                                    else c_i ← right
                                    end if
                             end if
                      else (a) j_i ← r + 1
                           (b) c_i ← left
                      end if
                        {The indices of the subsequence to be searched in the next iteration are
                        computed}
              (iii) if c_i ≠ c_(i−1) then (a) q ← j_(i−1) + 1
                                          (b) r ← j_i − 1
                      end if
               (iv) if (i = N and c_i ≠ c_(i+1)) then q ← j_i + 1
                      end if
              end for
        (3.3) g ← g − 1.
        end while. □
```

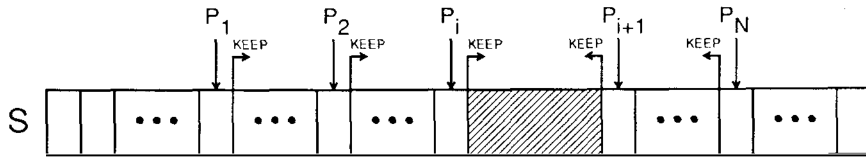


Figure 5.2 Searching sorted sequence with N processors.

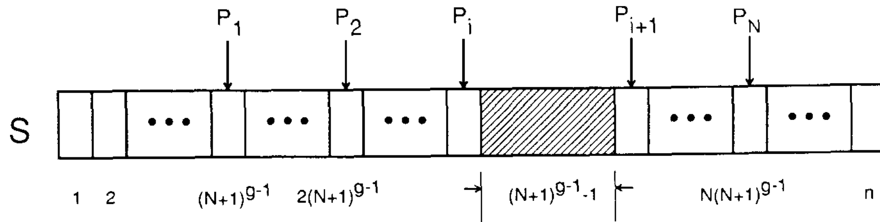


Figure 5.3 Derivation of number of stages required to search sequence.

Analysis

Steps 1, 2, 3.1, and 3.3 are performed by one processor, say, P_1, in constant time. Step 3.2 also takes constant time. As proved earlier, there are at most g iterations of step 3. It follows that procedure CREW SEARCH runs in $O(\log(n + 1)/\log(N + 1))$ time, that is, $t(n) = O(\log_{N+1}(n + 1))$. Hence $c(n) = O(N \log_{N+1}(n + 1))$, which is not optimal.

Example 5.1

Let $S = \{1, 4, 6, 9, 10, 11, 13, 14, 15, 18, 20, 23, 32, 45, 51\}$ be the sequence to be searched using a CREW SM SIMD computer with N processors. We illustrate two successful and one unsuccessful searches.

1. Assume that $N = 3$ and that it is required to find the index k of the element in S equal to 45 (i.e., $x = 45$). Initially, $q = 1$, $r = 15$, $k = 0$, and $g = 2$. During the first iteration of step 3, P, computes $j_1 = 4$ and compares s_4 to x. Since $9 < 45$, c, = right. Simultaneously, P_2 and P_3 compares, and s_{12}, respectively, to x:Since $14 < 45$ and $23 < 45$, c_2 = right and c_3 = right. Now $c_3 \neq c_4$; therefore $q = 13$ and r remains unchanged. The new sequence to be searched runs from s_{13} to s,,, *as* shown in Fig. 5.4(a), and $g = 1$. In the second iteration, illustrated in Fig. 5.4(b), P_1 computes $j_1 = 12 + 1$ and compares s_{13} to x: Since $32 < 45$, c_1 = right. Simultaneously, P_2 compares s_{14} to x, and since they are equal, it sets k to 14 (c_2 remains unchanged). Also, P_3 compares s_{15} to x: Since $51 > 45$, c_3 = left. Now $c_3 \neq c$,: Thus $q = 12 + 2 + 1 = 15$ and $r = 12 + 3 - 1 = 14$. The procedure terminates with $k = 14$.
2. Say now that $x = 9$, with N still equal to 3. In the first iteration, P_1 compares s_4 to x: Since they are equal, k is set to 4. All simultaneous and subsequent computations in this iteration are redundant since the following iteration is not performed and the procedure terminates early with $k = 4$.

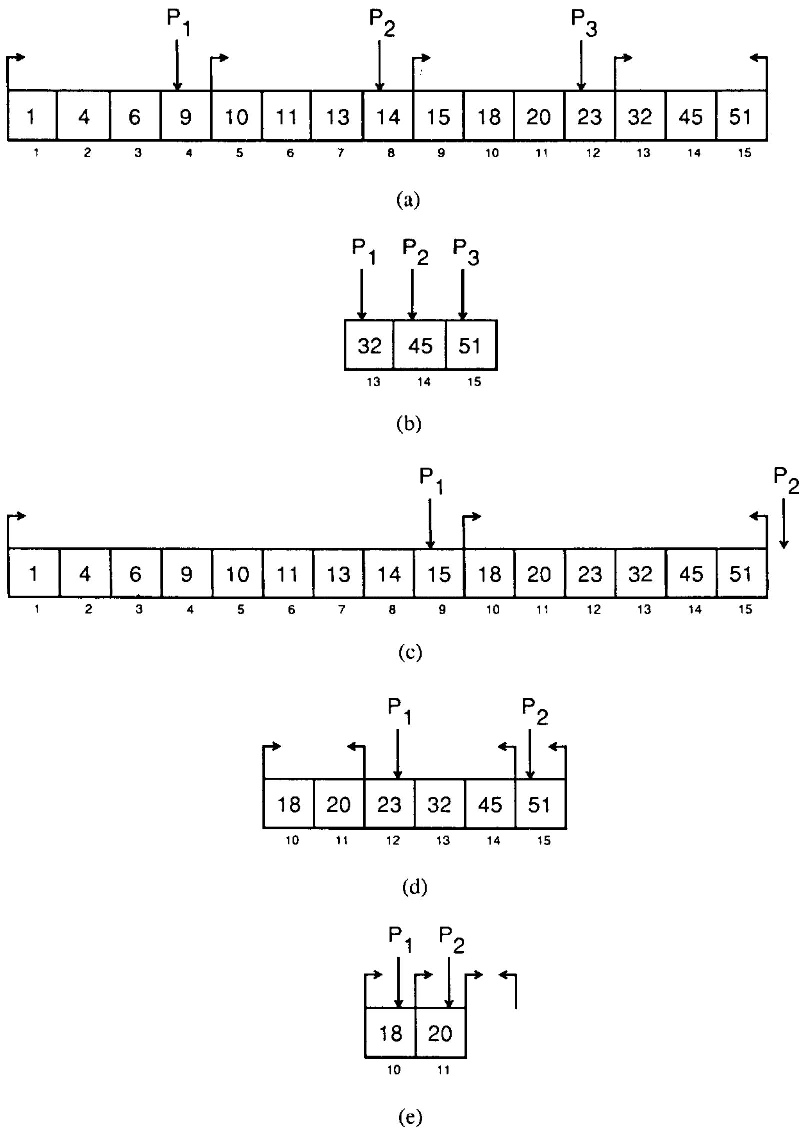


Figure 5.4 Searching sequence of fifteen elements using procedure CREW SEARCH.

3. Finally, let N = 2 and $x = 21$. Initially, $g = 3$. In the first iteration P_1 computes $j_1 = 9$ and compares s_9 to x: Since $15 < 21$, c_1 = right. Simultaneously, P_2 computes $j_2 = 18$: Since $18 > 15$, j_2 points to an element outside the sequence. Thus P_2 sets $j_2 = 16$ and c_2 = left. Now $c_2 \neq c_1$: Therefore $q = 10$ and $r = 15$, that is, the sequence to be searched in the next iteration runs from s_{10} to s_{15}, and $g = 2$. This is illustrated in Fig. 5.4(c). In the second iteration, P_1 computes $j_1 = 9 + 3$ and

compares s_{12} to x: since $23 > 21$, c_1 = left. Simultaneously, P_2 computes $j_2 = 15$: Since $51 > 21$, c_2 = left. Now $c_1 \neq c_0$, and therefore $r = 11$ and q remains unchanged, as shown in Fig. 5.4(d). In the final iteration, g = 1 and P_1 computes $j_1 = 9 + 1$ and compares s_{10} to x: Since $18 < 21$, c_1 = right. Simultaneously, P_2 computes $j_2 = 9 + 2$ and compares s_{11} to x: Since $20 < 21$, c_2 = right. Now $c_1 \neq c_2$, and therefore $q = 12$. Since q > r, the procedure terminates unsuccessfully with $k = 0$. □

We conclude our discussion of parallel searching algorithms for the CREW model with the following two observations:

1. Under the assumption that the elements of S are sorted and distinct, procedure CREW SEARCH, although not cost optimal, achieves the best possible running time for searching. This can be shown by noting that any algorithm using N processors can compare an input element x to at most N elements of S simultaneously. After these comparisons and the subsequent deletion of elements from S definitely not equal to **x**, a subsequence must be left whose length is at least

$$\lceil (n - N)/(N + 1) \rceil \geqslant (n - N)/(N + 1) = [(n + 1)/(N + 1)] - 1.$$

   After g repetitions of the same process, we are left with a sequence of length $[(n + 1)/(N + 1)^g] - 1$. It follows that the number of iterations required by any such parallel algorithm is no smaller than the minimum g such that

$$[(n + 1)/(N + 1)^g] - 1 \leqslant 0,$$

   which is

$$\lceil \log(n + 1)/\log(N + 1) \rceil.$$

2. Two parallel algorithms were presented in this section for searching a sequence of length n on a CREW SM SIMD computer with N processors. The first required $O(\log(n/N))$ time and the second $O(\log(n + 1)/\log(N + 1))$. In both cases, if N = n, then the algorithm runs in constant time. The fact that the elements of S are distinct still remains a condition for achieving this constant running time, as we shall see in the next section. However, we no longer need S to be sorted. The algorithm is simply as follows: In one step each P_i, $i = 1, 2, \ldots, n$, can read x and compare it to s_i; if x is equal to one element of S, say, s_k, then P_k returns k; otherwise k remains 0.

5.2.3 CRCW Searching

In the previous two sections, we assumed that all the elements of the sequence S to be searched are distinct. From our discussion so far, the reason for this assumption may have become apparent: If each s_i is not unique, then possibly more than one processor will succeed in finding a member of S equal to **x**. Consequently, possibly several

processors will attempt to return a value in the variable k, thus causing a write conflict, an occurrence disallowed in both the EREW and CREW models. Of course, we can remove the uniqueness assumption and still use the EREW and CREW searching algorithms described earlier. The idea is to invoke procedure STORE (see problem 2.13) whose job is to resolve write conflicts: Thus, in $O(\log N)$ time we can get the smallest numbered of the successful processors to return the index k it has computed, where $s_k = x$. The asymptotic running time of the EREW search algorithm in section 5.2.1 is not affected by this additional overhead. However, procedure CREW SEARCH now runs in

$$t(n) = O(\log(n + 1)/\log(N + 1)) + O(\log N).$$

In order to appreciate the effect of this additional $O(\log N)$ term, note that when $N = n$, $t(n) = O(\log n)$. In other words, procedure CREW SEARCH with n processors is no faster than procedure BINARY SEARCH, which runs on one processor!

Clearly, in order to maintain the efficiency of procedure CREW SEARCH while giving up the uniqueness assumption, we must run the algorithm on a CRCW SM SIMD computer with an appropriate write conflict resolution rule. Whatever the rule and no matter how many processors are successful in finding a member of S equal to x, only one index k will be returned, and that in constant time.

5.3 SEARCHING A RANDOM SEQUENCE

We now turn to the more general case of the search problem. Here the elements of the sequence $S = \{s_1, s_2, \ldots, s_n\}$ are not assumed to be in any particular order and are not necessarily distinct. As before, we have a file with n records that is to be searched using the s field of each record as the key. Given an integer x, a record is sought whose s field equals x; if such a record is found, then the information stored in the other fields may now be retrieved. This operation is referred to as querying the file. Besides querying, search is useful in file maintenance, such as inserting a new record and updating or deleting an existing record. Maintenance, as we shall see, is particularly easy when the s fields are in random order.

We begin by studying parallel search algorithms for shared-mernory SIMD computers. We then show how the power of this model is not really needed for the search problem. As it turns out, performance similar to that of SM SIMD algorithms can be obtained using a tree-connected SIMD computer. Finally, we demonstrate that a mesh-connected computer is superior to the tree for searching if signal propagation time along wires is taken into account when calculating the running time of algorithms for both models.

5.3.1 Searching on SM SIMD Computers

The general algorithm for searching a sequence in random order on a SM SIMD computer is straightforward and similar in structure to the algorithm in section 5.2.1.

We have an N-processor computer to search $S = \{s_1, s_2, \ldots, s_n\}$ for a given element x, where $1 < N \leqslant n$. The algorithm is given as procedure SM SEARCH:

```
procedure SM SEARCH (S, x, k)

Step 1: for i = 1 to N do in parallel
            Read x
        end for.

Step 2: for i = 1 to N do in parallel
          (2.1) S_i <- {s_((i-1)(n/N)+1), s_((i-1)(n/N)+2), ..., s_(i(n/N))}
          (2.2) SEQUENTIAL SEARCH (S_i, x, k_i)
        end for.

Step 3: for i = 1 to N do in parallel
            if k_i > 0 then k <- k_i end if
        end for. □
```

Analysis

We now analyze procedure SM SEARCH for each of the four incarnations of the shared-memory model of SIMD computers.

5.3.1.1 EREW. Step 1 is implemented using procedure BROADCAST and requires $O(\log N)$ time. In step 2, procedure SEQUENTIAL SEARCH takes $O(n/N)$ time in the worst case. Finally, procedure STORE (with an appropriate conflict resolution rule) is used in step 3 and runs in $O(\log N)$ time. The overall asymptotic running time is therefore

$$t(n) = O(\log N) + O(n/N),$$

and the cost is

$$c(n) = O(N \log N) + O(n),$$

which is not optimal.

5.3.1.2 ERCW. Steps 1 and 2 are as in the EREW case, while step 3 now takes constant time. The overall asymptotic running time remains unchanged.

5.3.1.3 CREW. Step 1 now takes constant time, while steps 2 and 3 are as in the EREW case. The overall asymptotic running time remains unchanged.

5.3.1.4 CRCW. Both steps 1 and 3 take constant time, while step 2 is as in the EREW case. The overall running time is now $O(n/N)$, and the cost is

$$c(n) = N \times O(n/N) = O(n),$$

which is optimal.

In order to put the preceding results in perspective, let us consider a situation where the following two conditions hold:

1. There are as many processors as there are elements in S, that is, $N = n$.
2. There are q queries to be answered, that is, q values of x are queued waiting for processing.

In the case of the EREW, ERCW, and CREW models, the time to process one query is now $O(\log n)$. For q queries, this time is simply multiplied by a factor of q. This is of course an improvement over the time required by procedure SEQUENTIAL SEARCH, which would be on the order of qn. For the CRCW computer, procedure SM SEARCH now takes constant time. Thus q queries require a constant multiple of q time units to be answered.

