Chapter 4

Partitioning and Divide-and-Conquer Strategies

In this chapter, we explore two of the most fundamental techniques in parallel programming, called *partitioning* and *divide and conquer,* which are related. In partitioning, the problem is simply divided into separate parts and each part is computed separately. Divide and conquer usually applies partitioning in a recursive manner by continually dividing the problem into smaller and smaller parts before solving the smaller parts and combining the results. First, we will review the technique of partitioning. Then we discuss recursive divide-and-conquer methods. Next, we outline some typical problems that can be solved with these approaches. As usual, we have a selection of scientific/numerical and real-life problems at the end of the chapter.

4.1 PARTITIONING

4.1.1 Partitioning Strategies

Partitioning simply divides the problem into parts. It is the basis of all parallel programming, in one form or another. The embarrassingly parallel problems in the last chapter used partitioning. Most partitioning formulations, however, require the results of the parts to be combined to obtain the desired result. Partitioning can be applied to the program data (that is, to dividing the data and operating upon the divided data concurrently). This is called *data partitioning* or *domain decomposition*. Partitioning can also be applied to the functions of a program (that is, dividing the program into independent functions and executing the functions concurrently). This is *functional decomposition*. The idea of per-forming a task by dividing it into a number of smaller tasks that when completed will complete the overall task is, of course, well known and can be applied in many situations, whether the smaller tasks operate upon parts of the data or are separate concurrent functions.

It is much less common to find concurrent functions in a problem, but data partitioning is a main strategy for parallel programming. To take a really simple data partitioning example, suppose a sequence of numbers, $x_0 \ldots x_{n-1}$, are to be added. This is a problem recurring in the text to demonstrate a concept; clearly unless there were a huge sequence of numbers, a parallel solution would not be worthwhile. However, the approach can be used for more realistic applications involving complex calculations on large databases.

We might consider dividing the sequence into m parts of n/m numbers each, $(x_0 \ldots x_{(n/m)-1})$, $(x_{n/m} \ldots x_{(2n/m)-1})$, …, $(x_{(m-1)n/m} \ldots x_{n-1})$, at which point m processors (or processes) can each add one sequence independently to create partial sums. The m partial sums need to be added together to form the final sum. Figure 4.1 shows the arrangement in which a single processor adds the m partial sums. Notice that each processor requires access to the numbers it has to accumulate. In a message-passing system, the numbers would need to be passed to the processors individually. (In a shared memory system, each processor could access those numbers it wanted from the shared memory, and in this respect, a shared memory system would clearly be more convenient for this and similar problems.)

The parallel code for this example is straightforward. For a simple master-slave approach, the numbers are first sent from the master processor to slave processors. The slave processors add their numbers, operating independently and concurrently. Next, the partial sums are sent from the slaves to the master processor. Finally, the master processor adds the partial sums to form the result. Often, we talk of processes rather than processors for code sequences, where one process is best mapped onto one processor.

It is a moot point whether broadcasting the whole list of numbers to every slave or only sending the specific numbers to each slave is best, since in both cases all numbers must be sent from the master. The specifics of the broadcast mechanism would need to be known in order to decide on the relative merits of that mechanism. A broadcast will have a single startup time rather than separate startup times when using multiple send routines and may be preferable.

First, we will send the specific numbers to each slave using individual `send()`s. Given

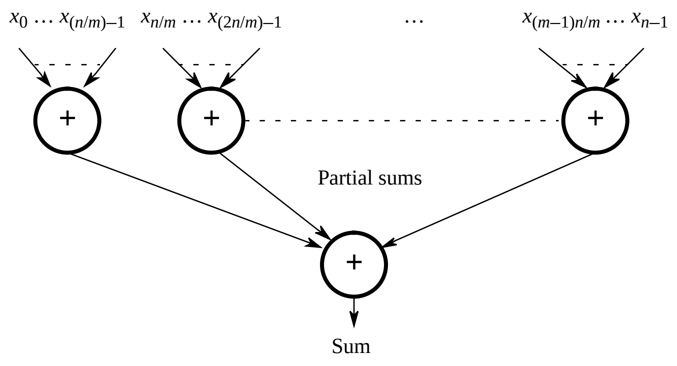

Figure 4.1 Partitioning a sequence of numbers into parts and adding the parts.

n numbers and *m* slaves, where *n*/*m* is an integer, the code using separate `send()`s and `recv()`s might look like the following:

Master

```
s = n/m;                           /* number of numbers for slaves*/
for (i = 0, x = 0; i < m; i++, x = x + s)
  send(&numbers[x], s, Pi);        /* send s numbers to slave */

result = 0;
for (i = 0; i < m; i++) {          /* wait for results from slaves */
  recv(&part_sum, PANY);
  sum = sum + part_sum;            /* accumulate partial sums */
}
```

Slave

```
recv(numbers, s, Pmaster);         /* receive s numbers from master */
sum = 0;
for (i = 0; i < s; i++)            /* add numbers */
  part_sum = part_sum + numbers[i];
send(&part_sum, Pmaster);          /* send sum to master */
```

If a broadcast or multicast routine is used to send the complete list to every slave, code is needed in each slave to select that part of the sequence to be used by the slave, adding additional computation steps within the slaves, as in

Master

```
s = n/m;                           /* number of numbers for slaves */
bcast(numbers, s, Pslave_group);   /* send all numbers to slaves */
result = 0;
for (i = 0; i < m; i++){           /* wait for results from slaves */
  recv(&part_sum, PANY);
  sum = sum + part_sum;            /* accumulate partial sums */
}
```

Slave

```
bcast(numbers, s, Pmaster);        /* receive all numbers from master*/
start = slave_number * s;          /* slave number obtained earlier */
end = start + s;
sum = 0;
for (i = start; i < end; i++)      /* add numbers */
  part_sum = part_sum + numbers[i];
send(&part_sum, Pmaster);          /* send sum to master */
```

Slaves are identified by a process ID, which can usually be obtained by calling a library routine. Most often, a group will first be formed and the slave number is an instance or rank within the group. The instance or rank is an integer from 0 to $m - 1$, where there are *m* processes in the group. MPI requires communicators to be established, and processes have rank within a communicator, as described in Chapter 2. Groups can be associated within communicators, and processes have a rank within groups. PVM has the concept of groups, but a unique process ID can also be obtained from the `PVM_spawn()` routine when the process is created, which could be used for very simple programs. (PVM process IDs are not small consecutive integers; they are similar to normal UNIX process IDs.)

If scatter and reduce routines are available,[1] the code could be

Master

```
s = n/m;                                 /* number of numbers */
scatter(numbers,&s,P_group,root=master);     /* send numbers to slaves */
reduce_add(&sum,&s,P_group,root=master);     /* results from slaves */
```

Slave

```
scatter(numbers,&s,P_group,root=master);     /* receive s numbers */
reduce_add(&part_sum,&s,P_group,root=master);/* send sum to master */
```

Remember, a simple pseudocode is used throughout. Scatter and reduce (and gather when used) have many additional parameters in practice to include both source and destination (see Chapter 2, Section 2.2.4). Normally, the operation of a reduce routine will be specified as a parameter and not as part of the routine name as here. Using a parameter does allow different operations to be selected easily. Code will also be needed to establish the group of processes participating in the broadcast, scatter, and reduce.

Although we are adding numbers, many other operations could be performed instead. For example, the maximum number of the group could be found and passed back to the master in order for the master to find the maximum number of all those passed back to it. Similarly, the number of occurrences of a number (or character, or a string of characters) can be found in groups and passed back to the master.

Analysis. The sequential computation requires $n - 1$ additions with a time complexity of $O(n)$. In the parallel implementation, the total number of processes used is $m + 1$. Our analyses throughout separate communication and computation. It is easier to visualize if we also separate the actions into distinct phases. As with many problems, there is a communication phase followed by a computation phase, and these phases are repeated.

Phase 1 — Communication. First, we need to consider the communication aspect of the m slave processes reading their n/m numbers. Using individual send and receive routines requires a communication time of

$$t_{\text{comm1}} = m(t_{\text{startup}} + (n/m)t_{\text{data}})$$

where t_{startup} is the constant time portion of the transmission and t_{data} is the time to transmit one data word. Using scatter might reduce the number of startup times; i.e.,

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

[1]MPI has these, but PVM version 3 only has reduce.

depending upon the implementation of scatter. In any event, the time complexity is still O(n).

Phase 2 — Computation. Next, we need to estimate the number of computational steps. The slave processes each add n/m numbers together, requiring $n/m - 1$ additions. Since all m slave processes are operating together, we can consider all the partial sums obtained in the $n/m - 1$ steps. Hence, the parallel computation time of this phase is

$$t_{\text{comp1}} = n/m - 1$$

Phase 3 — Communication. Returning partial results using individual send and receive routines has a communication time of

$$t_{\text{comm2}} = m(t_{\text{startup}} + t_{\text{data}})$$

Using gather and reduce has:

$$t_{\text{comm2}} = t_{\text{startup}} + mt_{\text{data}}$$

Phase 4 — Computation. For the final accumulation, the master has to add the m partial sums, which requires $m - 1$ steps:

$$t_{\text{comp2}} = m - 1$$

Overall. The overall execution time for the problem (with gather and scatter) is

$$\begin{aligned} t_p &= (t_{\text{comm1}} + t_{\text{comm2}}) + (t_{\text{comp1}} + t_{\text{comp2}}) \\ &= (m(t_{\text{startup}} + (n/m)t_{\text{data}}) + t_{\text{startup}} + mt_{\text{data}}) + (n/m - 1 + m - 1) \\ &= (m + 1)t_{\text{startup}} + (n + m)t_{\text{data}} + m + n/m \end{aligned}$$

or

$$t_p = \text{O}(n + m)$$

We see that the parallel time complexity is worse than the sequential time complexity of O(n). Of course, if we consider only the computation aspect, the parallel formulation is better than the sequential formulation. Ignoring the communication aspect, the speedup factor, S, is given by

$$S = \frac{t_s}{t_p} = \frac{n - 1}{n/m + m - 2}$$

The speedup tends to m for large n. However, for smaller n, the speedup will be quite low and worsen for an increasing number of slaves, because the m slaves are idle during the second phase forming the final result.

