5- ran’h

In this section, we study matrix multiplication in parallel. We present an algorithm for multiplying two n x n matrices. For clarity, we assume that n is a power of 2. We use the CREW PRAM model to allow multiple read operations from the same memory locations. Recall that in a CREW PRAM, multiple read operations can be conducted concurrently, but multiple write operations are performed exclusively. We start by presenting the algorithm on a CREW PRAM with n^3 processors. We will then show how to reduce the cost by using fewer processors. We assume that the two input matrices are stored in the shared memory in the arrays A[1 . . n, 1. . n], B[1 . . n, 1. . n].

5.1- Using n^3 Processors

We consider the n^3 processors as being arranged into a three-dimensional array. Processor $P_{i,j,k}$ is the one with index (i, j, k). A three-dimensional array C[i, j, k], where $1 \le i, j, k \le n$, in the shared memory will be used as working space. The resulting matrix will be stored in locations C[i, j, n], where $1 \le i, j \le n$.

The algorithm consists of two steps. In the first step, all n^3 processors operate in parallel to compute n^3 multiplications. For each of the n^2 cells in the output matrix, n products are computed. In the second step, the n products computed for each cell in the output matrix are summed to produce the final value of this cell. This summation can be performed in parallel in O(log n) time as shown in Algorithm Sum_EREW discussed earlier. The two steps of the algorithm are given as:

1. Each processor $P_{i,j,k}$ computes the product of A[i, k] * B[k, j] and stores it in C[i, j, k].
2. The idea of Algorithm Sum_EREW is applied along the k dimension n^2 times in parallel to compute C[i, j, n], where $1 \le i, j \le n$.

The details of these two steps are presented in Algorithm MatMult_CREW:

```
Algorithm MatMult_CREW
/* Step 1 */
forall P_{i,j,k}, where 1 ≤ i, j, k ≤ n do in parallel
  C[i,j,k] ← A[i,k]*B[k,j]
endfor

/* Step 2 */
for l=1 to log n do
  forall P_{i,j,k}, where 1 ≤ i, j ≤ n & 1 ≤ k ≤ n/2 do in parallel
    if (2k modulo 2^l)=0 then
      C[i,j,2k] ← C[i,j,2k] + C[i,j, 2k − 2^{l−1}]
    endif
  endfor
  /* The output matrix is stored in locations
    C[i,j,n], where 1 ≤ i,j ≤ n      */
  endfor
```

Figure 6.6 shows the activities of the active processors after each of the two steps of Algorithm MatMult_CREW when used to multiply two 2x2 matrices.

Complexity Analysis In the first step, the products are conducted in parallel in constant time, that is, O(1). These products are summed in O(log n) time during the second step. Therefore, the run time is O(log n). Since the number of processors used is n^3, the cost is $O(n^3 \log n)$. The complexity measures of the matrix multiplication on CREW PRAM with n^3 processors are summarized as:

1. Run time, $T(n) = O(\log n)$.
2. Number of processors, $P(n) = n^3$.
3. Cost, $C(n) = O(n^3 \log n)$.

Since an n x n matrix multiplication can be done sequentially in less than $O(n^3 \log n)$, this algorithm is not cost optimal. In order to reduce the cost of this parallel algorithm, we should try to reduce the number of processors.