Surprisingly, a performance slightly inferior to that of the CRCW algorithm but still superior to that of the EREW algorithm can be obtained using a much weaker model, namely, the tree-connected SIMD computer. Here a binary tree with $O(n)$ processors processes the queries in a pipeline fashion: Thus the q queries require a constant multiple of $\log n + (q - 1)$ time units to be answered. For large values of q (i.e., $q > \log n$), this behavior is equivalent to that of the CRCW algorithm. We now turn to the description of this tree algorithm.

5.3.2 Searching on a Tree

A tree-connected SIMD computer with n leaves is available for searching a file of n records. Such a tree is shown in Fig. 5.5 for $n = 16$. Each leaf of the tree stores one record of the file to be searched. The root is in charge of receiving input from the outside world and passing a copy of it to each of its two children. It is also responsible for producing output received from its two children to the outside world. As for the intermediate nodes, each of these is capable of:

1. receiving one input from its parent, making two copies of it, and sending one copy to each of its two children; and
2. receiving two inputs from its children, combining them, and passing the result to its parent.

The next two sections illustrate how the file stored in the leaves can be queried and maintained.

5.3.2.1 Querying. Given an integer x, it is required to search the file of records on the s field for x, that is, determine whether there is a value in $S = \{s_1, s_2, \ldots, s_n\}$ equal to x. Such a query only requires a yes or no answer. This is the most basic form of querying and is even simpler than the one that we have been concerned with so far in this chapter. The tree-connected computer handles this query

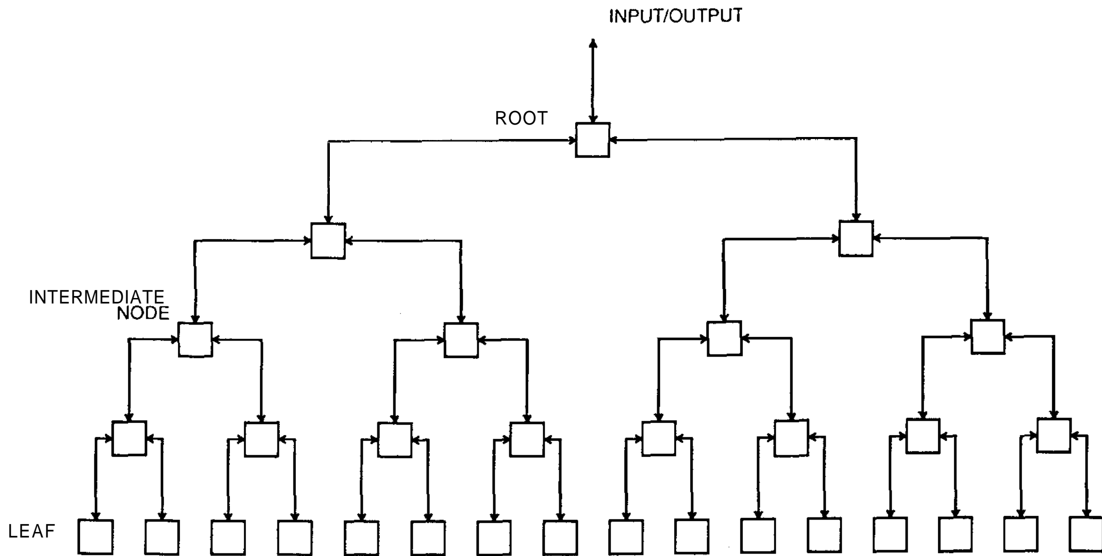


Figure 5.5 Tree-connected computer for searching.

in three stages:

Stage 1: The root reads x and passes it to its two children. In turn, these send x to their children. The process continues until a copy of x reaches each leaf.

Stage 2: Simultaneously, all leaves compare the s field of the record they store to x: If they are equal, the leaf produces a 1 as output: otherwise a 0 is produced.

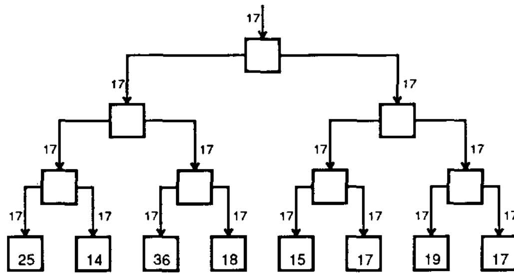
Stage 3: The outputs of the leaves are combined by going upward in the tree: Each intermediate node computes the logical **or** of its two inputs (i.e., 0 **or** 0 = 0, 0 **or** 1 = 1, 1 **or** 0 = 1, and 1 **or** 1 = 1) and passes the result to its parent. The process continues until the root receives two bits, computes their logical **or**, and produces either a 1 (for yes) or a 0 (for no).

It takes $O(\log n)$ time to go down the tree, constant time to perform the comparison at the leaves, and again $O(\log n)$ time to go back up the tree. Therefore, such a query is answered in $O(\log n)$ time.

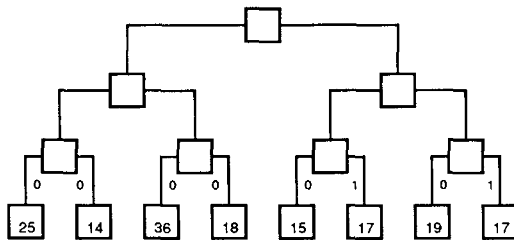
Example 5.2

Let $S = \{25, 14, 36, 18, 15, 17, 19, 17\}$ and $x = 17$. The three stages above are illustrated in Fig. 5.6. □

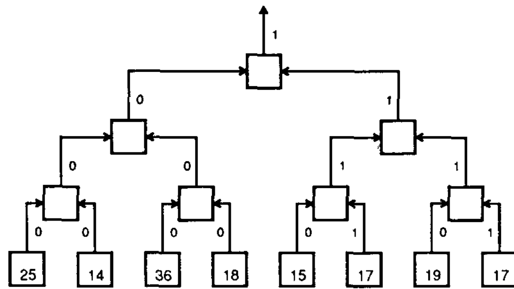
Assume now that q such queries are queued waiting to be processed. They can be pipelined down the tree since the root and intermediate nodes are free to handle the next query as soon as they have passed the current one along to their children. The same remark applies to the leaves: As soon as the result of one comparison has been



(a) STAGE 1



(b) STAGE 2



(c) STAGE 3

Figure 5.6 Searching sequence of eight elements using tree.

produced, each leaf is ready to receive a new value of **x**. The results are also pipelined upward: The root and intermediate nodes can compute the logical **or** of the next pair of bits as soon as the current pair has been cleared. Typically, the root and intermediate nodes will receive data flowing downward (queries) and upward (results) simultaneously: We assume that both can be handled in a single time unit; otherwise, and in order to keep both flows of data moving, a processor can switch its attention from one direction to the other alternately. It takes $O(\log n)$ time for the answer to the

first query to be produced at the root. The answer to the second query is obtained in the following time unit. The answer to the last query emerges $q - 1$ time units after the first answer. Thus the q answers are obtained in a total of $O(\log n) + O(q)$ time.

We now examine some variations over the basic form of a query discussed so far.

1. Position If a query is successful and element s_k is equal to x, it may be desired to know the index k. Assume that the leaves are numbered $1, \ldots, n$ and that leaf i contains s_i. Following the comparison with **x**, leaf i produces the pair (1, i) if $s_i = x$; otherwise it produces (0, i). All intermediate nodes and the root now operate as follows. If two pairs (1, i) and (0, j) are received, then the pair (1, i) is sent upward. Otherwise, if both pairs have a 1 as a first element or if both pairs have a 0 as a first element, then the pair arriving from the left son is sent upward. In this way, the root produces either

(i) (1, k) where k is the smallest index of an element in S equal to x or
(ii) (0, k) indicating that no match for x was found and, therefore, that the value of k is meaningless.

With this modification, the root in example 5.2 would produce (1, 6).

This variant of the basic query can itself be extended in three ways:

(a) When a record is found whose s field equals **x**, it may be desirable to obtain the entire record as an answer to the query (or perhaps some of its fields). The preceding approach can be generalized by having the leaf that finds a match return a triple of the form (1, i, required information). The intermediate nodes and root behave as before.
(b) Sometimes, the positions of all elements equal to x in S may be needed. In this case, when an intermediate node, or the root, receives two pairs (1, i) and (1, j), two pairs are sent upward consecutively. In this way the indices of all members of S equal to x will eventually emerge from the root.
(c) The third extension is a combination of (a) and (b): All records whose s fields match x are to be retrieved. This is handled by combining the preceding two solutions

It should be noted, however, that for each of the preceding extensions care must be taken with regards to timing if several queries are being pipelined. This is because the result being sent upward by each node is no longer a single bit but rather many bits of information from potentially several records (in the worst case the answer consists of the n entire records). Since the answer to a query is now of unpredictable length, it is no longer guaranteed that a query will be answered in $O(\log n)$ time, that the period is constant, or that q queries will be processed in $O(\log n) + O(q)$ time.

2. Count Another variant of the basic query asks for the number of records whose s field equals x. This is handled exactly as the basic query, except that now the

intermediate nodes and the root compute the sum of their inputs (instead of the logical **or).** With this modification, the root in example 5.2 would produce a 2.

3. Closest Element Sometimes it may be useful to find the element of S whose value is closest to x. As with the basic query, x is first sent to the leaves.. Leaf i now computes the absolute value of $s_i - x$, call it a_i, and produces (i, a,) as output.

Each intermediate node and the root now receive two pairs (i, a,) and (j, a_j): The pair with the smaller a component is sent upward. With this modification and $x = 38$ as input, the root in example 5.2 would produce (3, 2) as output. Note that the case of two pairs with identical a components is handled either by choosing one of the two arbitrarily or by sending both upward consecutively.

4. Rank The rank of an element **x** in S is defined as the number of **elements** of S smaller than **x** plus 1. We begin by sending x to the leaves and then having each leaf i produce a 1 if $s_i < x$, and a 0 otherwise. Now the rank of x in S is computed by making all intermediate nodes add their inputs and send the result upward. The root adds 1 to the sum of its two inputs before producing the rank. With this modification, the root's output in example 5.2 would be **3.**

It should be emphasized that each of the preceding variants, if **carefully** timed, should have the same running time as the basic query (except, of course, when the queries being processed do not have constant-length answers as pointed out earlier).

5.3.2.2 Maintenance. We now address the problem of maintaining a file of records stored at the leaves of a tree, that is, inserting a new record and updating or deleting an existing record.

1. Insertion In a typical file, records are inserted and deleted continually. It is therefore reasonable to assume that at any given time a number of leaves are unoccupied. We can keep track of the location of these unoccupied leaves by storing in each intermediate node and at the root

(i) the number of unoccupied leaves in its left subtree and
(ii) the number of unoccupied leaves in its right subtree.

A new record received by the root is inserted into an unoccupied leaf as follows:

(i) The root passes the record to the one of its two subtrees with unoccupied leaves. If both have unoccupied leaves, the root makes an arbitrary decision; if neither does, the root signals an ***overflow*** situation.
(ii) When an intermediate node receives the new record, it routes it to its subtree with unoccupied leaves (again, making an arbitrary choice, if necessary).
(iii) The new record eventually reaches an unoccupied leaf where it is stored.

Note that whenever the root, or an intermediate node, sends the new record to a subtree, the number of unoccupied leaves associated with that subtreee is decreased by

1. It should be clear that insertion is greatly facilitated by the fact that the file is not to be maintained in any particular order.

2. *Update* Say that every record whose s field equals x must be updated with new information in (some of) its other fields. This is accomplished by sending x and the new information to all leaves. Each leaf i for which $s_i = x$ implements the change.

3. Deletion If every record whose s field equals x must be deleted, then we begin by sending x to all leaves. Each leaf i for which $s_i = x$ now declares itself as unoccupied by sending a 1 to its parent. This information is carried upward until it reaches the root. On its way, it increments by 1 the appropriate count in each node of the number of unoccupied leaves in the left or right subtree.

Each of the preceding maintenance operations takes $O(\log n)$ time. As before, q operations can be pipelined to require $O(\log n) + O(q)$ time in total.

We conclude this section with the following observations.

1. We have obtained a search algorithm for a tree-connected computer that is more efficient than that described for a much stronger model, namely, the EREW SM SIMD. Is there a paradox here? Not really. What our result indicates is that we managed to find an algorithm that does not require the full power of the shared-memory model and yet is more efficient than an existing EREW algorithm. Since any algorithm for an interconnection network SIMD computer can be simulated on the shared-memory model, the tree algorithm for searching can be turned into an EREW algorithm with the same performance.

2. It may be objected that our comparison of the tree and shared-memory algorithms is unfair since we are using $2n - 1$ processors on the tree and only n on the EREW computer. This objection can be easily taken care of by using a tree with $n/2$ leaves and therefore a total of $n - 1$ processors. Each leaf now stores two records and performs two comparisons for every given x.

3. If a tree with N leaves is available, where $1 < N \leqslant n$, then n/N records are stored per leaf. A query now requires

(i) $O(\log N)$ time to send x to the leaves,

(ii) $O(n/N)$ time to search the records within each leaf for one with an s field equal to x, and

(iii) $O(\log N)$ time to send the answer back to the root,

that is, a total of $O(\log N) + O(n/N)$. This is identical to the time required by the algorithms that run on the more powerful EREW, ERCW, or CREW SM SIMD computers. Pipelining, however, is not as attractive as before: Searching within each leaf no longer requires constant time and q queries are not guaranteed to be answered in $O(\log n) + O(q)$ time.