Ideally, we want all the processes to be active all of the time, which cannot be achieved with this formulation of the problem. However, another formulation is helpful and is applicable to a very wide range of problems — namely, the divide-and-conquer approach.

4.1.2 Divide and Conquer

The divide-and-conquer approach is characterized by dividing a problem into subproblems that are of the same form as the larger problem. Further divisions into still smaller subproblems are usually done by recursion, a method well known to sequential programmers. The recursive method will continually divide a problem until the tasks cannot be broken down into smaller parts. Then the very simple tasks are performed and results combined, with the combining continued with larger and larger tasks. JáJá (1992) differentiates between when the main work is in dividing the problem and when the main work is combining the results (c.f., quicksort with mergesort). He categorizes the method as divide and conquer when the main work is combining the results, and categorizes the method as partitioning when the main work is dividing the problem. We will not make that distinction but will use the term *divide and conquer* anytime the partitioning is continued on smaller and smaller problems.

A sequential recursive definition for adding a list of numbers is[2]

```
int add(int *s)                          /* add list of numbers, s */
{
  if (number(s) =< 2) return (n1 + n2);   /* see explanation */
  else {
    Divide (s, s1, s2);      /* divide s into two parts, s1 and s2 */
    part_sum1 = add(s1);     /*recursive calls to add sub lists */
    part_sum2 = add(s2);
    return (part_sum1 + part_sum2);
  }
}
```

As in all recursive definitions, a method must be present to terminate the recursion when the division can go no further. In the code, `number(s)` returns the number of numbers in the list pointed to by `s`. If there are two numbers in the list, they are called `n1` and `n2`. If there is one number in the list, it is called `n1` and `n2` is zero. If there are no numbers, both `n1` and `n2` are zero. Separate `if` statements could be used for each of the cases: 0, 1, or 2 numbers in the list. Each would cause termination of the recursive call.

This method can be used for other global operations on a list, such as finding the maximum number. It can also be used for sorting a list by dividing the list into smaller and smaller lists to sort. Mergesort and quicksort sorting algorithms are usually described by such recursive definitions; see Kruse (1994). One would never actually use recursion to add a list of numbers when a simple iterative solution exists, but the following is applicable to any problem that is formulated by a recursive divide-and-conquer method.

When each division creates two parts, a recursive divide-and-conquer formulation forms a binary tree. The tree is traversed downward as calls are made and upward when the calls return (a preorder traversal given the recursive definition). A (complete) binary tree construction showing the "divide" part of divide and conquer is shown in Figure 4.2, with the final tasks at the bottom and the root at the top. The root process first divides the problem into two parts. These two parts are each divided into two parts, and so on until the leaves are reached. There the basic operations of the problem are performed. This construction can be used in the previous problem to divide the list of numbers first into two parts, then into four parts, and so on until each process has one equal part of the whole. After

[2]As in all of our pseudocode, implementation details are omitted. For example, the length of a list may need to be passed as a parameter.

adding pairs at the bottom of the tree, the accumulation occurs in a reverse tree construction.

Figure 4.2 shows a complete binary tree; that is, a perfectly balanced tree with all bottom nodes at the same level. This occurs if the task can be divided into a number of parts that is a power of 2. If not a power of 2, one or more bottom nodes will be at one level higher than the others. For convenience, we will assume that the task can be divided into a number of parts that is a power of 2, unless otherwise stated.

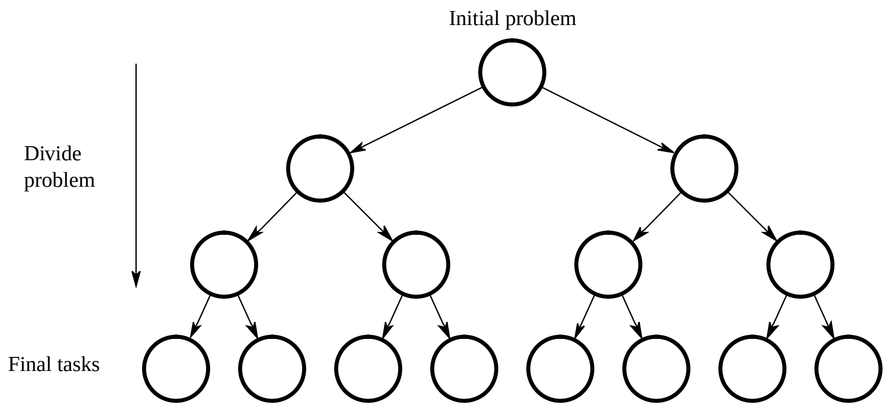

Figure 4.2 Tree construction.

Parallel Implementation. In a sequential implementation, only one node of the tree can be visited at a time. A parallel solution offers the prospect of traversing several parts of the tree simultaneously. Once a division is made into two parts, both parts can be processed simultaneously. Though a recursive parallel solution could be formulated, it is easier to visualize it without recursion. The key is realizing that the construction is a tree. One could simply assign one processor to each node in the tree. That would ultimately require $2^{m+1} - 1$ processors to divide the tasks into 2^m parts. Each processor would only be active at one level in the tree, leading to a very inefficient solution. (Problem 4-5 explores this method.)

A more efficient solution is to reuse processors at each level of the tree, as illustrated in Figure 4.3, which uses eight processors. The division stops when the total number of processors is committed. Until then, at each stage each processor keeps half of the list and passes on the other half. First, P_0 communicates with P_4, passing half of the list to P_4. Then P_0 and P_4 pass half of the list they hold to P_2 and P_6, respectively. Finally, P_0, P_2, P_4, and P_6 pass half of the list they hold to P_1, P_3, P_5, and P_7, respectively. Each list at the final stage will have $n/8$ numbers, or n/p in the general case of p processors. There are $\log p$ levels in the tree.

The "combining" act of summation of the partial sums can be done as shown in Figure 4.4. Once the partial sums have been formed, each odd-numbered processor passes its partial sum to the adjacent even-numbered processor; that is, P_1 passes its sum to P_0, P_3 to P_2, P_5 to P_4, and so on. The even-numbered processors then add the partial sum with its

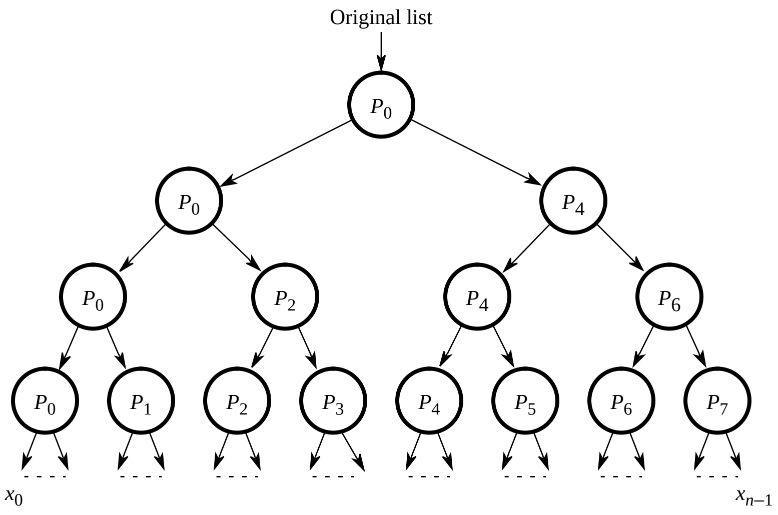

Figure 4.3 Dividing a list into parts.

own partial sum and pass the result onward, as shown. This continues until P_0 has the final result.

We can see that these constructions are the same as the binary hypercube broadcast and gather algorithms described in Chapter 2, Section 2.3.3. The constructions would map onto a hypercube perfectly but are also applicable to other systems. As with the hypercube broadcast/gather algorithms, processors that are to communicate with other processors can be found from their binary addresses. Processors communicate with processors whose addresses differ in one bit, starting with the most significant bit for the division phase and with the least significant bit for the combining phase (see Chapter 2, Section 2.3.3).

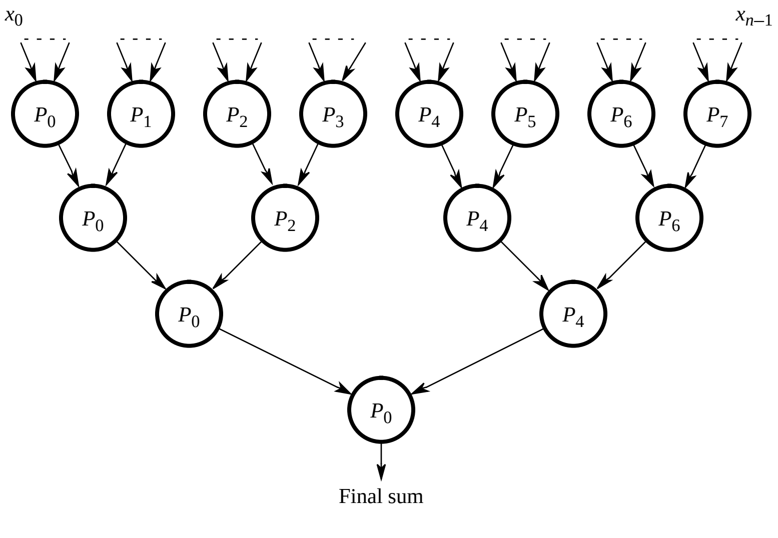

Figure 4.4 Partial summation.

