CHAPTER

14

Sorting

Had I been present at the creation, I would have given some useful hints for the better ordering of the universe.

Reaction of Alfonso X to a description of the intricacies of the Ptolemaic system

14.1 INTRODUCTION

Given a sequence of n numbers $\{a_0, a_1, a_2, \ldots, a_{n-1}\}$, the **sorting problem** is to find a permutation $\{a'_0, a'_1, a'_2, \ldots, a'_{n-1}\}$ such that $a'_0 \leq a'_1 \leq a'_2 \leq \ldots \leq a'_{n-1}$. Sorting is one of the most common activities performed on serial computers. Many algorithms incorporate a sort so that information may be accessed efficiently later.

Usually the numbers being sorted are part of data collections called **records.** Within each record, the value being sorted is called the **key.** The rest of the record contains **satellite data.** The information being accessed later is typically in the satellite data, so while it is the keys that are being compared, it is the complete records that must actually be permuted. If there are relatively little satellite data, entire records may be shuffled as the sort progresses. If there are large amounts of satellite data, the sort may actually permute an array of pointers to the records. For the purposes of this chapter, however, we will focus exclusively on the problem of sorting a sequence of numbers, leaving the issues associated with the satellite data as an implementation detail.

Researchers have developed many parallel sorting algorithms. Unfortunately, most of them are designed for theoretical models of parallel computation or special-purpose hardware, making them useless for those trying to implement an efficient sort on a general-purpose parallel computer.

Our focus in this chapter will be on methods suitable for multiple-CPU computers. We'll narrow our coverage in two additional ways. First, we'll be considering **internal sorts**—algorithms that sort sequences small enough to fit entirely in primary memory. (In contrast, an **external sort** orders a list of values too large to fit at one time in primary memory.) Second, the algorithms we consider here sort by comparing pairs of numbers. (Radix sort is an example of a sort that does not compare pairs of numbers.)

In this chapter we briefly summarize how quicksort works and then develop three parallel quicksort algorithms, assuming our target machine is a modern multicomputer that has equal latency and bandwidth between arbitrary pairs of processors.

14.2 QUICKSORT

Since you're probably familiar with the sequential quicksort algorithm, we'll only present a short refresher here. If you need a more in-depth review, consult Cormen et al. [18], Baase and Van Gelder [5], or another introductory analysis of algorithms textbook.

Quicksort, invented by C. A. R. Hoare about forty years ago [51], is a recursive algorithm that relies upon key comparisons to sort an unordered list. When passed a list of numbers, the algorithm selects one of these numbers to be the pivot. It partitions the list into two sublists: a "low list" containing numbers less than or equal to the pivot, and a "high list" containing those values greater than the pivot. It calls itself recursively to sort the two sublists. (If a sublist has no numbers, the call may be omitted.) The function ends by returning the concatenation of the low list, the pivot, and the high list.

For example, Figure 14.1 illustrates the operation of quicksort as it sorts the list of integers {79, 17, 14, 65, 89, 4, 95, 22, 63, 11}. Let's assume the algorithm always chooses the first list element to be the pivot value. With 79 as the pivot value, the low list contains {17, 14, 65, 4, 22, 63, 11} and the high list contains {89, 95}. The function calls itself recursively for each of these sublists.

The recursive execution of quicksort on the sublist containing {17, 14, 65, 4, 22, 63, 11} begins by removing 17 as the pivot value. The function creates a low list containing {14, 4, 11} and a high list containing {22, 63, 65}. Again, it calls itself recursively for both of these sublists.

The recursion eventually terminates because the removal of the pivot element guarantees the lengths of the lists continue to decrease. If a sublist has no elements, there is no need to sort it. If quicksort is called with a single element, that element becomes the pivot, and the algorithm simply returns that element as the sorted list.

Each function invocation results in the function returning the concatenation of the two sorted sublists and the pivot element. For example, look at the node Q(17,14,65,4,22,63,11). The call to Q(14,4,11) returns {4, 11, 14}. The pivot element is 17. The call to Q(65,22,63) returns {22, 63, 65}. Concatenating these lists, the function returns {4, 11, 14, 17, 22, 63, 65}.

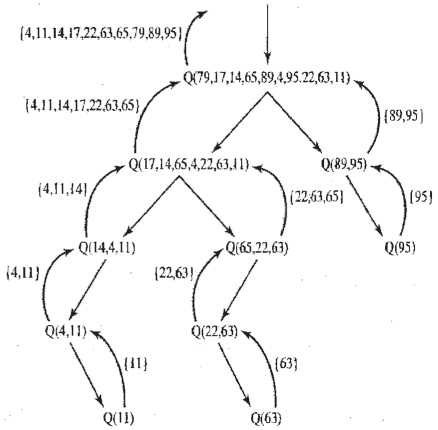


Figure 14.1 Sorting a 10-element list using quicksort. Each Q represents a call to quicksort. The algorithm removes the first element from the list, using it as a pivot to divide the list into two parts. It calls itself recursively to sort the two sublists. (The call is omitted for empty sublists.) It returns the concatenation of the sorted "low list," the pivot, and the sorted "high list."