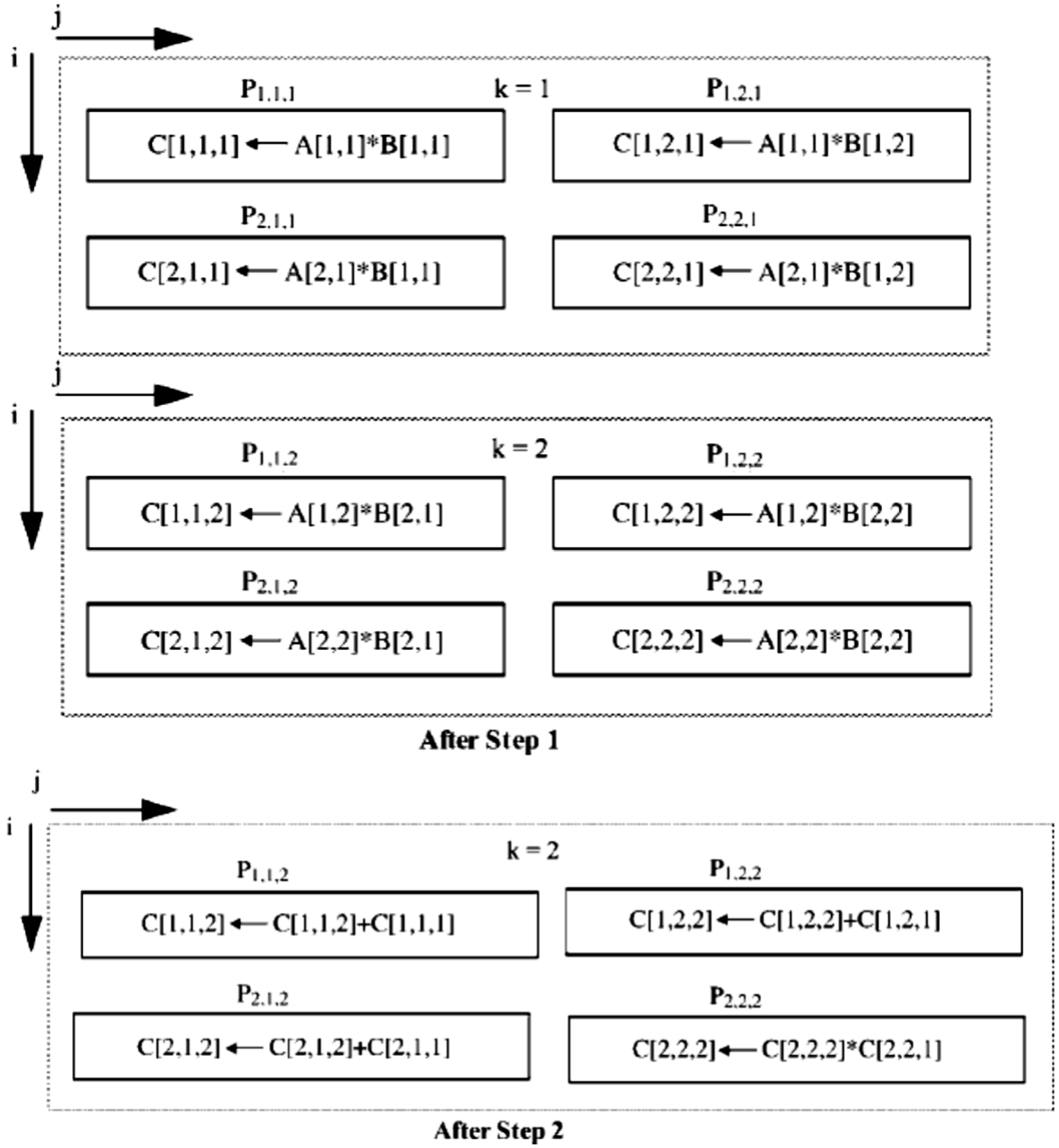

Figure 6.6 Multiplying two 2 x 2 matrices using Algorithm MatMult_CREW.

5.2- Reducing the Number of Processors

In the above algorithm, although all the processors were busy during the first step, not all of them performed addition operations during the second step. As you can see, the second step consists of log n iterations. During the first iteration, only $n^3/2$ processors performed addition operations, only $n^3/4$ performed addition operations in the second iteration, and so on. With this understanding, we may be able to use a smaller machine

with only $n^3/\log n$ processors. But how can this be done? The idea is to arrange the processors in n * n * n/log n three-dimensional array. The two steps of the Algorithm MatMult_CREW can be modified as:

1. Each processor $P_{i,j,k}$, where $1 \le k \le n/\log n$, computes the sum of log n products. This step will produce $(n^3/\log n)$ partial sums.
2. The sum of products produced in step 1 are added to produce the resulting matrix as discussed before.

