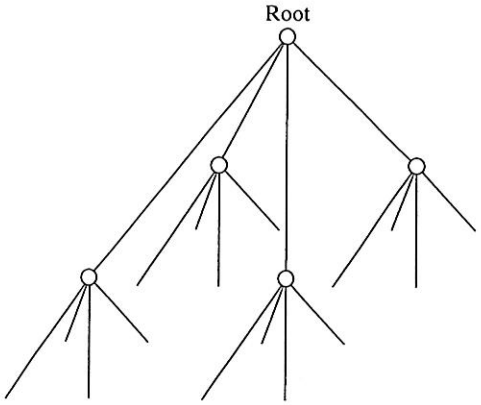


Figure 4.6 Quadtree.

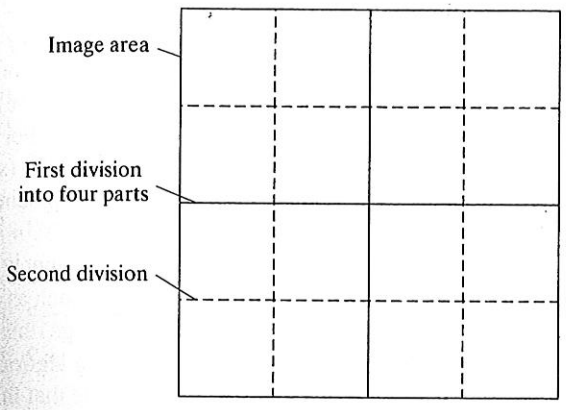


Figure 4.7 Dividing an image.

parallelism is available as m is increased because there are more parts that could be considered simultaneously. It is left as an exercise to develop the equations for computation time and communication time (Problem 4-7).

4.2 PARTITIONING AND DIVIDE-AND-CONQUER EXAMPLES

4.2.1 Sorting Using Bucket Sort

Suppose the problem is not simply to add together numbers in a list, but to sort them into numerical order. There are many practical situations that require numbers to be sorted, and in consequence, sequential programming classes spend a great deal of time developing the various ways that numbers can be sorted. Most of the sequential sorting algorithms are based upon the compare and exchange of pairs of numbers, and we will look at parallelizing such classical sequential sorting algorithms in Chapter 10. Let us look here at the sorting algorithm called *bucket sort*. Bucket sort is not based upon compare and exchange, but is naturally a partitioning method. However, bucket sort only works well if the original numbers are uniformly distributed across a known interval, say 0 to $a - 1$. This interval is

divided into m equal regions, 0 to $a/m - 1$, a/m to $2a/m - 1$, $2a/m$ to $3a/m - 1$, ... and one "bucket" is assigned to hold numbers that fall within each region. There will be m buckets. The numbers are simply placed into the appropriate buckets. The algorithm could be used with one bucket for each number (i.e., $m = n$). Alternatively, the algorithm could be developed into a divide-and-conquer method by continually dividing the buckets into smaller buckets. If the process is continued in this fashion until each bucket can only contain one number, the method is similar to quicksort, except that in quicksort the regions are divided into regions defined by "pivots" (see Chapter 10). Here we will use a limited number of buckets. The numbers in each bucket will be sorted using a sequential sorting algorithm, as shown in Figure 4.8.

Sequential Algorithm. To place a number into a specific bucket it is necessary to identify the region in which the number lies. One way to do this would be to compare the number with the start of regions; i.e., a/m, $2a/m$, $3a/m$, ... This could require as many as $m - 1$ steps for each number on a sequential computer. A more effective way is to divide the number by m/a and use the result to identify the buckets from 0 to $m - 1$, one computational step for each number (although division can be rather expensive in time). If m/a is a power of 2, one can simply look at the upper bits of the number in binary. For example, if $m/a = 2^3$ (eight), and the number is 1100101 in binary, it falls into region 110 (six), by considering the most significant three bits. In any event, let us assume that placing a number into a bucket requires one step, and that placing all the numbers requires n steps. If the numbers are uniformly distributed, there should be n/m numbers in each bucket.

