13

Collective Communication and Computation

Counting votes after elections[1] *is an early example of massively parallel computing. In a large country, there are millions and millions of ballots cast in thousands and thousands of polling places distributed all over the country. It is clearly a bad idea to ship all the ballot boxes to the capital and have them counted by a single clerk. Assuming the clerk can count one vote per second 24 h a day, counting 100 000 000 votes would take more than three years. Rather, the votes are counted in each station and the counts are aggregated in a hierarchy of election offices (e.g., polling place, city, county, state, capital). These counts are very compact and can be communicated by telephone in a few seconds. Overall, a well-organized counting process can yield a preliminary result in a few hours. We shall see that this is an example of a global reduction and that efficient parallel algorithms follow a very similar pattern.*

Most parallel algorithms in this book follow the SPMD (single program multiple data) principle. More often than not, the high symmetry of these algorithms leads to highly regular interaction patterns involving all the PEs. This is a mismatch to the low-level primitives of our machine models such as point-to-point message exchange (Sect. 2.4.2) or concurrent access to memory locations (Sect. 2.4.1). Fortunately, these interaction patterns usually come from a small set of operations for which we can provide a library of efficient algorithms. Hence, these *collective communication operations* definitively belong to the basic toolbox of parallel algorithms and thus get their own chapter in this book. Figure 13.1 and Table 13.1 give an overview. Note that all bounds stated in this table are better than what one obtains using trivial algorithms. While the table gives asymptotic running times, in the following we shall look also at the constant factors involved in the terms depending on α and β. We shall call this the *communication time*. A bound of $a\alpha + b\beta$ will in all cases imply a running time of $a\alpha + b\beta + \mathrm{O}(a+b)$.

[1] The illustration above shows a polling station in New York circa 1900. E. Benjamin Andrews, *History of the United States*, volume V. Charles Scribner's Sons, New York (1912).

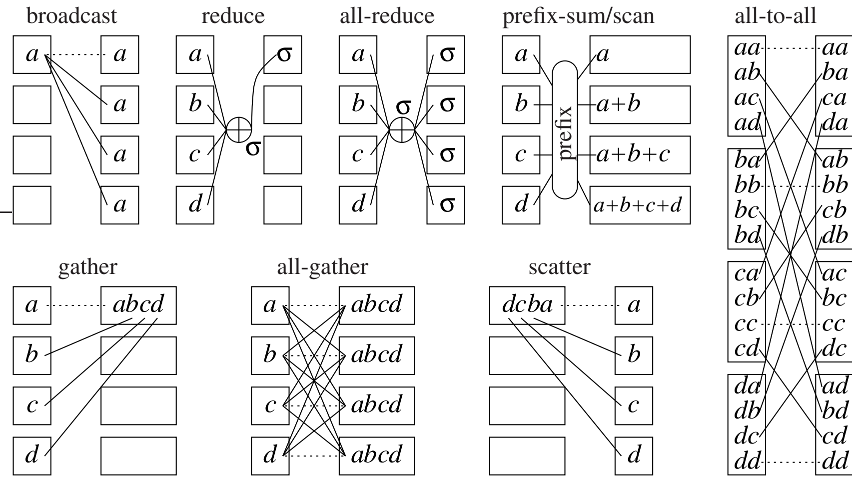


Fig. 13.1. Information flow (left to right) of collective communication operations. Note that the actual data flow might be different in order to improve performance.

Table 13.1. Collective communication operations and their asymptotic complexities; n = message size, p = #PEs, α = message startup latency, β = comm. cost per word

Name	# senders	# receivers	# messages	Computations?	Complexity	Section
Broadcast	1	p	1	no	$\alpha \log p + \beta n$	13.1
Reduce	p	1	p	yes	$\alpha \log p + \beta n$	13.2
All-reduce	p	p	p	yes	$\alpha \log p + \beta n$	13.2
Prefix sum	p	p	p	yes	$\alpha \log p + \beta n$	13.3
Barrier	p	p	0	no	$\alpha \log p$	13.4.2
Gather	p	1	p	no	$\alpha \log p + \beta pn$	13.5
All-gather	p	p	p	no	$\alpha \log p + \beta pn$	13.5
Scatter	1	p	p	no	$\alpha \log p + \beta pn$	13.5
All-to-all	p	p	p^2	no	$\log p(\alpha + \beta pn)$ or $p(\alpha + \beta n)$	13.6

We begin with a group of operations with essentially the same complexity, and some closely related algorithms: sending a message to all PEs (*broadcast*, Sect. 13.1), combining one message from each PE into a single message using an associative operator (*reduce*, Sect. 13.2), a combination of reduce and subsequent broadcast (*all-reduce*), and computation of the partial sums $\sum_{i \leq i_{\text{proc}}} x_i$, where, once more, the sum can be replaced by an arbitrary associative operator (*prefix sum* or *scan*, Sect. 13.3).

In Sect. 13.4, we discuss the *barrier* synchronization, which ensures that no PE proceeds without all the other PEs having executed the barrier operation. This operation is particularly important for shared memory, where it is used to ensure that no data is read by a consumer before being produced. On shared-memory machines,

we also need further synchronization primitives such as *locking*, which are also explained in Sect. 13.4.

The operations *gather* (concatenate p messages, one from each PE), *all-gather* (gather plus broadcast, also called *gossip* or *all-to-all broadcast*), and *scatter* (one PE sends one individual message to each PE) are much more expensive, since they scale only linearly with p and should be avoided whenever possible. Nevertheless, we sometimes need them, and Sect. 13.5 provides some interesting algorithms. Finally, in Sect. 13.6 we consider the most expensive operation, *all-to-all*, which delivers individual messages between all PEs.

We explain the collective operations first for the distributed-memory model. Then we comment on the differences for shared memory, where we have hardware support for some of the operations. For example, concurrent reading works well because the hardware internally implements a kind of broadcast algorithm. In small shared-memory systems, constant factors may also be more important than asymptotic complexity. For example, p concurrent fetch-and-add instructions may often be faster than a call to a tree-based reduction algorithm. However, this may not apply for large p and large objects to be combined.

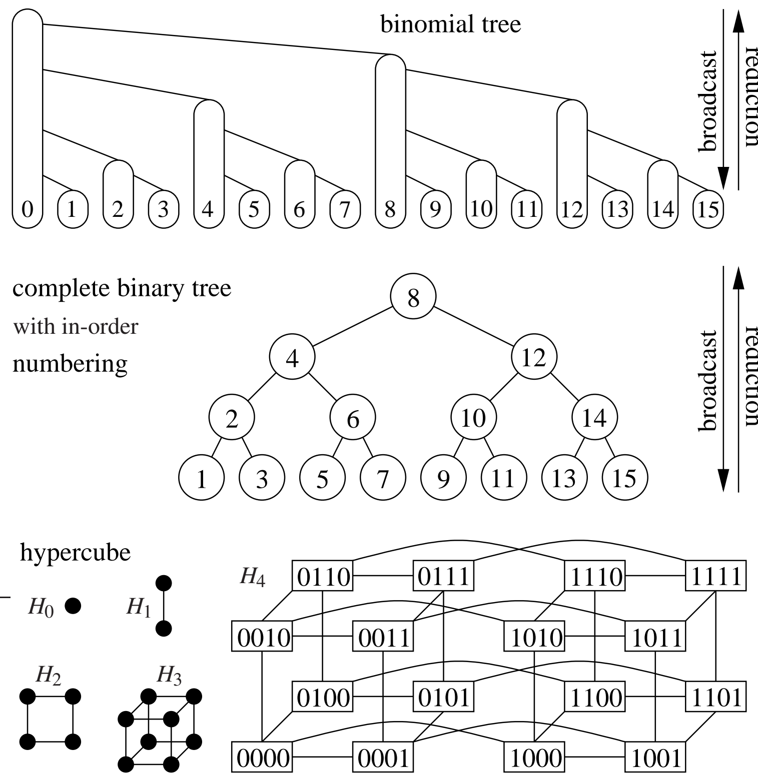


Fig. 13.2. Basic communication topologies for collective communications

All these algorithms are based on the three basic communication topologies *binomial tree*, *binary tree*, and *hypercube* shown in Fig. 13.2. The tree topologies are good for one-to-p patterns (broadcast and scatter) and p-to-one patterns (reduce and gather) and are introduced in Sect. 13.1 together with the broadcast operations. The hypercube introduced in Sect. 13.2.1 can in some cases accelerate p-to-p communication patterns and also leads to particularly simple algorithms.