Suppose we statically create eight processors (or processes) to add a list of numbers. The parallel code for process P_0 might take the form

Process P_0

```
                                  /* division phase */
divide(s1, s1, s2);               /* divide s1 into two, s1 and s2 */
send(s2, P4);                     /* send one part to another process */
divide(s1, s1, s2);
send(s2, P2);
divide(s1, s1, s2);
send(s2, P0};
part_sum = *s1;                   /* combining phase */
recv(&part_sum1, P0);
part_sum = part_sum + part_sum1;
recv(&part_sum1, P2);
part_sum = part_sum + part_sum1;
recv(&part_sum1, P4);
part_sum = part_sum + part_sum1;
```

The code for process P_4 might take the form

Process P_4

```
recv(s1, P0);                       /* division phase */
divide(s1, s1, s2);
send(s2, P6);
divide(s1, s1, s2);
send(s2, P5);
part_sum = *s1;                     /* combining phase */
recv(&part_sum1, P5);
part_sum = part_sum + part_sum1;
recv(&part_sum1, P6);
part_sum = part_sum + part_sum1;
send(&part_sum, P0);
```

Similar sequences are required for the other processes.

Analysis. We shall assume that n is a power of 2. The communication setup time, t_{startup}, is not included in the following for simplicity. It is left as an exercise to include the startup time.

The division phase essentially only consists of communication if we assume that dividing the list into two parts requires minimal computation. The combining phase requires both computation and communication to add the partial sums received and pass on the result.

Communication . There is a logarithmic number of steps in the division phase; i.e., $\log p$ steps with p processes. The communication time for this phase is given by

$$t_{\text{comm1}} = \frac{n}{2}t_{\text{data}} + \frac{n}{4}t_{\text{data}} + \frac{n}{8}t_{\text{data}} + \ldots + \frac{n}{p}t_{\text{data}} = \frac{n(p-1)}{p}t_{\text{data}}$$

where t_{data} is the transmission time for one data word. The time t_{comm1} is marginally better than a simple broadcast. The combining phase is similar, except only one data item is sent in each message (the partial sum); i.e.,

$$t_{\text{comm2}} = t_{\text{data}}\log p$$

for a total communication time of

$$t_{\text{comm}} = t_{\text{comm1}} + t_{\text{comm2}} = \frac{n(p-1)}{p}t_{\text{data}} + t_{\text{data}}\log p$$

or a time complexity of O(n) for constant p.

Computation. At the end of the divide phase, the n/p numbers are added together. Then one addition occurs at each stage during the combining phase, leading to

$$t_{\text{comp}} = \frac{n}{p} + \log p$$

again a time complexity of O(n) for constant p. For large n and variable p, we get O(n/p).

The total parallel execution time becomes

$$t_p = \frac{n(p-1)}{p}t_{\text{data}} + t_{\text{data}}\log p + \frac{n}{p} + \log p$$

assuming that p is a power of 2. The very best speedup we could expect with this method is, of course, p when all p processors are computing their partial sums. The actual speedup will be less than this due to the division and combining phases.

Clearly, another associative operator, such as subtraction, logical OR, logical AND, etc., can replace the addition operation in the previous example. The basic idea can also be applied to evaluating arithmetic expressions where operands are connected with an arithmetic operator. The tree construction can also be used for operations such as searching. In this case, the information passed upward is a Boolean flag indicating whether or not the specific item or condition has been found. The operation performed at each node is an OR operation, as shown in Figure 4.5.

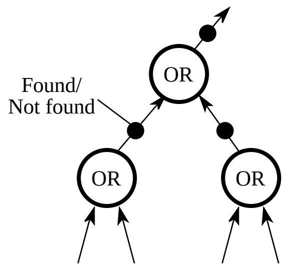

Figure 4.5 Part of a search tree.

4.1.3 *M*-ary Divide and Conquer

Divide and conquer can also be applied where a task is divided into more than two parts at each stage. For example, if the task is broken into four parts, the sequential recursive definition would be

```
int add(int *s)                    /* add list of numbers, s */
{
  if (number(s) =< 4) return(n1 + n2 + n3 + n4);
  else {
    Divide (s,s1,s2,s3,s4);        /* divide s into s1,s2,s3,s4*/
    part_sum1 = add(s1);           /*recursive calls to add sublists */
    part_sum2 = add(s2);
    part_sum3 = add(s3);
    part_sum4 = add(s4);
    return (part_sum1 + part_sum2 + part_sum3 + part_sum4);
  }
}
```

A tree in which each node has four children, as shown in Figure 4.6, is called a *quadtree*. A quadtree has particular applications in decomposing two-dimensional regions into four subregions. For example, a digitized image could be divided into four quadrants and then each of the four quadrants divided into four subquadrants, and so on, as shown in Figure 4.7. An *octtree* is a tree in which each node has eight children and has application for dividing a three-dimensional space recursively. An *m*-ary tree would be formed if the division is into *m* parts (i.e., a tree with *m* links from each node), which suggests that greater 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).

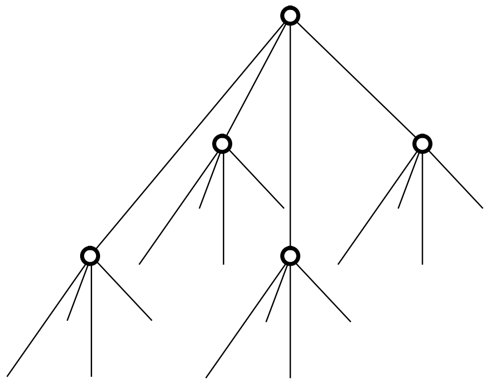

Figure 4.6 Quadtree.

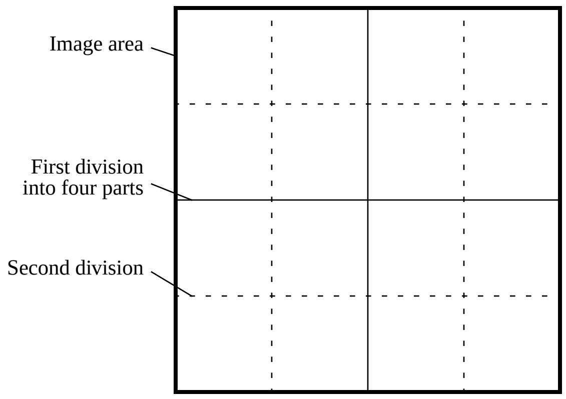

Figure 4.7 Dividing an image.

4.2 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 9. Let us look at one sorting algorithm here 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 could only contain one number, the method is similar to quicksort, except in quicksort the regions are divided into regions defined by "pivots" (see Chapter 9). 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 requires one 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 and use the result to identify the bucket from 0 to $m - 1$, one computational step for each number (although division can be rather expensive in time). If m is a power of 2, one can simply look at the upper bits of the number in binary. For example, if $m = 2^3$ (8), and the number is 1100101 in binary, it falls into region 110 (6), by considering the most

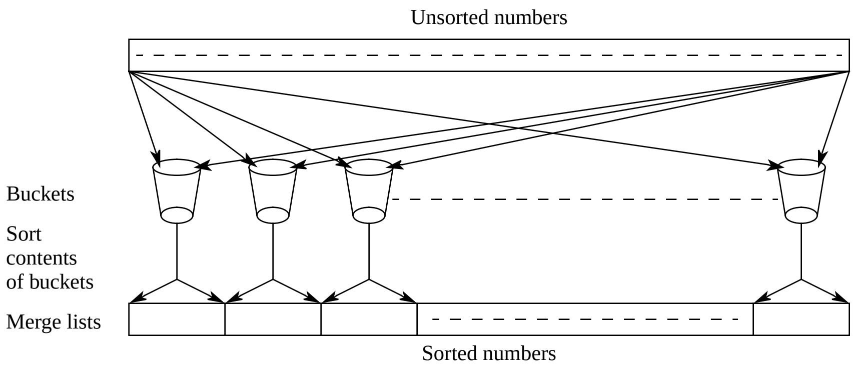

Figure 4.8 Bucket sort.

significant three bits. In any event, let us assume that placing a number into a bucket requires one step, and hence placing all the numbers requires n steps. If the numbers are uniformly distributed, there should be about 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

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

If $n = km$, where k is a constant, we get a time complexity of $O(n)$. Notice that this is much better than the lower bound for sequential compare and exchange sorting algorithms. However, it only applies when the numbers are well 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 processors actually remove numbers from the list into their buckets so that these numbers 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. Notice that this method is a simple partitioning method in which there is minimal work to create the partitions.

