CHAPTER 14

Parallel Algorithms

14.1

IN MANY INSTANCES THE BEST ALGORITHM for a parallel computer is not a parallelized serial algorithm. Rather it is an entirely new algorithm specifically designed for parallel computers. In this chapter we'll discuss a couple of parallel algorithms. Our purpose is not to provide an overview of the field; such an effort would require at least as much space as the rest of the book. Rather, we want to illustrate some of the issues in parallel algorithm design and give some indication of the breadth of the field.

Designing a Parallel Algorithm

Parallel algorithms can be roughly divided into two basic categories: those that are obtained by a direct parallelization of a standard serial algorithm, and those that either have no serial counterpart or have been obtained in an indirect way from a serial algorithm. The parallel trapezoidal rule program and the parallel Jacobi's method program are examples of direct parallelization, while the hypercube allgather function has no serial counterpart, and Fox's algorithm might be viewed as having been obtained in an indirect way from the standard serial matrix multiplication program. When we're designing parallel algorithms, we need to keep both approaches in mind. In most cases the design and development cost of programs that use the direct route will be substantially less. However, as a number of our examples illustrate, if we follow the indirect route, the improvement in performance can be substantial.

Our first example illustrates the indirect route. We design and code an algorithm for sorting that is based on a little known and not very efficient serial sorting algorithm. The reason we choose this indirect route is that the direct route doesn't produce a very good parallel algorithm. In our second example,

we take the direct route. We design an algorithm for tree searching that is a natural extension of serial depth-first search. The development illustrates the immense care needed to successfully parallelize a fairly simple and standard serial algorithm.

In our discussions we'll focus on coding and correctness. We leave it to you to decide whether the algorithms meet the criteria we've outlined in Chapters 11–12, i.e., performance, scalability, load balance, etc.

14.2 Sorting

In Chapter 10 we wrote a simple program for parallel sorting. It made the possibly unrealistic assumption that we knew the distribution of the keys to be sorted before we actually sorted them. In this section we'll discuss a parallel algorithm for sorting that doesn't make this assumption. We'll assume that there are n keys to be sorted, and, when the algorithm begins, each process is assigned n/p keys. When the algorithm completes, each process should contain a sorted subset of n/p keys, and if the rank of process q is less than the rank of process r, then each key on process q should be less than or equal to every key on process r.

Quicksort, the "standard" serial algorithm for sorting, might, at first glance, seem to be a good choice for the basis of a parallel algorithm for sorting. Recall that quicksort is a divide-and-conquer algorithm: It chooses a pivot from the list of keys, and splits the list into two sublists: the first sublist consists of keys less than or equal to the pivot, the second consists of keys greater than the pivot. It then calls itself recursively on the two sublists. The obvious parallelization would choose a global pivot and broadcast it to all the processes. It would then assign the first sublist to the processes with rank less than or equal to $p/2$, and the second sublist to the processes with rank greater than $p/2$. However, unless we have some a priori information on the distribution of the keys, there is no guarantee that the first sublist contains the same number of elements as the second, and the algorithm can quickly lead to serious imbalances. So we'll look at a parallel algorithm that is not based on serial quicksort.

14.3 Serial Bitonic Sort

The bitonic sorting algorithm is based on the idea of a sorting network. For further information on this approach, see the references at the end of the chapter. We'll simply view it as a (somewhat unusual) serial algorithm.

Recollect that a **monotonic** sequence is one that either increases or decreases, but not both. The word "bitonic" was coined to describe sequences that increase and then decrease. More formally, a sequence of keys $(a_0, a_1, \ldots, a_{n-1})$ is **bitonic** if

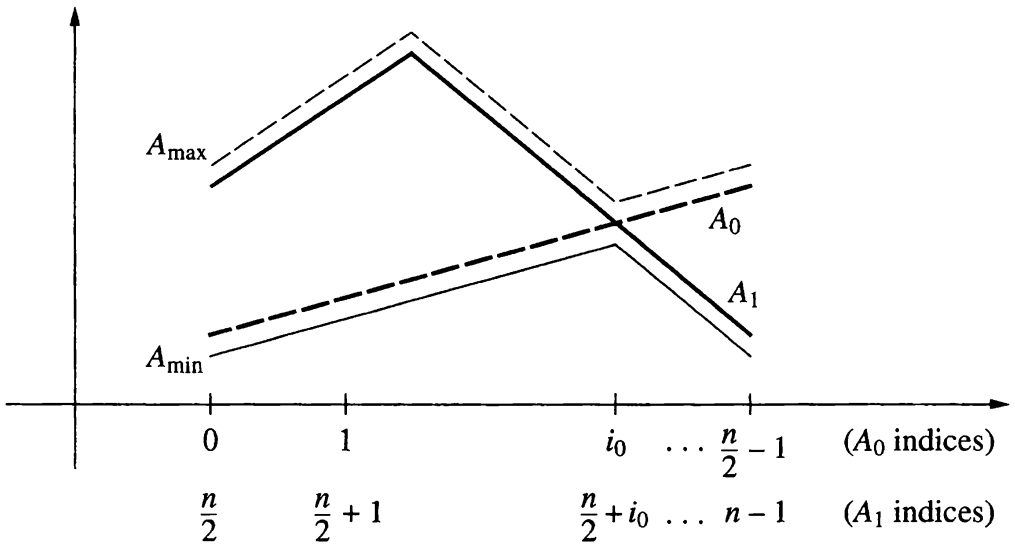

Figure 14.1 A bitonic split

1. there exists an index $m, 0 \leq m \leq n-1$, such that

$$a_0 \leq a_1 \leq \cdots \leq a_m \geq a_{m+1} \geq \cdots \geq a_{n-1},$$

or

2. there exists a cyclic shift σ of $(0, 1, \ldots, n-1)$ such that the sequence $(a_{\sigma(0)}, a_{\sigma(1)}, \ldots, a_{\sigma(n-1)})$ satisfies condition 1. A cyclic shift sends each index i to $(i + s)$ mod n, for some integer s.

Thus, the sequence $(1, 5, 6, 9, 8, 7, 3, 0)$ is bitonic, as is the sequence $(6, 9, 8, 7, 3, 0, 1, 5)$, since it can be obtained from the first by a cyclic shift.