13.1 Broadcast

In a broadcast, a root PE r wants to send a message m of size n to all PEs. Broadcasting is frequently needed in SPMD parallel programs to distribute parameters of the input or other globally needed values. Shared-memory machines with coherent caches support broadcasting by hardware to some extent. When one PE accesses a value, its cache line is copied to all levels of the cache hierarchy. Subsequently, other PEs accessing the same caches do not have to go down all the way to main memory. When all PEs access the same value, this hardware mechanism – with some luck[2] – will build a tree of caches and PEs in a similar fashion to the distributed-memory algorithms we shall see below. For example, the value may be copied from main memory to the L3 cache of all processor chips. From each L3 cache, it is then copied to the L2 and L1 caches of each core, and the hardware threads running on the same core need only to access their local L1 cache.

Going back to distributed memory, let us assume for now that $r = 0$ and that the PEs are numbered $0..p-1$. Other arrangements are easy to obtain by renumbering the PEs, for example as $i_{\text{proc}} - r \bmod p$. A naive implementation uses a for-loop and $p-1$ send operations. This strategy is purely sequential and needs time at least $\alpha(p-1)$; it could be a bottleneck in an otherwise parallel program. A better strategy is to use a divide-and-conquer strategy. PE 0 sends x to PE $\lceil p/2 \rceil$ and delegates to it the task of broadcasting x to PEs $\lceil p/2 \rceil ..p-1$ while PE 0 itself continues broadcasting to PEs $0..\lceil p/2 \rceil - 1$. More generally, a PE responsible for broadcasting to PEs $i..j$ delegates to PE $\lceil (i+j)/2 \rceil$.

13.1.1 Binomial Trees

Suppose now that p is also a power of two. Then the resulting communication topology is a *binomial tree*. We have already seen binomial trees in the context of addressable priority queues (see Fig. 6.5 in Sect. 6.2.2). Fig. 13.2 draws them differently to visualize the communication algorithm. Edges starting further up are used earlier in the broadcast algorithm. Interestingly, the binary representation of the PE numbers tells us the role of a node in the tree: A PE number i with k trailing 0's indicates that this PE has k children with numbers $i+1$, $i+2$, ..., $i+2^{k-1}$ and that PE $i-2^k$ is its parent. This is so convenient that we also adopt it when p is not a power of two. We

[2] In the worst case, all PEs may try to read the value exactly at the same time and will then produce a lot of contention in trying to access the main memory.

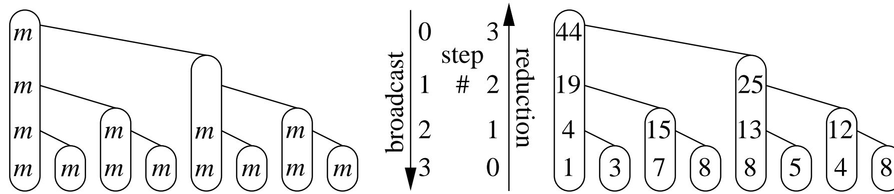


Fig. 13.3. Data flow for binomial tree broadcast (left) and reduction (right) with $p = 8$. The values at step i indicate the state of the computation at that step.

simply round up to the next power of two, use the binomial tree for that power of two and then drop the nodes with number $\geq p$. The resulting graph is still a tree, and it is easy to generalize the communication algorithms. Indeed, the following pseudocode works for any tree spanning all PEs:

```
Procedure Tree::broadcast(m)
    receive(parent, m)                          // does nothing on root
    foreach c ∈ children do send(c, m)          // largest first for binomial tree
```

Figure 13.3 (left) shows the data flow for $p = 8$. It is very important that the messages to the children are sent in decreasing order of their size. Is tree broadcast a good algorithm? We first show an upper bound for binomial trees.

Theorem 13.1. *The broadcast algorithm for binomial trees needs communication time*

$$\lceil \log p \rceil (\alpha + \beta n).$$

Proof. We use induction on p. The base case $p = 1$ is easy, PE 0 is the only node. The receive operation does nothing and the set of children is empty. Hence, no communication is needed.

The induction step assumes that the claim is true for all $p \leq 2^k$ and shows that this implies that the claim also holds for all $p \leq 2^{k+1}$. For $2^k < p \leq 2^{k+1}$, the first send operation sends m to PE 2^k in time $\alpha + \beta n$. From then on, PEs 0 and 2^k execute the broadcast algorithm for a subtree of size $\leq 2^k$. Hence, by the induction hypothesis, they need time $k(\alpha + \beta n)$. Overall, the time is $(k+1)(\alpha + \beta n) = \lceil \log p \rceil (\alpha + \beta n)$. □

This bound is in some sense the best possible: Since one communication operation can inform only a single PE about the content of message m, the number of PEs knowing anything about m can at most double with each communication. Hence, $\alpha \log p$ is a lower bound on the broadcasting time.

Exercise 13.1. Show that the running time of binomial tree broadcast becomes $\Omega(\alpha \log^2 p)$ if the order of the send operations is reversed, i.e., small children are served first.

Another obvious lower bound for broadcasting is βn – every nonroot PE must receive the entire message. However, binomial tree broadcast needs $\log p$ times more time. In fact, for long messages, there are better algorithms, approaching βn for large n.

13.1.2 Pipelining

There are two reasons why binomial trees are not good for broadcasting long messages. First, PE 0 sends the entire message to $\lceil \log p \rceil$ other PEs, so that it becomes a bottleneck. Hence, a faster algorithm should send only to a bounded number of other PEs. This problem is easy to fix – we switch to another tree topology where every PE has small outdegree. Outdegrees of one and two seem most interesting. For outdegree 1, the topology degenerates to a path, leading to communication time $\Omega(p\alpha)$. This is not attractive for large p. Hence, outdegree two – a binary tree – seems like a good choice. This is the smallest outdegree for which we can achieve logarithmic height of the tree.

Another problem is that any broadcasting algorithm which sends m as a whole will need time $\Omega(\log p(\alpha + \beta n))$. Hence, we should not immediately transfer the entire message. Rather, we should chop the message into smaller *packets*, which are sent independently. This way, broadcasting can spread its activity to all PEs much faster. Reinterpreting message m as an array of k packets of size $\lceil n/k \rceil$, we get the following algorithm that once more works for any tree topology:

Procedure *Tree*::*pipelinedBroadcast*(m : *Array*[1..k] **of** *Packet*)
  for $i := 1$ **to** k **do**
    receive(*parent*, $m[i]$)
    foreach $c \in$ *children* **do** *send*(c, $m[i]$)

Figure 13.4 gives an example.

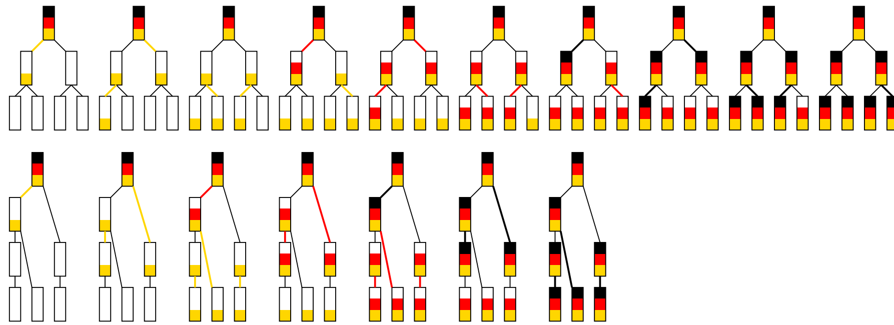


Fig. 13.4. Pipelined binary tree broadcast with $p = 7$ and $k = 3$. Top: 10 steps using a complete binary tree and half-duplex communication. Bottom: 7 steps using a skewed tree and full-duplex communication.

Lemma 13.2. *On a binary tree, the algorithm Tree*::*pipelinedBroadcast needs communication time*

$$T(L, n, k) \leq (2L + 3(k-1)) \left(\alpha + \beta \left\lceil \frac{n}{k} \right\rceil \right), \tag{13.1}$$

where L is the length of the longest root to leaf path in the tree.

Proof. Sending or receiving one packet takes time $\alpha+\beta\lceil n/k\rceil$. It remains to show that after $2L+3(k-1)$ such communication steps, every PE has received the entire message. A PE which has successfully received the first packet needs two steps to forward it to both of its neighbors. Hence, after $2L$ steps, every PE has received the first packet. An interior node of the tree can process one packet every three steps (receive, send left, send right). Hence, after $3(k-1)$ further steps, the last packet has arrived at the last PE. □

It remains to determine k and L. If we use a perfectly balanced binary tree, we get $L=\lfloor\log p\rfloor$. Ignoring rounding issues, we can find an optimal value for k using calculus. We get $k=\sqrt{\beta n(2L-3)/3\alpha}$ if this value is ≥ 2. This yields the following bound.