4. Throughout the preceding discussion we have assumed that the wire delay, that is, the time it takes a datum to propagate along a wire, from one level of the tree to the next is a constant. Thus for a tree with n leaves, each query or maintenance operation under this assumption requires a running time of $O(\log n)$ to be processed. In addition, the time between two consecutive inputs or two consecutive outputs is constant: In other words, searching on the tree has a constant period (provided, of course, that the queries have constant-length answers). However, a direct hardware implementation of the tree-connected computer would obviously have connections between levels whose length grows exponentially with the level number. As Fig. 5.5 illustrates, the wire connecting a node at level i to its parent at level $i + 1$ has length proportional to 2^i. The maximum wire length for a tree with n leaves is $O(n)$ and occurs at level $\log n - 1$. Clearly, this approach is undesirable from a practical point of view, as it results in a very poor utilization of the area in which the processors and wires are placed. Furthermore, it would yield a running time of $O(n)$ per query if the propagation time is taken to be proportional to the wire length. In order to prevent this, we can embed the tree in a mesh, as shown in Fig. 5.7. Figure 5.7 illustrates an n-

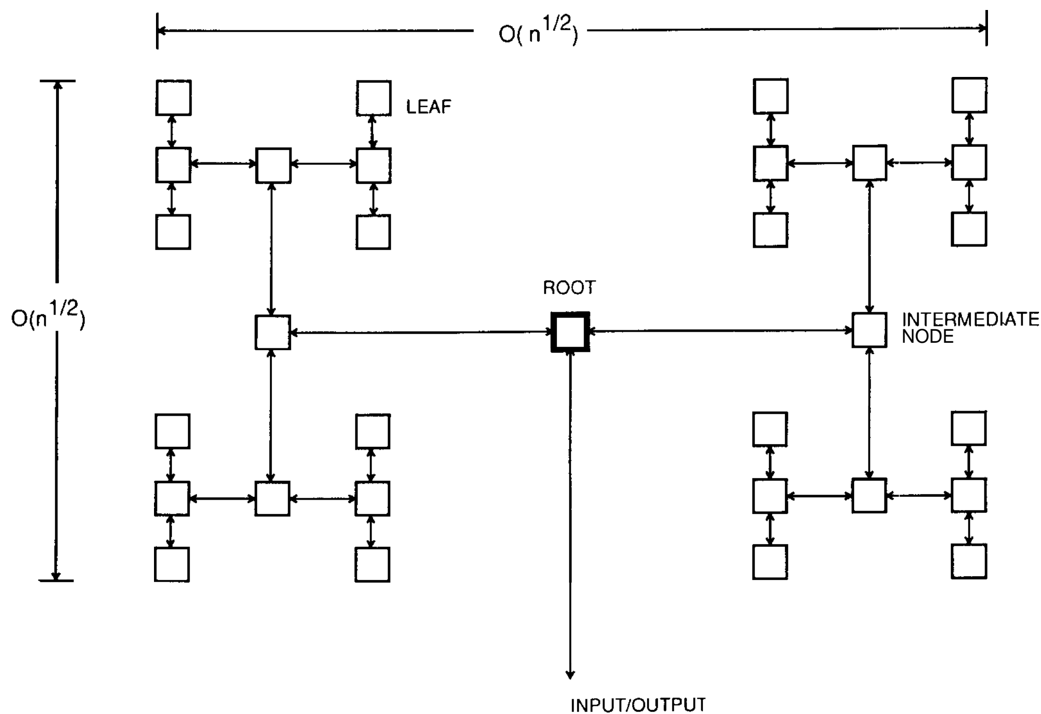


Figure 5.7 Tree-connected computer embedded in mesh.

node tree, with $n = 31$, where

(i) the maximum wire length is $O(n^{1/2})$,
(ii) the area used is $O(n)$, and
(iii) the running time per query or maintenance operation is $O(n^{1/2})$ and the period is $O(n^{1/2})$, assuming that the propagation time of a signal across a wire grows linearly with the length of the wire.

This is a definite improvement over the previous design, but not sufficiently so to make the tree the preferred architecture for search problems. In the next section we describe a parallel algorithm for searching on a mesh-connected SIMD computer whose behavior is superior to that of the tree algorithm under the linear propagation time assumption.

5.3.3 Searching on a Mesh

In this section we show how a two-dimensional array of processors can be used to solve the various searching problems described earlier. Consider the n-processor mesh-connected SIMD computer illustrated in Fig. 5.8 for $n = 16$, where each processor stores one record of the file to be searched. This architecture has the following characteristics:

1. The wire length is constant, that is, independent of the size of the array;
2. the area used is $O(n)$; and
3. the running time per query or maintenance operation is $O(n^{1/2})$ and the period is constant regardless of any assumption about wire delay.

Clearly, this behavior is a significant improvement over that of the tree architecture under the assumption that the propagation time of a signal along a wire is linearly proportional to the length of that wire. (Of course, if the wire delay is assumed to be a constant, then the tree is superior for the searching problem since $\log n < n^{1/2}$ for sufficiently large n.)

5.3.3.1 Querying. In order to justify the statement in 3 regarding the running time and period of query and maintenance operations on the mesh, we describe an algorithm for that architecture that solves the basic query problem; namely, given an integer x, it is required to search the file of records on the s field for x. We then show that the algorithm produces a yes or no answer to such a query in $O(n^{1/2})$ time and that q queries can be processed in $O(q) + O(n^{1/2})$ time. Let us denote by $s_{i,j}$ the s field of the record held by processor $P(i, j)$. The algorithm consists of two stages: unfolding and folding.

Unfolding. Processor $P(1, 1)$ reads x. If $x = s_{1,1}$, it produces an output $b_{1,1}$ equal to 1; otherwise $b_{1,1} = 0$. It then communicates $(b_{1,1}, x)$ to $P(1, 2)$. If $x = s_{1,2}$ or $b_{1,1} = 1$, then $b_{1,2} = 1$; otherwise $b_{1,2} = 0$. Now simultaneously, the two row

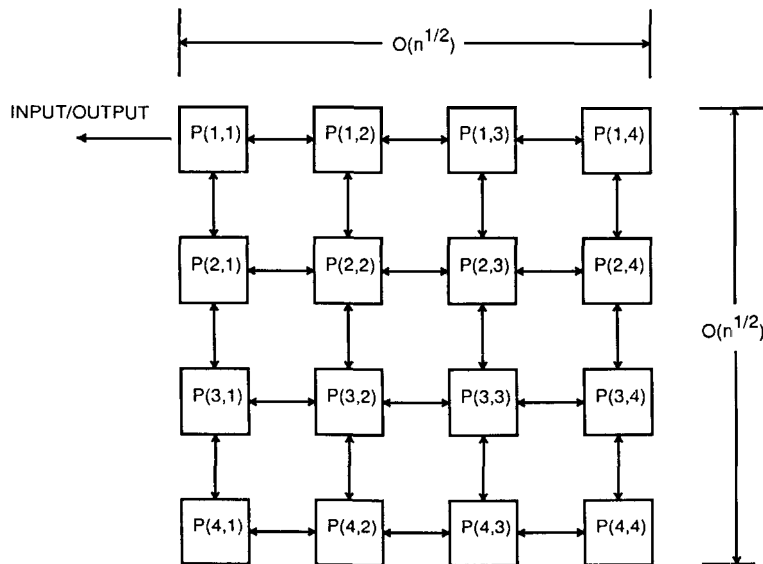


Figure 5.8 Mesh-connected computer for searching.

neighbors $P(1, 1)$ and $P(1, 2)$ send $(b_{1,1}, x)$ and $(b_{1,2}, x)$ to $P(2, 1)$ and $P(2, 2)$, respectively. Once $b_{2,1}$ and $b_{2,2}$ have been computed, the two column neighbors $P(1, 2)$ and $P(2, 2)$ communicate $(b_{1,2}, x)$ and $(b_{2,2}, x)$ to $P(1, 3)$ and $P(2, 3)$, respectively. This ***unfolding*** process, which alternates row and column propagation, continues until x reaches $P(n^{1/2}, n^{1/2})$.

Folding. At the end of the unfolding stage every processor has had a chance to "see" x and compare it to the s field of the record it holds. In this second stage, the reverse action takes place. The output bits are propagated from row to row and from column to column in an alternating fashion, right to left and bottom to top, until the answer emerges from $P(1, 1)$. The algorithm is given as procedure MESH SEARCH:

procedure MESH SEARCH (S, x, answer)

Step 1: {$P(1, 1)$ reads the input)
  if $x = s_{1,1}$ **then** b,,, $\leftarrow 1$
    else $b_{1,1} \leftarrow 0$
  end if.

Step 2: {Unfolding}
  for i = 1 **to** $n^{1/2} - 1$ **do**
    (2.1) **for** j = 1 **to** i **do in parallel**
      (i) $P(j, i)$ transmits $(b_{j,i}, x)$ to $P(j, i + 1)$
      (ii) **if** ($x = s_{j,i+}$, or $b_{j,i} = 1$) **then** $b_{j,i+}$, $\leftarrow 1$
        else $b_{j,i+}$, $\leftarrow 0$
      end if
    end for

(2.2) for j = 1 to i + 1 do in **parallel**
  (i) $P(i, j)$ transmits $(b_{i,j}, x)$ to $P(i+1, j)$
  (ii) **if** $(x = s_{i+1,j}$ or $b_{i,j} = 1)$ then $b_{i+1,j} \leftarrow 1$
    else $b_{i+1,j} \leftarrow 0$
   end if
  end for
end for.

Step 3: {Folding}
for i = $n^{1/2}$ **downto** 2 do
  (3.1) for j = 1 to i do in parallel
    $P(j, i)$ transmits $b_{j,i}$ to $P(j, i-1)$
   end for
  (3.2) for j = 1 to i − 1 do in parallel
    $b_{j,i-1} \leftarrow b_{j,i}$
   end for
  (3.3) if $(b_{i,i-1} = 1$ or $b_{i,i} = 1)$ **then** $b_{i,i-1} \leftarrow 1$
    else $b_{i,i-1} \leftarrow 0$
   end if
  (3.4) for j = 1 to i − 1 do in parallel
    $P(i, j)$ transmits $b_{i,j}$ to $P(i-1, j)$
   end for
  (3.5) for j = 1 to i − 2 do in parallel
    $b_{i-1,j} \leftarrow b_{i,j}$
   end for
  (3.6) **if** $(b_{i-1,i-1} = 1$ or $b_{i,i-1} = 1)$ **then** $b_{i-1,i-1} \leftarrow 1$
    else $b_{i-1,i-1} \leftarrow 0$
   end if
end for.

Step 4: {$P(1,1)$ produces the output}
if $b_{1,1} = 1$ then answer ← yes
  else answer ← no
end if. □

Analysis

As each of steps 1 and 4 takes constant time and steps 2 and 3 consist of $n^{1/2} - 1$ constant-time iterations, the time to process a query is $O(n^{1/2})$. Notice that after the first iteration of step 2, processor $P(1, 1)$ is free to receive a new query. The same remark applies to other processors in subsequent iterations. Thus queries can be processed in pipeline fashion. Inputs are submitted to $P(1, 1)$ at a constant rate. Since the answer to a basic query is of fixed length, outputs are also produced by $P(1, 1)$ at a constant rate following the answer to the first query. Hence the period is constant.

Example 5.3

Let a set of 16 records stored in a 4 x 4 mesh-connected SIMD computer be as shown in Fig. 5.9. Each square in Fig. 5.9(a) represents a processor and the number inside it is the s

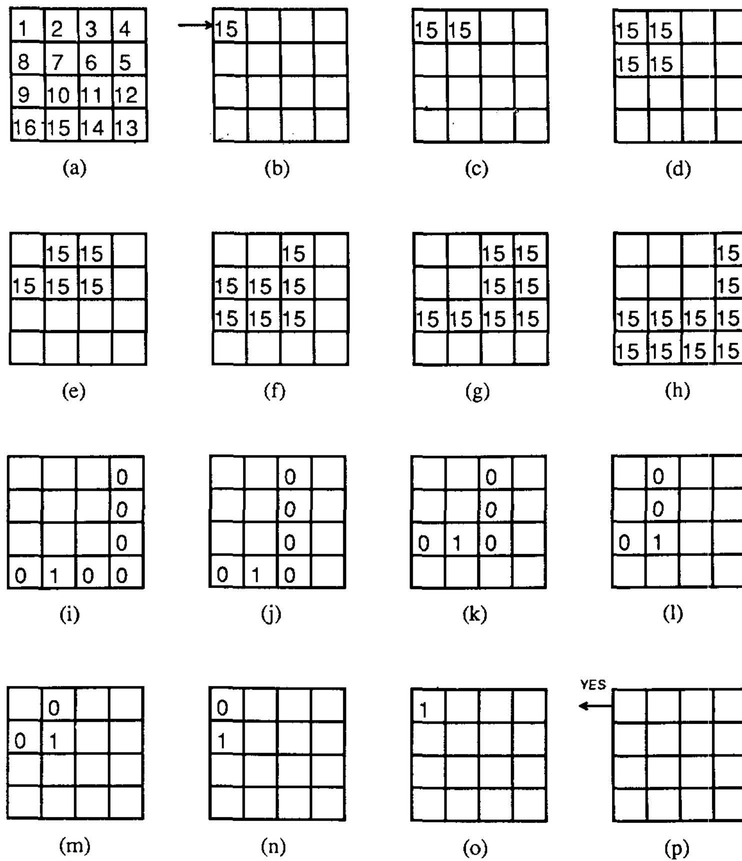


Figure 5.9 Searching sequence of sixteen elements using procedure MESH SEARCH.

field of the associated record. Wires connecting the processors are omitted for simplicity. It is required to determine whether there exists a record with s field equal to 15 (i.e., $x = 15$). Figures 5.9(b)–5.9(h) illustrate the propagation of 15 in the array. Figure 5.9(i) shows the relevant b values at the end of step 2. Figures 5.9(j)–5.9(o) illustrate the folding process. Finally Fig. 5.9(p) shows the result as produced in step 4. Note that in Fig. 5.9(e) processor $P(1, 1)$ is shown empty indicating that it has done its job propagating 15 and is now ready to receive a new query. □

Some final comments are in order regarding procedure MESH SEARCH.

1. No justification was given for transmitting $b_{i,j}$ along with x during the unfolding stage. Indeed, if only one query is to be answered, no processor needs to communicate its b value to a neighbor: All processors can compute and retain their outputs; these can then be combined during the folding stage. However, if

several queries are to be processed in pipeline fashion, then each processor must first transmit its current b value before computing the next one. In this way the $b_{i,j}$ are continually moving, and no processor needs to store its b value.

2. When several queries are being processed in pipeline fashion, the folding stage of one query inevitably encounters the unfolding stage of another. As we did for the tree, we assume that a processor simultaneously receiving data from opposite directions can process them in a single time unit or that every processor alternately switches its attention from one direction to the other.
3. It should be clear that all variations over the basic query problem described in section 5.3.2.1 can be easily handled by minor modifications to procedure MESH SEARCH.

5.3.3.2 Maintenance. All three maintenance operations can be easily implemented on the mesh.

1. ***Insertion*** Each processor in the top row of the mesh keeps track of the number of unoccupied processors in its column. When a new record is to be inserted, it is propagated along the top row until a column is found with an unoccupied processor. The record is then propagated down the column and inserted in the first unoccupied processor it encounters. The number of unoccupied processors in that column is reduced by 1.

2. ***Updating*** All records to be updated are first located using procedure MESH SEARCH and then the change is implemented.

3. ***Deletion*** When a record is to be deleted, it is first located, an indicator is placed in the processor holding it signifying it is unoccupied, and the count at the processor in the top row of the column is incremented by 1.

5.4 PROBLEMS

5.1 Show that $\Omega(\log n)$ is a lower bound on the number of steps required to search a sorted sequence of n elements on an **EREW** SM SIMD computer with n processors.

5.2 Consider the following variant of the **EREW** SM SIMD model. In one step, a processor can perform an arbitrary number of computations locally or transfer an arbitrary number of data (to or from the shared memory). Regardless of the amount of processing (computations or data transfers) done, one step is assumed to take a constant number of time units. Note, however, that a processor is allowed to gain access to a unique memory location during each step (as customary for the **EREW** model). Let n processors be available on this model to search a sorted sequence $S = \{s_1, s_2, \ldots, s_n\}$ of length n for a given value **x.** Suppose that any subsequence of S can be encoded to fit in one memory location. Show that under these conditions the search can be performed in $O(\log^{1/2} n)$ time. [Hint: Imagine that the data structure used to store the sequence in shared memory is a binary tree, as shown in Fig. 5.10(a) for $n = 31$. This tree can be encoded as shown in Fig. 5.10(b).]