Now observe that if the sequence $A = (a_0, a_1, \ldots, a_{n-1})$ is bitonic, then we can form two bitonic sequences from A as follows. Define

$$A_{min} = (\min\{a_0, a_{n/2}\}, \min\{a_1, a_{n/2+1}\} \ldots, \min\{a_{n/2-1}, a_{n-1}\}),$$

and

$$A_{max} = (\max\{a_0, a_{n/2}\}, \max\{a_1, a_{n/2+1}\} \ldots, \max\{a_{n/2-1}, a_{n-1}\}).$$

A_{min} and A_{max} are bitonic sequences. Furthermore, each element of A_{min} is less than every element of A_{max}. It's easy to see that this is the case if we represent the two subsequences $A_0 = (a_0, a_1, \ldots, a_{n/2-1})$ and $A_1 = (a_{n/2}, \ldots, a_{n-1})$ of A in a graph. See Figure 14.1. The figure makes it clear that there is an index $i_0, 0 \leq i_0 \leq n/2 - 1$, with the property that up to index i_0 all the elements of A_{min} are the corresponding elements of A_0, and after i_0, all the elements of A_{min} are the corresponding elements of A_1. This process of creating two bitonic sequences from a single bitonic sequence is called a **bitonic split.**

Because every element in A_{min} is less than every element in A_{max}, and both sequences are bitonic, we can recursively split A_{min} and A_{max} until we

Table 14.1 Bitonic sort of a bitonic sequence

Step	*Elements*							
Start	1	5	6	9	8	7	3	0
1	1	5	3	0	8	7	6	9
2	1	0	3	5	6	7	8	9
3	0	1	3	5	6	7	8	9

obtain sequences of length 1. When this has happened, we will have obtained a sorted sequence.

As an example, consider the sequence $(1,5,6,9,8,7,3,0)$. Table 14.1 illustrates the process of successively splitting to ultimately obtain a sorted sequence.

Of course, we still need to arrange that we have a bitonic sequence to start off with. Let's start small: observe that any sequence containing two elements is bitonic—in fact, any sequence containing two elements is monotonic. Now suppose for the moment that we have an unsorted list of four elements, $A = (a_0, a_1, a_2, a_3)$. In order for A to be bitonic, it's not enough for both two element sequences, (a_0, a_1) and (a_2, a_3) to be monotonic. We need for, say, the first to be increasing and the second to be decreasing. Of course this is easy enough to arrange: just compare the two elements in each two element sequence and make the appropriate switch.

How does this generalize? If we have an eight-element sequence, $A = (a_0, a_1, \ldots, a_7)$, we can use the preceding method to convert it into two bitonic sequences. That is, we can assume that (a_0, a_1, a_2, a_3) and (a_4, a_5, a_6, a_7) are bitonic. Indeed we can assume that $a_0 \leq a_1$, $a_2 \geq a_3$, $a_4 \leq a_5$, and $a_6 \geq a_7$. But how can we arrange that the sequence A itself is bitonic? Recall that the iterated bitonic split converts a bitonic sequence into a sorted sequence. Thus, we can arrange that (a_0, a_1, a_2, a_3) is increasing. Can we also arrange that (a_4, a_5, a_6, a_7) is decreasing? It's easy; we just "reverse" our bitonic splits. That is, we put the minima in the second half and the maxima in the first half.

An example will help. Suppose we have the sequence

$$(5,3,6,2,1,9,0,8).$$

We can convert it into two bitonic sequences easily enough: $(3,5,6,2)$ and $(1,9,8,0)$. Now we can convert the sequence $(3,5,6,2)$ into an increasing sequence using bitonic splits:

Step	*Elements*			
Start	3	5	6	2
1	3	2	6	5
2	2	3	5	6

Similarly, we can convert $(1, 9, 8, 0)$ into a decreasing sequence:

Step	*Elements*			
Start	1	9	8	0
1	8	9	1	0
2	9	8	1	0

Now we can perform a sequence of bitonic splits on $(2, 3, 5, 6, 9, 8, 1, 0)$ to obtain a sorted list.

If we assume that n is a power of two, it's easy to code a serial bitonic sort. The following code contains some excerpts:

```
        ⋮
/* Successive subsequences will switch between
 * increasing and decreasing bitonic splits.
 */
#define INCR 0
#define DECR 1
#define Reverse(ordering) ((ordering) == INCR ? DECR : INCR)

main() {
        ⋮

    for (list_length = 2; list_length <= n;
         list_length = list_length*2)
        for (start_index = 0, ordering = INCR;
             start_index < n;
             start_index = start_index + list_length,
             ordering = Reverse(ordering))
            if (ordering == INCR)
                Bitonic_sort_incr(list_length,
                    A + start_index);
            else
                Bitonic_sort_decr(list_length,
                    A + start_index);
        ⋮
void Bitonic_sort_incr(
        int       length /* in     */,
        KEY_T     B[]    /* in/out */) {
    int i;
    int half_way;

    /* This is the bitonic split */
    half_way = length/2;
    for (i = 0; i < half_way; i++)
        if (B[i] > B[half_way + i])
            Swap(B[i],B[half_way+i]);
```

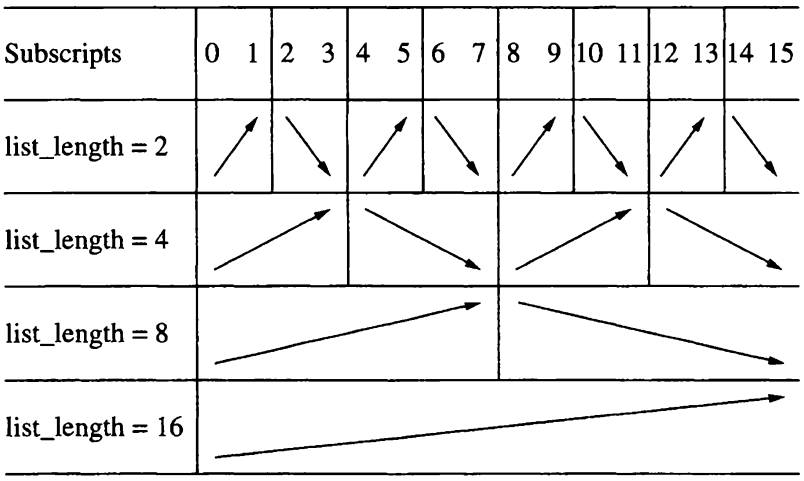

Figure 14.2 Orientation of subsequences during stages of bitonic sort

```
    if (length > 2) {
        Bitonic_sort_incr(length/2, B);
        Bitonic_sort_incr(length/2, B + half_way);
    }
} /* Bitonic_sort_incr */
```

The nested `for` loops in the main program arrange first that successive two-element subsequences are monotone (increasing or decreasing), then successive four-element subsequences are monotone, etc. For a 16-element sequence, the effect of the algorithm can be visualized as illustrated in Figure 14.2. An up arrow indicates an increasing subsequence, a down arrow a decreasing subsequence.