Theorem 13.3. *Using pipelined broadcast on a perfectly balanced binary tree, we can obtain communication time*

$$T_2(k)=3\beta n+2\alpha\log p+\mathrm{O}\left(\sqrt{\alpha\beta n\log p}\right)=\mathrm{O}(\beta n+\alpha\log p).$$

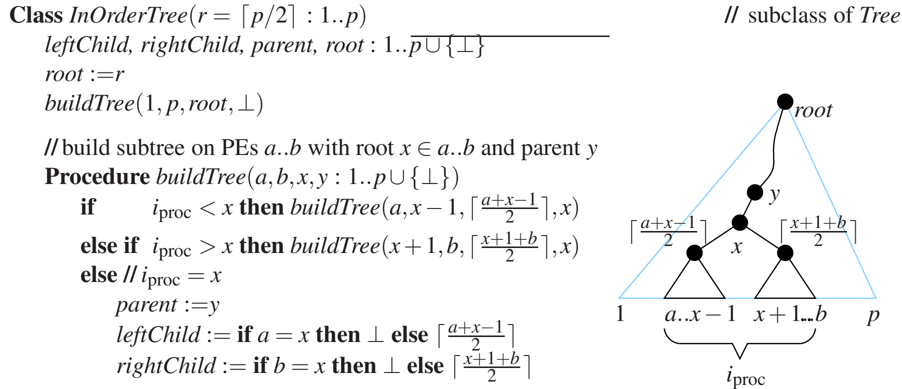
Exercise 13.2. Prove Theorem 13.3.

13.1.3 Building Binary Trees

We have not yet explained how to efficiently build the topology for a perfectly balanced binary tree – each node (PE) needs to know the identity of its parent, its children, and the root. We describe a construction principle that is simple, is easy to compute, can also be used for reduction and prefix sums, and uses subtrees with contiguous numbers. The latter property might be useful for networks that exhibit a hierarchy reflected by the PE numbering. Figure 13.2 gives an example. Figure 13.5 gives pseudocode for a tree class defining a perfectly balanced binary tree whose nodes are numbered *in-order*, i.e., for any node v, the nodes in the subtree rooted at the left child of v have a smaller number than v and the nodes in the subtree rooted at the right child of v have a larger number than v. The constructor takes a user-defined root and calls a recursive procedure *buildTree* for defining a binary tree on PE numbers $a..b$. This procedure adopts the root x and parent y passed by the caller and calls itself to define the subtrees for PEs $a..x-1$ and $x+1..b$. The roots of the subtrees are placed in the middle of these subranges. Since PE i_{proc} needs only the local values for the parent, left child, and right child, only that branch of the recursion needs to be executed which contains i_{proc}. Thus, we get execution time $\mathrm{O}(\log p)$ without any communication.

Exercise 13.3. Prove formally that the tree constructed in Fig. 13.5 has height $\leq\lceil\log p\rceil$ regardless of the root chosen.

Exercise 13.4. Design an algorithm that runs in time $\mathrm{O}(\log p)$ and defines a balanced binary tree where the nodes are numbered layer by layer as in Sect. 6.1.



```
Class InOrderTree(r = ⌈p/2⌉ : 1..p)                    // subclass of Tree
    leftChild, rightChild, parent, root : 1..p ∪ {⊥}
    root := r
    buildTree(1, p, root, ⊥)

    // build subtree on PEs a..b with root x ∈ a..b and parent y
    Procedure buildTree(a, b, x, y : 1..p ∪ {⊥})
        if       i_proc < x then buildTree(a, x − 1, ⌈(a+x−1)/2⌉, x)
        else if  i_proc > x then buildTree(x + 1, b, ⌈(x+1+b)/2⌉, x)
        else // i_proc = x
            parent := y
            leftChild := if a = x then ⊥ else ⌈(a+x−1)/2⌉
            rightChild := if b = x then ⊥ else ⌈(x+1+b)/2⌉
```


Fig. 13.5. Constructor of an in-order binary tree

13.1.4 *Faster Broadcasting

With binomial tree broadcasting we have a very simple algorithm that is optimal for small messages. Pipelined binary tree broadcasting is simple and asymptotically optimal for long messages also. However, it has several weaknesses that can be improved on.

First, we are not exploiting full-duplex communication. For *very* long messages, we can exploit full-duplex communication by using a *unary tree* – a path consisting of p nodes. The following *linear pipeline* algorithm exploits full-duplex communication. While receiving packet i, it forwards the previous packet $i-1$ down the line:

```
Procedure Path::pipelinedBroadcast(m : Array[1..k] of Packet)
    receive(parent, m[1])
    for i := 2 to k do  receive(parent, m[i]) ‖ send(child, m[i − 1])
    send(child, m[k])
```

The algorithm terminates in $p+k-1$ steps. Optimizing for k as in Sect. 13.1.2 yields running time

$$T_1(k) = \beta n + p\alpha + \mathrm{O}\left(\sqrt{\alpha\beta n p}\right).$$

This is optimal for fixed p and $n \to \infty$.

The linear pipeline is much worse than pipelined binary tree broadcasting for large p unless n is extremely large. We can also use bidirectional communication for binary tree broadcasting, as in the following exercise.

Exercise 13.5. Adapt pipelined binary tree broadcasting so that it exploits full-duplex communication for communicating with one of its children and runs in time

$$T_2^* = 2\beta n + 2\alpha \log p + \mathrm{O}\left(\sqrt{\alpha\beta n \log p}\right).$$

Hint: Use the same trick as for the linear pipeline. Figure 13.4 gives an example.

For large n, this is still a factor of two away from optimality. We seem to have the choice between two evils – a linear pipeline that does not scale with p and a binary tree that is slow for large n. This seems to be unavoidable for any tree that is not a path – the interior nodes have to do three communications for every packet and are thus overloaded, whereas the leaves do only one communication and are thus underloaded. Another trick solves the problem. By carefully scheduling communications, we can run *several* tree-based broadcast algorithms simultaneously. One approach uses $\log p$ spanning binomial trees embedded into a hypercube [170]. Another one uses just two binary trees [277]. The principle behind the latter algorithm is simple. By ensuring that an interior node in one tree is a leaf in the other tree, we cancel out the imbalance inherent in tree-based broadcasting. This algorithm is also easy to adapt to arbitrary values of p and to prefix sums and noncommutative reduction.

Another weakness of binary tree broadcasting can be seen in the example in Fig. 13.4. Some leaves receive the packets later than others. It turns out that one can reduce the latency of broadcasting by building the trees in such a way that all leaves receive the first message at the same time. This implies that, towards the right, the tree becomes less and less deep. Interestingly, one can obtain such a tree by modifying the procedure *buildTree* in Fig. 13.5 to place the root of a subtree not in the middle but according to the *golden ratio* $1 : (1+\sqrt{5})/2$. Fig. 13.4 gives an example where all leaves receive a packet at the same time. When an interconnection network does not support arbitrary concurrent communications, one can adapt the tree structure so that not too many concurrent communications use the same wire. For example, Fig. 13.6 suggests how to embed a spanning binary tree of depth $\log p + \mathrm{O}(1)$ into two-dimensional square meshes such that at most two tree edges are embedded into connections between neighboring PEs.

****Exercise 13.6.** (Research problem) Is there an embedding which uses any edge of the mesh for only a single tree edge?

****Exercise 13.7.** Construct a binary tree of depth $\mathrm{O}(\log p)$ such that only a constant number of tree edges are embedded into connections between neighboring PEs of a $p = k \times 2^k$ mesh. Hint: Use horizontal connections spanning 2^i PEs in layer i.

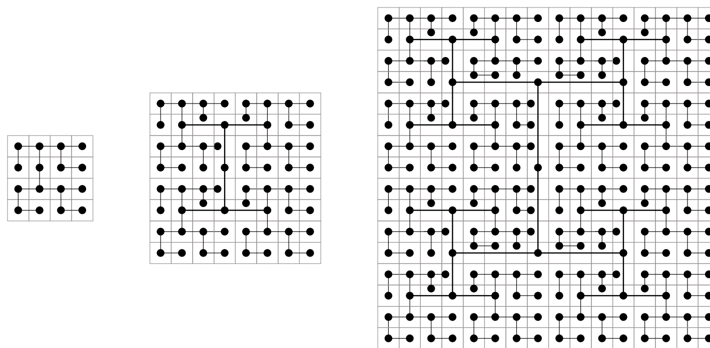


Fig. 13.6. *H-tree* embedding of a binary trees into 2D meshes.