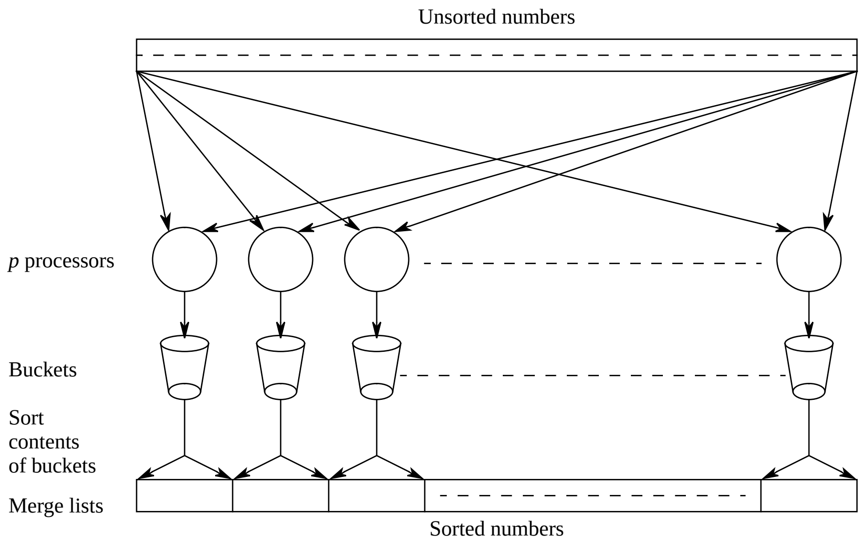

Figure 4.9 One parallel version of bucket sort.

The following phases are needed:

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

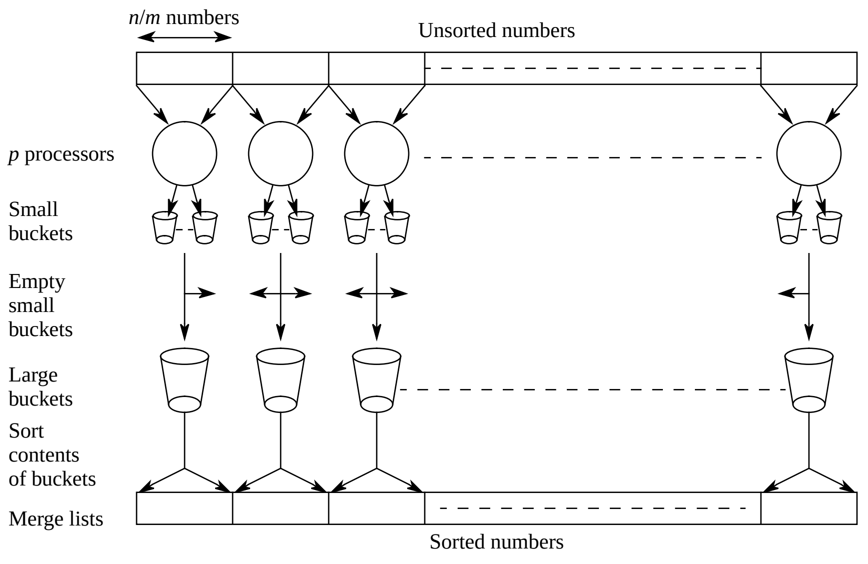

Figure 4.10 Parallel version of bucket sort.

Phase 1 — Computation and Communication. If we assume n computational steps to partition n numbers into p regions, then the computation is

$$t_{\text{comp1}} = n$$

After partitioning, the p partitions containing n/p numbers each are sent to the processes. Using a broadcast or scatter routine, the communication time is:

$$t_{\text{comm1}} = t_{\text{startup}} + t_{\text{data}}n$$

including the communication startup time.

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

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

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

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

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

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

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 and is 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 Chapter 9, Section 9.2.3).

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

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

Overall. The overall run time including communication is

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

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!

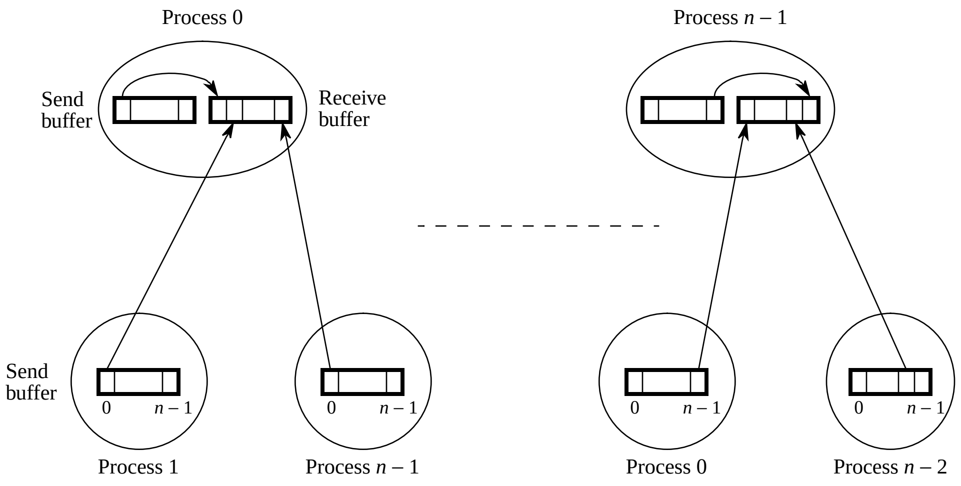

Figure 4.11 "All-to-all" broadcast.

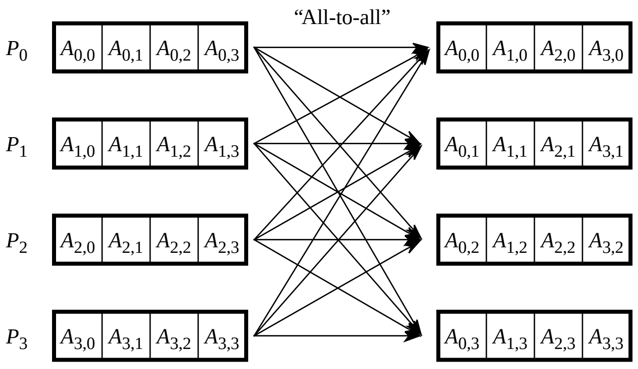
"All-to-all"
P_0
P_1
P_2
P_3
$A_{0,0}$ $A_{0,1}$ $A_{0,2}$ $A_{0,3}$
$A_{1,0}$ $A_{1,1}$ $A_{1,2}$ $A_{1,3}$
$A_{2,0}$ $A_{2,1}$ $A_{2,2}$ $A_{2,3}$
$A_{3,0}$ $A_{3,1}$ $A_{3,2}$ $A_{3,3}$
$A_{0,0}$ $A_{1,0}$ $A_{2,0}$ $A_{3,0}$
$A_{0,1}$ $A_{1,1}$ $A_{2,1}$ $A_{3,1}$
$A_{0,2}$ $A_{1,2}$ $A_{2,2}$ $A_{3,2}$
$A_{0,3}$ $A_{1,3}$ $A_{2,3}$ $A_{3,3}$

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

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 simple partitioning was employed. Sometimes 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, see Kumar et al. (1994).

A general divide-and-conquer technique divides the region continually into parts and lets some 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 construction has the advantage that the errors on each side of the midpoint end tend to cancel. Another more obvious construction is to use the actual intersections of the vertical lines with the function to create trapezoidal regions, as shown in Figure 4.15. Each region is now calculated as $1/2(f(p) + f(q))\delta$. Such approximate numerical methods for computing a definite integral using a linear combination of values are called *quadrature* methods.

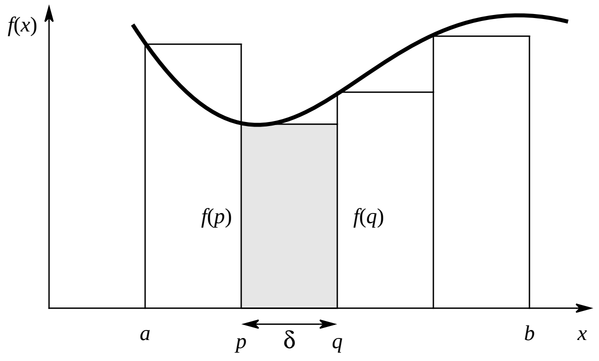

Figure 4.13 Numerical integration using rectangles.

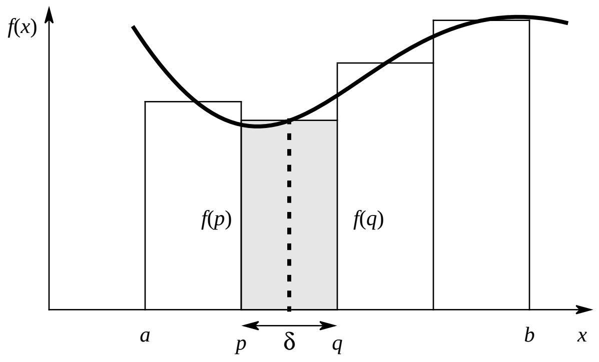

Figure 4.14 More accurate numerical integration using rectangles.

Static Assignment. Let us consider the *trapezoidal* method. Prior to the start of the computation, one process is statically assigned to be responsible for computing each region. By making the interval smaller, we come closer to attaining the exact solution.

Since each calculation is of the same form, the SPMD (single program multiple data) model is appropriate. Suppose we were to sum the area from $x = a$ to $x = b$ using p processes numbered 0 to $p - 1$. The size of the region for each process is $(b - a)/p$. To calculate the area in the described manner, a section of SPMD pseudocode could be

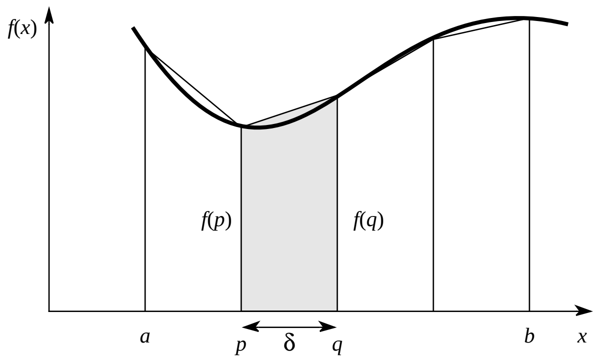

Figure 4.15 Numerical integration using the trapezoidal method.