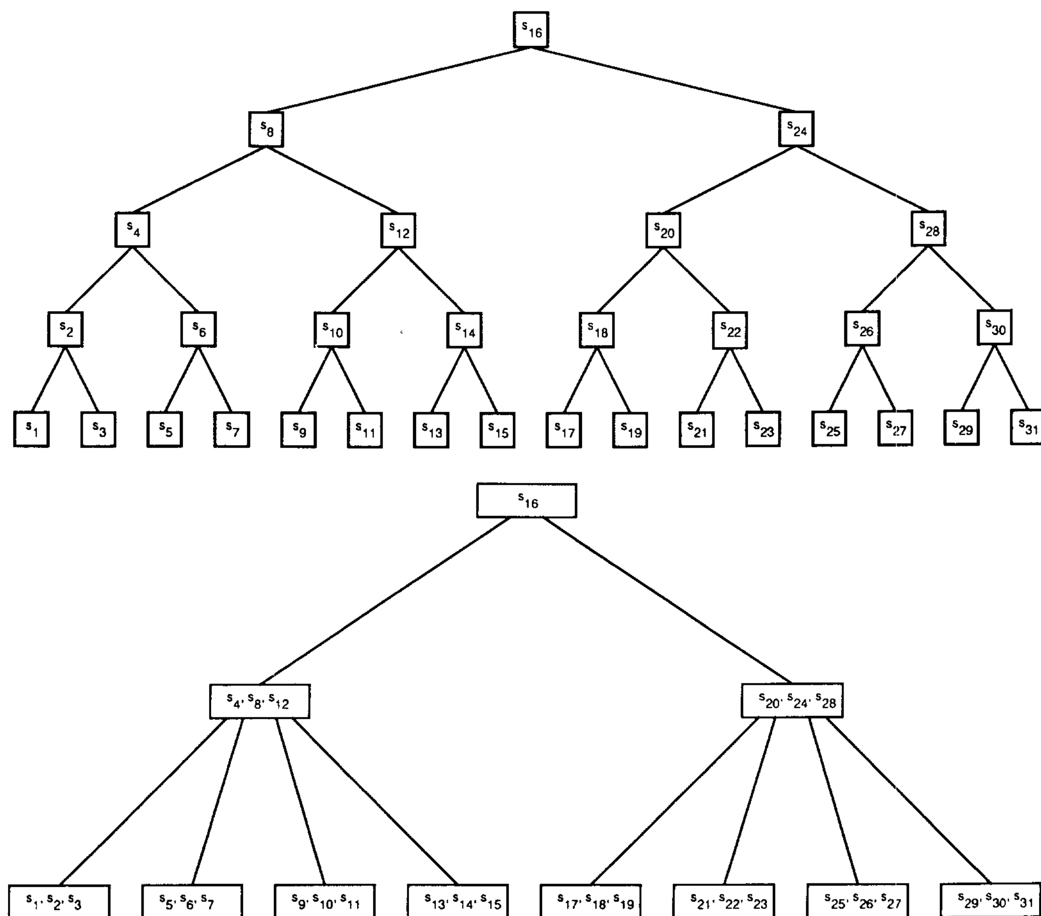


Figure 5.10 Data structures for problem 5.2.

5.3 Prove that $\Omega(\log^{1/2} n)$ is a lower bound on the number of steps required to search a sorted sequence of n elements using n processors on the EREW SM SIMD computer of problem 5.2.

5.4 Let us reconsider problem 5.2 but without the assumption that arbitrary subsequences of S can be encoded to fit in one memory location and communicated in one step. Instead, we shall store the sequence in a tree with d levels such that a node at level i contains d − i elements of S and has d − i + 1 children, as shown in Fig. 5.11 for $n = 23$. Each node of this tree is assigned to a processor that has sufficient local memory to store the elements of S contained in that node. However, a processor can read only one element of S at every step. The key **x** to be searched for is initially available to the processor in charge of the root. An additional array in memory, with as many locations as there are processors, allows processor P_i to communicate x to P_j by depositing it in the location associated with P_j. Show that $O(n)$ processors can search a sequence of length n in $O(\log n/\log \log n)$.

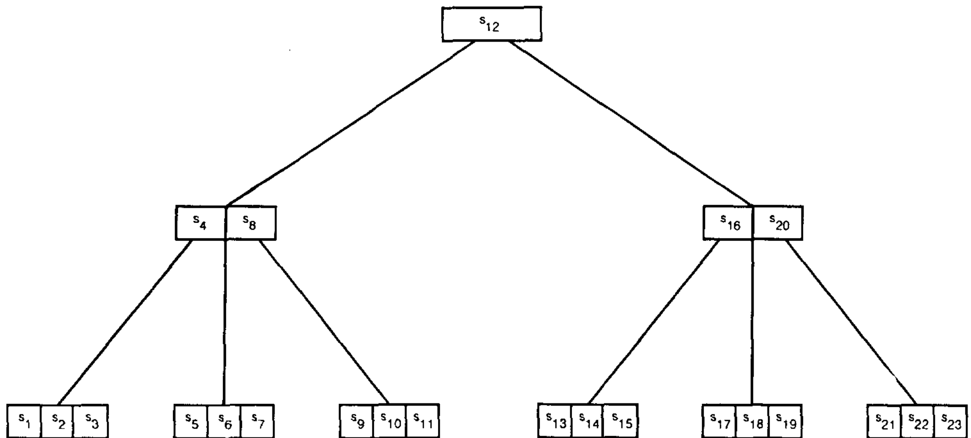


Figure 5.11 **Data** structure for problem 5.4.

5.5 Let $M(N, r, s)$ be the number of comparisons required by an N-processor CREW SM SIMD computer to merge two sorted sequences of length r and s, respectively. Prove that $M(N, 1, s) = \lceil \log(s + 1)/\log(N + 1) \rceil$.

5.6 Let $1 \leqslant r \leqslant N$ and $r \leqslant s$. Prove that

$$M(N, r, s) \leqslant \lceil \log(s + 1)/\log(\lfloor N/r \rfloor + 1) \rceil.$$

5.7 Let $1 \leqslant N \leqslant r \leqslant s$. Prove that

$$M(n, r, s) \leqslant \lceil r/N \rceil \lceil \log(s + 1) \rceil.$$

5.8 Consider an interconnection-network SIMD computer with n processors where each processor has a fixed-size local memory and is connected to each of the other $n - 1$ processors by a two-way link. At any given step a processor can perform any amount of computations locally but can communicate at most one input to at most one other processor. A sequence S is stored in this computer one element per processor. It is required to search S for an element x initially known to one of the processors. Show that $\Omega(\log n)$ steps are required to perform the search.

5.9 Assume that the size of the local memory of the processors in the network of problem 5.8 is no longer fixed. Show that if each processor can send or receive one element of S or x at a time, then searching S for some x can be done in $O(\log n/\log \log n)$ time.

5.10 Reconsider the model in problem 5.8 but without any restriction on the kind of information that can be communicated in one step from one processor to another. Show that in this case the search can be performed in $O(\log^{1/2} n)$ time.

5.11 Let the model of computation described in problem 2.9, that is, a linear array of N processors with a bus, be available. Each processor has a copy of a sorted sequence S of n distinct elements. Describe an algorithm for searching S for a given value x on this model and compare its running time to that of procedure CREW SEARCH.

5.12 An algorithm is described in example 1.4 for searching a file with n entries on a CRCW SM SIMD computer. The n entries are not necessarily distinct or sorted in any order. The

algorithm uses a location F in shared memory to determine whether early termination is possible. Give a formal description of this algorithm.

5.13 Give a formal description of the tree algorithm for searching described in section 5.3.2.1.

5.14 Given a sequence S and a value x, describe tree algorithms for solving; the following extensions to the basic query:

(a) Find the predecessor of x in S, that is, the largest element of S smaller than x.

(b) Find the successor of x in S, that is, the smallest element of S larger than x.

5.15 A file of n records is stored in the leaves of a tree machine one record per leaf. Each record consists of several fields. Given $((i, x_i), (j, x_j), \ldots, (m, x_m))$, it is required to find the records with the ith field equal to x_i, the jth field equal to x_j, and so on. Describe an algorithm for solving this version of the search problem.

5.16 Consider a tree-connected SIMD computer where each node contains a record (not just the leaves). Describe algorithms for querying and maintaining such a file of records.

5.17 Repeat problem 5.14 for a mesh-connected SIMD computer.

5.18 Consider the following modification to procedure MESH SEARCH. As usual, $P(1, 1)$ receives the input. During the unfolding stage processor $P(i, j)$ can send data simultaneously to $P(i + 1, j)$ and $P(i, j + 1)$. When the input reaches $P(n^{1/2}, n^{1/2})$, this processor can compute the final answer and produce it as output (i.e., there is no folding stage). Describe the modified procedure formally and analyze its running time.

5.19 Repeat problem 5.11 for the case where the number of processors is n and each processor stores one element of a sequence S of n distinct elements.

5.20 A binary sequence of length n consisting of a string of 0's followed by a string of 1's is given. It is required to find the length of the string of 0's using an EREW SM SIMD computer with N processors, $1 < N \leqslant n$. Show that this can be done in $O(\log(n/N))$ time.

5.21 In a storage and retrieval technique known as hashing, the location of a dlata element in memory is determined by its value. Thus, for every element x, the address of x is $f(x)$, where f is an appropriately chosen function. This approach is used when the data space (set of potential values to be stored) is larger than the storage space (memory locations) but not all data need be stored at once. Inevitably, collisions occur, that is, $f(x) = f(y)$ for $x \neq y$, and several strategies exist for resolving them. Describe a parallel algorithm for the hashing function, collision resolution strategy, and model of computation of your choice.

5.22 The algorithms in this chapter addressed the discrete search problem, that is, searching for a value in a given sequence. Similar algorithms can be derived for the *continuous* case, that is, searching for points at which a continuous function takes a given value. Describe parallel algorithms for locating (within a given tolerance) the point at which a certain function (i) assumes its largest value and (ii) is equal to zero.

5.23 It was shown in section 5.2.2 that procedure CREW SEARCH achieves the best possible running time for searching. In view of the lower bound in problem 5.1, show that no procedure faster than MULTIPLE BROADCAST of section 3.4 exists for simulating a CREW algorithm on an EREW computer.

5.5 BIBLIOGRAPHICAL REMARKS

The problem of searching a sorted sequence in parallel has attracted a good deal of attention since searching is an often-performed and time-consuming operation in most database, information retrieval, and office automation applications. Algorithms similar to procedure

7

Matrix Operations

7.1 INTRODUCTION

Problems involving matrices arise in a multitude of numerical and nonnumerical contexts. Examples range from the solution of systems of equations (see chapter 8) to the representation of graphs (see chapter 10). In this chapter we show how three operations on matrices can be performed in parallel. These operations are matrix transposition (section 7.2), matrix-by-matrix multiplication (section 7.3), and matrix-by-vector multiplication (section 7.4). Other operations are described in chapters 8 and 10. One particular feature of this chapter is that it illustrates the use of all the interconnection networks described in chapter 1, namely, the one-dimensional array, the mesh, the tree, the perfect shuffle, and the cube.

7.2 TRANSPOSITION

An $n \times n$ matrix A is given, for example:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix};$$

it is required to compute the *transpose* of A:

$$A^T = \begin{bmatrix} a_{11} & a_{21} & a_{31} & a_{41} \\ a_{12} & a_{22} & a_{32} & a_{42} \\ a_{13} & a_{23} & a_{33} & a_{43} \\ a_{14} & a_{24} & a_{34} & a_{44} \end{bmatrix}.$$

In other words, every ***row*** in matrix A is now a *column* in matrix A^T. The elements of A are any data objects; thus a_{ij} could be an integer, a real, a character, and so on.

Sequentially the transpose of a matrix can be computed very easily as shown in procedure TRANSPOSE. The procedure transposes A *in place,* that is, it returns A^T in the same memory locations previously occupied by A.

```
procedure TRANSPOSE (A)

  for i = 2 to n do
    for j = 1 to i − 1 do
      a_ij ↔ a_ji
    end for
  end for. □
```

This procedure runs in $O(n^2)$ time, which is optimal in view of the $R(n^2)$ steps required to simply read A.

In this section we show how the transpose can be computed in parallel on three different models of parallel computation, namely, the mesh-connected, shuffle-connected, and the shared-memory SIMD computers.

7.2.1 Mesh Transpose

The parallel architecture that lends itself most naturally to matrix operations is the mesh. Indeed, an n x n mesh of processors can be regarded as a matrix and is therefore perfectly fitted to accommodate an n x n data matrix, one element per processor. This is precisely the approach we shall use to compute the transpose of an n x n matrix A initially stored in an n x n mesh of processors, as shown in Fig. 7.1 for $n = 4$. Initially, processor $P(i,j)$ holds data element a_{ij}; at the end of the computation $P(i,j)$ should hold a_{ji}. Note that with this arrangement $\Omega(n)$ is a lower bound on the running time of any matrix transposition algorithm. This is seen by observing that a,, cannot reach $P(n, 1)$ in fewer than $2n - 2$ steps.

The idea of our algorithm is quite simple. Since the diagonal elements are not affected during the transposition, that is, element a,, of A equals element a,, of A^T, the data in the diagonal processors will stay stationary. Those below the diagonal are sent to occupy symmetrical positions above the diagonal (solid arrows in Fig. 7.1). Simultaneously, the elements above the diagonal are sent to occupy symmetrical positions below the diagonal (dashed arrows in Fig. 7.1). Each processor $P(i,j)$ has three registers:

1. $A(i,j)$ is used to store a_{ij} initially and a_{ji} when the algorithm terminates;
2. $B(i,j)$ is used to store data received from $P(i,j+1)$ or $P(i-1,j)$, that is, from its right or top neighbors; and
3. $C(i,j)$ is used to store data received from $P(i,j-1)$ or $P(i+1,j)$, that is, from its left or bottom neighbors.

The algorithm is given as procedure MESH TRANSPOSE. Note that the contents of registers $A(i,i)$, initially equal to a_{ii}, $1 \leqslant i \leqslant n$, are not affected by the procedure.

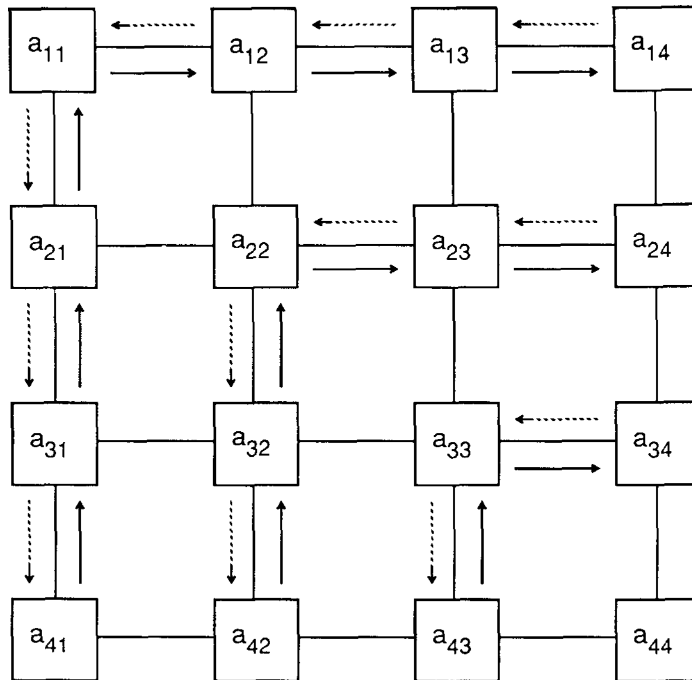


Figure 7.1 Matrix to be transposed, stored in mesh of processors.

