64 KB. Thus, on a machine with 256-KB caches, the miss rate drops from 5% to 1% as the configuration scales from four to sixteen or more processors. Indeed, the sum of the computational time over all the processors drops as the configuration scales, so perfect speedup is obtained even though the time spent communicating increases. This effect is less pronounced on the SP-2 because the basic node is optimized for operation on data that does not fit in the cache.

## 7.9 SYNCHRONIZATION

Scalability is a primary concern in the combination of software and hardware that implements synchronization operations in large-scale distributed-memory machines. With a message-passing programming model, mutual exclusion is a given since each process has exclusive access to its local address space. Point-to-point events are implicit in every message operation. The more interesting case is orchestrating global or group synchronization from point-to-point messages. An important issue here is balance: it is important that the communication pattern used to achieve the synchronization be balanced among nodes, in which case high message rates and efficient synchronization can be realized. In the extreme, we should avoid having all processes communicate with or wait for one process at a given time. Machine designers and implementers of message-passing layers attempt to maximize the message rate in such circumstances, but only the program can relieve the load imbalance. Other issues for global synchronization are similar to those for a shared address space.

In a shared address space, the issues for mutual exclusion and point-to-point events are essentially the same as those discussed in Chapter 5. As in small-scale shared memory machines, the trend in scalable machines is to build user-level synchronization operations (like locks and barriers) in software on top of basic atomic exchange primitives. Two major differences, however, may affect the choice of algorithms. First, the interconnection network is not centralized but has many parallel paths. On one hand, this means that disjoint sets of processors can coordinate with one another in parallel on entirely disjoint paths; on the other hand, it can complicate the implementation of synchronization primitives. Second, physically distributed memory may make it important to allocate synchronization variables appropriately among memories. The importance of this depends on whether the machine caches nonlocal shared data or not and is clearly greater for machines that do not, such as the ones described in this chapter. This section covers new algorithms for locks and barriers appropriate for machines with physically distributed memory and interconnect, starting from the algorithms discussed for shared memory machines. We will return to this comparison once we have studied scalable cache-coherent systems in the next chapter. Let us begin with algorithms for locks.

### 7.9.1 Algorithms for Locks

Section 5.5 presents the basic test&set lock, the test&set lock with backoff, the test-and-test&set lock, the ticket lock, and the array-based lock. Each successive step

went further in reducing bus traffic and fairness but often at a cost in overhead. For example, the ticket lock allowed only one process to issue a test&set when a lock was released, but all processors were notified of the release through an invalidation and a subsequent read miss to determine who should issue the test&set. The array-based lock fixed this problem by having each process wait on a different location and the releasing process notify only one process of the release by writing the corresponding location.

However, the array-based lock has two potential problems for scalable machines with physically distributed memory. First, each lock requires space proportional to the number of processors. Second, and more important for machines that do not cache remote data, there is no way to know ahead of time which location a process will spin on since this is determined at run time through a fetch&increment operation. This makes it impossible to allocate the synchronization variables in such a way that the variable a process spins on is always in its local memory (in fact, all of the locks in Chapter 5 have this problem). On a distributed-memory machine without coherent caches, such as the CRAY T3D and T3E, this is a big problem since processes will spin on remote locations, causing inordinate amounts of traffic and contention. Fortunately, a software lock algorithm is available that both reduces the space requirements and ensures all spinning will be on locally allocated variables. This lock, known as a *software queuing lock,* is a software implementation of a lock originally proposed for an all-hardware implementation by the Wisconsin Multicube project (Goodman, Vernon, and Woest 1989). The idea is to have a distributed linked list or a queue of waiters on the lock. The head node in the list represents the process that holds the lock. Every other node is a process that is waiting on the lock and is allocated in that process's local memory. A node points to the process (node) that tried to acquire the lock just after it. There is also a tail pointer that points to the last node in the queue, that is, the last node to have tried to acquire the lock. Let us look pictorially at how the queue changes as processes acquire and release the lock; then we will examine the code for the acquire and release methods.