Next, each bucket must be sorted. Sequential sorting algorithms, such as quicksort or mergesort, have a time complexity of $O(n \log n)$ to sort n numbers (average time complexity for quicksort). The lower bound on any compare and exchange sorting algorithm is about $n \log n$ comparisons (Aho, Hopcroft, and Ullman, 1974). Let us assume that the sequential sorting algorithm actually requires $n \log n$ comparisons, one comparison being regarded as one computational step. Thus, it will take $(n/m)\log(n/m)$ steps to sort the n/m numbers in each bucket using these sequential sorting algorithms. The sorted numbers must be concatenated into the final sorted list. Let us assume that this concatenation requires no additional steps. Combining all the actions, the sequential time becomes

$$t_s = n + m((n/m)\log(n/m)) = n + n\log(n/m) = O(n\log(n/m))$$

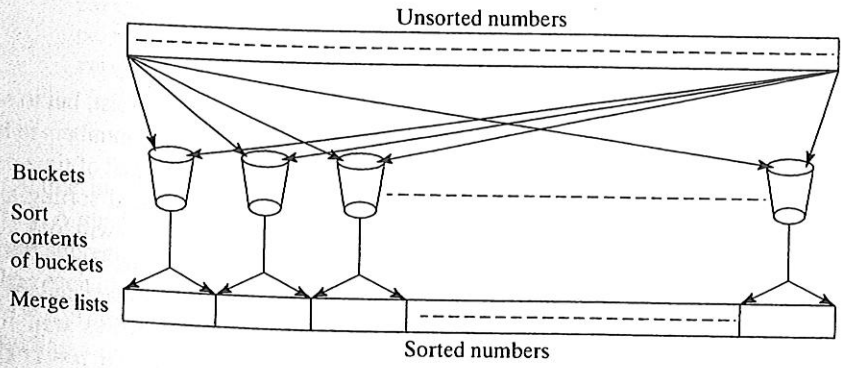


Figure 4.8 Bucket sort.

If $n = km$, where k is a constant, we get a time complexity of $O(n)$. Note that this is much better than the lower bound for sequential compare and exchange sorting algorithms. However, it only applies when the numbers are uniformly distributed.

Parallel Algorithm. Clearly, bucket sort can be parallelized by assigning one processor for each bucket, which reduces the second term in the preceding equation to $(n/p)\log(n/p)$ for p processors (where $p = m$). This implementation is illustrated in Figure 4.9. In this version, each processor examines each of the numbers, so that a great deal of wasted effort takes place. The implementation could be improved by having the processors actually remove numbers from the list into their buckets so that they are not reconsidered by other processors.

We can further parallelize the algorithm by partitioning the sequence into m regions, one region for each processor. Each processor maintains p "small" buckets and separates the numbers in its region into its own small buckets. These small buckets are then "emptied" into the p final buckets for sorting, which requires each processor to send one small bucket to each of the other processors (bucket i to processor i). The overall algorithm is shown in Figure 4.10. Note that this method is a simple partitioning method in which there is minimal work to create the partitions.

The following phases are needed:

1. Partition numbers.
2. Sort into small buckets.
3. Send to large buckets.
4. Sort large buckets.

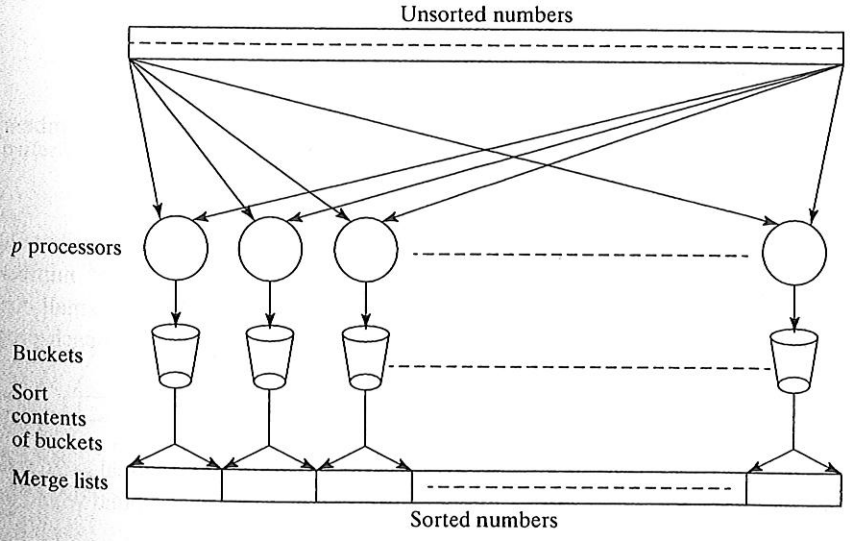


Figure 4.9 One parallel version of bucket sort.

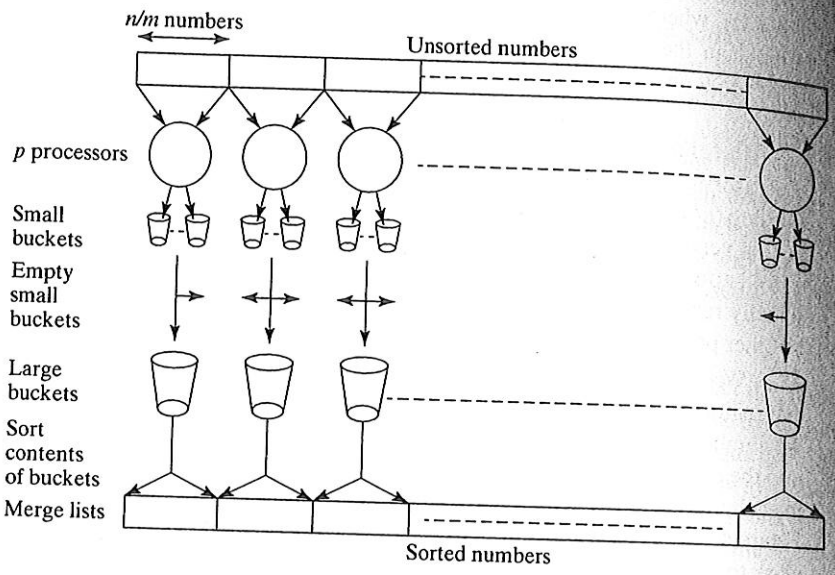


Figure 4.10 Parallel version of bucket sort.

Phase 1 — Computation and Communication. The first step is to send groups of numbers to each processor. Marking a group of numbers into partitions can be done in constant time. This time will be ignored in the overall compassion time. Rather than make the partitions and then send a partition to each processor, a more efficient solution is to simply broadcast all the numbers to each processor and let each processor make its partition. (One must ensure that each partition so created is disjoint but the partitions together include all the numbers.) Using a broadcast or scatter routine, the communication time is:

$$t_{comm1} = t_{startup} + nt_{data}$$

including the communication startup time.

Phase 2 — Computation. To separate each partition of n/p numbers into p small buckets requires the time

$$t_{comp2} = n/p$$

Phase 3 — Communication. Next, the small buckets are distributed. (There is no computation in Phase 3.) Each small bucket will have about n/p^2 numbers (assuming uniform distribution). Each process must send the contents of $p - 1$ small buckets to other processes (one bucket being held for its own large bucket). Since each process of the p processes must make this communication, we have

$$t_{comm3} = p(p-1)(t_{startup} + (n/p^2)t_{data})$$

if these communications cannot be overlapped in time and individual `send()`s are used. This is the upper bound on this phase of communication. The lower bound would occur if all the communications could overlap, leading to

$$t_{comm3} = (p-1)(t_{startup} + (n/p^2)t_{data})$$

In essence, each proces
all" mechanism would
to every other process,
MPI (`MPI_Alltoall()`),
using individual `send()`
of an array to columns
Section 10.2.3).