procedure MESH TRANSPOSE (A)

Step 1: do steps 1.1 and *1.2* **in** parallel
(1.1) for $i = 2$ to n do in parallel
for $j = 1$ to $i - 1$ do in parallel
$C(i - 1, j) \leftarrow (a_{ij}, j, i)$
end for
end for
(1.2) for $i = 1$ to $n - 1$ do in parallel
for $j = i + 1$ to n do in parallel
$B(i, j - 1) \leftarrow (a_{ij}, j, i)$
end for
end for.

Step 2: do steps *2.1, 2.2,* and *2.3* in parallel
(2.1) for $i = 2$ to n do in parallel
for $j = 1$ to $i - 1$ do in parallel
while $P(i, j)$ receives input from its neighbors do
(i) if (a,,, m, k) is received from $P(i + 1, j)$
then send it to $P(i - 1, j)$
end if

```
                (ii) if (a,,,  m, k) is received from P(i − 1, j)
                     then if i = m and j = k
                          then A(i, j) ← a,  {a,  has reached its destination}
                          else send (a,,,  m, k) to P(i + 1, j)
                          end if
                     end if
                end while
              end for
           end for
(2.2) for i = 1 to n do in parallel
         while P(i, i) receives input from its neighbors do
          (i) if (a,,,  m, k) is received from P(i + 1, i)
              then send it to P(i, i + 1)
              end if
         (ii) if (a,,,  m, k) is received from P(i, i + 1)
              then send it to P(i + 1, i)
              end if
         end while
      end for
(2.3) for i = 1 to n − 1 do in parallel
         for j = i + 1 to n do in parallel
            while P(i, j) receives input from its neighbors do
               (i) if (a,,,  m, k) is received from P(i, j + 1)
                   then send it to P(i, j − 1)
                   end if
              (ii) if (a,,,  m, k) is received from P(i, j − 1)
                   then if i = m and j = k
                        then A(i, j) ← a,  {a,  has reached its destination}
                        else send (a,,,  m, k) to P(i, j + 1)
                        end if
                   end if
            end while
         end for
      end for. □
```

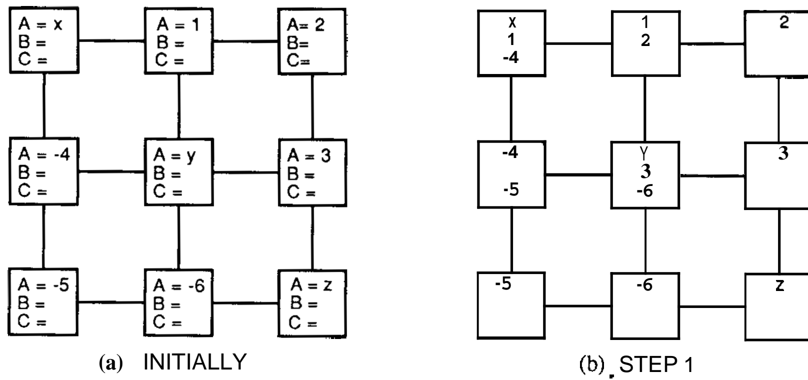
Analysis. Each element a_{ij}, $i > j$, must travel up its column until it reaches $P(j, j)$ and then travel along a row until it settles in $P(j, i)$. Similarly for a_{ij}, $j > i$. The longest path is the one traversed by **a,,** (or **a,,**), which consists of $2(n - 1)$ steps. The running time of procedure MESH TRANSPOSE is therefore

$$t(n) = O(n),$$

which is the best possible for the mesh. Since $p(n) = n^2$, the procedure has a cost of $O(n^3)$, which is not optimal.

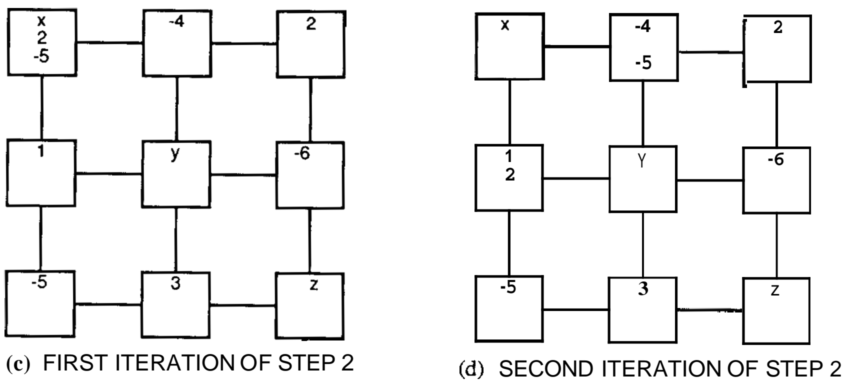
Example 7.1

The behavior of procedure MESH TRANSPOSE is illustrated in Fig. 7.2 for the input matrix



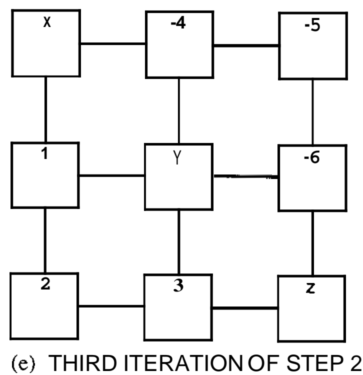
(a) INITIALLY

(b) STEP 1



(c) FIRST ITERATION OF STEP 2

(d) SECOND ITERATION OF STEP 2



(e) THIRD ITERATION OF STEP 2

Figure 7.2 Transposing matrix using procedure MESH TRANSPOSE.

$$A = \begin{bmatrix} x & 1 & 2 \\ -4 & y & 3 \\ -5 & -6 & z \end{bmatrix}.$$

The contents of registers A, B, and C in each processor are shown. Note that for clarity only the a_{ij} component of (a_{ij}, j, i) is shown for registers B and C. Also when either B or C receives no new input, it is shown empty. □

7.2.2 Shuffle Transpose

We saw in the previous section that procedure MESH TRANSPOSE computes the transpose of an $n \times n$ matrix in $O(n)$ time. We also noted that this running time is the fastest that can be obtained on a mesh with one data element per processor. However, since the transpose can be computed sequentially in $O(n^2)$ time, the speedup achieved by procedure MESH TRANSPOSE is only linear. This speedup may be considered rather small since the procedure uses a quadratic number of processors. This section shows how the same number of processors arranged in a different geometry can transpose a matrix in logarithmic time.

Let $n = 2^q$ and assume that an $n \times n$ matrix A is to be transposed. We use for that purpose a perfect shuffle interconnection with n^2 processors $P_0, P_1, \ldots, P_{2^{2q}-1}$. Element a_{ij} of A is initially stored in processor P_k, where $k = 2^q(i - 1) + (j - 1)$, as shown in Fig. 7.3 for $q = 2$.

We claim that after exactly q shuffle operations processor P_k contains element a_{ji}. To see this, recall that if P_k is connected to P_m, then m is obtained from k by cyclically shifting to the left by one position the binary representation of k. Thus P_{0000} is connected to itself, P_{0001} to P_{0010}, P_{0010} to P_{0100}, ..., P_{1001} to P_{0011}, P_{1010} to

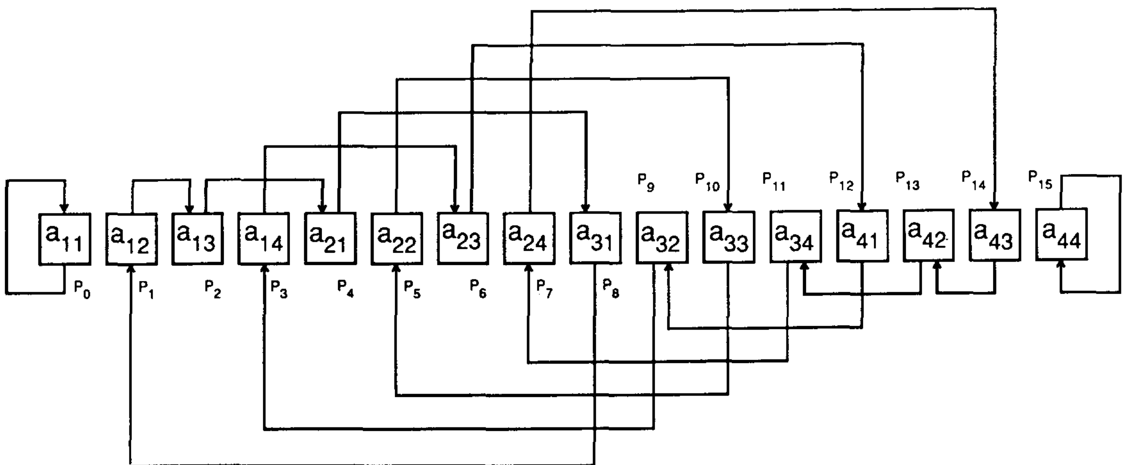


Figure 7 3 Matrix to be transposed, stored in perfect shuffle-connected computer.

$P_{0101}, \ldots$, and P_{1111} to itself. Now consider a processor index k consisting of $2q$ bits. If $k = 2^q(i-1) + (j-1)$, then the q most significant bits of k represent $i-1$ while the q least significant bits represent $j-1$. This is illustrated in Fig. 7.4(a) for $q = 5$, $i = 5$, and $j = 12$. After q shuffles (i.e., q cyclic shifts to the left), the element originally held by P_k will be in the processor whose index is

$$s = 2^q(j-1) + (i-1),$$

as shown in Fig. 7.4(b). In other words a_{ij} has been moved to the position originally occupied by a_{ji}. The algorithm is given as procedure SHUFFLE TRANSPOSE. In it we use the notation $2k \bmod (2^{2q}-1)$ to represent the remainder of the division of $2k$ by $2^{2q}-1$.

procedure SHUFFLE TRANSPOSE (A)

for i = 1 **to** q **do**
  for k = 1 **to** $2^{2q} - 2$ **do in parallel**
    P_k sends the element of A it currently holds to $P_{2k \bmod (2^{2q}-1)}$
  end for
end for. □

Analysis. There are q constant time iterations and therefore the procedure runs in $t(n) = O(\log n)$ time. Since $p(n) = n^2$, $c(n) = O(n^2 \log n)$, which is not optimal. Interestingly, the shuffle interconnection is faster than the mesh in computing the transpose of a matrix. This is contrary to our original intuition, which suggested that the mesh is the most naturally suited geometry for matrix operations.

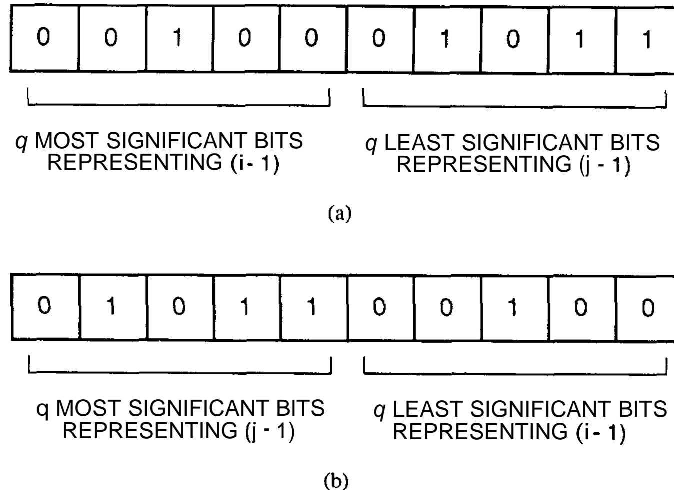


Figure 7.4 Derivation of number of shuffles required to transpose matrix.

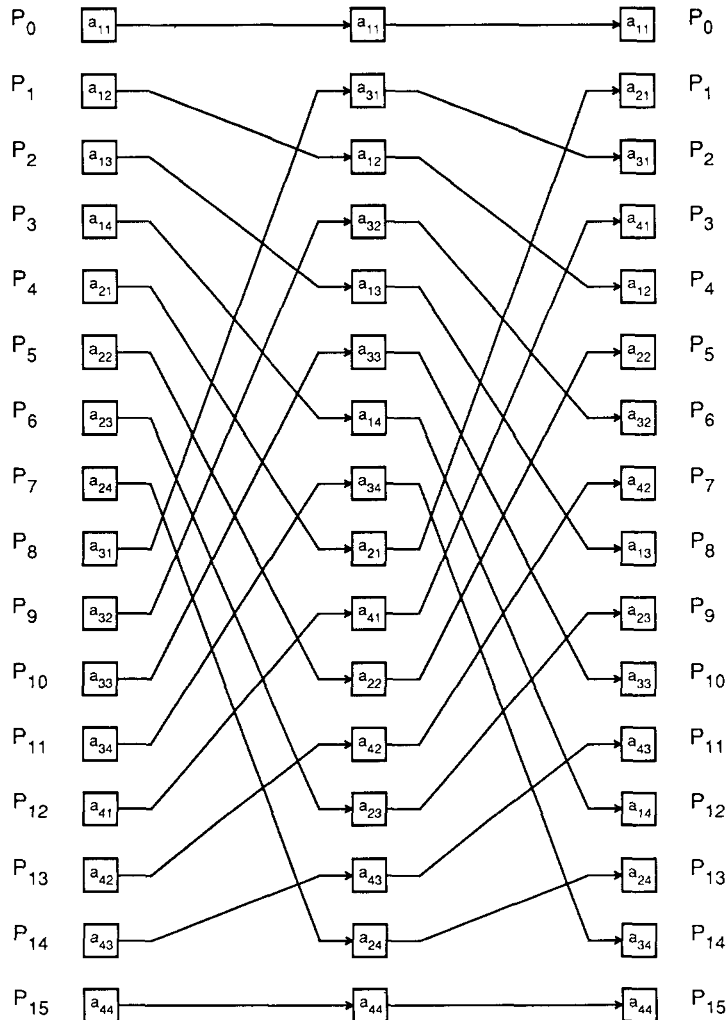


Figure 7.5 Transposing matrix using procedure SHUFFLE TRANSPOSE.