Complexity Analysis Since each processor is responsible for computing log n product terms and obtaining their sum in step 1, each processor performs log n multiplications and (log n)-1 additions during this step. These products are summed in log (n/log n) time during step 2. Therefore, the execution time of step 1 and step 2 of the algorithm is 2 (log n) –1+ log(n/log n), which can be approximated as O(log n) for large values of n. The complexity measures of the matrix multiplication on CREW PRAM with $n^3/\log n$ processors are summarized as:

1. Run time, T(n) =O(log n).
2. Number of processors, $P(n) = n^3/\log n$.
3. Cost, $C(n) = O(n^3)$.

Is Algorithm MatMult_CREW after modification cost optimal?

6- Sorting

The sorting algorithm we present here is based on the enumeration idea. Given an unsorted list of n elements a_1 , a_2 , . . . , a_i , . . . , a_n , an enumeration sort determines the position of each element a_i in the sorted list by computing the number of elements smaller than it. If c_i elements are smaller than a_i, then it is the $(c_i + 1)^{th}$ element in the sorted list. If two or more elements have the same value, the element with the largest index in the unsorted list will be considered as the largest in the sorted list. For example, suppose that $a_i = a_j$, then a_i will be considered the larger of the two if i > j, otherwise a_j is the larger.

We present this simple algorithm on a CRCW PRAM with n^2 processors. Recall that in a CRCW PRAM multiple read operations can be conducted concurrently; so are multiple write operations. However, write conflicts must be resolved according to a certain policy. In this algorithm, we assume that when multiple processors try to write different values into the same address, the sum of these values will be stored in that address.

Consider the n^2 processors as being arranged into n rows of n elements each. The processors are numbered as follows: $P_{i,j}$ is the processor

located in row i and column j in the grid of processors. We assume that the sorted list is stored in the global memory in an array A[1 . . n]. Another array C[1 . . n] will be used to store the number of elements smaller than every element in A. The algorithm consists of two steps:

1. Each row of processors i computes C[i], the number of elements smaller than A[i]. Each processor $P_{i,j}$ compares A[i] and A[j], then updates C[i] appropriately.
2. The first processor in each row $P_{i,1}$ places A[i] in its proper position in the sorted list (C[i] + 1).

The details of these two steps are presented in Algorithm Sort_CRCW:

```
Algorithm Sort_CRCW

/* Step 1 */
forall P(i,j), where 1 ≤ i, j ≤ n do in parallel

 if A[i] > A[j] or (A[i]=A[j] and i > j) then
   C[i] ← 1
 else
   C[i] ← 0
 endif
endfor

/* Step 2 */
forall P(i,1), where 1 ≤ i ≤ n do in parallel
  A[C[i] + 1] ← A[i]
endfor
```

Complexity Analysis The complexity measures of the enumerating sort on CRCW PRAM are summarized as:

1. Run time, T(n) = O(1).
2. Number of processors, P(n) = n^2.
3. Cost, C(n) =O(n^2).

The run time of this algorithm is constant because each of the two steps of the algorithm consumes a constant amount of time. Since the number of processors used is n^2, the cost is obviously O(n^2). Since a good sequential algorithm can sort a list of n elements in O(n log n), this algorithm is not cost optimal. Although the above algorithm sorts n elements in constant time, it has no practical value because it uses a very large number

of processors in addition to its reliance on a very powerful PRAM model (CRCW). How can you reduce the cost?

Example 2 Let us sort the array A = [6, 1, 3]. As shown in Figure 6.7, we need nine processors to perform the sort. The figure shows the contents of the shared memory and the elements that each processor compares.