The only differences between `Bitonic_sort_incr` and `Bitonic_sort_decr` will be that the conditional

```
if (B[i] > B[half_way + i])
```

should be replaced by

```
if (B[i] < B[half_way + i])
```

and `Bitonic_sort_decr` should recursively call itself.

14.4 Parallel Bitonic Sort

As with the serial bitonic sort, we'll assume that the number of keys n is a power of two. We'll also assume that the number of processes p is a power of two, and that the underlying architecture of our system is a hypercube. (We'll relax the first restriction later on.) So we'll assume that the process

ranking in MPI_COMM_WORLD is the natural hypercube ranking we discussed in Chapter 13. At each stage of the algorithm, each process will have a sublist of the global list containing n/p keys.

Let's first consider the case $p = 4$. We can begin by using a fast local sorting algorithm (e.g., quicksort) to convert the local keys on each process into an increasing or decreasing sequence. The ordering will be determined by the parity of the process rank: even-numbered processes will sort into increasing order, odd-numbered processes will sort into decreasing order. Then, each pair of processes, $(0, 1)$ and $(2, 3)$, jointly owns a bitonic sequence, and, if they perform the appropriate bitonic splits, each pair will own a monotonic sequence, and the global sequence will be bitonic. That is, the 4-tuple of processes, $(0, 1, 2, 3)$, jointly owns a bitonic sequence.

Observe that we can now carry out a bitonic split on the entire distributed sequence by carrying out a bitonic split on the sequence shared by processes 0 and 2 and by carrying out a bitonic split on the sequence shared by processes 1 and 3.

An example will clarify this. Suppose that the following sequences have been assigned to processes 0, 1, 2, and 3:

Process 0: 9, 12, 16, 23
Process 1: 26, 39, 42, 61
Process 2: 43, 17, 14, 13
Process 3: 12, 7, 6, 5

If we perform an (increasing) bitonic split on the sequence

$$(9, 12, 16, 23, 26, 39, 42, 61, 43, 17, 14, 13, 12, 7, 6, 5),$$

and divide the keys among the processes, we'll get the following assignment:

Process 0: 9, 12, 14, 13
Process 1: 12, 7, 6, 5
Process 2: 43, 17, 16, 23
Process 3: 26, 39, 42, 61

However, if we perform a bitonic split on the subsequence shared by processes 0 and 2,

$$(9, 12, 16, 23, 43, 17, 14, 13),$$

and assign the lower half of the result to process 0 and the upper half to process 2, processes 0 and 2 will get the same keys. Similarly, if we perform a bitonic split on the sequence shared by processes 1 and 3, $(12, 7, 6, 5, 26, 39, 42, 61)$, and assign the lower half to process 1 and the upper half to process 3, processes 0 and 3 will also get the same keys. That is, a bitonic split on the entire sequence has the same effect as two separate, independent bitonic splits: one on the subsequence shared by processes 0 and 2, and the other on the subsequence shared by processes 1 and 3. Further, in a hypercube, the paired processes are adjacent.

To finish up and obtain a sorted sequence on the four processes, we can perform a bitonic split on the sequence shared by processes 0 and 1 and a bitonic split on the sequence shared by processes 2 and 3.

The difference between the four-processor case and the p-processor case, $p = 2^d$, should be pretty clear. We obtain successively larger bitonic sequences by applying the process-pair bitonic splits to larger collections of processes. For example, suppose a bitonic sequence is distributed across eight processes. Then process pairs $(0, 4)$, $(1, 5)$, $(2, 6)$, and $(3, 7)$ will perform the initial bitonic splits. The next set of bitonic splits will pair processes $(0, 2)$, $(1, 3)$, $(4, 6)$, and $(5, 7)$. The final splits will pair processes $(0, 1)$, $(2, 3)$, $(4, 5)$, and $(6, 7)$.

A final observation. Suppose processes A and B, $A < B$, share a bitonic sequence, with A's keys increasing and B's keys decreasing. We can obtain a monotonic sequence if, instead of performing a bitonic split, we sort the keys on B in increasing order, merge the keys from the two processes, and assign the smaller keys to A and the larger keys to B. In other words, we can replace our bitonic splits with "merge splits." This is a cheaper operation, since it always maintains a sorted list. If there are n/p keys per process, we only have to sort these n/p keys once. If we used bitonic splits, we would sort them every time we sorted across a set of processes. Note also that if we use merge splits, we can relax the restriction that the number of keys is a power of two.

Let's make all of this precise by writing some of the code for carrying it out. The heart of the algorithm, the bitonic split, is replaced by a merge split.

```
#include "mpi.h"

KEY_T temp_list[MAX]; /* buffer for keys received */
                      /* in Merge_split           */
                      /* KEY_T is a C type        */

/* Merges the contents of the two lists. */
/* Returns the smaller keys in list1     */
void Merge_list_low(
        int    list_size  /* in     */,
        KEY_T  list1[]    /* in/out */,
        KEY_T  list2[]    /* in     */);

/* Returns the larger keys in list 1.    */
void Merge_list_high(
        int    list_size  /* in     */,
        KEY_T  list1[]    /* in/out */,
        KEY_T  list2[]    /* in     */);

void Merge_split(
        int       list_size    /* in     */,
        KEY_T     local_list[] /* in/out */,
        int       which_keys   /* in     */,
        int       partner      /* in     */,
        MPI_Comm  comm         /* in     */ ) {
```

```
    MPI_Status status;

    /* key_mpi_t is an MPI (derived) type */
    MPI_Sendrecv(local_list, list_size, key_mpi_t,
                 partner, 0, temp_list, list_size,
                 key_mpi_t, partner, 0, comm, &status);
    if (which_keys == HIGH)
        Merge_list_high(list_size, local_list,
                        temp_list);
    else
        Merge_list_low(list_size, local_list,
                       temp_list);
} /* Merge_split */
```