Assume that the lock in Figure 7.39 is initially free. When process A tries to acquire the lock, it gets it, and the queue looks as shown in Figure 7.39(a). In step (b), process B tries to acquire the lock, so it is put on the queue and the tail pointer now points to it. Process C is treated similarly when it tries to acquire the lock in step (c). B and C are now spinning on local flags associated with their queue nodes while A holds the lock. In step (d), process A releases the lock. It then "wakes up" the next process, B, in the queue by writing the flag associated with B's node, and leaves the queue. B now holds the lock and is at the head of the queue. The tail pointer does not change. In step (e), B releases the lock similarly, passing it on to C. There are no other waiting processes, so C is at both the head and tail of the queue. If C releases the lock before another process tries to acquire it, then the lock pointer will be NULL and the lock will be free again. In this way, processes are granted the lock in FIFO order with regard to the order in which they tried to acquire it. The latter order will be defined next.

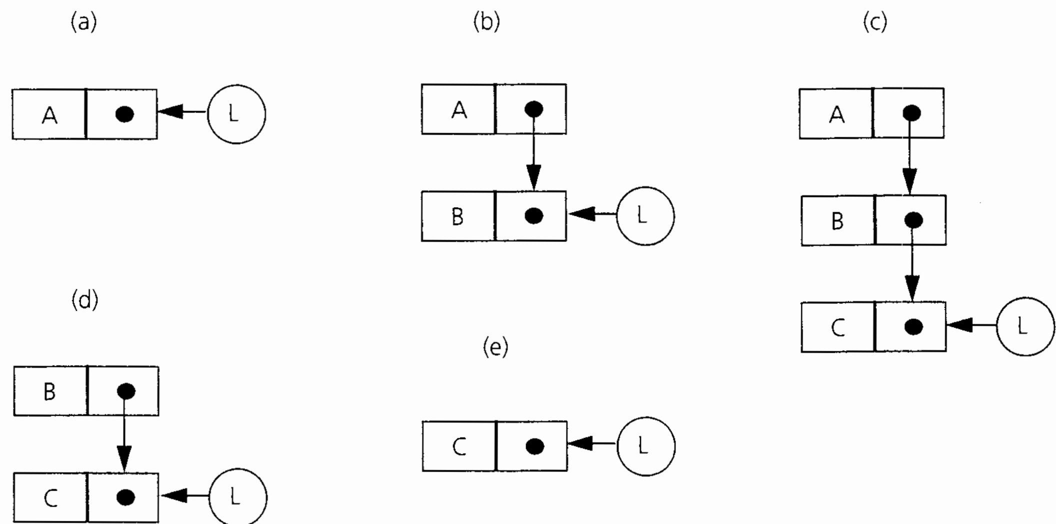


**FIGURE 7.39 States of the queue for a lock as processes try to acquire and as processes release.** The queue grows as new waiters are added to the tail. When the lock is released, the next waiter at the head is notified. Waiters always spin on local locations.

The code for the acquire and release methods is shown in Figure 7.40. In terms of primitives needed, the key is to ensure that changes to the tail pointer are atomic. In the acquire method, the acquiring process wants to change the lock pointer to point to its node. It does this using an atomic *fetch&store* operation, which takes two operands: it returns the current value of the first operand (here the current tail pointer) and then sets it to the value of the second operand, returning only when it succeeds. The order in which the atomic fetch&store operations of different processes succeed defines the order in which they acquire the lock.