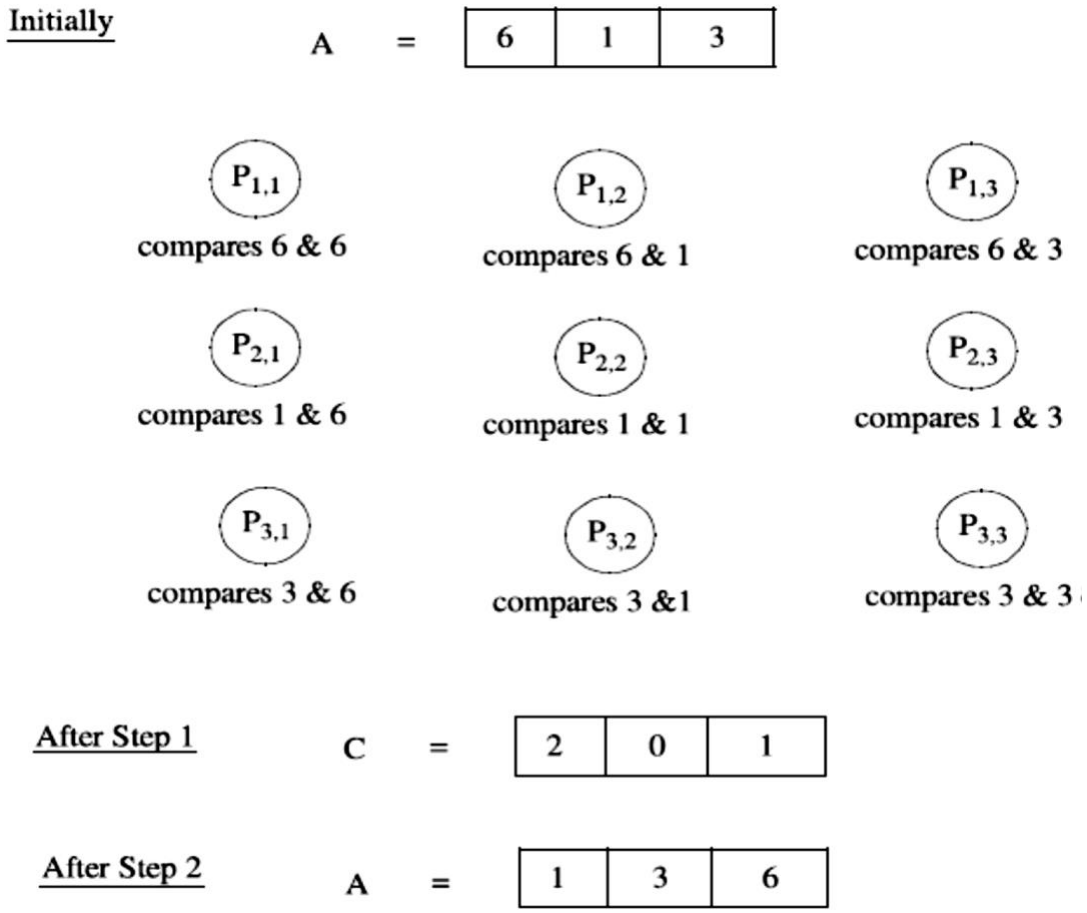

Figure 6.7 Enumeration sort of [6,1,3] on a CRCW PRAM.

7- Message Passing Model

An algorithm designed for a message passing system consists of a collection of local programs running concurrently on the different processing units in a distributed system. Each local program performs a sequence of computation and message passing operations. Message passing in distributed systems can be modeled using a communication graph. The nodes of the graph represent the processors (or the processes running on them) and the

edges represent communication links between processors. Throughout this lecture, we will not distinguish between a processor and its process. Each node representing a process has a set of neighbors with which it can communicate. The communication graph may be directed or undirected.

A directed edge indicates unidirectional communication, while an undirected edge implies bidirectional communication. The two graphs of Figure 6.8 are examples of unidirectional and bidirectional communication. For example, it can be seen in Figure 6.8a that the outgoing neighbors of P3 are P1 and P5, while its incoming neighbors are P2 and P4. In Figure 6.8b, processes P1, P2, P4, and P5 are incoming as well as outgoing neighbors of P3.

A message passing distributed system may operate in synchronous, asynchronous, or partially synchronous modes. In the synchronous extreme, the execution is completely lock-step and local programs proceed in synchronous rounds.