Process P_i

```
if (i == master) {             /* read number of intervals required */
  printf("Enter number of intervals ");
  scanf(%d",&n);
}
bcast(&n, Pgroup);            /* broadcast interval to all processes */
region = (b - a)/p;            /* length of region for each process */
start = a + region * i;        /* starting x coordinate for process */
end = start + region;          /* ending x coordinate for process */
d = (b - a)/n;                               /* size of interval */
area = 0.0;
for (x = start; x < end; x = x + d)
  area = area + 0.5 * (f(x) + f(x+d)) * d;
reduce_add(&integral, &area, Pgroup);        /* form sum of areas */
```

A reduce operation is used to add the areas computed by the individual processes. For computational efficiency, computing each area is better if written as

```
area = 0.0;
for (x = start; x < end; x = x + d)
  area = area + f(x) + f(x+d);
area = 0.5 * area * d;
```

We assume that the variable `area` does not exceed the allowable maximum value (a possible disadvantage of this variation). For further efficiency, we can simplify the calculation somewhat by algebraic manipulation as follows:

$$\text{Area} = \frac{\delta(f(a) + f(a+\delta))}{2} + \frac{\delta(f(a+\delta) + f(a+2\delta))}{2} \ldots + \frac{\delta(f(a+(n-1)\delta) + f(b))}{2}$$

$$= \delta\left(\frac{f(a)}{2} + f(a+\delta) + f(2a+\delta) \ldots + f(a+(n-1)\delta) + \frac{f(b)}{2}\right)$$

given n intervals each of width δ. One implementation would be to use this formula for the region handled by each process:

```
area = 0.5 * (f(start) + f(end));
for (x = start + d; x < end; x = x + d)
  area = area + f(x);
area = area * d;
```

Adaptive Quadrature. The methods used so far are fine if we know beforehand the size of the interval δ that will give a sufficiently close solution. We also assumed that a fixed interval is used across the whole region. If a suitable interval is not known, some form of iteration is necessary to converge on the solution. For example, we could start with one interval and reduce it until a sufficiently close approximation is obtained. This implies that the area is recomputed with different intervals, so we cannot simply divide the total region into a fixed number of subregions, as in the summation example.

One approach is for each process to double the number of intervals successively until two successive approximations are sufficiently close. The tree construction could be used for dividing regions. The depth of the tree will be limited by the number of available processes/processors. In our example, it may be possible to allow the tree to grow in an unbalanced fashion as regions become computed to a sufficient accuracy. The phrase *sufficiently close* will depend upon the accuracy of the arithmetic and the application.

Another way to terminate is use three areas, *A*, *B*, and *C*, as shown in Figure 4.16. The computation is terminated when the area computed for the largest of the *A* and *B* regions is sufficiently close to the sum of the areas computed for the other two regions. For example, if region *B* is the largest, terminate when the area of *B* is sufficiently close to the area of *A* plus the area of *C*. Alternatively, we could simply terminate when *C* is sufficiently small. Such methods are known as *adaptive quadrature* because the solution adapts to the shape of the curve. (Simplified formulas can be derived for adaptive quadrature methods; see Freeman and Phillips, 1992.)

Computations of areas under slowly varying parts of the curve stop earlier than computations of areas under more rapidly varying parts. The spacing, δ, will vary across the interval. The consequence of this is that a fixed process assignment will not lead to the most efficient use of processors. The load-balancing techniques described in Chapter 3, Section 3.2.2, and in more detail in Chapter 7 are more appropriate. We should point out that some care might be needed in choosing when to terminate. For example, the function such as shown in Figure 4.17 might cause us to terminate early, as two large regions are the same (i.e., $C = 0$).

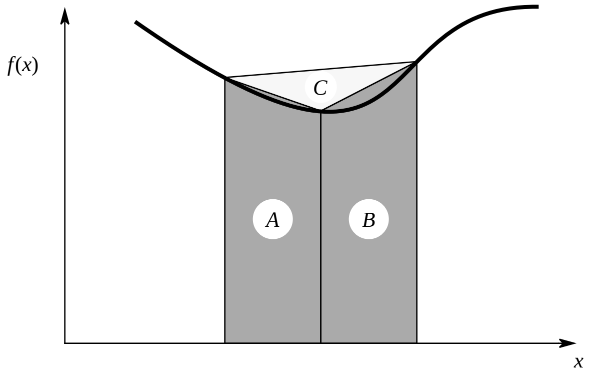

Figure 4.16 Adaptive quadrature construction.

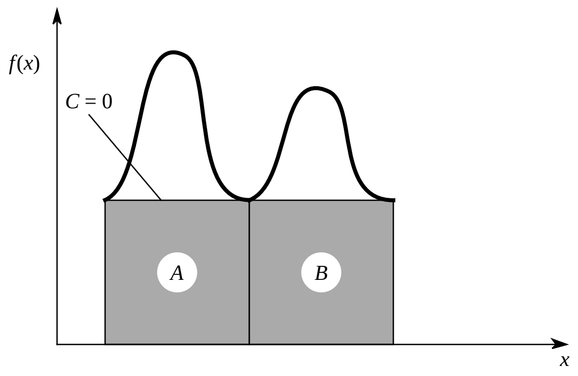

Figure 4.17 Adaptive quadrature with false termination.

4.2.3 *N*-Body Problem

The *N*-body problem is concerned with determining the effects of forces between “bodies” (for example, astronomical bodies that are attracted to each other through gravitational forces). The *N*-body problem appears in other areas, including molecular dynamics and fluid dynamics. Let us examine the problem in terms of astronomical systems, although the techniques apply to other applications. We provide the basic equations to enable the application to be coded as a programming exercise, which could use the same graphic routines as the Mandelbrot problem of Chapter 3 for interesting graphical output.

Gravitational *N*-Body Problem. The objective is to find the positions and movements of the bodies in space (say planets) that are subject to gravitational forces from other bodies using Newtonian laws of physics. The gravitational force between two bodies of masses m_a and m_b is given by

$$F = \frac{Gm_a m_b}{r^2}$$

where G is the gravitational constant and r is the distance between the bodies. We see that gravitational “forces” are described by an inverse square law. That is, the force between a pair of bodies is proportional to $1/r^2$, where r is the distance between the bodies. Each body will feel the influence of each of the other bodies according to the inverse square law, and the forces will sum together (taking into account the direction of each force). Subject to forces, a body will accelerate according to Newton’s second law:

$$F = ma$$

where m is the mass of the body, F is the force it experiences, and a is the resultant acceleration. Hence, all the bodies will move to new positions due to these forces and have new velocities. For a precise numeric description, differential equations would be used (that is, $F = mdv/dt$ and $v = dx/dt$, where v is the velocity). However, an exact “closed” solution to the *N*-body problem is not known for systems with greater than three bodies.

For a computer simulation, we use values at particular times, t_0, t_1, t_2 etc., the time intervals being as short as possible to achieve the most accurate solution. Let the time interval be Δt. Then, for a particular body of mass m, the force is given by

$$F = \frac{m(v^{t+1} - v^t)}{\Delta t}$$

and a new velocity

$$v^{t+1} = v^t + \frac{F\Delta t}{m}$$

where v^{t+1} is the velocity of the body at time $t + 1$ and v^t is the velocity of the body at time t. If a body is moving at a velocity v over the time interval Δt, its position changes by

$$x^{t+1} - x^t = v\Delta t$$

where x^t is its position at time t. Once bodies move to new positions, the forces change and the computation has to be repeated.

The velocity is not actually constant over the time interval, Δt, and hence only an approximate answer is obtained. It can help to use a "leap-frog" computation in which velocity and position are computed alternately; i.e.,

$$F^t = \frac{m(v^{t+1/2} - v^{t-1/2})}{\Delta t}$$

and

$$x^{t+1} - x^t = v^{t+1/2}\Delta t$$

where the velocities are computed for times t, $t + 1$, $t + 2$, etc. and the positions are computed for times $t + 1/2$, $t + 3/2$, $t + 5/2$, etc.

Three-Dimensional Space. Since the bodies are in a three-dimensional space, all values are vectors and have to be resolved into three directions, x, y, and z. In a three-dimensional space having a coordinate system (x, y, z), the distance between the bodies at (x_a, y_a, z_a) and (x_b, y_b, z_b) is given by

$$r = \sqrt{(x_b - x_a)^2 + (y_b - y_a)^2 + (z_b - z_a)^2}$$

The forces are resolved in the three directions, using, for example,

$$F_x = \frac{Gm_a m_b}{r^2}\left(\frac{x_b - x_a}{r}\right)$$

$$F_y = \frac{Gm_a m_b}{r^2}\left(\frac{y_b - y_a}{r}\right)$$

$$F_z = \frac{Gm_a m_b}{r^2}\left(\frac{z_b - z_a}{r}\right)$$

where the particles are of mass m_a and m_b and have the coordinates (x_a, y_a, z_a) and (x_b, y_b, z_b). Finally, the new position and velocity are computed. The velocity can also be resolved in three directions. For a simple computer solution, usually we assume a three-dimensional space with fixed boundaries. Actually, the universe is continually expanding and does not have fixed boundaries!

Other Applications. Although we describe the problem in terms of astronomical bodies, the concept can be applied to other situations. For example, charged particles are also influenced by each other, in this case according to Coulomb's electrostatic law (also an inverse square law of distance); particles of opposite charge are attracted and those of like charge are repelled. A subtle difference between the problem and astronomical bodies is that charged particles may move away from each other, whereas astronomical bodies are only attracted and hence will tend to cluster.