Example 7.2

The behavior of procedure SHUFFLE TRANSPOSE is illustrated in Fig. 7.5 for the case where $q = 2$. For clarity, the shuffle interconnections are shown as a mapping from the set of processors to itself. □

7.2.3 EREW Transpose

Although faster than procedure MESH TRANSPOSE, procedure SHUFFLE TRANSPOSE is not cost optimal. We conclude this section by describing a cost-

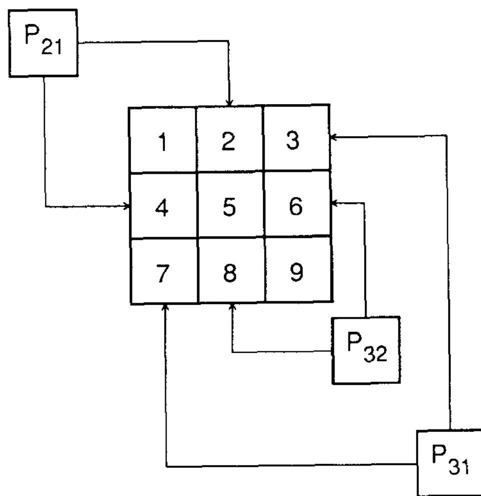


Figure 7.6 Transposing matrix using procedure EREW TRANSPOSE.

optimal algorithm for transposing an $n \times n$ matrix A. The algorithm uses $(n^2 - n)/2$ processors and runs on an EREW SM SIMD computer. Matrix A resides in the shared memory. For ease of exposition, we assume that each processor has two indices i and j, where $2 \leqslant i \leqslant n$ and $1 \leqslant j \leqslant i - 1$. With all processors operating in parallel, processor P_{ij} swaps two elements of A, namely, a_{ij} and a_{ji}. The algorithm is given as procedure EREW TRANSPOSE.

procedure EREW TRANSPOSE (A)

for i = 2 **to** n **do in parallel**
  for j = 1 **to** i − 1 **do in parallel**
    $a_{ij} \leftrightarrow a_{ji}$
  end for
end for. □

Analysis. It takes constant time for each processor to swap two elements. Thus the running time of procedure EREW TRANSPOSE is $t(n) = O(1)$. Since $p(n) = O(n^2)$, $c(n) = O(n^2)$, which is optimal.

Example 7.3

The behavior of procedure EREW TRANSPOSE is illustrated in Fig. 7.6 for n = 3. The figure shows the two elements swapped by each processor. □

7.3 MATRIX-BY-MATRIX MULTIPLICATION

In this section we assume that the elements of all matrices are numerals, say, integers. The product of an $m \times n$ matrix A by an $n \times k$ matrix B is an $m \times k$ matrix C whose

elements are given by

$$c_{ij} = \sum_{s=1}^{n} a_{is} \times b_{sj}, \qquad 1 \leqslant i \leqslant m, \quad 1 \leqslant j \leqslant k.$$

A straightforward sequential implementation of the preceding definition is given by procedure MATRIX MULTIPLICATION.

```
procedure MATRIX MULTIPLICATION (A, B, C)

  for i = 1 to m do
    for j = 1 to k do
      (1) c_ij ← 0
      (2) for s = 1 to n do
            c_ij ← c_ij + (a_is × b_sj)
          end for
    end for
  end for. □
```

Assuming that $m \leqslant n$ and $k \leqslant n$, it is clear that procedure MATRIX MULTIPLICATION runs in $O(n^3)$ time. As indicated in section 7.6, however, there exist several sequential matrix multiplication algorithms whose running time is $O(n^x)$, where $2 < x < 3$. It is not known at the time of this writing whether the fastest of these algorithms is optimal. Indeed, the only known lower bound on the number of steps required for matrix multiplication is the *trivial* one of $\Omega(n^2)$. This lower bound is obtained by observing that n^2 outputs are to be produced, and therefore any algorithm must require at least that many steps. In view of this gap between n^2 and n^x, $2 < x < 3$, we will find ourselves unable to exhibit cost-optimal parallel algorithms for matrix multiplication. Rather, we present algorithms whose cost is matched against the running time of procedure MATRIX MULTIPLICATION.

7.3.1 Mesh Multiplication

As with the problem of transposition, again we feel compelled to use a mesh-connected parallel computer to perform matrix multiplication. Our algorithm uses $m \times k$ processors arranged in a mesh configuration to multiply an $m \times n$ matrix A by an $n \times k$ matrix B. Mesh rows are numbered $1, \ldots, m$ and mesh columns $1, \ldots, k$. Matrices A and B are fed into the *boundary* processors in column 1 and row 1, respectively, as shown in Fig. 7.7 for $m = 4$, $n = 5$, and $k = 3$. Note that row i of matrix A lags one time unit behind row $i - 1$ for $2 \leqslant i \leqslant m$. Similarly, column j of matrix B lags one time unit behind column $j - 1$ for $2 \leqslant j \leqslant k$. This ensures that a_{is} meets b_{sj} in processor $P(i, j)$ at the right time. At the end of the algorithm, element c_{ij} of the product matrix C resides in processor $P(i, j)$. Initially c_{ij} is zero. Subsequently, when $P(i, j)$ receives two inputs a and b, it

(i) multiplies them,

(ii) adds the result to c_{ij},

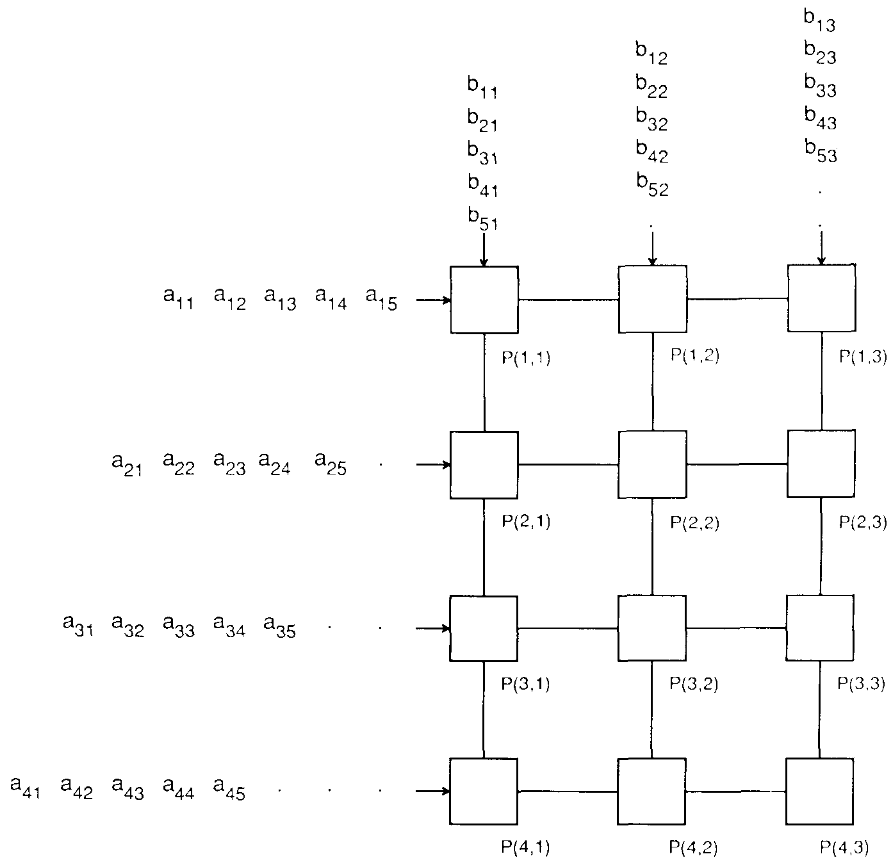


Figure 7.7 Two matrices to be multiplied, being fed as input to mesh of processors.

(iii) sends **a** to $P(i, j+1)$ unless $j = k$, and
(iv) sends b to $P(i+1, j)$ unless $i = m$.

The algorithm is given as procedure MESH MATRIX MULTIPLICATION.

procedure MESH MATRIX MULTIPLICATION (A, B, C)

```
for i = 1 to m do in parallel
  for j = 1 to k do in parallel
    (1) c_ij ← 0
    (2) while P(i, j) receives two inputs a and b do
          (i) c_ij ← c_ij + (a x b)
          (ii) if i < m then send b to P(i + 1, j)
               end if
```

(iii) **if** j < k **then** send **a** to $P(i, j+1)$
end if
end while
end for
end for. □

Analysis. Elements a_{m1} and b_{1k} take $m + k + n - 2$ steps from the beginning of the computation to reach $P(m, k)$. Since $P(m, k)$ is the last processor to terminate, this many steps are required to compute the product. Assuming that $m \leqslant n$ and $k \leqslant n$, procedure MESH MATRIX MULTIPLICATION therefore runs in time $t(n) = O(n)$. Since $p(n) = O(n^2)$, $c(n) = O(n^3)$, which matches the running time of the sequential procedure MATRIX MULTIPLICATION. It should be noted that the running time of procedure MESH MATRIX MULTIPLICATION is the fastest achievable for matrix multiplication on a mesh of processors assuming that only boundary processors are capable of handling input and output operations. Indeed, under this assumption $\Omega(n)$ steps are needed for the input to be read (by the processors in row 1 and column 1, say) and/or for the output to be produced (by the processors in row m and column k, say).

Example 7.4

The behavior of procedure MESH MATRIX MULTIPLICATION is illustrated in Fig. 7.8 for

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} -55 & -6 \\ -7 & -8 \end{bmatrix}.$$

The value of c_{ij} after each step is shown inside $P(i, j)$. □

7.3.2 Cube Multiplication

The running time of procedure MESH MATRIX MULTIPLICATION not only is the best achievable on the mesh, but also provides the highest speedup over the sequential procedure MATRIX MULTIPLICATION using n^2 processors. Nevertheless, we seek to obtain a faster algorithm, and as we did in section 7.2.2, we shall turn to another architecture for that purpose. Our chosen model is the cube-connected SIMD computer introduced in chapter 1 and that we now describe more formally.

Let $N = 2^g$ processors $P_0, P_1, \ldots, P_{2^g-1}$ be available for some $g \geqslant 1$. Further, let i and $i^{(b)}$ be two integers, $0 \leqslant i, i^{(b)} \leqslant 2^g - 1$, whose binary representations differ only in position b, $0 \leqslant b < g$. In other words, if $i_{g-1} \ldots i_{b+1}\, i_b\, i_{b-1} \ldots i_1\, i_0$ is the binary representation of i, then $i_{g-1} \ldots i_{b+1}\, i'_b\, i_{b-1} \ldots i_1\, i_0$ is the binary representation of $i^{(b)}$, where i'_b is the binary complement of bit i_b. The cube connection specifies that every processor P_i is connected to processor $P_{i^{(b)}}$ by a two-way link for all $0 \leqslant b < g$. The g processors to which P_i is connected are called P_i's ***neighbors.*** An example of such a connection is illustrated in Fig. 7.9 for the case g = 4. Now let $n = 2^q$. We use a cube-connected SIMD computer with $N = n^3 = 2^{3q}$ processors to multiply two n x n matrices A and B. (We assume for simplicity of presentation that the two matrices

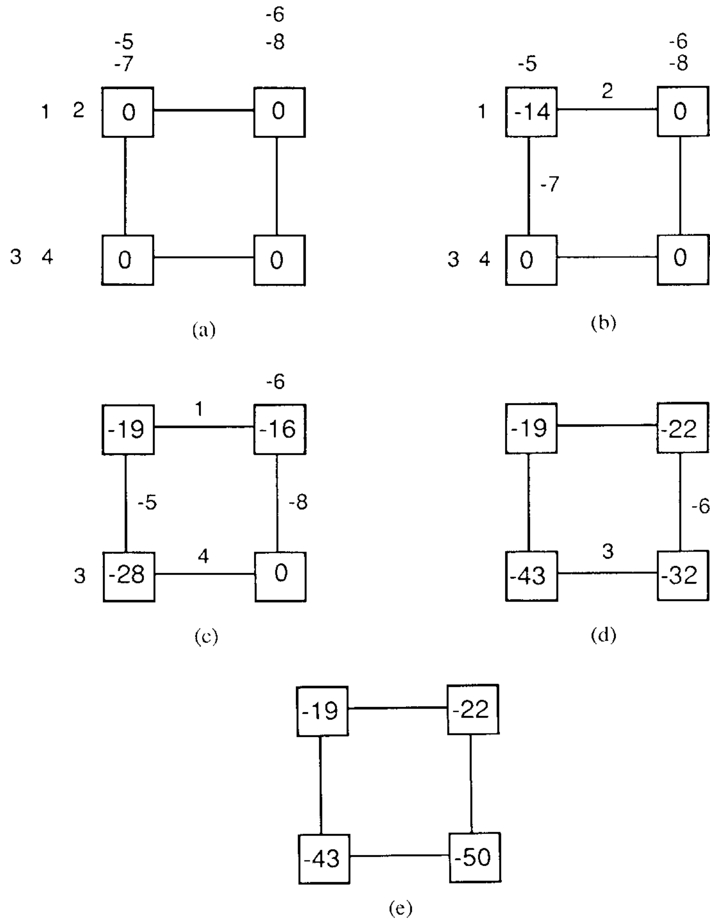


Figure 7.8 Multiplying two matrices using procedure MESH MATRIX MULTIPLICATION.

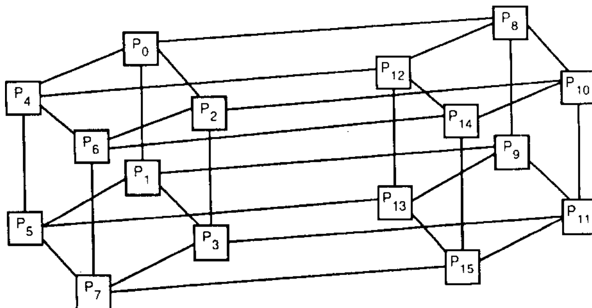


Figure 7.9 Cube-connected computer with sixteen processors.

have the same number of rows and columns.) It is helpful to visualize the processors as being arranged in an $n \times n \times n$ array pattern. In this array, processor P_r occupies position (i, j, k), where $r = in^2 + jn + k$ and $0 \leqslant i, j, k \leqslant n - 1$ (this is referred to as *row-major order*). Thus if the binary representation of r is $r_{3q-1} r_{3q-2} \ldots r_0$, then the binary representations of i, j, and k are $r_{3q-1} \ldots r_{2q}$, $r_{2q-1} \ldots r_q$, and $r_{q-1} \ldots r_0$, respectively. Each processor P_r has three registers A_r, B_r, and C_r, also denoted $A(i, j, k)$, $B(i, j, k)$, and $C(i, j, k)$, respectively. Initially, processor P_s in position $(0, j, k)$, $0 \leqslant j < n$, $0 \leqslant k < n$, contains a_{jk} and b_{jk} in its registers A_s and B_s, respectively. The registers of all other processors are initialized to zero. At the end of the computation, C should contain c_{jk}, where

$$c_{jk} = \sum_{i=0}^{n-1} a_{ji} \times b_{ik}.$$

The algorithm is designed to perform the n^3 multiplications involved in computing the n^2 entries of C simultaneously. It proceeds in three stages.

Stage 1: The elements of matrices A and B are distributed over the n^3 processors. As a result, $A(i, j, k) = a_{ji}$ and $B(i, j, k) = b_{ik}$.