13.2 Reduction

Reduction is important in many parallel programs because it is a scalable way to aggregate partial results computed on different PEs into a global result.

Given a message m_i on each PE i, a reduction computes $\bigotimes_{i<p} m_i$, where $\otimes$ can be any associative operator, i.e., $\forall a,b,c : (a \otimes b) \otimes c = a \otimes (b \otimes c)$. Some algorithms also require $\otimes$ to be commutative, and sometimes it is convenient if we have a neutral element (e.g., 0 for $\otimes = +$). In practice, the most frequent operators are $+$, min, and max, which are all commutative and have a neutral element.

Exercise 13.8. Explain how to compute both the minimum and the location of the minimum using a reduction. Is the resulting operation commutative?

Exercise 13.9. Show that floating-point addition is, strictly speaking, *not* associative. What are the consequences of treating it like an associative operation?

When long messages are involved in a reduction, we shall assume that they represent vectors of smaller objects and thus the reduction can be computed componentwise. This will be important in order to use pipelining techniques.

Shared-memory machines have limited support for reduction in the form of fetch-and-add instructions. However, when all processors are adding to a global value at the same time, software implementations along the lines of what is presented below may still be faster.

Mathematically speaking, associativity is the key to reducing in parallel. When we write $\bigoplus_{i<p} m_i$, this is conventionally interpreted as

$$((\cdots((m_0 \otimes m_1) \otimes m_2) \cdots) \otimes m_{p-2}) \otimes m_{p-1}$$

which looks inherently sequential. However, associativity allows us to rewrite the sum as any binary tree whose interior nodes are labeled with $\otimes$ and whose ith leaf is m_i.

Parallel reduction algorithms thus use trees much like the broadcasting algorithms we have seen in Sect. 13.1. Indeed, that similarity goes much further. The algorithms we propose here are basically obtained by running a broadcasting algorithm "backwards". The leaves of the tree send their values to their parents, which add them up before forwarding the partial sum to their parents. This works even for noncommutative operators if the nodes of the tree are numbered in-order.

For binomial trees, we have already seen the resulting code in Sect. 2.4.2. Figure 13.3 gives an example and illustrates the symmetry with respect to broadcasting. Note that compared to broadcasting, the "running backwards" rule means that we also reverse the order in which we receive from the children – this time from left to right. This is important to achieve the same communication time as for the broadcasting algorithm.

For binary trees, we get the following pseudocode (for the nonpipelined case). To simplify some special cases, we use a neutral element $\mathbf{0}$:

Procedure *InOrderTree*::*reduce*(m)
$x := \mathbf{0}$; *receive*$(leftChild, x)$
$z := \mathbf{0}$; *receive*$(rightChild, z)$
send$(parent, x \otimes m \otimes z)$ // *root* returns result

We leave formulating pseudocode for pipelined tree reduction as an exercise but prove its complexity here, which is the same as for pipelined broadcasting (Lemma 13.2).

Lemma 13.4. *Pipelined binary tree reduction needs communication time*

$$T^{\otimes}(L,n,k) \leq (2L+3(k-1))\left(\alpha+\beta\left\lceil\frac{n}{k}\right\rceil\right), \tag{13.2}$$

where k is the number of packets and L is the length of the longest root-leaf path in the tree.

Proof. We have to show that $2L+3(k-1)$ packet communication steps are needed. Reduction activity propagates bottom up, one level every two steps. Hence, the root performs its first addition after $2L$ steps. An interior node of the graph (e.g., the right child of the root) can process one packet every three steps (receive left, receive right, send sum). Hence, after $3(k-1)$ further steps, the last packet has arrived at the root. □

Since the execution time is the same as for broadcasting, we can also optimize k in the same way and obtain the following corollary.

Corollary 13.5. *Using pipelined reduction on a perfectly balanced binary tree, we can obtain communication time*

$$T_2^{\otimes}(k) = 3\beta n + 2\alpha\log p + \mathrm{O}\left(\sqrt{\alpha\beta n\log p}\right) = \mathrm{O}(\beta n+\alpha\log p).$$

All the optimizations in Sect. 13.1.4 also transfer to reduction, except that using $\log p$ spanning trees [170] does not work for noncommutative reduction – it seems impossible to have in-order numberings of all the trees involved. However, this works for the two trees used in [277].

13.2.1 All-Reduce and Hypercubes

Often, *all* PEs (rather than only one root PE) need to know the result of a reduction. This *all-reduce* operation can be implemented by a reduction followed by a broadcast. For long messages and for shared-memory machines (where broadcast is supported by the hardware) this is also a good strategy. However, for short messages we can do better reducing the latency from $2\alpha\log p$ to $\alpha\log p$. We show this for the case where $p(=2^d)$ is a power of two.

In this case, all-to-all communication patterns can often be implemented using a *hypercube*: A d-dimensional hypercube is a graph $H_d = (0..2^d-1, E)$ where $(x,y) \in E$ if and only if the binary representations of x and y differ in exactly one bit. Edges

corresponding to changing bit i are the edges along *dimension* i. Figure 13.2 shows the hypercubes for $d \in 0..4$. The hypercube communication pattern considers the PEs as nodes of a hypercube. It iterates through the dimensions, and in iteration i communicates along edges of dimension i.

For the all-reduce problem, we get the following simple code:

for $i := 0$ **to** $d-1$ **do**
  $send(i_{\text{proc}} \oplus 2^i, m) \parallel receive(i_{\text{proc}} \oplus 2^i, m')$ // Or, for short:
  $m := m \otimes m'$ // $m\otimes = m@(i_{\text{proc}} \oplus 2^i)$

Understanding how and why a hypercube algorithm works often involves loop invariants dealing with i-dimensional *subcubes*. All the 2^i nodes sharing the most significant bits $i..d-1$ in their number form an i-dimensional hypercube. For all-reduction, the loop invariant says that after iteration i, all $i+1$-dimensional subcubes have computed an all-reduce within that subcube. Figure 13.7 gives an example.

13.3 Prefix Sums

A *prefix sum* or *scan* operation is a generalization of a reduction where we are interested not only in the overall sum but also in all the prefixes of the sum. More precisely, PE i wants to compute $\otimes_{i \le i_{\text{proc}}} m_i$. A variant of this definition computes the *exclusive prefix sum* $\otimes_{i < i_{\text{proc}}} m_i$ on PE i.

Prefix sums with $\otimes = +$ are frequently used for distributing work between PEs; see also Sect. 14.2. We have seen an example for a different operator in Sect. 1.1.1, where it was used for computing carry lookaheads in parallel addition. Prefix sums are equally important for shared-memory and distributed-memory algorithms, and there is no hardware support for them on shared-memory machines.

Here is a very simple hypercube algorithm for computing prefix sums which just adds two lines to the all-reduce algorithm presented in Sect. 13.2.1.

$x := m$; **invariant** *x is the prefix sum in current subcube*
for $i := 0$ **to** $d-1$ **do**
  $send(i_{\text{proc}} \oplus 2^i, m) \parallel receive(i_{\text{proc}} \oplus 2^i, m')$
  $m := m \otimes m'$ // update overall sum in subcube
  if i_{proc} **bitand** 2^i **then** $x := m' \otimes x$ // update prefix sum in subcube

Figure 13.7 gives an example. Note that this algorithm computes *both* the prefix sum *and* the overall sum on each PE. This is handy, since in many applications we actually need both values, for example for distributed-memory quicksort (Sect. 5.7.1). We obtain the following result:

Theorem 13.6. *The hypercube algorithm computes a prefix sum plus all-reduce in communication time*

$$\log p(\alpha + n\beta).$$

Hypercube algorithms only work when p is a power of two. Also, for long messages, it is problematic that all PEs are active in every step and hence we cannot use

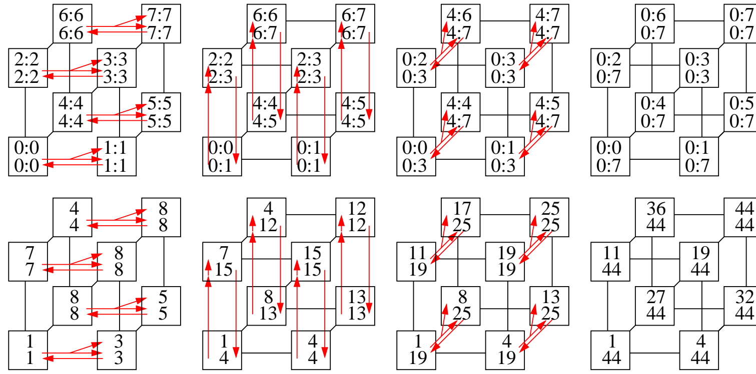


Fig. 13.7. A hypercube prefix sum (top values)/all-reduce (bottom values) for $p = 8$. The *top* row shows the general computation pattern, with $x : y$ as an abbreviation for $\bigoplus_{i=x}^{y} m_i$. The *bottom* row shows a concrete numeric example.