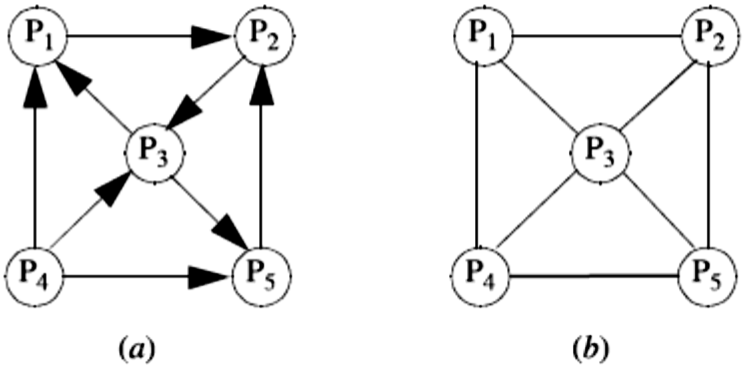

Figure 6.8 Unidirectional and bidirectional communication graphs (a) directed communication graph; and (b) unidirected communication graph.

For example, in one round each local program sends messages to its outgoing neighbors, waits for the arrival of messages from its incoming neighbors, and performs some computation upon the receipt of the messages. In the other extreme, in asynchronous mode the local programs execute in arbitrary order at arbitrary rate. The partially synchronous systems work at an intermediate degree of synchrony, where there are restrictions on the relative timing events.

In this section we focus on studying synchronous message passing systems. We outline the basic elements of a formal model of a distributed system as described by Lynch (1996). The models presented here may be more detailed than needed in some cases, but they help give the reader a good idea of the formal representation of different and more general

systems. We also discuss the complexity analysis of algorithms described in terms of these formal models as we define the time and message complexity.

7.1- Synchronous Message Passing Model

Given a message passing system consisting of n processes, each of which is running on a separate processor, we show how to model such a system in the synchronous mode. As mentioned earlier, the communication among processors is represented using a communication graph G = (V,E). The behavior of this system can be described as follows:

1. System is initialized and set to an arbitrary initial state.
2. For each process $i \in V$, repeat the following two steps in synchronized rounds (lock-step fashion):

(a) Send messages to the outgoing neighbors by applying some message generation function to the current state.

(b) Obtain the new state by applying a state transition function to the current state and the messages received from incoming neighbors.

An execution of this synchronized system can be represented as a sequence of (1) states, (2) sent messages, and (3) received messages as follows:

$$\text{state}_0, \text{sent-msg}_1, \text{rcvd-msg}_1, \text{state}_1, \text{sent-msg}_2, \text{rcvd-msg}_2, \text{state}_2, \ldots, \text{sent-msg}_j, \text{rcvd-msg}_j, \text{state}_j, \ldots$$

The system changes its current state to a new state based on the messages sent and received among the processes. Note that the messages received may not be the same as the messages sent because some of them may be lost as a result of a faulty channel. For example, the system starts at state_0 and is changed to state_1 after the sending and receiving of sent-msg_1 and rcvd-msg_1. The system then changes from state_1 to state_2, and so on.

Thus, a synchronous system can be modeled as a state machine with the following components:

1. M, a fixed message alphabet.
2. A process i can be modeled as:

(a) Q_i, a (possibly infinite) set of states. The system state can be represented using a set of variables.

(b) $q_{0,i}$, the initial state in the state set Q_i. The state variables have initial values in the initial state.

(c) GenMsg_i, a message generation function. It is applied to the current system state to generate messages to the outgoing neighbors from elements in M.

(d) $Trans_i$, a state transition function that maps the current state and the incoming messages into a new state.

Suppose that the communication links are reliable and the messages received by process i are the same as the ones sent by its incoming neighbors. Figure 6.9 shows a simple example of a state diagram for process i. Starting at state $q_{0,i}$, process i receives the messages Msg_1 from its incoming neighbors and changes to state $q_{1,i}$. Process i at state $q_{1,i}$ now receives the messages Msg_2 from its incoming neighbors and changes its state to $q_{2,i}$. This process is repeated any number of times as shown in the figure. Note that after k rounds process i will be at state $q_{k,i}$. In order to provide a description for algorithms studied under the synchronous model, the following is a template that we will be using in this chapter. The template, which is referred to as S_Template, describes the computation carried out by process $i \in V$. The prefix S_ in the algorithm's name is meant to indicate that it is synchronous.