In the release method, we want to atomically check if the process doing the release is the last one in the queue, and if so, set the lock pointer to NULL. We can do this using an atomic *compare&swap* operation, which takes three operands: it compares the first two (here the tail pointer and the node pointer of the releasing process), and if they are equal, it sets the first (the tail pointer) to the third operand (here NULL) and returns TRUE; if they are not equal, it does nothing and returns FALSE. The setting of the lock pointer to NULL must be atomic with the comparison since otherwise another process could slip in between and add itself to the queue, in which case setting the lock pointer to NULL would be the wrong thing to do. Recall from Chapter 5 that a compare&swap is difficult to implement as a single machine instruction since it requires three operands in a memory instruction (the functionality can, however, be implemented using load-locked and store-conditional instructions). It is possible to implement this queuing lock without a compare&swap—using only a fetch&store—but the implementation is more complicated (it allows the queue to be broken and then repairs it), and it loses the FIFO property of lock granting (Michael and Scott 1996).

```
struct node {
   struct node *next;
   int locked;
} *mynode, *prev_node;
shared struct node *Lock;

lock (Lock, mynode) {
   mynode->next = NULL;                 /*make me last on queue*/
   prev_node = fetch&store(Lock, mynode);
                                        /*Lock currently points to the previous tail of
                                        the queue; atomically set prev_node to the
                                        Lock pointer and set Lock to point to my node
                                        so I am last in the queue*/
   if (prev_node != NULL) {             /*if by the time I get on the queue I am not the
                                        only one, i.e., some other process on queue
                                        still holds the lock*/
      mynode->locked = TRUE;            /*Lock is locked by other process*/
      prev_node->next = mynode;         /*connect me to queue*/
      while (mynode->locked) {};        /*busy-wait till I am granted the lock*/
   }
}

unlock (Lock, mynode) {
   if (mynode->next == NULL) {          /*no one to release, it seems*/
      if compare&swap(Lock,mynode,NULL) /*really no one to release*/
         return;                        /*i.e., Lock points to me, then set Lock to
                                        NULL and return*/
      while (mynode->next == NULL);     /*if I get here, someone just got on the
                                        queue and made my c&s fail, so I should wait
                                        till they set my next pointer to point to
                                        them before I grant them the lock*/
   }
   mynode->next->locked = FALSE;        /*someone to release; release them*/
}
```

**FIGURE 7.40 Algorithm for the software queuing lock.** The data for the lock is a list of length equal to the number of waiters. A node requests the lock by atomically adding an item to the tail of the list and spinning on the local item until an unlock by a previous requestor provides notification.

It should be clear that the software queuing lock needs only as much space per lock as the number of processes waiting on or participating in the lock, not space proportional to the number of processes in the program. It is the lock of choice for machines that support a shared address space with distributed memory but without coherent caching (Kägi, Burger, and Goodman 1997).

## 7.9.2 Algorithms for Barriers

In both message-passing and shared address space models, global events like barriers are a key concern. A question of considerable debate is whether special hardware support is needed for global operations or whether sophisticated software algorithms upon point-to-point operations are sufficient. The CM-5 represented one end of the spectrum, with a special "control" network providing barriers, reductions, broadcasts, and other global operations over a subtree of the machine. The CRAY T3D provided hardware support for barriers also. Since it is easy to construct barriers that spin only on local variables or use only point-to-point messages, many scalable machines provide no special support for barriers at all but build them in software libraries.

In the centralized barrier used on bus-based machines, all processors used the same lock to increment the same counter when they signaled their arrival, and all waited on the same flag variable until they were released. On a large machine, the allowing for all processors to access the same lock and to read and write the same variables can lead to a lot of traffic and contention. Again, this is particularly true of machines that are not cache coherent, where the variable quickly becomes a hot spot as several processors spin on it without caching it.

It is possible to implement the arrival and departure in a more distributed way, in which not all processes have to access the same variable or lock. The coordination of arrival or release can be performed in phases or rounds with subsets of processes coordinating with one another in each round, such that after a few rounds all processes are synchronized. The coordination of different subsets can proceed in parallel with no serialization needed across them. In a bus-based machine, distributing the necessary coordination actions wouldn't matter much since the bus serializes all actions that require communication anyway; however, it can be very important in machines with distributed memory and interconnect where different subsets can coordinate in different parts of the network. The techniques used in a shared address space closely reflect natural message-passing approaches. Let us examine a few such distributed-barrier algorithms.