Since we are effectively parallelizing the inner loop of the serial main program, the main program will now have a single for loop:

```
main(int argc, char* argv[]) {
          :
    Local_sort(list_size, local_list);

    /* and_bit is a bitmask that, when "anded" with  */
    /* my_rank, tells us whether we're working on an */
    /* increasing or decreasing list                 */
    for (proc_set_size = 2, and_bit = 2;
         proc_set_size <= p;
         proc_set_size = proc_set_size*2,
         and_bit = and_bit << 1)
        if ((my_rank & and_bit) == 0)
            Par_bitonic_sort_incr(list_size,
                      local_list, proc_set_size, comm);
        else
            Par_bitonic_sort_decr(list_size,
                      local_list, proc_set_size, comm);
          :
```

As with most hypercube algorithms, the processes that have been grouped for a bitonic sort are paired by "exclusive or'ing" the process rank with a bitmask consisting of a single bit successively right-shifted. (See section 13.2 for a detailed discussion.)

```
void Par_bitonic_sort_incr(
        int       list_size      /* in     */,
        KEY_T     local_list[]   /* in/out */,
        int       proc_set_size  /* in     */,
        MPI_Comm  comm           /* in     */ ) {
          :
```

```
    /* type of eor_bit should be unsigned: otherwise
     * right shift may fill most significant bits with
     * sign bit */
    proc_set_dim = log_base2(proc_set_size);
    eor_bit = 1 << (proc_set_dim - 1);
    for (stage = 0; stage < proc_set_dim; stage++) {
        partner = my_rank ^ eor_bit;
        if (my_rank < partner)
            Merge_split(list_size, local_list, LOW,
                partner, comm);
        else
            Merge_split(list_size, local_list, HIGH,
                partner, comm);
        eor_bit = eor_bit >> 1;
    }
} /* Par_bitonic_sort_incr */
```

We have effectively replaced the recursive calls in `Bitonic_sort_incr` by an iterative loop. The only difference between `Par_bitonic_sort_incr` and `Par_bitonic_sort_decr` is that the test

```
if (my_rank < partner)
```

should be replaced with

```
if (my_rank > partner)
```

We'll leave to the exercises discussions of the performance of parallel bitonic sort and parallel quicksort.

14.5 Tree Searches and Combinatorial Optimization

Many optimization problems can be solved by searching a tree. As an example, consider the famous travelling salesman problem: a salesman has a list of cities he must visit and a set of costs associated to travelling between each city (e.g., airfare, car rental). His problem is to visit each city exactly once and return to the starting city, ordering the cities so that the cost of his trip is minimized.

The problem can be formalized by defining a weighted, directed graph. Each vertex of the graph corresponds to a city, and the weight of each edge is the cost of travelling from the city corresponding to the tail of the edge to the city corresponding to its head. Thus, the problem can be viewed as finding a listing of the vertices or cities so that the first vertex is the same as the last, and every other vertex appears exactly once in the list. Any such listing is called a **tour**. We want to find a tour with the property that the sum of the weights on the edges joining consecutive vertices is minimized.

The problem can be solved in several ways by using a tree search. In one approach the root of the tree corresponds to all possible tours. Children are

generated by choosing an edge: the left subtree corresponds to all tours that traverse the chosen edge, and the right subtree corresponds to all tours that don't traverse the edge.

Another combinatorial optimization problem that has applications to parallel computing is graph partitioning. Many problems in scientific computing can be abstracted as (possibly weighted) graphs: solving differential equations using finite elements, modelling electronic circuits, etc. If we are solving one of these problems on a parallel system, we may wish to partition the graph among the processes. Typically the vertices of the graphs correspond to computations, and the edges correspond to communications. Thus, we would like to partition the graph so that the sums of the weights of the vertices assigned to the processes are equalized (i.e., the computational load is balanced), and the weights of the edges cut by the partitioning is minimized. A simple-minded solution to the problem of partitioning a graph into two subsets is to build a binary tree with the root corresponding to all solutions. Children are generated by choosing an unassigned vertex: the left subtree corresponds to all solutions that assign that vertex to the first subset in the partition; the right subtree corresponds to all solutions that assign that vertex to the second.

Most combinatorial optimization problems of interest are NP-hard or NP-complete, which means, for all practical purposes, that there is no general algorithm that is better in every instance than the brute force algorithm, which examines every possible solution and chooses the optimal solution. Furthermore, if we try to express the size of the set of all possible solutions in terms of the size of the input (e.g., the number of cities in the travelling salesman problem), it will usually contain an exponential term. For such problems, parallel computing won't improve matters unless we happen to have a parallel system with a processor set that also grows exponentially with the problem size. However, there are many important special cases that can be solved in less than exponential time, and for practical applications, we are often willing to settle for a "good" solution rather than an optimal one.

14.6 Serial Tree Search

There are a number of approaches to searching the trees generated during the solution of a combinatorial optimization problem. One of the simplest and most commonly used is **depth-first search.** In depth-first search, we begin at the root; after a node in the tree is "expanded" (i.e., its children are generated), one of the children is chosen for expansion. This process continues until we reach a node that is either a solution or cannot possibly lead to a solution, at which point we backtrack to the first ancestor with children that are still unexpanded. When we find a solution, we can compare it to the best known solution and either save it or discard it. A simple example using the travelling salesman problem is illustrated in Figures 14.3 and 14.4.