Phase 4 — Comp
neously. Each large buc

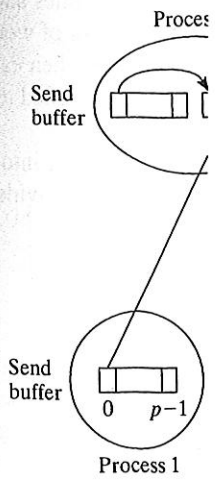
Overall Executio

$t_p = t_{startup} +$
$= (n/p$

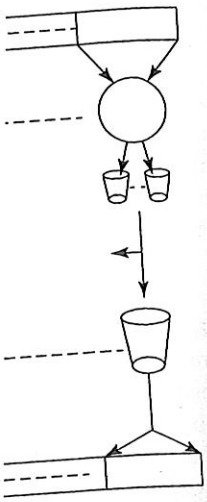
Speedup Factor.

Speedup factor = $\frac{t}{t}$

Speedup factor is actu
the problem. The lov
exchange operations
sequence is $n \log n$ st
has the assumption of



Fi



step is to send groups of
partitions can be done in
n time. Rather than make
ore efficient solution is to
processor make its parti-
but the partitions together
ne communication time is:

f n/p numbers into p small

are distributed. (There is no
ut n/p^2 numbers (assuming
$p-1$ small buckets to other
Since each process of the p

ata)

lividual `send()`s are used. This
er bound would occur if all the

lata)

Conquer Strategies Chap. 4

In essence, each processor must communicate with every other processor, and an "all-to-all" mechanism would be appropriate. An all-to-all routine sends data from each process to every other process, as illustrated in Figure 4.11. This type of routine is available in MPI (`MPI_Alltoall()`), which we assume would be implemented more efficiently than using individual `send()`s and `recv()`s. The all-to-all routine will actually transfer the rows of an array to columns, as illustrated in Figure 4.12 (and hence transpose a matrix; see Section 10.2.3).

Phase 4 — Computation. In the final phase, the large buckets are sorted simultaneously. Each large bucket contains about n/p numbers. Thus

$$t_{comp4} = (n/p)\log(n/p)$$

Overall Execution Time. The overall run time, including communication, is

$$t_p = t_{comm1} + t_{comp2} + t_{comm3} + t_{comp4}$$

$$t_p = t_{startup} + nt_{data} + n/p + (p-1)(t_{startup} + (n/p^2)t_{data}) + (n/p)\log(n/p)$$

$$= (n/p)(1 + \log(n/p)) + pt_{startup} + (n + (p-1)(n/p^2))t_{data}$$

Speedup Factor. The speedup factor, when compared to sequential bucket sort, is

$$\text{Speedup factor} = \frac{t_s}{t_p} = \frac{n + n\log(n/m)}{(n/p)(1 + \log(n/p)) + pt_{startup} + (n + (p-1)(n/p^2))t_{data}}$$

Speedup factor is actually defined where t_s is the time for the best sequential algorithm for the problem. The lower bound for a sequential sorting algorithm using compare and exchange operations and no requirement upon the distribution or special features of the sequence is $n \log n$ steps. However, bucket sort is better than this and is used for t_s, but it has the assumption of uniformly distributed numbers.