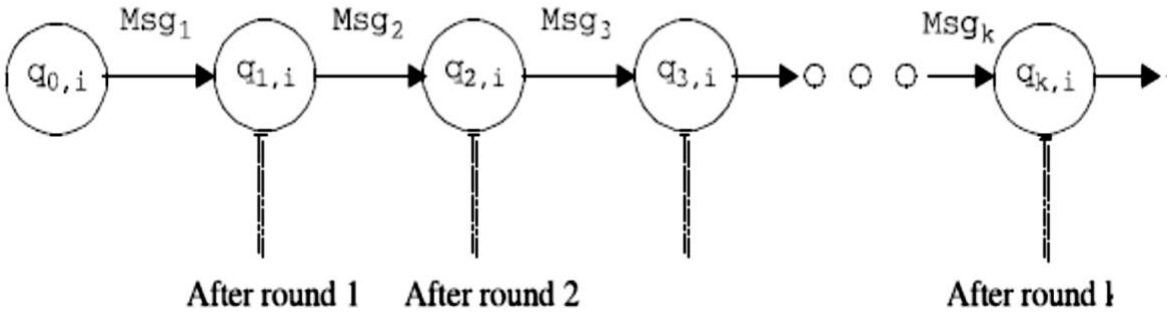

Figure 6.9 An example of a state diagram for process i.

```
Algorithm S_Template
Qi
    <state variables used by process i>
q0,i
    <state variables> ← <initial values>
GenMsgi
    <Send one message to each of a (possibly empty) subset of
    outgoing neighbors>
Transi
    <update the state variables based on the incoming messages>
```

7.2- Complexity Analysis

As discussed earlier, complexity analysis of algorithms is usually expressed in terms of the amount of resources needed by the computation to be completed. In addition to the run time, the amount of communication is an important resource in message passing systems. The complexity of

algorithms in such systems will be measured quantitatively using time complexity and message complexity. The measures of complexity will be expressed in the usual asymptotic fashion as functions of the number of nodes and edges in the communication graph representing the distributed system.

Message Complexity The message complexity is defined as the number of messages sent between neighbors during the execution of the algorithm. We usually consider the worst-case message complexity, which is the maximum number of messages.

Time Complexity The time complexity is defined generally as the time spent during the execution of the algorithm. The definition of the time complexity in the synchronous model is rather simple, and amounts to the number of rounds until the termination of the algorithm. Defining the time complexity of an algorithm in asynchronous executions is not so straightforward. The time complexity in this case only considers messages that happen sequentially. It can be obtained by assigning occurrence times to events in the execution under some restrictions.

Example 3 Consider a synchronous hypercube made of n processors. Assume that each link connecting two adjacent processors can perform communication in both directions at the same time (bidirectional communication). Suppose that each process i has its own data x_i, and we would like to compute the summation of data at all processes. The summation can be computed in log n rounds. At each round each process sends its data to one of its neighbors along one of the hypercube dimensions. At the end, each process will have the summation stored in its local space. A formal description of this method using the above model is shown below in *Algorithm S_Sum_Hypercube*:

Algorithm S_Sum_Hypercube

Q_i

buff, an integer
dim, a value in {1, 2, ..., log n}

$q_{0,i}$

buff $\leftarrow x_i$
dim $\leftarrow$ log n

$GenMsg_i$

If the current value of dim = 0, do nothing. Otherwise, send the current value of buff to the neighbor along the dimension dim.

$Trans_i$

if the incoming message is v & dim > 0, then
buff $\leftarrow$ buff + v, dim $\leftarrow$ dim - 1

Figure 6.10 illustrates the rounds of the algorithm for a hypercube of dimension 3. Assume that the data values 1, 2, 3, 3, 9, 5, 4, 22 are distributed to the eight processes as shown in Figure 6.10a. The states after each of the first three rounds are shown in Figures 6.10b, c, and d.