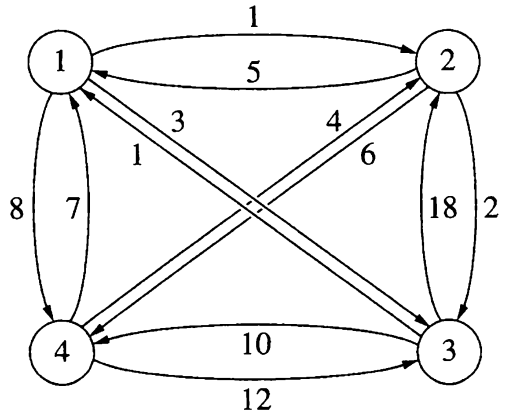

Figure 14.3 Four-vertex travelling salesman problem

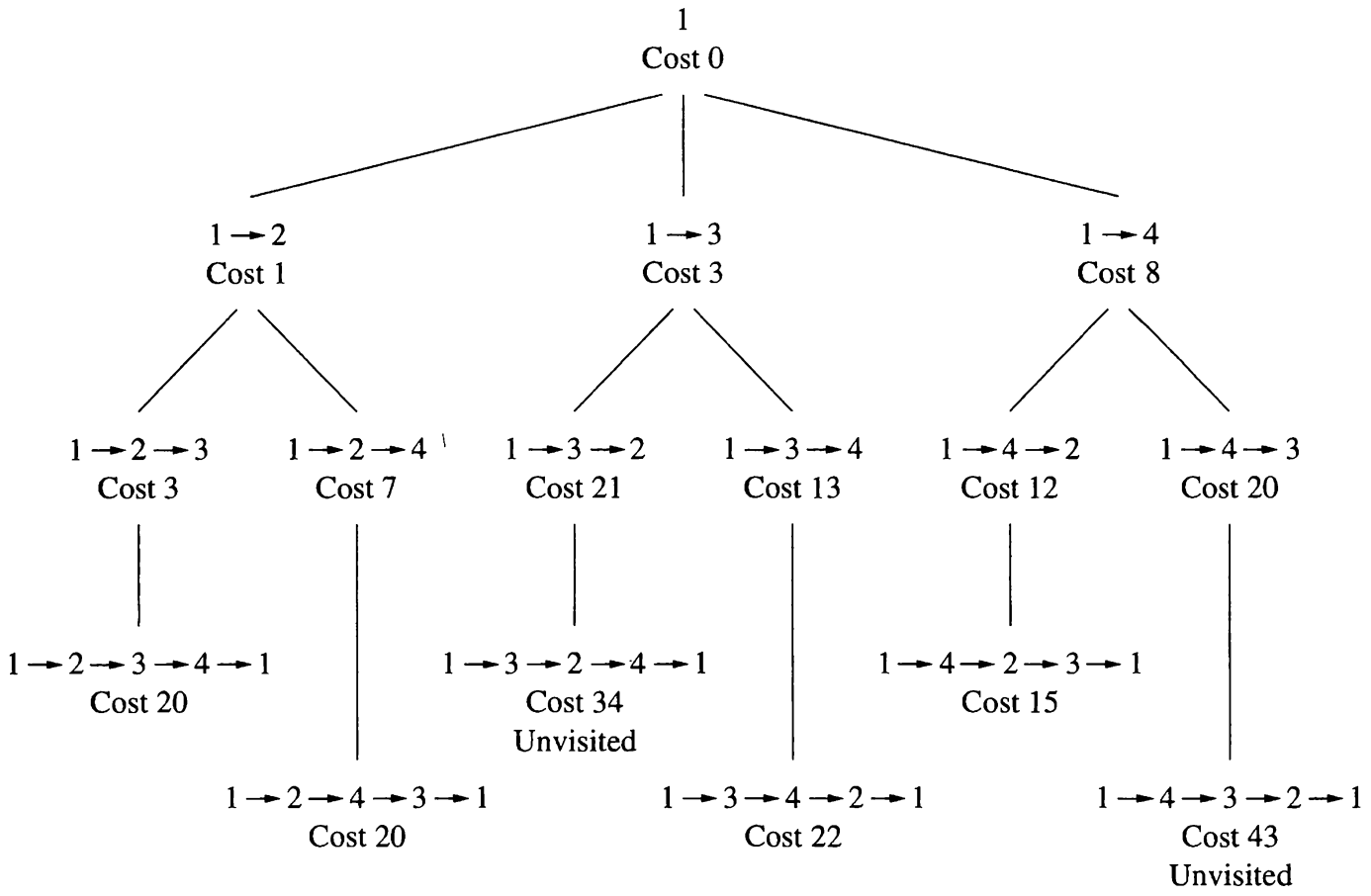

Figure 14.4 Search tree for four-vertex travelling salesman problem

There are two standard implementations of serial depth-first search. The first uses recursion:

```
/* Recursive depth-first search                    */
/* Assume the parameter node is "feasible" --      */
/*     i.e., it can lead to a solution             */
void Dfs_recursive(
        NODE_T node  /* in */) {
```

```
    int    i;
    int    num_children;
    NODE_T child_list[MAX_CHILDREN];

    Expand(node, child_list, &num_children);

    for (i = 0; i < num_children; i++)
        if (Solution(child_list[i])) {
            if (Evaluate(child_list[i])
                < best_solution)
                /* A solution has no children */
                best_solution =
                   Evaluate(child_list[i]);
        } else if (Feasible(child_list[i])) {
            Dfs_recursive(child_list[i]);
        }
} /* Dfs_recursive */
```

Note that the function Expand may order the children so that more promising nodes are expanded first. Also note that in its simplest form, depth-first search will not be able to determine whether a node that doesn't correspond to a solution is feasible, and the test

```
if (Feasible(child_list[i]))
```

will be omitted.

The second standard implementation eliminates recursion by pushing unexpanded nodes onto a stack.

```
/* Iterative depth-first search using a stack */
void Dfs_stack(
         NODE_T root  /* in */) {
    NODE_T   node;
    STACK_T  stack;

    /* Allocate empty stack */
    Initialize(&stack);

    /* Expand root; push children onto stack */
    Expand(root, stack);

    while (!Empty(stack)) {
        node = Pop(stack);
        if (Solution(node)) {
            if (Evaluate(node) < best_solution)
                best_solution = Evaluate(node);
        } else if (Feasible(node)) {
            Expand(node, stack);
        }
    }
```

```
      Free(stack);
   } /* Dfs_stack */
```

A possible problem with depth-first search becomes evident in the iterative solution: can it happen that there are no acceptable solutions? If so, what does the program do? Clearly the answer to the first question should be yes. For example, we might want all solutions to an optimization problem that are better than some value, and there might not be any. In such a case, we should have initialized `best_solution` to be the cut-off value, and the program can simply check whether the result in `best_solution` is better than this initialization. We'll return to this problem when we look at parallel depth-first search.