### Software Combining Trees

A simple distributed way to coordinate the arrival or release of processes is through a tree structure (see Figure 7.41), just as was suggested for avoiding hot spots in Chapter 3. An arrival tree is a tree that processors use to signal their arrival at a barrier. It replaces the single lock and counter of the centralized barrier by a tree of counters. The tree may be of any chosen degree or branching factor, say, $k$. In the simplest case, each leaf of the tree is a process that participates in the barrier. When a process arrives at the barrier, it signals its arrival by performing a fetch&increment on the counter associated with its parent (or by sending a message to the parent). It then checks the value returned by the fetch&increment to see if it was the last of its siblings to arrive. If not, its work for the arrival is done and it simply waits for the release. If so, it considers itself chosen to represent its siblings at the next level of the

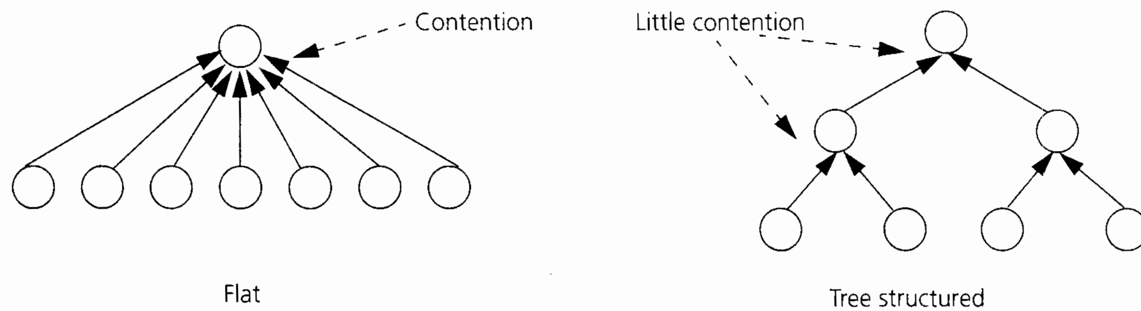


**FIGURE 7.41 Replacing a flat arrival structure for a barrier by an arrival tree (here of degree 2).** A deeper tree with smaller branching utilizes the many paths through the network of a large-scale machine to avoid serialization.

tree and so does a fetch&increment on the counter at that level. In this way, each tree node sends only a single representative process up to the next higher level in the tree when all the processes represented by that node's children have arrived. For a tree of degree $k$, it takes $\log_k p$ levels and hence that many steps to complete the arrival notification of $p$ processes. If subtrees of processes are placed in different parts of the network and if the counter variables at the tree nodes are distributed appropriately across memories, fetch&increment operations on nodes that do not have an ancestor-descendent relationship need not be serialized at all.

A similar tree structure can be used for the release as well, so all processors don't busy-wait on the same flag. That is, the last process to arrive at the barrier sets the release flag associated with the root of the tree, on which only $k - 1$ processes are busy-waiting. Each of the $k$ processes then sets a release flag at the next level of the tree, on which $k - 1$ other processes are waiting, and so on down the tree until all processes are released. (Similarly, messages can be passed down the tree.) The critical path length of the barrier in terms of the number of dependent or serialized operations (e.g., network transactions) is thus $O(\log_k p)$ as opposed to $O(p)$ for the centralized barrier or $O(p)$ for any barrier on a centralized bus. The code for a simple combining tree barrier with sense reversal is shown in Figure 7.42.

Although this tree barrier distributes traffic in the interconnect, it has the same problem as the simple lock for machines that do not cache remote shared data: the variables that processors spin on are not necessarily allocated in their local memory. Multiple processors spin on the same variable, and which processors reach the higher levels of the tree and spin on the variables there depends on the order in which processors reach the barrier and perform their fetch&increment instructions, which is impossible to predict. This leads to a lot of network traffic while spinning.