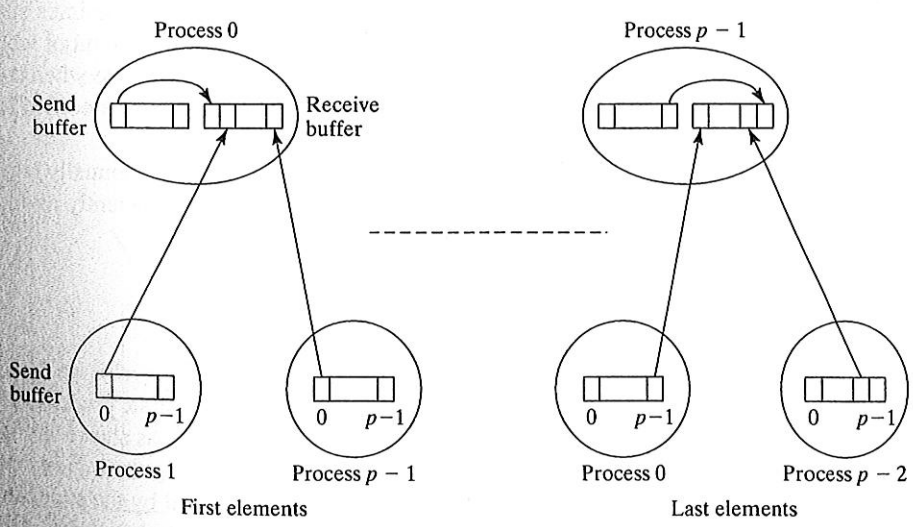


Figure 4.11 All-to-all broadcast.

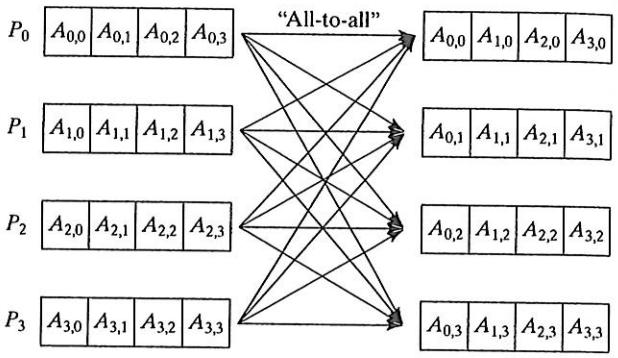


Figure 4.12 Effect of all-to-all on an array.

Computation/communication Ratio. The computation/communication ratio is given by

$$\text{Computation/communication ratio} = \frac{t_{comp}}{t_{comm}} = \frac{(n/p)(1+\log(n/p))}{pt_{startup} + (n+(p-1))(n/p^2)t_{data}}$$

It is assumed that the numbers are uniformly distributed to obtain these formulas. If the numbers are not uniformly distributed, some buckets would have more numbers than others, and sorting them would dominate the overall computation time. The worst-case scenario would occur when all the numbers fell into one bucket!

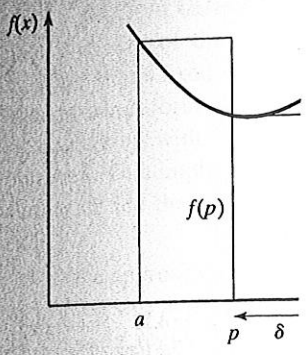
4.2.2 Numerical Integration

Previously, we divided a problem and solved each subproblem. The problem was assumed to be divided into equal parts, and partitioning was employed. Sometimes such simple partitioning will not give the optimum solution, especially if the amount of work in each part is difficult to estimate. Bucket sort, for example, is only effective when each region has approximately the same number of numbers. (Bucket sort can be modified to equalize the work.)

A general divide-and-conquer technique divides the region continually into parts and lets an optimization function decide when certain regions are sufficiently divided. Let us take a different example, numerical integration:

$$I = \int_a^b f(x)\,dx$$

To integrate this function (i.e., to compute the "area under the curve"), we can divide the area into separate parts, each of which can be calculated by a separate process. Each region could be calculated using an approximation given by rectangles, as shown in Figure 4.13, where $f(p)$ and $f(q)$ are the heights of the two edges of a rectangular region, and δ is the width (the *interval*). The complete integral can be approximated by the summation of the rectangular regions from a to b. A better approximation can be obtained by aligning the rectangles so that the upper midpoint of each rectangle intersects with the function, as



shown in Figure 4.14. This c
the midpoint end tend to ca
intersections of the vertical l
in Figure 4.15. Each regior
numerical methods for comp
are called *quadrature* metho

Static Assignment.
the computation, one proces
region. By making the interv

Since each calculation
model is appropriate. Suppo
numbered 0 to $p-1$. The si
area in the described manne

Process P_i

```
if (i == master) {
   printf("Enter num
   scanf(%d",&n);
}
```