Sequential Code. The overall gravitational N-body computation can be described by the algorithm

```
for (t = 0; t < tmax; t++)            /* for each time period */
  for (i = 0; i < N; i++) {           /* for each body */
    F = Force_routine(i);             /* compute force on ith body */
    v[i]new = v[i] + F * dt;          /* compute new velocity and
    x[i]new = x[i] + v[i]new * dt;    /* new position (leap-flog) */
  }
for (i = 0; i < nmax; i++) {        /* for each body */
  x[i] = x[i]new;                   /* update velocity and position*/
  v[i] = v[i]new;
}
```

Parallel Code. Parallelizing the sequential algorithm code can use simple partitioning whereby groups of bodies are the responsibility of each processor, and each force is "carried" in distinct messages between processors. However, a large number of messages could result. The algorithm is an $O(N^2)$ algorithm (for one iteration) as each of the N bodies is influenced by each of the other $N - 1$ bodies. It is not feasible to use this direct algorithm for most interesting N-body problems where N is very large.

The time complexity can be reduced using the observation that a cluster of distant bodies can be approximated as a single distant body of the total mass of the cluster sited at the center of mass of the cluster, as illustrated in Figure 4.18. This clustering idea can be applied recursively.

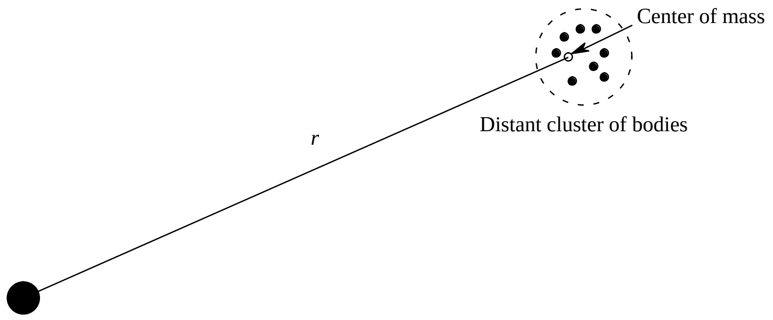

Figure 4.18 Clustering distant bodies.

Barnes-Hut Algorithm. A clever divide-and-conquer formation to the problem using this clustering idea starts with the whole space in which one cube contains the bodies (or particles). First, this cube is divided into eight subcubes. If a subcube contains no particles, the subcube is deleted from further consideration. If a subcube contains more than one body, it is recursively divided until every subcube contains one body. This process creates an *octtree*; that is, a tree with up to eight edges from each node. The leaves represent cells each containing one body. (We assumed the original space is a cube so that cubes result at each level of recursion, but other assumptions are possible.)

For a two-dimensional problem, each recursive subdivision will create four subareas and a *quadtree* (a tree with up to four edges from each edge; see Section 4.1.3). In general, the tree will be very unbalanced. Figure 4.19 illustrates the decomposition for a two-dimensional space (which is easier to draw) and the resultant quadtree. The three-dimensional case follows the same construction except with up to eight edges from each node.

In the *Barnes-Hut algorithm* (Barnes and Hut, 1986), after the tree has been constructed, the total mass and center of mass of the subcube is stored at each node. The force on each body can then be obtained by traversing the tree starting at the root, stopping at a node when the clustering approximation can be used for the particular body, and otherwise continuing to traverse the tree downward. In astronomical *N*-body simulations, a simple criterion for when the approximation can be made is as follows. Suppose the cluster is enclosed in a cubic volume given by the dimensions $d \times d \times d$, and the distant to the center of mass is *r*. Use the clustering approximation when

$$r \geq \frac{d}{\theta}$$

where θ is a constant typically 1.0 or less (θ is called the opening angle). This approach can substantially reduce the computational effort.

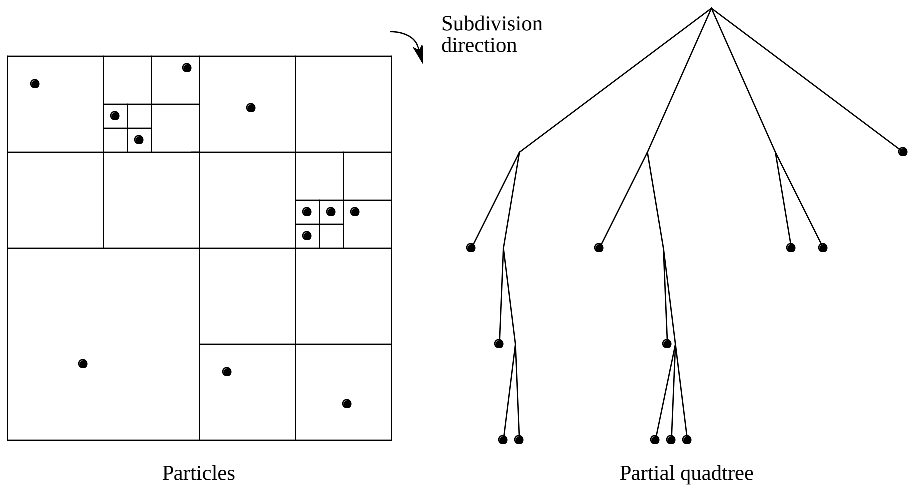

Figure 4.19 Recursive division of two-dimensional space.

Once all the bodies have been given new positions and velocities, the process is repeated for each time period. This means that the whole octtree must be reconstructed for each time period (because the bodies have moved). Constructing the tree requires a time of O($n\log n$), and so does computing all the forces, so that the overall time complexity of the method is O($n\log n$) (Barnes and Hut, 1986).

The algorithm can be described by the following:

```
for (t = 0; t < tmax; t++) {     /* for each time period */
   Build_Octtree();              /* construct Octtree (or Quadtree) */
   Tot_Mass_Center();            /* compute total mass & center /*
   Comp_Force();                 /* traverse tree/computing forces */
   Update();                     /* update position/velocity */
}
```

The `Build_Octtree()` routine can be constructed from the positions of the bodies, considering each body in turn. The `Tot_Mass_Center()` routine must traverse the tree, computing the total mass and center of mass at each node. This could be done recursively. The total mass, M, is given by the simply sum of the total masses of the children:

$$M = \sum_{i=0}^{7} m_i$$

where m_i is the total mass of the ith child. The center of mass, C, is given by

$$C = \frac{1}{M}\sum_{i=0}^{7} (m_i \times c_i)$$

where position of the centers of mass have three components, in the x, y, and z directions. The `Comp_Force()` routine must visit nodes ascertaining whether the clustering approximation can be applied to compute the force of all the bodies in that cell. If the clustering approximation cannot be applied, the children of the node must be visited.

The octtree will, in general, be very unbalanced, and its shape changes during the simulation. Hence, a simple static partitioning strategy will not be very effective in load balancing. A better way of dividing the bodies into groups is called *orthogonal recursive bisection* (Salmon, 1990). Let us describe this method in terms of a two-dimensional square area. First, a vertical line is found that divides the area into two areas each with an equal number of bodies. For each area, a horizontal line is found that divides it into two areas each with an equal number of bodies. This is repeated until there are as many areas as processors, and then one processor is assigned to each area. An example of the division is illustrated in Figure 4.20.

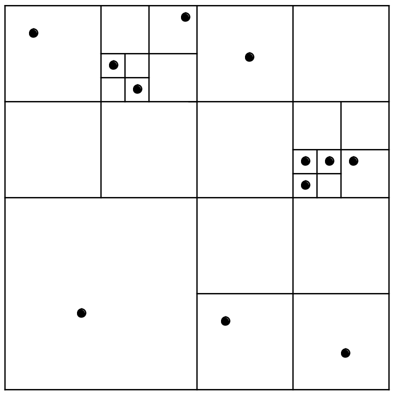

Figure 4.20 Orthogonal recursive bisection method.

4.3 SUMMARY

This chapter introduced the following concepts:

- Partitioning and divide-and-conquer concepts as the basis for parallel computing techniques
- Tree constructions
- Examples of partitioning and divide-and-conquer problems — namely, bucket sort, numerical integration, and the *N*-body problem

FURTHER READING

The divide-and-conquer technique is described in many sequential programming texts (for example, Kruse, 1994). As we have seen, this technique results in a tree structure. It is possible to construct a multiprocessor with a tree network, which would then be amenable to divide-and-conquer problems. One or two tree network machines have been constructed with the thought that most applications can be formulated as divide and conquer. However, as we have seen in Chapter 1, trees can be embedded into meshes and hypercubes so that it is not necessary to have tree network. Mapping divide-and-conquer algorithms onto different architectures is the subject of research papers, such as that done by Lo and Rajopadhye (1990).

Once a problem is partitioned, in some contexts a scheduling algorithm is appropriate for allocating processors to partitions or processes. Several texts explore scheduling, including Quinn (1994), Lewis and El-Rewini (1992), Moldovan (1993), and Sarkar (1989). Mapping (static scheduling) is not considered in this text. However, dynamic load balancing, in which tasks are assigned to processors during the execution of the program, is considered in Chapter 7.

Bucket sort is described in texts on sorting algorithms (see Chapter 9) and can be found specifically in Lindstrom (1985) and Wagner and Han (1986). Numerical evaluation of integrals in the context of parallel programs can be found in Freeman and Phillips (1992), Gropp, Lusk, and Skjellum (1994), Lester (1993), and Smith (1993) and is often used as a simple application of parallel programs. The original source for the Barnes-Hut algorithm is Barnes and Hut (1986). Other papers include Bhatt et al. (1992) and Warren and Salmon (1992). Liu and Wu (1997) consider programming the algorithm inn C++. Apart from the Barnes-Hutt divide-and-conquer algorithm, another approach is the fast multipole method (Greengard and Rokhlin, 1987). Hybrid methods exist.