## Tree Barriers with Local Spinning

There are two ways to ensure that a processor spins on a local variable. One is to predetermine which processor moves up from a node to its parent in the tree, based on the process identifier and the number of processes participating in the barrier. In this

```
struct tree_node {
   int count = 0;                        /*counter initialized to 0*/
   int local_sense;                      /*release flag implementing sense reversal*/
   struct tree_node *parent;
}

struct tree_node tree[P];                /*each element (node) allocated in a different
                                           memory*/
private int sense = 1;
private struct tree_node *myleaf;        /*pointer to this process's leaf in the tree*/

barrier () {
   barrier_helper(myleaf);
   sense = !(sense);                     /*reverse sense for next barrier call*/
}

barrier_helper(struct tree_node *mynode) {
   if (fetch&increment (mynode->count) == k-1){      /*last to reach node*/
     if (mynode->parent != NULL)
       barrier_helper(mynode->parent);               /*go up to parent node*/
     mynode->count = 0;                              /*set up for next time*/
     mynode->local_sense = !(mynode->local_sense);   /*release*/
   endif
   while (sense != mynode->local_sense) [];          /*busy-wait*/
```

**FIGURE 7.42 A software combining barrier algorithm with sense reversal.** Each time the barrier is used, the sense of the flag is reversed, so the flag does not need to be reset.

case, a binary tree makes local spinning easy since the flag to spin on can be allocated in the local memory of the spinning processor rather than the one that goes up to the parent level. In fact, in this case, it is possible to perform the barrier without any atomic operations like fetch&increment but with only simple reads and writes as follows. For arrival, one process arriving at each node simply spins on an arrival flag associated with that node. The other process associated with that node simply writes the flag when it arrives. The process whose role was to spin now simply spins on the release flag associated with that node while the other process now proceeds up to the parent node. Such a static binary tree barrier has been called a "tournament barrier" in the literature, since one process can be thought of as dropping out of the tournament at each step in the arrival tree. (As an exercise, think about how you might modify this scheme to handle the case where the number of participating processes is not a power of two and to use a nonbinary tree.)

The other way to ensure local spinning is to use $p$-node trees to implement a barrier among $p$ processes, where each tree node (leaf or internal) is assigned to a unique process. The arrival and wake-up trees can be the same, or they can be main-

```
struct tree_node {
  struct tree_node *parent;
  int parent_sense = 0;
  int wkup_child_flags[2]; /*flags for children in wake-up tree*/
  int child_ready[4];      /*flags for children in arrival tree*/
  int child_exists[4];
}
                           /*nodes are numbered from 0 to P – 1 level-by-level starting
                           from the root*/
struct tree_node tree[P];  /*each element (node) allocated in a different memory*/
private int sense = 1, myid;
private me = tree[myid];

barrier() {
  while (me.child_ready is not all TRUE) {}; /*busy-wait*/
  set me.child_ready to me.child_exists; /*reinitialize for next barrier call*/
  if (myid !=0) {            /*set parent's child_ready flag, and wait for release*/
    tree[⌊(myid-1)/4⌋].child_ready[(myid-1) mod 4] = true;
    while (me.parent_sense != sense) {};
  }
  me.child_pointers[0] = me.child_pointers[1] = sense;
  sense = !sense;
```

**FIGURE 7.43 A combining tree barrier that spins on local variables only.** Each tree node is assigned to a unique process and allocated in the memory that is local to the process.

tained as different trees with different branching factors. Each internal node (process) in the tree maintains an array of arrival flags, with one entry per child, allocated in that node's local memory. When a process arrives at the barrier, if its tree node is not a leaf, then it first checks its arrival flag array and waits until all its children have signaled their arrival by setting the corresponding array entries. Then it sets its entry in its parent's (remote) arrival flag array and busy-waits on the release flag associated with its tree node in the wake-up tree. When the root process arrives and when all its arrival flag array entries are set, this means that all processes have arrived. The root then sets the (remote) release flags of all its children in the wake-up tree; these processes break out of their busy-wait loop and set the release flags of their children, and so on until all processes are released. The code for this barrier is shown in Figure 7.43, assuming an arrival tree of branching factor 4 and a wake-up tree of branching factor 2. In general, choosing branching factors in tree-based barriers is largely a trade-off between contention and critical path length counted in network transactions. Either of these types of barriers may work well for scalable machines without coherent caching.