For alternatives to depth-first search, see the references at the end of the chapter.

14.7 Parallel Tree Search

The obvious, perhaps too obvious, parallelization of depth-first search is simply to distribute disjoint subtrees among the processes. For example, if the tree is a binary tree, and there are four processes, we can have process 0 expand the root and its two children, and distribute the four grandchildren among the processes: one grandchild to each process. Indeed this approach works very well if we know that the tree is *balanced.* In this case "balanced" means that each of the subtrees rooted at the grandchildren contains approximately the same number of nodes. Of course, this won't necessarily be the case, and we may run into serious problems with load imbalance. Indeed, some solutions to optimization problems that use tree search intentionally generate a highly unbalanced tree.

In order to avoid this problem, parallel implementations of depth-first search usually rely on a *dynamic* load-balancing scheme. There are several methods for dynamic load balancing. On a shared-memory machine, we can simply put the stack in shared memory, and processes can push nodes onto and pop nodes from the stack. On a distributed-memory machine, a single process could be responsible for managing the stack, and the other processes could access the stack through message passing. However, even on a shared-memory machine, this can lead to serious problems with processes competing for access to the stack. Thus, it may be better to distribute the stack among the processes. If a process exhausts its stack, it can send requests to other processes for work.

The basic outline for a parallel tree search based on serial depth-first search might be something like this:

```
/* root == NULL if rank != 0 */
void Par_tree_search(
        NODE_T       root       /* in */,
        MPI_Comm     comm       /* in */ ) {
```

```
    STACK_T local_stack;
    NODE_T* node_list;
    NODE_T  node;
    int     p;
    int     my_rank;

    MPI_Comm_size(comm, &p);
    MPI_Comm_rank(comm, &my_rank);

    /* Generate initial set of nodes, 1 per process */
    if (my_rank == 0) {
        Generate(root, &node_list, p);
    }

    Scatter(node_list, &node, comm);
    Initialize(node, &local_stack);

    do {
        /* Search for a while */
        Par_dfs(local_stack, comm);

        /* Service requests for work. */
        Service_requests(local_stack, comm);

        /* If local_stack isn't empty, return.          */
        /* If local_stack is empty, send                */
        /* requests until either we get work, or we     */
        /* receive a message terminating program.       */
    } while(Work_remains(local_stack, comm));

    Update_solution(comm);
    Free(local_stack);
    Print_solution(comm);
} /* Par_tree_search */
```

Thus, process 0 receives the root of the tree from the calling function and generates a list of p nodes—possibly using depth-first search. It then scatters the nodes among the processes, and each process does a local depth-first search. The local depth-first search continues until either the process exhausts its subtree, or it expands some predefined maximum number of nodes. In either case, it returns and services any requests it has received for work: if `local_stack` doesn't contain enough work, it will send a rejection; otherwise it will split its stack and send part to the requesting process. After returning from the call to `Service_requests`, it either continues with more searching or, if its stack is empty, it requests further work. Eventually, it will either receive work from another process, in which case it will continue, or it will receive a termination

message. After termination, each process will update its value for the solution, process 0 will print the result, and Par_tree_search will return.

Let's look at each of the functions Par_dfs, Service_requests, and Work_remains.

14.7.1 Par_dfs

Par_dfs is basically the same as Dfs_stack. There are two main differences: The first is that Par_dfs doesn't necessarily run until the local stack is empty: it will return after it has expanded MAX_WORK nodes regardless of the state of the local stack. The reason for this is that the calling function Par_tree_search needs to periodically service requests for work from other processes. The parameter MAX_WORK must be chosen so that it balances two conflicting requirements. On the one hand, we want each process to do as much local work as possible—i.e., the amount of communication should be minimized. This suggests that MAX_WORK should be chosen to be as large as possible. On the other hand, we don't want processes sitting idle, and if a process is assigned a small subtree and MAX_WORK is large, it may wait a long time before it receives any new work from another process.

The second main difference with Dfs_stack is the fact that in Par_dfs the "quality" of a solution may depend on the work done by other processes. For example, in the solution of the travelling salesman problem, we only want the *best* tour. When we are executing Par_dfs and we generate a new solution, we can evaluate it strictly on the basis of whether it is better than all local solutions so far, or we can compare it to the best solution on all processes so far. Similarly, if we can evaluate partial solutions (e.g., a partial tour in the travelling salesman problem), we will not want to complete all partial solutions. Those that cannot possibly be better than the current best solution or upper bound on the best possible solution (given the current state of our knowledge) should not be completed. Once again, this decision can be made on the basis of local or global information. In either case, we have a trade-off: We can improve the quality of our estimates if we use global information; however, the use of global information involves communication and, as a consequence, is expensive. In our code we have chosen to make the decisions on the basis of global information. Consequently, the function Feasible takes a communicator argument as well as a node argument, and we define a new function, Best_solution, which checks for new solutions from other processes.

```
void Par_dfs(
        STACK_T   local_stack  /* in/out */,
        MPI_Comm  comm         /* in     */) {
    int     count;
    NODE_T  node;
    float   temp_solution;
```

```
    /* Search local subtree for a while */
    count = 0;
    while (!Empty(local_stack) && (count < MAX_WORK)) {
        node = Pop(local_stack);
        if (Solution(node)) {
            temp_solution = Evaluate(node);
            if (temp_solution < Best_solution(comm)) {
                Local_solution_update(temp_solution, node);
                Bcast_solution(comm);
            }
        } else if (Feasible(node, comm)) {
            Expand(node, local_stack);
        }
        count++;
    } /* while */
} /* Par_dfs */
```

Note that the stack is no longer local to the depth-first search routines: other functions (e.g., those that service requests) will also need to access the stack. Also note that we will need to use at least two communication functions: a function that checks the message queue for new best solutions (to be called by `Best_solution` and `Feasible`) and a broadcast function.