BIBLIOGRAPHY

AHO, A. V., J. E. HOPCROFT, AND J. D. ULLMAN (1974), *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Massachusetts.

BARNES, J. E., AND P. HUT (1986), “A Hierarchical O($N\log N$) Force Calculation Algorithm,” *Nature*, Vol. 324, No. 4 (December), pp. 446–449.

BHATT, S., M. CHEN, C. Y. LIN, AND P. LIU (1992), “Abstractions for Parallel *N*-Body Simulations,” *Proc. Scalable High Performance Computing Conference*, pp. 26–29.

BLELLOCH, G. E. (1996), “Programming Parallel Algorithms,” *Comm. ACM*, Vol. 39, No. 3, pp. 85–97.

BOKHARI, S. H. (1981), “On the Mapping Problem,” *IEEE Trans. Comput.*, Vol. C-30, No. 3, pp. 207–214.

FREEMAN, T. L., AND C. PHILLIPS (1992), *Parallel Numerical Algorithms*, Prentice Hall, London.

GREENGARD, L., AND V. ROKHLIN (1987), “A Fast Algorithm for Particle Simulations,” *J. Comp. Phys.*, Vol. 73, pp. 325–348.

GROPP, W., E. LUSK, AND A. SKJELLUM (1994), *Using MPI Portable Parallel Programming with the Message-Passing Interface*, MIT Press, Cambridge, Massachusetts.

JÁJÁ, J. (1992), *An Introduction to Parallel Algorithms*, Addison Wesley, Reading, Massachusetts.

KRUSE, R. L. (1994), *Data Structures and Program Design*, 3rd ed., Prentice Hall, Englewood Cliffs, New Jersey.

KUMAR, V., A. GRAMA, A. GUPTA, AND G. KARYPIS (1994), *Introduction to Parallel Computing*, Benjamin/Cummings, Redwood City, California.

LESTER, B. (1993), *The Art of Parallel Programming*, Prentice Hall, Englewood Cliffs, New Jersey.

LEWIS, T. G., AND H. EL-REWINI (1992), *Introduction to Parallel Computing*, Prentice Hall, Englewood Cliffs, New Jersey.

LINDSTROM, E. E. (1985), “The Design and Analysis of BucketSort for Bubble Memory Secondary Storage,” *IEEE Trans. Comput.*, Vol. C-34, No. 3, pp. 218–233.

LIU, P., AND J.-J. WU (1997), “A Framework for Parallel Tree-Based Scientific Simulations,” *Proc. 1997 Int. Conf. Par. Proc.*, pp. 137–144.

LO, V. M., AND S. RAJOPADHYE (1990), “Mapping Divide-and-Conquer Algorithms to Parallel Architectures,” *Proc. 1990 Int. Conf. Par. Proc.*, Part III, pp. 128–135.

MILLER, R., AND Q. F. STOUT (1996), *Parallel Algorithms for Regular Architectures: Meshes and Pyramids*, MIT Press, Cambridge, Massachusetts.

MOLDOVAN, D. I. (1993), *Parallel Processing from Applications to Systems*, Morgan Kaufmann, San Mateo, California.

PREPARATA, F. P., AND M. I. SHAMOS (1985), *Computational Geometry: An Introduction*, Springer-Verlag, New York.

QUINN, M. J. (1994), *Parallel Computing Theory and Practice*, McGraw-Hill, New York.

SALMON, J. K. (1990), *Parallel Hierarchical N-Body Methods*, Ph.D. thesis, California Institute of Technology.

SARKAR, V. (1989), *Partitioning and Scheduling Parallel Programs for Multiprocessing*, MIT Press, Cambridge, Massachusetts.

SMITH, J. R. (1993), *The Design and Analysis of Parallel Algorithms*, Oxford UP, Oxford.

WAGNER, R. A., AND Y. HAN (1986), "Parallel Algorithms for Bucket Sorting and Data Dependent Prefix Problem," *Proc. 1986 Int. Conf. Par. Proc.*, pp. 924–929.

WARREN, M., AND J. SALMON (1992), "Astrophysical *N*-Body Simulations Using Hierarchical Tree Data Structures," *Proc. Supercomputing 92*, IEEE CS Press, Los Alamitos, pp. 570–576.

PROBLEMS

Scientific/Numerical

4-1. Write a program that will prove that the maximum speedup of adding a series of numbers using a simple partition described in Section 4.1.1 is $m/2$, where there are m processes.

4-2. Using the equations developed in Section 4.1.1 for partitioning a list of numbers into m partitions that are added separately, show that the optimum value for m to give the minimum parallel execution time is when $m = \sqrt{p/(1 + t_{\text{startup}})}$, where there are p processors. (Clue: Differentiate the parallel execution time equation.)

4-3. Section 4.1.1 gives three implementations of adding numbers, using separate `send()`s and `recv()`s, using a broadcast routine with separate `recv()`s to return partial results and using scatter and reduce routines. Write parallel programs for all three implementations, instrumenting the programs to extract timing information (Chapter 2, Section 2.3.4), and compare the results.

4-4. Suppose the structure of a computation consists of a binary tree with n leaves (final tasks) and $\log n$ levels. Each node in the tree consists of one computational step. What is the lower bound of the execution time if the number of processors is less than n?

4-5. Analyze the divide-and-conquer method of assigning one processor to each node in a tree for adding numbers (Section 4.1.2) in terms of communication, computation, overall parallel execution time, speedup, and efficiency.

4-6. Complete the parallel pseudocode given in Section 4.1.2 for the (binary) divide-and-conquer method for all eight processes.

4-7. Develop the equations for computation and communication times for m-ary divide and conquer, following the approach used in Section 4.1.2.

4-8. Develop a divide-and-conquer algorithm that finds the smallest value in a set of n values in $O(\log n)$ steps using $n/2$ processors. What is the time complexity if there are fewer than $n/2$ processors?

4-9. Write a parallel program with a time complexity of $O(\log n)$ to compute the polynomial

$$f = a_0x^0 + a_1x^1 + a_2x^2 + \ldots + a_{n-1}x^{n-1}$$

to any degree, n, where the a's, x, and n are input.

4-10. Write a parallel program that uses a divide-and-conquer approach to find the first zero in a list of integers stored in an array. Use 16 processes and 256 numbers.

4-11. Write parallel programs to compute the summation of *n* integers in each of the following ways and assess their performance. Assume that *n* is a power of 2.

(a) Partition the *n* integers into *n*/2 pairs. Use *n*/2 processes to add together each pair of integers resulting in *n*/2 integers. Repeat the method on the *n*/2 integers to obtain *n*/4 integers and continue until the final result is obtained. (This is a binary tree algorithm.)

(b) Divide the *n* integers into *n*/log *n* groups of log *n* numbers each. Use *n*/log *n* processes, each adding the numbers in one group sequentially. Then add the *n*/log *n* results using method (a). This algorithm is shown in Figure 4.21.

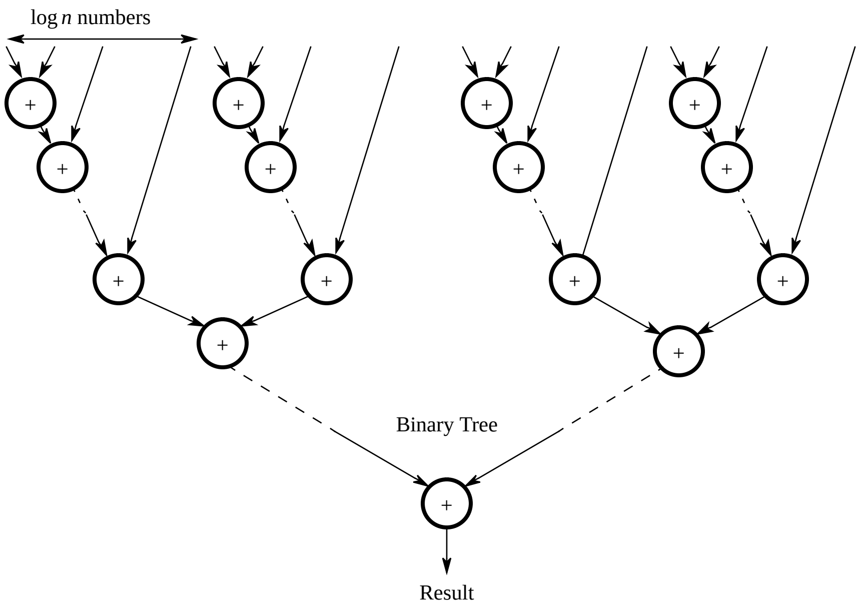

Figure 4.21 Process diagram for Problem 4-12(b).

4-12. Write parallel programs to compute *n*! in each of the following ways and assess their performance. The number, *n*, may be odd or even but is a positive constant.

(a) Compute *n*! using two concurrent processes, each computing approximately half of the complete sequence. A master process then combines the two partial results.

(b) Compute *n*! using a producer process and a consumer process connected together. The producer produces the numbers 1, 2, 3, … *n* in sequence. The consumer accepts the sequence of numbers from the producer and accumulates the result; i.e., $1 \times 2 \times 3 \ldots$.

4-13. Write a divide-and-conquer parallel program that determines whether the number of 1's in a binary file is even or odd (i.e., create a parity checker). Modify the program so that a bit is attached to the contents of the file, and set to a 0 or a 1 to make the number of 1's even (a parity generator).