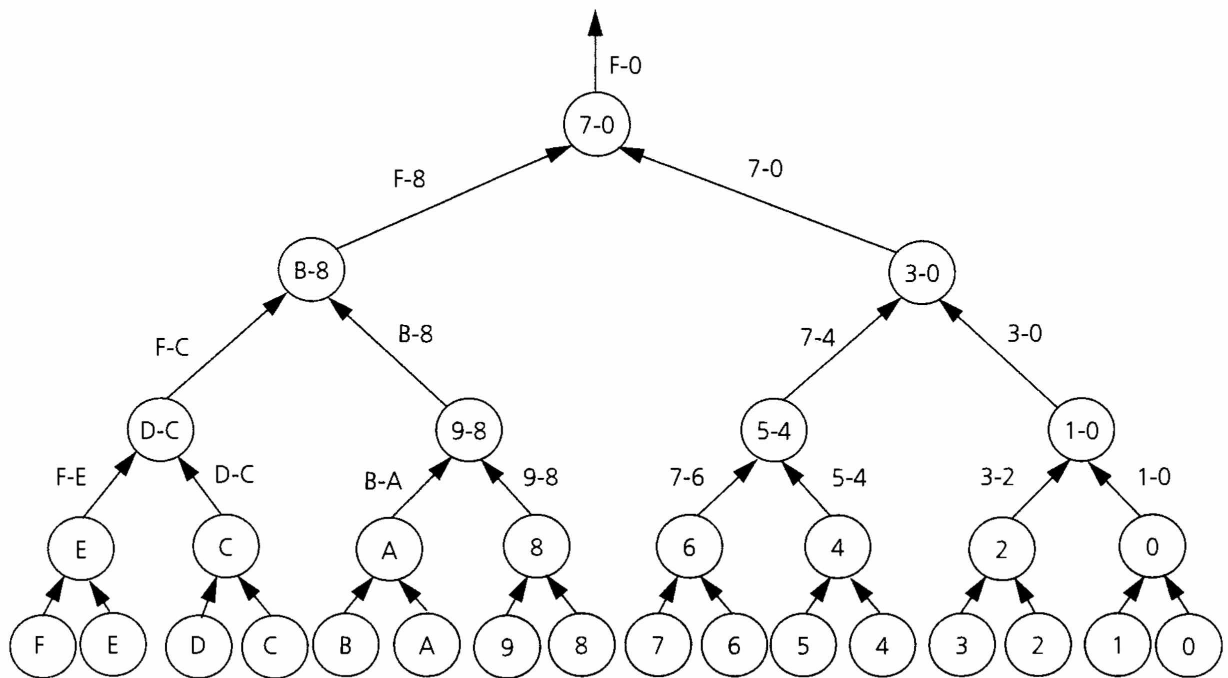


**FIGURE 7.44 Upward sweep of the parallel prefix operation.** Each node receives two elements from its children, combines them and passes the result to its parent, and holds the element from the least significant (right) child.

### Parallel Prefix

In many parallel applications, a point of global synchronization is associated with combining information that has been computed by many processors and distributing a result based on the combination. Parallel prefix operations are an important, widely applicable generalization of reductions and broadcasts (Blelloch 1993). Given some associative binary operator $\oplus$, we want to compute $S_i = x_i \oplus x_{i-1} \ldots \oplus x_0$ for $i = 0, \ldots, P$. A canonical example is a running sum, but several other operators are useful. The carry-lookahead operator from adder design is actually a special case of a parallel prefix circuit. The surprising fact about parallel prefix operations is that they can be performed as quickly as a reduction followed by a broadcast, with a simple pass up a binary tree and back down. Figure 7.44 shows the upward sweep, in which each node applies the operator to the pair of values it receives from its children and passes the result to its parent, just as with a binary reduction. (The value that is transmitted is indicated by the range of indices next to each arc; this is the subsequence over which the operator is applied to get that value.) In addition, each node holds onto the value it received from its least significant child (rightmost in the figure). Figure 7.45 shows the downward sweep. Each node waits until it receives a value from its parent. It passes this value along unchanged to its rightmost child. It combines this value with the value that was held over from the upward pass and passes the result to its left child. The nodes along the right edge of the tree are