Checking the message queue can be implemented with the MPI function

```
int MPI_Iprobe(
    int         source /* in  */,
    int         tag    /* in  */,
    MPI_Comm    comm   /* in  */,
    int*        flag   /* out */,
    MPI_Status* status /* out */ )
```

This function searches for an incoming message that matches `source`, `tag`, and `comm`. If it finds a message that matches, it returns `flag = TRUE`, and `status` will contain the same information that would be contained in the status argument returned by a call to `MPI_Recv`. In particular, it will contain information on the number of elements in the message. Thus, `MPI_Iprobe` can be used when we want to receive a message of unknown size. The source and tag arguments can be wildcards; i.e., we can call `MPI_Iprobe` with

```
source = MPI_ANY_SOURCE;
tag    = MPI_ANY_TAG;
```

If there are multiple messages that can match the call, information is returned on the message that would be received by a call to `MPI_Recv` at that point. Thus, in order to correctly implement checking the message queue, it will be necessary to loop until all messages that contain new best solutions have been received. Note that `MPI_Iprobe` doesn't actually receive the message: you still need to call `MPI_Recv`.

Since MPI_Bcast requires the participation of all the processes in comm, and the only process that "knows" about the new solution is the one that finds it, MPI_Bcast cannot be used in Par_dfs. A simple, albeit unscalable, solution is to use a loop of buffered sends. In most cases, the number of new solutions will be quite small and this won't be a serious problem. If it is, it's possible to write a nonblocking broadcast using attributes (see Chapter 8).

14.7.2 Service_requests

The function that services requests needs to read all requests for data and either service them or send a refusal. It might look something this.

```
void Service_requests(
         STACK_T   local_stack /* in/out */,
         MPI_Comm  comm        /* in     */) {
    STACK_T send_stack;
    int     destination;

    while (Work_requests_pending(comm)) {
        destination = Get_dest(comm);
        if (Nodes_available(local_stack)) {
            Split(local_stack, &send_stack);
            Send_work(destination, send_stack, comm);
        } else {
            Send_reject(destination, comm);
        }
    }
} /* Service_requests */
```

The function Work_requests_pending uses MPI_Iprobe to see if there's a message with a previously defined tag that is used only for requests. If there is such a message, Get_dest will return the source of the message.

Both Nodes_available and Split are highly problem dependent. We'll explore some alternatives in the exercises. For further information, see the references at the end of the chapter.

14.7.3 Work_remains

The function Work_remains determines whether there are any remaining nodes to be expanded either locally or globally. If there are, it returns TRUE and a nonempty stack. Otherwise it returns FALSE.

```
int Work_remains(
        STACK_T    local_stack /* in/out */,
        MPI_Comm   comm        /* in     */ ) {

    int      work_available;
```

```
    int       request_sent = FALSE;
    int       work_request_process;

    if (!Empty(local_stack)) {
        return TRUE;
    } else {
        Out_of_work(comm);
        while (TRUE) {
            Send_all_rejects(comm);
            if (Search_complete(comm)) {
                return FALSE;
            } else if (!request_sent) {
                work_request_process = New_request(comm);
                Send_request(work_request_process, comm);
                request_sent = TRUE;
            } else if (Reply_received(work_request_process,
                        &work_available, local_stack, comm)) {
                if (work_available)
                    return TRUE;
                else
                    request_sent = FALSE;
                }
            }
        } /* while (TRUE) */
    }
} /* Work_remains */
```

The function `Out_of_work` posts a notice that we've finished all local work. `Search_complete` tests whether all the processes are done. We'll discuss these in more detail in the next section.

The function `Send_all_rejects` sends rejection messages to all processes that have requested work. This is necessary to avoid a "busy" deadlock. If we omitted it, two processes that are out of work could send requests to each other. These requests would never be either satisfied or rejected, and the other processes would do all the remaining work.

The function `New_Request` simply returns the rank of a process to which the work request will be sent. There are many possibilities for this function. One possibility is for it to just generate a random process rank. Another possibility would be for each process to simply increment, modulo p, the rank of the process from which work was last requested, with the initial request being sent to process (`my_rank` + 1) mod p. A third possibility is for a single process, say process 0, to maintain the rank of the process to which the next request will be sent, and when a process runs out of work, it gets the rank from 0, and 0 increments it. Since the first two approaches make no use of global information, it can happen that many processes simultaneously request work from the same process. In the third approach, process 0 may find itself spending all its time servicing requests for a process address, and each request

for work requires four communications (send request for process rank to 0, receive process rank, send work request, receive work) and hence $4t_s$ instead of just $2t_s$.

The function `Send_request` simply sends a message using the request tag to the process `work_request_process`. `Reply_received` uses `MPI_Iprobe` to check whether there has been a reply to our request. If there hasn't, it returns false. Otherwise, if the reply was affirmative, it copies the received data into `local_stack` and sets `work_available` to true. If the reply was negative, it sets `work_available` to false.

Alternatively, `Send_request` can post a nonblocking receive, and `Reply_received` can use `MPI_Test` to check for a reply.

14.7.4 Distributed Termination Detection

This is a popular topic in computer science, and there is an extensive literature on the subject. We'll limit our discussion to a single approach that is especially well suited to parallel tree search. See the references at the end of the chapter for further discussion.

The idea behind this algorithm is that when the program begins, process 0 has a certain quantity of some indestructible resource: call it energy. Process 0 can transfer some of the energy to other processes, and they can, in turn, transfer some to yet other processes, but, of course, energy is conserved. So no matter how the energy is distributed among the processes, the total energy will always be the same as it was when the program started. The rules governing the energy are as follows:

1. When the program begins, process 0 has all the energy: its total energy is 1.
2. Energy is only transferred in three circumstances:
   a. during the initial distribution of the tree
   b. when a process fills a request for work
   c. when a process runs out of work
3. During the initial distribution phase, the energy is divided into p parts. So after the initial distribution, each process has energy $1/p$.
4. When a process fills a request for work, it keeps half its energy and transfers half to the process that requested the work.
5. When a process runs out of work, it transfers whatever energy it has left to process 0.

Now observe that energy is conserved: we never actually dispose of energy. Also observe that when every process (except possibly 0) has run out of work, each process will have 0 energy, and process 0 will have energy 1. Thus, whenever process 0 completes its work and finds it has energy 1, it should broadcast a termination message to all the processes.