pipelining. In order to obtain a fast algorithm for arbitrary p and for long messages, we have therefore developed an algorithm based on in-order binary trees [197]. Figure 13.8 gives such an algorithm. For simplicity, we use neutral elements and do not show the code for pipelining. The algorithm consists of two phases. First, there is an upward phase, which works in the same way as reduction. Then comes a downward phase, which resembles a broadcast but computes the prefix sum and sends different data to the left and the right child. We exploit the fact that the PEs are numbered in-order, i.e., each PE i_{proc} is the root of a subtree containing all the PEs in a range $a..b$ of integers. During the upward phase, PE i_{proc} receives the sum x of the elements $a..i_{\text{proc}} - 1$ from its left subtree. The downward phase is implemented in such a way that PE i_{proc} receives the sum ℓ of the elements on PEs $0..a-1$. Hence, PE i_{proc} can compute its local prefix sum as $\ell + x + m$. The left subtree is numbered $a..i_{\text{proc}} - 1$, so that i_{proc} forwards ℓ there. The right subtree is numbered $i_{\text{proc}} + 1..b$, so that i_{proc} sends the local prefix sum $\ell + x + m$. Figure 13.9 gives an example.

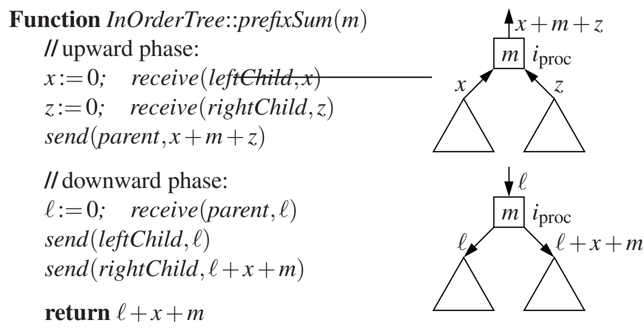


Fig. 13.8. Prefix sum using an *InOrderTree*

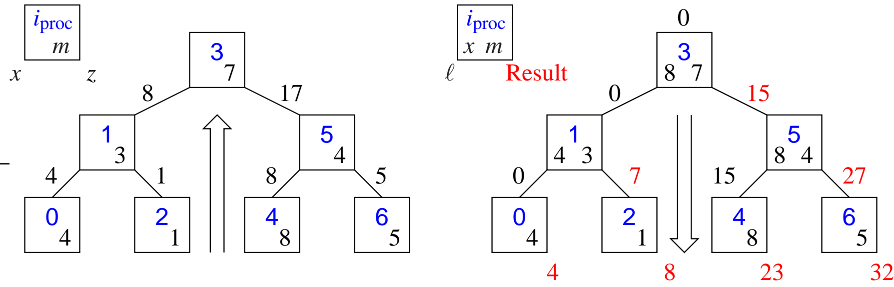


Fig. 13.9. Tree-based prefix sum computation for $p = 7$ over the sequence $\langle 4,3,1,7,8,4,5\rangle$. See also Fig. 13.8.

Since the communication operations are exactly the same as in a reduction followed by a broadcast, we can adopt the strategy for pipelining and the analysis presented in the previous sections and get obtain the following corollary.

Corollary 13.7. *Using pipelined prefix sums on binary trees, we can obtain communication time*

$$T_2(k) = 6\beta n + 4\alpha \log p + \mathrm{O}\left(\sqrt{\alpha\beta n \log p}\right) = \mathrm{O}(\alpha n + \beta \log p).$$

The optimizations in Sect. 13.1.4 also transfer to prefix sums except that we cannot use $\log p$ spanning trees simultaneously [170]. We need the same in-order numbering of all trees involved in order to compute prefix sums, even for commutative operations. Using the result for two compatibly numbered trees [277], we obtain execution time $2\beta n + 4\alpha \log p + \mathrm{O}\left(\sqrt{\alpha\beta n \log p}\right)$.

13.4 Synchronization

In a synchronization event, different threads inform each other that they have reached a certain location in their code. Thus synchronization is communication even if no additional information is exchanged. We have already introduced shared-memory locks in Chap. 2. In Sect. 13.4.1, we shall fill in some details. We discuss global synchronization of all PEs in Sect. 13.4.2.

13.4.1 Locks

In Sect. 2.5, we learned about binary locks. A lock is represented by a memory cell $S[i]$ which takes on values 0 and 1. A thread acquires this lock by changing the value from 0 to 1 and releases the lock by changing it back to 0. In order to make sure that only one thread can acquire the lock, the variable is set using a CAS instruction:

repeat until $CAS(i, desired{:=}0, 1)$

This kind of lock is called a *spin lock* because the thread trying to acquire it simply waits in a loop (spin) while repeatedly checking whether the lock is available. If many threads contend for the same lock, a basic spin lock may become inefficient, and a more careful implementation is called for:

Procedure $lock(i : \mathbb{N})$ // acquire lock in memory cell $S[i]$
  $desired = 0 : \mathbb{N}$
  loop
    if $S[i] = 1$ **then** *backoff* // wait a bit
    else if $CAS(i, desired, 1)$ **then return**

We have made two improvements here. First, we attempt the expensive CAS instruction only when an ordinary memory read tells us that we "might" be successful. Second, we do not spin in a tight loop gobbling up a lot of hardware resources but instead call a procedure *backoff* that uses various measures to save resources. First of all, *backoff* should call the machine instruction `pause`, which tells the processor to yield resources to another thread on the same core. In addition, *backoff* may explicitly wait before even attempting to read $S[i]$ again in order to avoid situations where many threads wait for $S[i] = 0$ and each of them executes an expensive CAS when $S[i]$ becomes 0. One such adaptive strategy is *exponential backoff* – the backoff period is multiplied by a constant in every loop iteration until it hits a maximum in $\Theta(p)$. Thus, even if all PEs do little else but contend for the same lock, we can achieve a constant success probability for the CAS instruction. After a successful lock, the backoff period is reduced by the same factor. Waiting can be done with a for-loop that does nothing other than execute the `pause` instruction. Unlocking a spin lock is very simple – simply set $S[i] := 0$. No CAS instruction is necessary.[3]

Distinguishing readers and writers is slightly more complicated but can also be implemented with the spin-locking idea. Now the lock variable $S[i]$ is set to $p+1$ if a writer has exclusive access, where p is an upper bound on the number of threads. Otherwise, $S[i]$ gives the number of readers. Locking for writing has to wait for $S[i] = 0$. Locking for reading has to wait for $S[i] \leq p$ and it then increments $S[i]$. Unlocking from reading means decrementing $S[i]$ atomically.

Sometimes locks are held for a very long time. For example, a thread holding a lock may be waiting for I/O or a user interaction. Then a more complicated kind of lock makes sense, where the runtime system, in cooperation with the operating system, suspends threads that are waiting for a lock. An *unlock* then has to activate the first thread in the queue of waiting threads.

[3] This is true at least on the x86 architecture. Other architectures may need additional memory fence operations: see also Sect. B.3.

13.4.2 Barrier Synchronization

When a PE in an SPMD program calls the procedure *barrier*, it waits until all other PEs have called this routine. A barrier is thus a global synchronization of all PEs. This is needed when a parallel computation can only proceed if all PEs have finished a computation, for example updating a shared data structure.

The behavior of a barrier is entailed by calling an all-reduce with an empty (or dummy) operand.

Corollary 13.8. *A barrier needs communication time* $\mathrm{O}(\alpha \log p)$.

On distributed-memory machines, there is little else to say. On shared-memory machines, barriers are so important that it is worth looking at some implementation details affecting the constant factors involved. Figure 13.10 gives pseudocode. Here, we adopt the basic idea to implement an all-reduce and choose a combination of a binomial tree reduce and a (hardware supported) broadcast. The idea is to replace a send operation by a write operation to a single memory cell (*readyEpoch*) that no other PE ever writes to. A receive operation becomes waiting for this memory cell to change. If there were only a single call to the procedure *barrier*, it would suffice to use a simple flag variable. However, in general, we would need an additional mechanism to reset these flags, which costs additional time. We circumvent this complication by using counters rather than flags.