4-14. One way to compute π is to compute the area under the curve $f(x) = 4/(1 + x^2)$ between 0 and 1, which is numerically equal to π. Write a parallel program to calculate π this way using 10 processes. Another way to compute π is to compute the area of a circle of radius $r = 1$ (i.e., $\pi r^2 = \pi$). Determine the appropriate equation for a circle, and write a parallel program to compute π this way. Comment on the two ways of computing π.

4-15. Derive a formula to evaluate numerically an integral using the adaptive quadrature method described in Section 4.2.2. Use the approach given for the trapezoidal method.

4-16. Using any method, write a parallel program that will compute the integral

$$I = \int_{0.01}^{1}\left(x + \sin\left(\frac{1}{x}\right)\right)dx$$

4-17. Write a static assignment parallel program to compute π using the formula

$$\int_0^1 \sqrt{1-x^2}\,dx = \frac{\pi}{4}$$

using each of the following ways:

1. Rectangular decomposition, as illustrated in Figure 4.13
2. Rectangular decomposition, as illustrated in Figure 4.14
3. Trapezoidal decomposition, as illustrated in Figure 4.15

Evaluate each method in terms of speed and accuracy.

4-18. Find the zero crossing of a function by a bisection method. In this method, two points on the function are computed, say *f*(*a*) and *f*(*b*), where *f*(*a*) is positive and *f*(*b*) is negative. The function must cross somewhere between *f*(*a*) and *f*(*b*), as illustrated in Figure 4.22. By

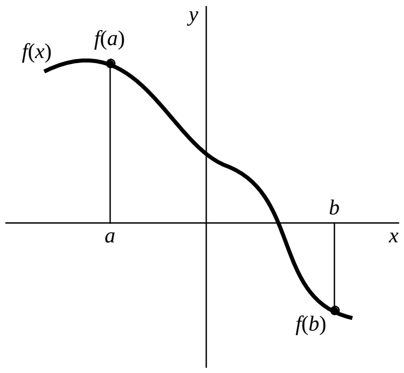

Figure 4.22 Bisection method for finding the zero crossing location of a function.

successively dividing the interval, the exact location of the zero crossing can be found. Write a divide-and-conquer program that will find the zero crossing locations of the function $f(x) = x^2 - 3x + 2$. (This function has two zero crossing locations, $x = 1$ and $x = 2$.)

4-19. Write a parallel program to integrate a function using Simpson's rule, which is given as follows:

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

$$\frac{\delta}{3}[f(a) + 4f(a+\delta) + 2f(a+2\delta) + 4f(a+3\delta) + 2f(a+4\delta) + \ldots 4f(a+(n-1)\delta) + f(b)]$$

where δ is fixed [$\delta = (b - a)/n$ and n must be even]. Choose a suitable function (or arrange it so that the function can be input).

4-20. Write a sequential program and a parallel program to simulate an astronomical *N*-body system, but in two-dimensions. The bodies are initially at rest. Their initial positions and masses are to be selected randomly (using a random number generator). Display the movement of the bodies using the graphical routines used for the Mandelbrot program found in http://

www.cs.uncc.edu/par_prog, or otherwise, showing each body in a color and size to indicate its mass.

4-21. Develop the *N*-body equations for a system of charged particles (e.g., free electrons and positrons) that are governed by Coulumb's law. Write a sequential and a parallel program to model this system, assuming that the particles lie in a two-dimensional space. Produce graphical output showing the movement of the particles. Provide your own initial distribution and movement of particles and solution space.

4-22. (Research problem) Given a set of *n* points in a plane, develop an algorithm and parallel program to find the points that are on the perimeter of the smallest region containing all of the points, and join the points, as illustrated in Figure 4.23. This problem is known as the planar convex hull problem and can be solved by a recursive divide-and-conquer approach very similar to quicksort, by recursively splitting regions into two parts using "pivot" points. There are several sources for information on the planar convex hull problem, including Blelloch (1996), Preparata and Shamos (1985), and Miller and Stout (1996).

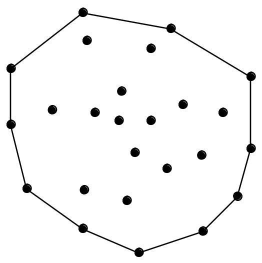

Figure 4.23 Convex hull (Problem 4-22).

Real Life

4-23. Write a sequential and a parallel program to model the planets around the sun (or another astronomical system). Produce graphical output showing the movement of the planets. Provide your own initial distribution and movement of planets. (Use real data if available.)

4-24. A major bank in your state processes an average of 30 million checks a day for its 2 million customer accounts. One time-consuming problem is that of sorting the checks into individual customer-account bundles so they can be returned with the monthly statements. (Yes, the bank handles check sorting for several client banks in addition to its own.) The bank has been using a very fast mainframe-based check sorter and the quicksort method. However, you have told the bank that you know of a way to use *N* smaller computers in parallel, with each sorting 1/*N*th of the 30 million checks, and then merge those partial sorts into a single sorted result. Prior to the bank actually investing in the new technology, you have been hired as a consultant to simulate the process using message-passing parallel programming. Under the following assumptions, simulate this new approach for the bank.

Assumptions:

1. Each check has three identification numbers: a nine-digit bank-identification number, a nine-digit account-identification number, and a three-digit check number (leading zeros are not printed or shown).
2. All checks with the same bank-identification number are to be sorted by customer account for transmittal to that client bank.

Estimate the speedup if *N* is 10; if *N* is 1000. Estimate the percentage of time spent in communications versus time spent in processing.

4-25. Sue, 21 years old, comes from a very financially astute family. She has been watching her parents save and invest for several years now, reads the *Wall Street Journal* daily in the university library (for free!), and has concluded that she will not be able to rely on social security when she retires in 49 years. For graduation from college, her parents got her a CD-ROM containing historical daily closing prices covering every exchange-listed security, from January 1, 1900 to the end of last month.

For simplicity you may think of the data on the CD-ROM as being organized into date/symbol/closing price records for each of the 358,000 securities that have been listed since 1900. (Only a fraction are listed at any given date; firms go out of business and new ones start daily.) Similarly, you may assume that the format of a record is given by

date	Last three digits of the year, followed by the "Julian date" (where January 15 is Julian 15, February 1 is Julian 32, etc.)
symbol	Up to 10 characters, such as PCAI, KAUFX, or IBM.AZ, representing a NASDAQ stock (PCA International), a mutual fund (Kaufman Aggressive Growth), and an option to buy IBM stock at a particular price for a particular length of time, respectively.
closing price	Three integers, *X* (representing a whole number of dollars per unit), *Y* (representing the numerator of a fractional dollar per unit), and *Z* (representing the denominator of a fractional dollar per unit).

For example, "996033/PCAI/10/3/4" indicates that on February 2, 1996, PCA International stock closed at $10.75 per share. Sue wants to know how many of the stocks that were listed as of last month's end have had 50 or more consecutive trading days in which they closed either unchanged from the previous day or closed at a higher price, anytime in the CD-ROM's "recorded history."

4-26. The more Samantha recalled her grandfather's stories about the time he won the 1963 World Championship Dominos Match, the more she wanted to improve her skills at the game. She had a basic problem though; she had no playing partners left, having already improved to the point where she consistently won every game against the few friends who still remained!

Samantha knew that computerized versions of Go, chess, bridge, poker, and checkers had been developed, and saw no reason someone skilled in the science of computers could not do the same for dominos. One of her computer science professors at the new campus of the University of Canada, U-Can-II, had told her she could do anything she wanted (within theoretical limitations, of course), and she *really* wanted to win that next World Championship!

Pulling out her slow, old, nearly obsolete 300 MHz/64 Meg (RAM)/6 Gbyte (disk) Pentium, she quickly developed a straightforward, single processor simulator that she could practice against. The basic outline of her approach was to have the program compare every one of its pieces to the pieces already played in order to determine the computer's best move. This appeared to involve enough computation, including rotations and trial placements of pieces, that Samantha found herself waiting for the program to produce the computer's next move, and becoming as bored with its game performance as with that of her old friends. Thus, she is seeking your assistance in developing a parallel processor version.

1. Outline her single processor algorithm.
2. Outline your parallel processor algorithm.
3. Estimate the speedup that could be obtained if you were to network 50 old computers like hers, and make a recommendation to her about either going ahead with the task or spending $2500 to buy the new 800 MHz "Octium," which is reputed to be 50 times faster than her old Pentium for these kinds of simulations.

4-27. Area, Inc., provides a numerical integration service for several small engineering firms in the region. When any of those firms has a continuous function defined over a domain and is unable to integrate it, Area, Inc., gets the call. You have just been hired to help Area, Inc., improve its slow delivery of computed integration results. Area, Inc. has lost money each year of its existence and is so "nonprofit" that payment of next week's payroll is in question. Given your desire to continue eating (and for that to continue, Area, Inc., has to pay you), you have considerable incentive to help Area, Inc.

Given also that you have a considerable background in parallel computing, you recognize the problem immediately: Area, Inc., has been using a single processor to implement a standard numerical integration algorithm.

Step 1: Divide the independent axis into *N* even intervals.

Step 2: Approximate the area under the function in any interval (its integral over that interval), by the product of the interval width times the function value when it is evaluated at the left edge of the interval.

Step 3: Add up all *N* approximations to get the total area.

Step 4: Divide the interval width in half.

Step 5: Repeat steps 1 – 4 until the total from the *i*th repetition differs from the $(i-1)th$ repetition by less than 0.001% of the magnitude of the *i*th total.

Since your manager is skeptical about new-fangled parallel computing approaches, she wants you to simulate two different machine configurations: two processors in the first, and eight processors in the second. She has told you that a successful demonstration is key to being able to buy more processors and to your getting paid next week.