Stage 2: The products $C(i, j, k) = A(i, j, k) \times B(i, j, k)$ are computed.

Stage 3: The sums $\Sigma_{i=0}^{n-1} C(i, j, k)$ are computed.

The algorithm is given as procedure CUBE MATRIX MULTIPLICATION. In it we denote by $\{N, r_m = d\}$ the set of integers r, $0 \leqslant r < N - 1$, whose binary representation is $r_{3q-1} \ldots r_{m+1}\, d\, r_{m-1} \ldots r_0$.

procedure CUBE MATRIX MULTIPLICATION (A, B, C)

Step 1: **for** $m = 3q - 1$ **downto** $2q$ **do**
  for all r **in** $\{N, r_m = 0\}$ **do in parallel**
    (1.1) $A_{r^{(m)}} = A_r$
    (1.2) $B_{r^{(m)}} = B_r$
  end for
end for.

Step 2: **for** $m = q - 1$ **downto** 0 **do**
  for all r **in** $\{N, r_m = r_{2q+m}\}$ **do in parallel**
    $A_{r^{(m)}} \leftarrow A_r$
  end for
end for.

Step 3: **for** $m = 2q - 1$ **downto** q **do**
  for all r **in** $\{N, r_m = r_{q+m}\}$ **do in parallel**
    $B_{r^{(m)}} \leftarrow B_r$
  end for
end for.

Step 4: **for** $r = 1$ **to** N **do in parallel**
  $C_r \leftarrow A_r \times B_r$
end for.

```
Step 5: for m = 2q to 3q − 1 do
          for r = 1 to N do in parallel
            C_r ← C_r + C_r(m)
          end for
        end for. □
```

Stage 1 of the algorithm is implemented by steps 1–3. During step 1, the data initially in $A(0,j,k)$ and $B(0,j,k)$ are copied into the processors in positions (i,j,k), where $1 \leqslant i < n$, so that at the end of this step $A(i,j,k) = a_{jk}$ and $B(i,j,k) = b_{jk}$ for $0 \leqslant i < n$. Step 2 copies the contents of $A(i,j,i)$ into the processors in position (i,j,k), so that at the end of this step $A(i,j,k) = a_{ji}$, $0 \leqslant k < n$. Similarly, step 3 copies the contents of $B(i,i,k)$ into the processors in position (i,j,k), so that at the end of this step $B(i,j,k) = b_{ik}$, $0 \leqslant j < n$. In step 4 the product $C(i,j,k) = A(i,j,k) \times B(i,j,k)$ is computed by the processors in position (i,j,k) for all $0 \leqslant i,j,k < n$ simultaneously. Finally, in step 5, the n^2 sums

$$C(0, j, k) = \sum_{i=0}^{n-1} C(i, j, k)$$

are computed simultaneously.

Analysis. Steps 1, 2, 3, and 5 consist of q constant time iterations, while step 4 takes constant time. Thus procedure CUBE MATRIX MULTIPLICATION runs in $O(q)$ time, that is, $t(n) = O(\log n)$. We now show that this running time is the fastest achievable by any parallel algorithm for multiplying two $n \times n$ matrices on the cube. First note that each c_{ij} is the sum of n elements. It takes $\Omega(\log n)$ steps to compute this sum on any interconnection network with n (or more) processors. To see this, let s be the smallest number of steps required by a network to compute the sum of n numbers. During the final step, at most one processor is needed to perform the last addition and produce the result. During step $s - 1$ at most two processors are needed, during step $s - 2$ at most four processors, and so on. Thus after s steps, the maximum number of useful additions that can be performed is

$$\sum_{i=0}^{s-1} 2^i = 2^s - 1.$$

Given that exactly $n - 1$ additions are needed to compute the sum of n numbers, we have $n - 1 \leqslant 2^s - 1$, that is, $s \geqslant \log n$.

Since $p(n) = n^3$, procedure CUBE MATRIX MULTIPLICATION has a cost of $c(n) = O(n^3 \log n)$, which is higher than the running time of sequential procedure MATRIX MULTIPLICATION. Thus, although matrix multiplication on the cube is faster than on the mesh, its cost is higher due to the large number of processors it uses.

Example 7.5

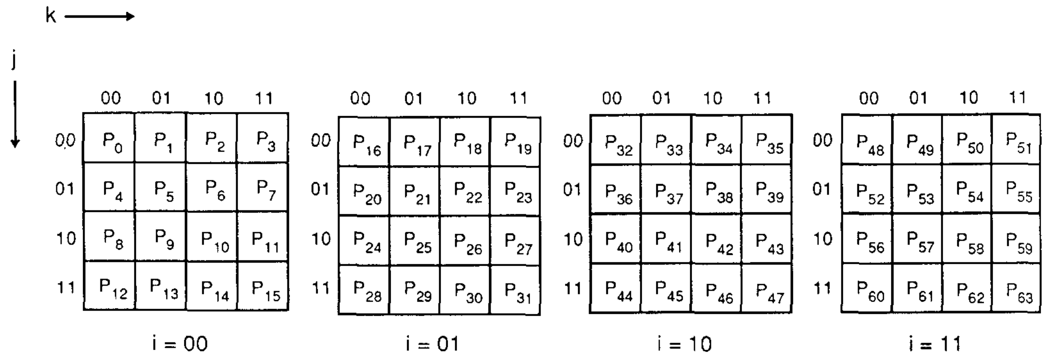
Let n = 2^2 and assume that the two 4 x 4 matrices to be multiplied are

$$A = \begin{bmatrix} 17 & 23 & 27 & 3 \\ 9 & 1 & 14 & 16 \\ 31 & 26 & 22 & 8 \\ 15 & 4 & 10 & 29 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} -7 & -25 & -19 & -5 \\ -18 & -30 & -28 & -12 \\ -13 & -21 & -11 & -32 \\ -20 & -2 & -6 & -24 \end{bmatrix}$$

There are N = 2^6 processors available on a cube-connected SIMD computer $P_0, P_1, \ldots, P_{63}$. The processors are arranged in a three-dimensional array as shown in Fig. 7.10(a). (Note that this three-dimensional array is in fact a six-dimensional cube with connections omitted for simplicity.) Each of i, j, k contributes two bits to the binary representation $r_5 r_4 r_3 r_2 r_1 r_0$ of the index r of processor P,: i = $r_5 r_4$, $j = r_3 r_2$, and $k = r_1 r_0$. Initially the matrices A and B are loaded into registers $P_0, \ldots, P_{15}$, as shown in Fig. 7.10(b).

Since q = 2, step 1 is iterated twice: once for m = 5 and once for m = 4. In the first iteration, all processors whose binary index $r_5 r_4 r_3 r_2 r_1 r_0$ is such that $r_5 = 0$ copy their contents into the processors with binary index $r'_5 r_4 r_3 r_2 r_1 r_0$ (i.e., $r'_5 = 1$). Thus $P_0, \ldots, P_{15}$ copy their initial contents into $P_{32}, \ldots, P_{47}$, respectively, and simultaneously $P_{16}, \ldots, P_{31}$ copy their initial contents (all zeros) into $P_{48}, \ldots, P_{63}$, respectively. In the second iteration, all processors whose binary index $r_5 r_4 r_3 r_2 r_1 r_0$ is such that $r_4 = 0$ copy their contents into the processors with binary index $r_5 r'_4 r_3 r_2 r_1 r_0$ (i.e., $r'_4 = 1$). Thus $P_0, \ldots, P_{15}$ copy their contents into $P_{16}, \ldots, P_{31}$, respectively, and simultaneously $P_{32}, \ldots, P_{47}$ copy their new contents (acquired in the previous iteration) into $P_{48}, \ldots, P_{63}$, respectively. At the end of step 1, the contents of the sixty-four processors are as shown in Fig. 7.10(c).

There are two iterations of step 2: one for m = 1 and one for m = 0. During the first iteration all processors with binary index $r_5 r_4 r_3 r_2 r_1 r_0$ such that $r_1 = r_5$ copy the contents of their A registers into those of processors with binary index $r_5 r_4 r_3 r_2 r'_1 r_0$. Thus, for example, P_0 and P_1 copy the contents of their A registers into the A registers of P_2 and P,, respectively. During the second iteration all processors with binary index $r_5 r_4 r_3 r_2 r_1 r_0$ such that $r_0 = r_4$ copy the contents of their A registers into the A registers of processors with binary index $r_5 r_4 r_3 r_2 r_1 r'_0$. Again, for example, P_0 and P_2 copy the contents of their A registers into the A registers of P_1 and P_3, respectively. At the end of this step one element of matrix A has been replicated across each "row" in Fig. 7.10(a). Step 3 is equivalent except that it replicates one element of matrix B across each "column." The contents of the sixty-four processors at the end of steps 2 and 3 are shown in Fig. 7.10(d). In step 4, with all processors operating simultaneously, each processor computes the product of its A and B registers and stores the result in its C register. Step 5 consists of two iterations: one for m = 4 and one for m = 5. In the first iteration the contents of the C registers of processor pairs whose binary indices differ in bit r_4 are added. Both processors keep the result. The same is done in the second iteration for processors differingin bit r,. The final answer, stored in $P_0, \ldots, P_{15}$ is shown in Fig. 7.10(e). □



(a)

17 -7	23 -25	27 -19	3 -5
9 -18	1 -30	14 -28	16 -12
31 -13	26 -21	22 -11	8 -32
15 -20	4 -2	10 -6	29 -24

(b)

17 -7	23 -25	27 -19	3 -5
9 -18	1 -30	14 -28	16 -12
31 -13	26 -21	22 -11	8 -32
15 -20	4 -2	10 -6	29 -24

17 -7	23 -25	27 -19	3 -5
9 -18	1 -30	14 -28	16 -12
31 -13	26 -21	22 -11	8 -32
15 -20	4 -2	10 -6	29 -24

17 -7	23 -25	27 -19	3 -5
9 -18	1 -30	14 -28	16 -12
31 -13	26 -21	22 -11	8 -32
15 -20	4 -2	10 -6	29 -24

17 -7	23 -25	27 -19	3 -5
9 -18	1 -30	14 -28	16 -12
31 -13	26 -21	22 -11	8 -32
15 -20	4 -2	10 -6	29 -24

(c)

17 -7	17 -25	17 -19	17 -5
9 -7	9 -25	9 -19	9 -5
31 -7	31 -25	31 -19	31 -5
15 -7	15 -25	15 -19	15 -5

23 -18	23 -30	23 -28	23 -12
1 -18	1 -30	1 -28	1 -12
26 -18	26 -30	26 -28	26 -12
4 -18	4 -30	4 -28	4 -12

27 -13	27 -21	27 -11	27 -32
14 -13	14 -21	14 -11	14 -32
22 -13	22 -21	22 -11	22 -32
10 -13	10 -21	10 -11	10 -32

3 -20	3 -2	3 -6	3 -24
16 -20	16 -2	16 -6	16 -24
8 -20	8 -2	8 -6	8 -24
29 -20	29 -2	29 -6	29 -24

(d)

-944	-1688	-1282	-1297
-583	-581	-449	-889
-1131	-2033	-1607	-1363
-887	-763	-681	-1139

(e)

Figure 7.10 Multiplying two matrices using procedure CUBE MATRIX MULTIPLICATION.

7.3.3 CRCW Multiplication

We conclude this section by presenting a parallel algorithm for matrix multiplication that is faster and has lower cost than procedure CUBE MATRIX MULTIPLICATION. The algorithm is designed to run on a CRCW SM SIMD computer. We assume that ***write conflicts*** are resolved as follows: When several processors attempt to write in the same memory location, the ***sum*** of the numbers to be written is stored in that location. The algorithm is a direct parallelization of sequential procedure MATRIX MULTIPLICATION. It uses $m \times n \times k$ processors to multiply an $m \times n$ matrix A by an $n \times k$ matrix B. Conceptually the processors may be thought of as being arranged in a $m \times n \times k$ array pattern, each processor having three indices (i, j, s), where $1 \leqslant i \leqslant m$, $1 \leqslant j \leqslant n$, and $1 \leqslant s \leqslant k$. Initially matrices A and B are in shared memory; when the algorithm terminates, their product matrix C is also in shared memory. The algorithm is given as procedure CRCW MATRIX MULTIPLICATION.

procedure CRCW MATRIX MULTIPLICATION (A, B, C)

```
for i = 1 to m do in parallel
  for j = 1 to k do in parallel
    for s = 1 to n do in parallel
      (1) c_ij ← 0
      (2) c_ij ← a_is x b_sj
    end for
  end for
end for. □
```

Analysis. It is clear that procedure CRCW MATRIX MULTIPLICATION runs in constant time. Since $p(n) = n^3$,

$$\begin{aligned} c(n) &= p(n) \times t(n) \\ &= n^3 \times O(1) \\ &= O(n^3), \end{aligned}$$

which matches the running time of sequential procedure MATRIX MULTIPLICATION.

Example 7.6

A CRCW SM SIMD computer with sixty-four processors can multiply the two matrices A and B of example 7.5 in constant time. All sixty-four products shown in Fig. 7.10(d) are computed simultaneously and stored (i.e., added) in groups of four in the appropriate position in C. Thus, for example, P_1, P_2, P_3, and P_4 compute 17 x (−7), 23 x (−18), 27 x (−13), and 3 x (−20), respectively, and store the results in c_{11}, yielding $c_{11} = -944$. □

7.4 MATRIX-BY-VECTOR MULTIPLICATION

The problem addressed in this section is that of multiplying an m x n matrix A by an n x 1 vector U to produce an m x 1 vector V, as shown for m = 3 and n = 4:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{bmatrix} \times \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}.$$

The elements of V are obtained from

$$v_i = \sum_{j=1}^{n} a_{ij} \times u_j, \qquad 1 \leqslant i \leqslant m.$$

This of course is a special case of matrix-by-matrix multiplication. We study it separately in order to demonstrate the use of two interconnection networks in performing matrix operations, namely, the linear (or one-dimensional) array and the tree. In addition, we show how a parallel algorithm for matrix-by-vector multiplication can be used to solve the problem of convolution.

7.4.1 Linear Array Multiplication

Our first algorithm for matrix-by-vector multiplication is designed to run on a linear array with m processors $P_1, P_2, \ldots, P_m$. Processor P_i is used to compute element v_i of V. Initially, v_i is zero. Matrix A and vector U are fed to the array, as shown in Fig. 7.11, for n = 4 and m = 3. Each processor P_i has three registers a, u, and v. When P_i receives two inputs a_{ij} and u_j, it

(i) stores a_{ij} in a and u_j in u,
(ii) multiplies a by u
(iii) adds the result to v_i, and
(iv) sends u_j to P_{i-1} unless i = 1.