$epoch = 0 : \mathbb{N}$ // how often was *barrier* called locally?
$readyEpoch = 0 : \mathbb{N}$ // how often did my subtree finish a barrier?
Procedure *barrier*
  $epoch$++
  for $i := 0$ **to** $|childrenInBinomialTree|$ **do** // that number should be precomputed
    while $readyEpoch@(i_{\text{proc}} + 2^i) \neq epoch$ **do** *backoff*
  $readyEpoch := epoch$
  while $readyEpoch@0 \neq epoch$ **do** *backoff* // implicit broadcast

Fig. 13.10. SPMD code for a shared-memory barrier.

13.4.3 Barrier Implementation

In this section we provide a C++ implementation of the binomial tree barrier and benchmark it against the simple folklore barrier shown in Listing 13.1. The latter implementation uses a PE counter c and an *epoch* counter that are aligned with cache-line boundaries (64 bytes on x86) to avoid false sharing. The barrier wait method accepts the ID of PE *iPE* and the total number of PEs p. In this implementation, only p is used but for the sake of generality we stick with this interface to be compatible with other barrier implementations. Each PE atomically increments the counter c with the last PE (that passes the barrier) incrementing the *epoch* and resetting the PE

counter *c* (lines 6–8). All other PEs wait for this epoch transition in line 10. Note that the *epoch* variable is declared with the `volatile` storage class (see Sect. B.3) to prevent caching it into a local stack variable or a register through the compiler. The barrier semantics (proceed if all PEs have finished a computation) also requires a CPU memory fence because otherwise the processor is, in general, allowed to reorder computations including loads/stores before and after the barrier, a contradiction to the semantics. For this purpose we have inserted a CPU memory fence (line 12). On x86 architectures, this fence is not required, because the code flow includes an atomic operation on counter *c* (line 6), which is an implicit CPU memory fence on x86.

Listing 13.1. A simple barrier in C++

```
class SimpleBarrier {                                                    1
public:                                                                  2
  SimpleBarrier(): c(0), epoch(0) {}                                     3
  void wait(const int /* iPE */, const int p) {                          4
    register const int startEpoch = epoch;                               5
    if(c.fetch_add(1) == p-1) {                                          6
      c = 0;                                                             7
      ++epoch;                                                           8
    } else {                                                             9
      while(epoch == startEpoch) backoff();                              10
    }                                                                    11
    atomic_thread_fence(memory_order_seq_cst); // not required on x86    12
  }                                                                      13
protected:                                                               14
  atomic<int> c __attribute__((aligned(64)));                            15
  char pad[64 - sizeof(atomic<int>)];                                    16
  volatile int epoch __attribute__((aligned(64)));                       17
};//SPDX-License-Identifier: BSD-3-Clause; Copyright(c) 2018 Intel Corporation   18
```

The implementation of the binomial tree barrier is shown in Listing 13.2. It stores the PE-local *epoch* and the number of children *numChildren* in the binomial tree in the array *peData*. The number of children is precomputed in the recursive function *initNumChildren* (lines 8–17) called in the constructor. The items in the arrays *peData* and *readyEpoch* are padded to avoid false sharing (lines 6, 36, and 37). The *readyEpoch* array is initialized in line 21. Note that the actual value is accessed by the *first* member of the *paddedInt* C++ pair class.

The barrier *wait* function increments the local PE's *epoch* and caches it together with *numChildren* into registers in lines 24 and 25. In the following `for` loop the PE waits until all its children (if any) have reached *myEpoch*. The reference to $readyEpoch@(iPE+2^i)$ is precomputed (line 27) before the polling loop. When all children are ready, the PE's *readyEpoch* is updated with *myEpoch* (line 30). The following CPU memory fence is required to prevent possible CPU memory reordering of this update (and, in general, other operations before or after the barrier). The reference *e* to $readyEpoch@(0)$ is precomputed in line 32 before the loop that polls *e* until it reaches the value of *myEpoch*.

Listing 13.2. An implementation of a binomial tree barrier in C++

```
class BinomialBarrier {
  BinomialBarrier();
  struct EpochNumChildren {
    EpochNumChildren() : epoch(0), numChildren(0) {}
    int epoch, numChildren; // co-locate to avoid additional cache miss
    char padding[64 - 2*sizeof(int)];
  };
  template <class It>
  void initNumChildren(It begin, int size) {
    if(size < 2) return;
    for(int i = 1; i < size; i *= 2) {
      ++(begin->numChildren);
      int child = i, childSize = i;
      if(size < child + childSize) childSize = size - child;
      initNumChildren(begin + child, childSize);
    }
  }
public:
  BinomialBarrier(int p) : readyEpoch(p), peData(p) {
    initNumChildren(peData.begin(), p);
    for(auto && e : readyEpoch) e.first = 0;
  }
  void wait(const int iPE, const int /* p */) {
    register const int myEpoch = ++(peData[iPE].epoch);
    register const int numC = peData[iPE].numChildren;
    for(int i=0; i < numC; ++i) {
      auto & e = readyEpoch[iPE + (1<<i)].first;
      while(e != myEpoch) backoff();
    }
    readyEpoch[iPE].first = myEpoch;
    atomic_thread_fence(memory_order_seq_cst);
    auto & e = readyEpoch[0].first;
    while(e != myEpoch) backoff();
  }
private:
  typedef std::pair<volatile int, char [64-sizeof(int)]> paddedInt;
  vector<paddedInt> readyEpoch;
  vector<EpochNumChildren> peData;
};//SPDX-License-Identifier: BSD-3-Clause; Copyright(c) 2018 Intel Corporation
```

To benchmark the barrier implementations, we chose a diffusion-like application that iteratively averages neighboring entries of an array:

Procedure *diffuse(a* : *Array* $[0..n+1]$ **of** *double, k* : *int)*
    b : *Array* $[0..n+1]$ **of** *double;*
    for $i := 1$ **to** k **step** *2* **do**
        for $j := 1$ *to* n **do**$\|$
            $b[i] := (a[i-1] + a[i] + a[i+1])/3.0$*;*
        barrier;
        for $j := 1$ **to** n **do**$\|$
            $a[j] := (b[j-1] + b[j] + b[j+1])/3.0$*;*
        barrier;

Table 13.2 shows running times and speedups using 64 and 128 threads. We used $k = 10^8 p/n$ iterations, i.e., we kept the number of arithmetic operations per thread fixed. The tests were done on the four-socket machine described in Sect. B. If the barrier synchronization overhead is significant compared with the amount of computation (small values of n/p), we observe speedups of up to 2.35 compared with the simple barrier. We also tested the standard barrier implementation from the POSIX PThread library. Unfortunately, it does not scale at all and did not finish even after several hours. The reason is that it uses an exclusive lock to manage the internal structure for every barrier synchronization; see `github.com/lattera/glibc/blob/master/nptl/pthread_barrier_wait.c`. On the other hand, we did experiments with the OpenMP barrier implementation of the Intel® C++ Compiler (version 18.0.0), which shows similar performance than our binomial tree barrier. It turns out that it uses a similar algorithm.

Table 13.2. Running times (in milliseconds, median of nine runs) of the barrier benchmark with 64 and 128 threads.

p	n/p	simple barrier	binomial tree barrier	speedup
64	100	6 826	3 702	1.84
64	1 000	895	580	1.54
64	10 000	335	318	1.05
64	100 000	282	279	1.01
64	1 000 000	2 371	2 341	1.01
128	100	8 012	3 413	2.35
128	1 000	1465	790	1.85
128	10 000	632	571	1.11
128	100 000	3 006	2 917	1.03
128	1 000 000	3 945	3 897	1.01

13.5 (All)-Gather/Scatter

13.5.1 (All)-Gather

Given a local message m on each PE, a gather operation moves all these messages $m@0, m@1, \ldots, m@(p-1)$ to PE 0 (or some other specified PE). All-gather moves these messages to *all* PEs. The all-gather operation is also called *gossiping* or *all-to-all broadcast*. These operations are sometimes needed to assemble partial results

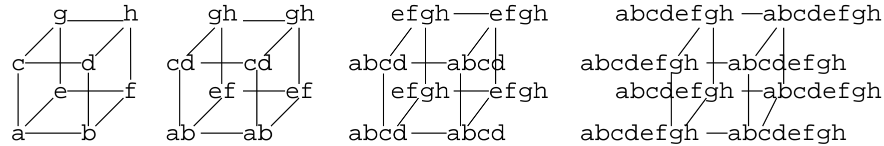


Fig. 13.11. Hypercube all-gather on eight PEs.

into a global picture. Whenever possible, gathers should be avoided since they can easily become a performance bottleneck. In a naive implementation of gather, all PEs send their message directly to PE 0. This takes communication time $(p-1)(\alpha + n\beta)$. We cannot do much about the term involving β, since PE 0 needs to receive all the messages in some way or the other. However, we can reduce the number of startup latencies. We simply use the binomial tree reduction algorithm presented in Sect. 13.2 using the concatenation of messages as the operator (see Fig. 13.12). This is correct since concatenation is associative. However, we have to redo the algorithm analysis, since the communicated messages have nonuniform length.