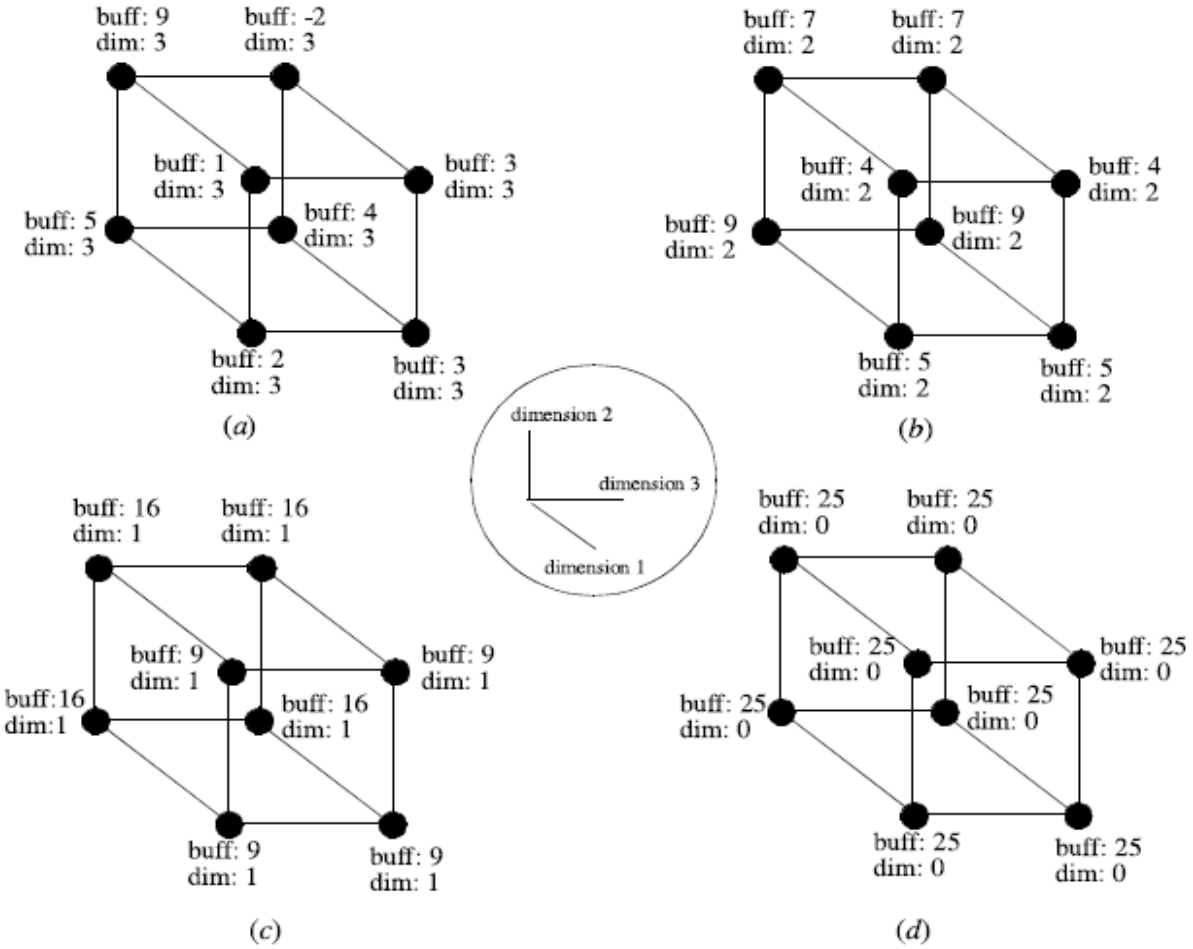

Figure 6.10 Computing the summation synchronously on a three-dimensional hypercube (a) initial states; (b) after first round; (c) after second round;
and (d) after third round.

Complexity Analysis Given n processors connected via a hypercube, S_Sum_Hypercube needs log n rounds to compute the sum. Since n messages are sent and received in each round, the total number of messages is O(n log n).

Therefore, the complexity of Algorithm S_Sum_Hypercube is summarized:

1. Time complexity: O(log n).
2. Message complexity: O(n log n).