Of course, there will be problems with keeping track of the energy if we try to use floating point arithmetic and keep halving the energy. However, we can avoid this by representing fractions as pairs of integers: $a/b = (a, b)$. Then dividing by p converts (a, b) to (a, pb) and adding $(a, b) + (c, d) = (ad + bc, bd)$. The only work here is reducing to lowest terms. However, the only possible divisors of the denominator are 2 and divisors of p.

Also note that even if process 0 is receiving lots of energy returns, it only needs to tally them when it runs out of work. So this shouldn't seriously interfere with process 0's performance. Alternatively, we can form a spanning tree, rooted at process 0. Then rather than returning energy to process 0, each process can return energy to its parent in the spanning tree.

14.8 Summary

In this chapter we took a very brief look at the field of parallel algorithms. We saw that parallel algorithms can be roughly divided into two categories: those that are more or less direct parallelizations of standard serial algorithms and those that aren't.

We took in-depth looks at the design and coding of two parallel algorithms: parallel bitonic sort and parallel tree search. The first algorithm used the indirect approach to the development of parallel algorithms, since the standard serial algorithm for sorting, quicksort, doesn't seem to parallelize very well.

The second algorithm, parallel tree search, was a natural generalization of serial depth-first search. However, in order to obtain load balance, we had to be very careful about the details of our implementation.

In our discussion of parallel tree search we touched briefly on a well-known problem in parallel and distributed computing: distributed termination detection. The problem is to determine when a collection of processes working asynchronously (as in the parallel tree search) are finished computing. The solution we looked at assigns a fixed quantity of an indestructible resource to process 0 at the start of execution. Each time work is sent from one process to another process, the process sending the work also sends half of its resource. When a process runs out of work, it sends its resource back to process 0. When process 0 has recovered all of the original resource, the program should terminate.

We only learned one new MPI function, `MPI_Iprobe`. It can be used to check to see whether a message from a specified source, a message with a given tag, or a message in a certain communicator is available. Its syntax is

```
int MPI_Iprobe(
    int         source /* in  */,
    int         tag    /* in  */,
    MPI_Comm    comm   /* in  */,
    int*        flag   /* out */,
    MPI_Status* status /* out */ )
```

If `flag` is nonzero, then a message matching the criteria specified by `source`, `tag`, and `comm` has arrived. The `source` and `tag` arguments can be wildcards, but there is no wildcard for communicators. The `status` will return a reference to the status that would be returned if `MPI_Recv` were called with the same `source`, `tag`, and `comm` arguments. Note that since the `status` specifies the amount of storage needed for the message, we can guarantee that the buffer in a subsequent call to `MPI_Recv` will be large enough to receive the entire message.

14.9 References

The literature on parallel algorithms is quite extensive. Two excellent starting points are Fox et al. [18] and Kumar et al. [26]. Both books provide discussions of parallel sorting algorithms. Fox et al. provide empirical performance data on several sorting algorithms, including bitonic sort and quicksort. The efficiency of parallel quicksort actually begins to decrease on large datasets. Kumar et al. also discuss how bitonic sort can be mapped to a mesh.

Much of our discussion of parallel tree search is based on Kumar et al. They also discuss alternatives to depth-first search, and they provide performance analyses and empirical data on the performance of various implementations of parallel depth-first search.

There is also an extensive literature on serial algorithms. Perhaps the best known reference is Knuth's classic, *The Art of Computer Programming* [25]. The third volume discusses searching and sorting.

Reingold et al. [32] have an extensive discussion of approaches to the travelling salesman problem.

14.10 Exercises

1. If we wish to predict the performance of any *comparison-based* sorting algorithm— serial or parallel—we run into difficulties with the constants in our formulas. Unlike the algorithms we've considered thus far, it is not possible to say something like "The runtime of a serial bubble sort is $T(n) = kn^2$." The reason is that the exact time of a bubble sort (and other comparison-based algorithms) depends in large measure on the number of swaps, and for fixed n, this can vary considerably. A reasonable approach to this difficulty is to empirically determine an average value for k by timing the program on a large number of datasets.

   Use this approach to derive a formula for the average runtimes of serial bitonic sort on your system.

2. Use the method outlined in exercise 1 to derive a formula for the average runtime for the local sorting algorithm you're using in your implementation of parallel bitonic sort. Use this formula to develop a formula for the

runtime of parallel bitonic sort. How do your predicted runtimes compare to the actual runtimes?

3. Modify the parallel bitonic sort program so that it can handle lists whose length isn't divisible by p. There are several ways this can be done. For example, we might pad the list with "huge" keys—keys larger than any actual key—or we might modify `Merge_split` so it uses two list sizes: the size of the local list and the size of the partner's list. Which method do you think will result in better performance? Which method is easier to implement?

14.11 Programming Assignments

1. Reingold et al. [32] provide an extensive discussion of the *branch and bound* solution to the travelling salesman problem (TSP).
   a. Write a serial program that uses branch and bound to solve the travelling salesman problem. Input to the program will consist of n, the number of cities, and an adjacency matrix for the graph representing intercity costs. Output should be an optimal tour.
   b. Complete the coding of the parallel tree search program so that it uses branch and bound to solve the travelling salesman problem. Add `MAX_WORK` to the input data. For the `Nodes_available` function, return `TRUE` if there are at least two nodes in `local_stack`. For the `Split` function, send every other node in the `local_stack`. If a process runs out of work, it should simply generate a random process rank, and send a request for work to this process. How do different values of `MAX_WORK` affect the performance of the program?
   c. Compare the performance of the serial program to the parallel program. Does the program seem to be scalable?
   d. Kumar et al. [26] suggest that the `Nodes_available` function should depend on a cut-off depth. (The depth of a node in a tree is its distance from the root.) The idea is that if a node is very deep in the tree, it won't have many descendants, and hence it won't provide much additional work to a process that's run out of data. Modify your tree-search program so that it takes a cut-off depth as an additional input parameter. The `Nodes_available` function should return `TRUE` only if there are nodes in `local_stack` above the cut-off depth. If there are, the `Split` function should send every other node above the cut-off depth. Can you significantly improve the performance of the program by choosing a good cut-off depth?
2. Both Fox et al. [18] and Kumar et al. [26] discuss parallel shellsort. Implement a parallel shellsort algorithm. Compare its performance to parallel bitonic sort. Is one algorithm consistently better than the other?