Theorem 13.9. *Gather using binomial tree reduction needs communication time*

$$\lceil \log p \rceil \alpha + (p-1)n\beta.$$

Proof. Counting from the bottom, in level i of the reduction tree, messages of length $2^i n$ are sent. This takes time $\alpha + 2^i n\beta$. Summing over all levels we get

$$\sum_{i=0}^{\lceil \log p \rceil - 1} \alpha + 2^i n\beta = \lceil \log p \rceil \alpha + n\beta \sum_{i=0}^{\lceil \log p \rceil - 1} 2^i = \lceil \log p \rceil \alpha + (p-1)n\beta. \qquad \square$$

Using the hypercube algorithm for all-reduce in Sect. 13.2.1 instead, we get an all-gather with the same complexity (when p is a power of two). Figure 13.11 gives an example.

Corollary 13.10. *All-gather using hypercube all-reduce needs communication time* $\alpha \log p + (p-1)n\beta$.

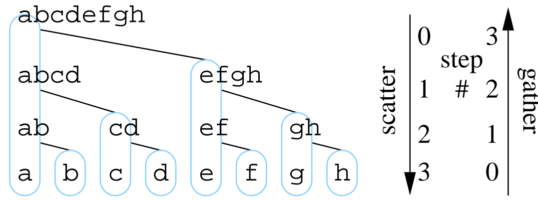


Fig. 13.12. Binomial tree gather and scatter for $p = 8$.

13.5.2 Scatter

When PE 0 (or some other specified PE) has messages $m_0, \ldots, m_{p-1}$ and delivers them so that PE i gets message m_i, this is called a *scatter* operation. This operation can be useful for distributing data evenly to all PEs. As with gather, the presence of a scatter operation may be an indicator of a program with poor scalability.

Scatter and gather and broadcast and reduction are, in a certain sense, duals of each other. Indeed, by slightly modifying the binomial tree broadcast algorithm in Sect. 13.1.1, we can get an efficient scatter algorithm: When sending to a subtree containing the nodes $a..b$, the concatenation of messages $m_a, \ldots, m_b$ is sent to the root of that subtree. Figure 13.12 gives an example. An analysis analogous to that in Theorem 13.9 yields execution time

$$\lceil \log p \rceil \alpha + (p-1)n\beta .$$

13.6 All-to-All Message Exchange

The most general communication operation is when every PE i has a message m_{ij} for every other PE j. We have seen this pattern for several sorting algorithms (bucket sort, radix sort, sample sort, and multiway mergesort) and for bulk operations on hash tables. We first look at the case where all messages have the same size n. In Sect. 13.6.1, we shall see how to achieve this task with direct delivery of data in communication time

$$T_{\mathrm{all}\to\mathrm{all}}(n) \leq (p-1)(n\beta + \alpha). \tag{13.3}$$

For small messages, indirect delivery is better. Using a hypercube algorithm, we obtain

$$T_{\mathrm{all}\to\mathrm{all}}(n) \leq \log p \left(n\frac{p}{2}\beta + \alpha \right), \tag{13.4}$$

see Sect. 13.6.2. In Sect. 13.6.3 we consider the more general case when messages are allowed to have different sizes. Let $h = \max_i \max(\sum_j |m_{ij}|, \sum_j |m_{ji}|)$ denote the maximum amount of data to be sent or received by a PE. We shall see a simple algorithm that needs communication time

$$T^*_{\mathrm{all}\to\mathrm{all}}(h) \leq 2T_{\mathrm{all}\to\mathrm{all}}\left(\frac{h}{p} + 2p \right). \tag{13.5}$$

13.6.1 Uniform All-to-All with Direct Data Delivery

At first glance, all-to-all communication looks easy – simply send each message to its destination. However, our model of communication allows only one message to be sent at a time, and senders and receivers have to agree in which order the messages are sent. To keep the notation short, we abbreviate $i_{\mathrm{proc}} \in 0..p-1$ to i in this section.

There is a particularly simple solution if p is a power of two: In step $k \in 1..p-1$, PE i communicates with PE $i \oplus k$. Since

$$(i \oplus k) \oplus k = i \oplus (k \oplus k) = i,$$

the senders and receivers agree on their communication partner. Moreover, every pair of PEs communicates in some step – PEs i and j communicate in step $i \oplus j$ (since $i \oplus (i \oplus j) = (i \oplus i) \oplus j = j$). We obtain an algorithm that needs communication time $(p-1)(\alpha + n\beta)$. This is optimal if messages are to be sent directly. Moreover, in each step the PEs are paired and the partners in each pair exchange their messages. This is referred to as the *telephone model* of point-to-point communication.

With full-duplex communication, we may also arrange the PEs into longer cycles (note that pairs are cycles of length two). The following code implements this idea:

for $k := 1$ **to** $p-1$ **do**
  send$((i+k) \bmod p, m_{i,(i+k) \bmod p}) \parallel$ *receive*$((i-k) \bmod p, m_{(i-k) \bmod p, i})$

Exercise 13.10. Show the correctness of this solution.

We next give a solution in the telephone model for general p. We first show how to achieve the communication task for odd p with p steps and then derive a solution for even p with $p-1$ steps. The solution also applies to speed-dating and speed chess.

So assume p is odd. We first observe that the sequence $2r \bmod p$, $r \in 0..p-1$, first enumerates the even numbers 0, 2, … $p-1$ and then the odd numbers 1, 3, …, $p-2$ less than p. Consider any round r and let $k = 2r \bmod p$. We pair PEs i and j such that $i + j = k \bmod p$. Then $j = k - i \bmod p$ or

$$j = \begin{cases} k-i & \text{if } i \leq k \\ p+k-i & \text{if } i > k. \end{cases}$$

In this way, PE i will send to any PE (including itself) exactly once, and if PE i sends to j, j will send to i, i.e., we are within the telephone model. In each step, one PE communicates superfluously with itself. This PE has number $k/2$ if k is even and number $(p+k)/2$ if k is odd. Since $k = 2r$ if k is even and $k = 2r - p$ if k is odd, the idle PE has number r. In order to have this nice correspondence between round number and number of idle PE, we have k enumerate the numbers less than p in the order $0, 2, 4, \ldots, p-1, 1, 3, \ldots, p-2$.

The algorithm for even p is only slightly more complicated. We essentially run the algorithm for the case $p-1$. The only modification is that the "idle" PE communicates with PE $p-1$. We obtain the following protocol:

$p' := p - isEven(p)$
for $r := 0$ **to** $p' - 1$ **do**
  $k := 2r \bmod p'$
  $j := (k - i) \bmod p'$
  if $isEven(p) \wedge i = r$ **then** $j := p - 1$
  if $isEven(p) \wedge i = p - 1$ **then** $j := r$
  $send(j, m_{ij}) \parallel receive(j, m_{ji})$

Figure 13.13 gives an example of the resulting communication pattern for this *1-factor algorithm*. The resulting communication time is $(p - isEven(p))(\alpha + n\beta)$.

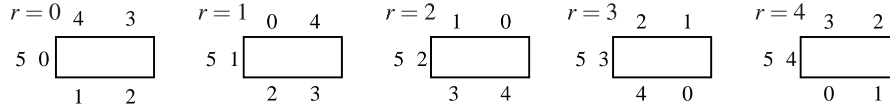


Fig. 13.13. Speed-dating for $p = 6$. PEs 0 to 4 sit around a rectangular table. Initially, PE 0 sits on one of the short sides of the table and the other PEs sit in counter-clockwise order along the long sides of the table. PE 5 sits separately next to PE 0. The PEs sitting on the long sides of the table partner with their counterpart on the other side of the table. In each round, PEs 0 to 4 move clockwise by one position. The index of the partner grows by two modulo $p - 1$ in each round. For example, the partners of PE 3 are 2, 4, 1, 5, 0 in the five rounds.