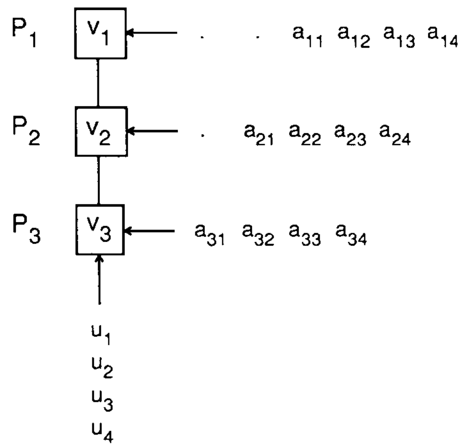


Figure 7.11 Matrix and vector to be multiplied, being fed as input to linear array.

Note that row i of matrix A lags one time unit behind row i + 1 for $1 \leqslant i \leqslant m - 1$. This ensures that a_{ij} meets u_j at the right time. The algorithm is given as procedure LINEAR MV MULTIPLICATION.

procedure LINEAR MV MULTIPLICATION (A, U, V)

for $i = 1$ **to** m **do in parallel**
  (1) $v_i \leftarrow 0$
  (2) **while** P_i receives two inputs a and u **do**
    (2.1) $v_i \leftarrow v_i + (a \times u)$
    (2.2) **if** $i > 1$ **then** send u **to** P_{i-1}
      end if
    end while
end for. □

Analysis. Element a,, takes $m + n - 1$ steps to reach P_1. Since P_1 is the last processor to terminate, this many steps are required to compute the product. Assuming $m \leqslant n$, procedure LINEAR MV MULTIPLICATION therefore runs in time $t(n) = O(n)$. Since m processors are used, the procedure has a cost of $O(n^2)$, which is optimal in view of the $\Omega(n^2)$ steps required to read the input sequentially.

Example 7.7

The behavior of procedure LINEAR MV MULTIPLICATION for

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \quad \text{and} \quad U = \begin{bmatrix} 5 \\ 6 \end{bmatrix}$$

is illustrated in Fig. 7.12. □

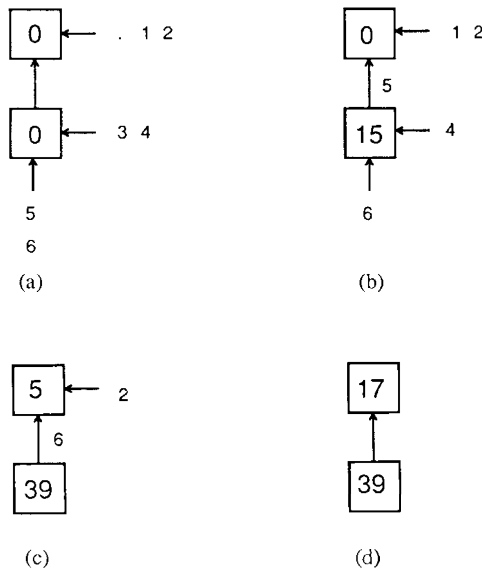


Figure 7.12 Multiplying matrix by vector using procedure LINEAR MV **MULTIPLICATION.**

7.4.2 Tree Multiplication

As observed in the previous section, matrix-by-vector multiplication requires $m + n - 1$ steps on a linear array. It is possible to reduce this time to $m - 1 + \log n$ by performing the multiplication on a tree-connected SIMD computer. The arrangement is as shown in Fig. 7.13 for $m = 3$ and $n = 4$. The tree has n leaf processors $P_1, P_2, \ldots, P_n$, $n - 2$ intermediate processors $P_{n+1}, P_{n+2}, \ldots, P_{2n-2}$, and a root processor P_{2n-1}. Leaf processor P_i stores u_i throughout the execution of the algorithm. The matrix A is fed to the tree row by row, one element per leaf. When leaf processor P_i receives a_{ji}, it computes $u_i \times a_{ji}$ and sends the product to its parent. When intermediate or root processor P_k receives two inputs from its children, it adds them and sends the result to its parent. Eventually v_j emerges from the root. If the rows of A are input at the leaves in consecutive time units, then the elements of V are also produced as output from the root in consecutive time units. The algorithm is given as procedure TREE MV MULTIPLICATION.

procedure TREE MV MULTIPLICATION (A, U, V)

do steps 1 and **2 in parallel**
  (1) for i = 1 **to** n **do in parallel**
    for $j = 1$ **to** m **do**
      (1.1) compute $u_i \times a_{ji}$
      (1.2) send result to parent
    end for
  end for

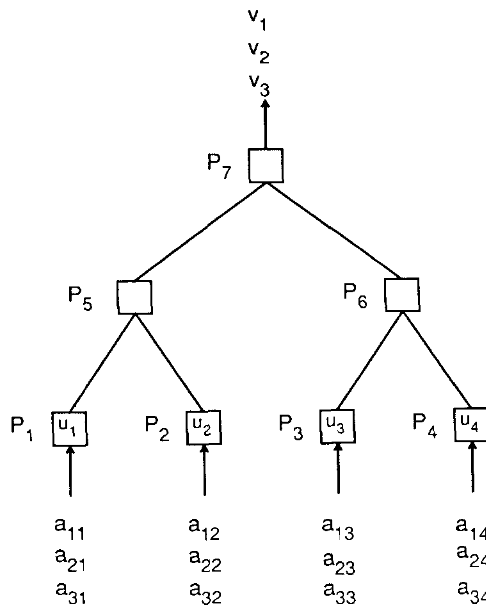


Figure 7.13 Tree-connected computer for matrix-by-vector multiplication.

(2) **for** i = n + 1 to 2n − 1 **do in parallel**
  while P_i receives two inputs **do**
    (2.1) compute the sum of the two inputs
    (2.2) **if** i < 2n − 1 **then** send the result to parent
                else produce the result as output
      end if
  end while
end for. □

Analysis. It takes log n steps after the first row of A has been entered at the leaves for v_1 to emerge from the root. Exactly m − 1 steps later, v_m emerges from the root. Procedure TREE MV MULTIPLICATION thus requires m − 1 + log n steps for a cost of $O(n^2)$ when $m \leqslant n$. The procedure is therefore faster than procedure LINEAR MV MULTIPLICATION while using almost twice as many processors. It is cost optimal in view of the $\Omega(n^2)$ time required to read the input sequentially.

Example 7.8

The behavior of procedure TREE MV MULTIPLICATION is illustrated in Fig. 7.14 for the same data as in example 7.7. □

7.4.3 Convolution

We conclude this section by demonstrating one application of matrix-by-vector multiplication algorithms. Given a sequence of constants $\{w_1, w_2, \ldots, w_n\}$ and an

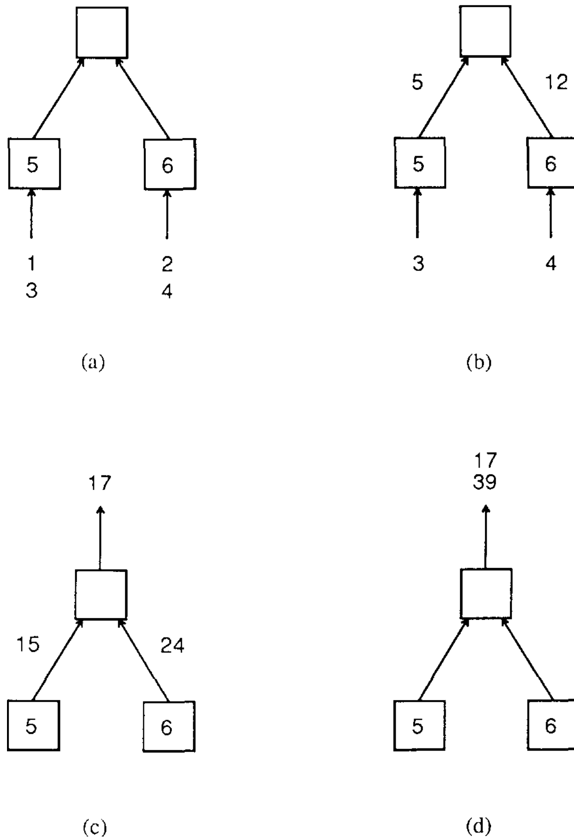


Figure 7.14 Multiplying matrix by vector using procedure TREE MV **MULTIPLICATION.**

input sequence $\{x_1, x_2, \ldots, x_n\}$, it is required to compute the output sequence $\{y_1, y_2, \ldots, y_{2n-1}\}$ defined by

$$y_i = \sum_{j=1}^{n} x_{i-j+1} \times w_j, \qquad 1 \leqslant i \leqslant 2n - 1.$$

This computation, known as ***convolution,*** is important in digital signal processing. It can be formulated as a matrix-by-vector multiplication. This is shown for the case $n = 3$:

$$\begin{bmatrix} x_1 & 0 & 0 \\ x_2 & x_1 & 0 \\ x_3 & x_2 & x_1 \\ 0 & x_3 & x_2 \\ 0 & 0 & x_3 \end{bmatrix} \times \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix}.$$

7.5 PROBLEMS

7.1 Procedure MESH TRANSPOSE requires that the destination (j, i) of each element a_{ij} be sent along with it during the computation of the transpose of a matrix A. Design an algorithm for transposing a matrix on a'mesh where it is not necessary for each element to carry its new destination along.

7.2 Is the running time of procedure SHUFFLE TRANSPOSE the smallest achievable when transposing a matrix on a shuffle-connected SIMD computer?

7.3 Can the transpose of an $n \times n$ matrix be obtained on an interconnection network, other than the perfect shuffle, in $O(\log n)$ time?

7.4 Is there an interconnection network capable of simulating procedure EREW TRANSPOSE in constant time?

7.5 Assume that every processor of an $n \times n$ mesh-connected computer contains one element of each of two $n \times n$ matrices A and B. Use a "distance" argument to show that, regardless of input and output considerations, this computer requires $\Omega(n)$ time to obtain the product of A and B.

7.6 Modify procedure MESH MULTIPLICATION so it can be used in a pipeline fashion to multiply several pairs of matrices. By looking at Fig. 7.7, we see that as soon as processor $P(1, 1)$ has multiplied a_{11} and b_{11}, it is free to receive inputs from a new pair of matrices. One step later, $P(1, 2)$ and $P(2, 1)$ are ready, and so on. The only problem is with the results of the previous computation: Provision must be made for c_{ij}, once computed, to vacate $P(i, j)$ before the latter becomes involved in computing the product of a new matrix pair.

7.7 Consider an $n \times n$ mesh of processors with the following additional links: (i) the rightmost processor in each row is directly connected to the leftmost, (ii) the bottommost processor in each column is directly connected to the topmost. These additional links are called wraparound connections. Initially, processor $P(i, j)$ stores elements a_{ij} and b_{ij} of two matrices A and B, respectively. Design an algorithm for multiplying A and B on this architecture so that at the end of the computation, $P(i, j)$ contains (in addition to a_{ij} and b_{ij}) element c_{ij} of the product matrix C.

7.8 Repeat problem **7.7** for the mesh under the same initial conditions but without the wraparound connections.

7.9 Design an algorithm for multiplying two $n \times n$ matrices on a mesh with fewer than n^2 processors.

7.10 Design an algorithm for multiplying two $n \times n$ matrices on an $n \times n$ mesh of trees architecture (as described in problem 4.2).

7.11 Extend the mesh of trees architecture to three dimensions. Show how the resulting architecture can be used to multiply two $n \times n$ matrices in $O(\log n)$ time using n^3 processors. Show also that m pairs of $n \times n$ matrices can be multiplied in $O(m + 2\log n)$ steps.

7.12 Assume that every processor of a cube-connected computer with n^2 processors contains one element of each of two $n \times n$ matrices A and B. Use a "distance" argument to show that, regardless of the number of steps needed to evaluate sums, this computer requires $\Omega(\log n)$ time to obtain the product of A and B.

7.13 Design an algorithm for multiplying two $n \times n$ matrices on a cube with n^2 processors in $O(n)$ time.

7.14 Combine procedure CUBE MATRIX MULTIPLICATION and the algorithm in problem 7.13 to obtain an algorithm for multiplying two n x n matrices on a cube with n^2m processors in $O((n/m) + \log m)$ time, where $1 \leqslant m \leqslant n$.

7.15 Design an algorithm for multiplying two matrices on a perfect shuffle-connected SIMD computer.

7.16 Repeat problem 7.15 for a tree-connected SIMD computer.

7.17 It is shown in section 7.3.2 that n processors require $\Omega(\log n)$ steps to add n numbers. Generalize this bound for the case of k processors, where $k < n$.

7.18 Modify procedure CRCW MATRIX MULTIPLICATION to run on an EREW SM SIMD computer. Can the modified procedure be made to have a cost of $O(n^3)$?

7.19 Design an MIMD algorithm for multiplying two matrices.

7.20 Given m n x n matrices $A,, A,, \ldots, A,,$ design algorithms for two different interconnection networks to compute the product matrix

$$C = A_1 \; x \; A, \; x \cdots x \; A,.$$

7.21 Let w be a primitive nth root of unity, that is, $w^n = 1$ and $w^i \neq 1$ for $1 \leqslant i < n$. The Discrete Fourier Transform (DFT) of the sequence $\{a,, a,, \ldots, a_{n-1}\}$ is the sequence $\{b_0, b_1, \ldots, b_{n-1}\}$ where

$$b_j = \sum_{i=0}^{n-1} a_i \; x \; w^{ij} \quad \text{for } 0 \leqslant j < n.$$

Show how the DFT computation can be expressed as a matrix-by-vector product.

7.22 The inverse of an n x n matrix A is an n x n matrix A^{-1} such that A x $A^{-1} = A^{-1}$ x $A = I$, where I is an n x n identity matrix whose entries are 1 on the main diagonal and 0 elsewhere. Design a parallel algorithm for computing the inverse of a given matrix.

7.23 A q-dimensional cube-connected SIMD computer with $n = 2^q$ processors $P_0, P_1, \ldots, P_{n-1}$ is given. Each processor P_i holds a datum x_i. Show that each of the following computations can be done in $O(\log n)$ time:
(*a*) Broadcast x_0 to $P_1, P_2, \ldots, P_{n-1}$.
(*b*) Replace x_0 with $x_0 + x_1 + \cdots + x_{n-1}$.
(*c*) Replace x_0 with the smallest (or largest) of $x_0, x_1, \ldots, x_{n-1}$.

7.24 An Omega network is a multistage interconnection network with n inputs and n outputs. It consists of $k = \log n$ rows numbered $1, 2, \ldots, k$ with n processors per row. The processors in row i are connected to those in row $i + 1$, $i = 1, 2, \ldots, k - 1$, by a perfect shuffle interconnection. Discuss the relationship between the Omega network and a k-dimensional cube.

7.6 BIBLIOGRAPHICAL REMARKS

A mesh algorithm is described in [Ullman] for computing the transpose of a matrix that, unlike procedure MESH TRANSPOSE, does not depend directly on the number of processors on the mesh. Procedure SHUFFLE TRANSPOSE is based on an idea proposed in [Stone 1].

For references to sequential matrix multiplication algorithms with $O(n^x)$ running time, $2 < x < 3$, see [Gonnet], [Strassen], and [Wilf]. A lower bound on the number of parallel steps required to multiply two matrices is derived in [Gentleman]. Let $f(k)$ be the maximum number