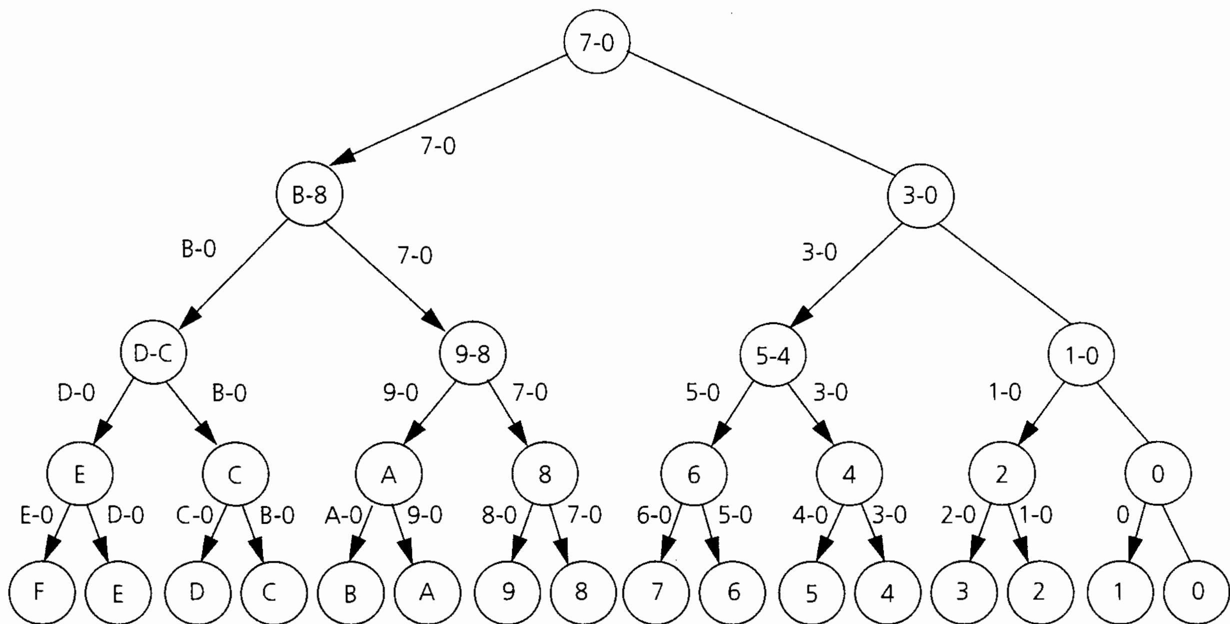


**FIGURE 7.45 Downward sweep of the parallel prefix operation.** When a node receives an element from above, it passes the data down to its right child, combines it with its stored element, and passes the result to its left child. Nodes along the rightmost branch need nothing from above.

special because they do not need to receive anything from their parent. This parallel prefix tree can be implemented either in hardware or in software.

## All-to-All Personalized Communication

All-to-all personalized communication occurs when each process has a distinct set of data to transmit to every other process. The canonical example of this is a transpose operation, say, where each process owns a set of rows of a matrix and needs to access data in a set of columns. Another important example is remapping a data structure between blocked and cyclic layouts. Many other permutations of this form are widely used in practice. Quite a bit of work has been done in implementing all-to-all personalized communication operations efficiently on specific network topologies (i.e., with no contention internal to the network). If the network is highly scalable, the internal communication flows within the network become secondary, but contention at the endpoints of the network is critical, regardless of the quality of the network. A simple, widely used scheme is to schedule the sequence of communication events so that $P$ rounds of disjoint pairwise exchanges are performed. In round $i$, process $p$ transmits the data it has for process $q = p \oplus i$ obtained as the exclusive-or of the binary number for $p$ and the binary representation of $i$. Since exclusive-or is commutative, $p = q \oplus i$, and the round is indeed an exchange.