13.6.2 Hypercube All-to-All

For small n, direct data delivery is wasteful since the communication time will be dominated by startup overheads. Once more, we can reduce the number of startup overheads to $\log p$ using a hypercube algorithm when p is a power of two. The following pseudocode defines the algorithm:

$M := \{m_{i_{\text{proc}} j} : j \in 0..p-1\}$ // messages i_{proc} has to deliver
for $k := d - 1$ **downto** 0 **do**
  $M_s := \{m_{ij} \in M : i_{\text{proc}}$ **bitand** $2^k \neq j$ **bitand** $2^k\}$ // messages for other k-cube
  $send(i \oplus 2^k, M_s) \parallel receive(i \oplus 2^k, M_r)$
  $M := M_r \cup M \setminus M_s$

Each PE maintains a set M of messages it has to deliver. This time, we iterate from $d - 1$ downward because this simplifies formulating the loop invariant: After iteration k, all messages destined for a k-dimensional subcube are located somewhere in that subcube. Thus, the messages $m_{ij} \in M_s$ to be sent in each iteration are those where the kth bit of the recipient j differs from the kth bit of the local PE i_{proc}. Figure 13.14 gives an example. The algorithm sends and receives $p/2$ messages in each iteration. Thus, its communication complexity is

$$\log p \left(n \frac{p}{2} \beta + \alpha \right).$$

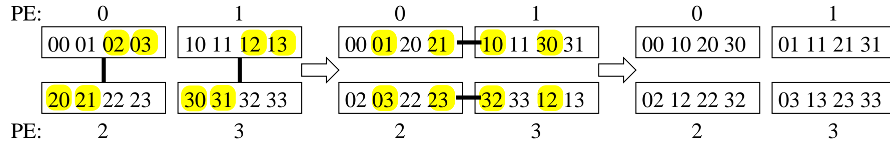


Fig. 13.14. Hypercube all-to-all with $p = 4$. The digit pair ij is a shorthand for m_{ij}. The boxes contain the message set in the current step. The shaded messages are moved in the next step along the communication links given by the bold lines.

This is good for small n but wasteful for large n since a message is moved $\log p/2$ times on average.

There are compromises between moving the data with p startups and moving it a logarithmic number of times with a logarithmic number of startups.

***Exercise 13.11.** Suppose $p = k^d$ for integers k and d. Design an algorithm that delivers all messages using communication time $\leq kd\alpha + dnp\beta$. Hint: Arrange the PEs as a d-dimensional grid of side length k. Generalize the hypercube algorithm for this case. Use the direct delivery algorithm within each dimension. Generalize further for the case where $p = k_1 \cdot k_2 \cdots k_d$. Give an algorithm that needs communication time $\alpha \sum_i k_i + \beta dnp$.

13.6.3 All-to-All with Nonuniform Message Sizes

We now consider the case where the messages m_{ij} to be delivered have arbitrary sizes. A good way to measure the difficulty of such a problem is to look for the *bottleneck* communication volume – what is the maximum amount of data to be sent or received by any PE? In the full-duplex model of point-to-point communication, this is

$$h = \max_i \max\left(\sum_j |m_{ij}|, \sum_j |m_{ji}|\right).$$

We shall therefore also call this problem an *h-relation*. The uniform all-to-all algorithms we have seen so far can become quite slow if we use them naively by padding all messages to the maximum occurring message size. Even optimized implementations that deliver only the data actually needed could become slow because PEs currently delivering short messages will have to wait for those delivering long messages. We therefore need new algorithms.

An elegant solution is to reduce the all-to-all problem with variable message lengths to two all-to-all problems with a uniform message length. We chop each message m_{ij} into p pieces m_{ijk} of size $\lceil |m_{ij}|/p \rceil$ and send piece m_{ijk} from PE i to PE j via PE k. For the first uniform data exchange, PE i combines all messages m_{ijk}, $0 \leq j \leq p-1$, into a single message m_{i*k} and sends it to PE k. The message consists of p fields specifying where the pieces begin plus the p pieces itself. Thus,

$$|m_{i*k}| = p + \sum_j |m_{ijk}| \leq p + \sum_j \left\lceil \frac{|m_{ij}|}{p} \right\rceil \leq p + \sum_j \left(\frac{|m_{ij}|}{p} + 1\right) \leq \frac{h}{p} + 2p.$$

Note that this bound is independent of the individual message sizes and depends only on the global values h and p. Thus, the first data exchange can be performed in time $T_{\text{all}\to\text{all}}(h/p+2p)$ using any uniform all-to-all operation.

For the second data exchange, PE k combines the pieces m_{ijk}, $0 \leq i \leq p-1$, into a message m_{*jk} and sends it to PE j. We have

$$|m_{*jk}| = p + \sum_i |m_{ijk}| \leq p + \sum_i \left\lceil \frac{|m_{ij}|}{p} \right\rceil \leq p + \sum_i \left(\frac{|m_{ij}|}{p} + 1 \right) \leq \frac{h}{p} + 2p.$$

Once more, this can be done with a uniform all-to-all with message size $h/p+2p$. Figure 13.15 gives an example. Overall, assuming the regular all-to-all is implemented using direct data delivery, we get a total communication cost

$$T^*_{\text{all}\to\text{all}}(h) \leq 2T_{\text{all}\to\text{all}}\left(\frac{h}{p} + 2p^2\right) \approx 2p\alpha + (2h+4p)\beta.$$

This is not quite what we may want. For large h this is a factor of 2 away from the lower bound $h\beta$ that we may hope to approximate. For $h \ll p$, the overhead for rounding and communicating message sizes is an even worse penalty. There are approaches to improve the situation, but we are not aware of a general solution – routing h-relations in a practically and theoretically satisfactory way is still a research problem.

For example, there is an intriguing generalization of the graph theoretical model discussed in Sect. 13.6.1. Suppose the messages are partitioned into packets of uniform size. Now h measures the maximum communication cost in number of packets. We can model the communication problem as a bipartite graph $H = (\{s_1,\ldots,s_p\} \cup \{r_1,\ldots,r_p\}, E)$. Node s_i models the sending role of PE i and node r_i models its receiving role. A packet to be sent from PE i to PE j is modeled as an edge (s_i, r_j). Since messages can consist of multiple packets, *parallel edges* are allowed, i.e., we are dealing with a *multigraph*. Assuming packets are to be delivered directly, scheduling communication of an h-relation is now equivalent to coloring the edges of H. Consider a coloring $\chi : E \to 1..k$ where no two incident edges have the same color. Then all the packets corresponding to edges with color c can be delivered in a single step. Conversely, we can obtain a coloring of H for any schedule for the h-relation: If a packet x is delivered in step c, we can assign color c to the corresponding edges in H.

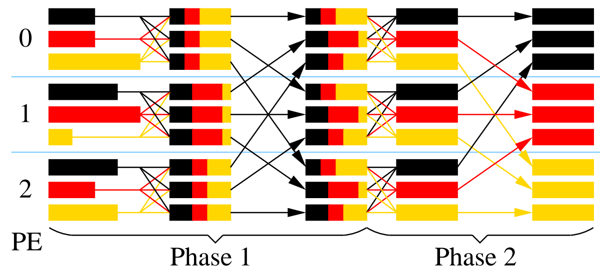


Fig. 13.15. Two-phase nonuniform all-to-all for $p=3$.

Exercise 13.12. Prove this equivalence formally.

The maximum degree of H is h. It is known that a bipartite multigraph with maximum degree h can be edge-colored with h colors [187]; the maximum degree is clearly also a lower bound. Moreover, such a coloring can be found in polynomial time – actually, in time $O(|E|\log h)$ [77]. There are also fast parallel algorithms [201]

The discussion in the preceding paragraph suggests that delivering an h-relation may be possible in time close to $h\beta$ for large h. However, it is not so clear how to use an algorithm based on edge coloring in practice, since we would need to compute the coloring in parallel and highly efficiently.

13.7 Asynchronous Collective Communication

Sometimes one wants to overlap a collective communication with other computations. For example, the double counting termination detection protocol used in Sect. 14.5.3 requires a reduction operation running concurrently with the application. Most of the collective communication operations described here can be adapted to an asynchronous scenario. We must take care, however, to use asynchronous message send operations that do not have to wait for the actual delivery. We should also be aware that additional delays may be introduced by PEs that do not immediately react to incoming messages. One way to avoid such delays is to run the asynchronous operations using separate threads. Another useful concept is *active messages* that trigger a predefined behavior on the receiving side. For example, an asynchronous broadcast might trigger a transfer of the message to the children in a tree data structure. One can also emulate active messages by using a message handler within the application that can handle all possible asynchronous events by calling appropriate callback functions.