14.3 A PARALLEL QUICKSORT ALGORITHM

Quicksort is a good starting point for a parallel sorting algorithm for two reasons. First, it is generally recognized as the fastest sorting algorithm based on comparison of keys, in the average case. We always prefer to base our parallel algorithms on the fastest sequential algorithms. Second, quicksort has some natural concurrency. When quicksort calls itself recursively, the two calls may be executed independently.

14.3.1 Definition of Sorted

We want an algorithm suitable for implementation on commodity clusters and multicomputers. Before we go any further, we must determine what it means for a multicomputer to sort an unordered list. We could say that at the beginning of the algorithm a single processor contains the unsorted list in its primary memory, and at the end of the algorithm the same processor would contain the sorted list in its primary memory. The problem with this definition is that it does not allow the maximum problem size to increase with the number of processors.

Instead, we'll adopt a different definition of what we mean by parallel sorting on a multicomputer. We assume that the list of unordered values is initially distributed evenly among the primary memories of the processors. At the completion of the algorithm, (1) the list stored in every processor's memory is sorted, and (2) the value of the last element on P_i's list is less than or equal to the value of the first element on P_{i+1}'s list, for $0 \leq i \leq p-2$. Note that the sorted values do not need to be distributed evenly among the processors.

14.3.2 Algorithm Development

Let's imagine how a parallel quicksort algorithm could work. Because the quicksort function calls itself twice, the number of "leaves" in the call graph is a power of 2 (ignoring omitted calls due to empty sublists). For this reason we're going to assume that the number of active processes is also a power of 2.

Take a look at Figure 14.2. The unsorted values are distributed among the memories of the processes. We choose a pivot value from one of the processes and broadcast it (Figure 14.2a). Each process divides its unsorted numbers into two lists: those less than or equal to the pivot, and those greater than the pivot. Each process in the upper half of the process list sends its "low list" to a partner process in the lower half of the process list and receives a "high list" in return (Figure 14.2b). Now the processes in the upper half of the process list have values greater than the pivot, and the processes in the lower half of the process list have values less than or equal to the pivot (Figure 14.2c).

At this point the processes divide themselves into two groups and the algorithm recurses. In each process group a pivot value is broadcast (Figure 14.2c). Processes divide their lists and swap values with partner processes (Figure 14.2d).

After $\log p$ recursions, every process has an unsorted list of values completely disjoint from the values held by the other processes. In other words, the largest value held by process i is smaller than the smallest value held by process $i+1$. Each process can sort the list it controls using (what else?) sequential quicksort, and the parallel algorithm terminates.

14.3.3 Analysis

If we were to implement this algorithm, how well would it perform? The execution time of the algorithm begins when the first process starts execution and ends when the last process finishes execution. That is why it is important to make sure all processes have about the same amount of work, so that they will all terminate at about the same time. In this algorithm, the amount of work is related to the number of elements controlled by the process.

Unfortunately, this algorithm is likely to do a poor job of balancing list sizes. For example, take another look at the example of sequential quicksort illustrated in Figure 14.1. The original list-splitting step produces one list of size 7 and another list of size 2. If the pivot value were equal to the median value, we could divide the list into equal parts, but in order to find the median we must go a long way toward sorting the list, which is what we're trying to do in the first place. So it's not practical to insist that the pivot value be the median value.

However, it is clear that we could do a better job balancing the list sizes among the processes if instead of choosing an arbitrary list element to be the

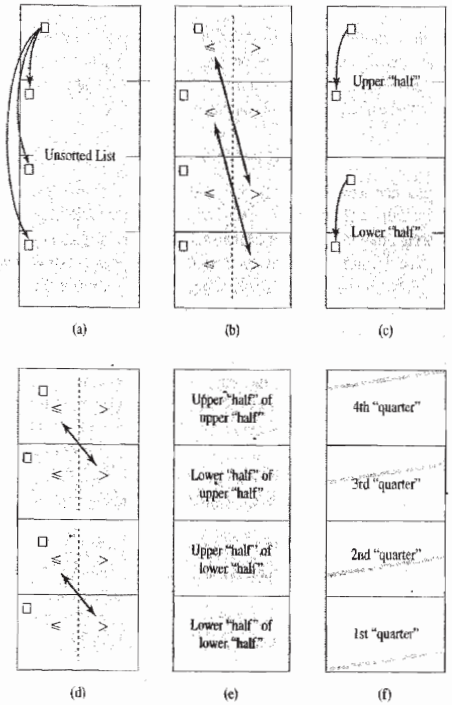


Figure 14.2 High-level view of a parallel quicksort algorithm. (a) Initially the unsorted values are distributed among the memories of all the processes. A single value is chosen as the pivot. The pivot is broadcast to the other processes. (b) Processes use the pivot to divide their numbers into those in the "lower half" and those in the "upper half." Each process in the upper half swaps values with a partner in the lower half. (c) The algorithm recurses. A single value from each "half" is chosen as the pivot for that "half" and broadcast to the other process responsible for that "half." (d) As in step (b), processes use the pivot to divide their numbers. Upper processes swap with lower processes, swapping smaller values for larger values. (e) At this point the largest value held by process i is less than the smallest value held by process $i + 1$. (f) Each process uses quicksort to sort the elements it controls. The list is now sorted.

pivot value, we chose a value more likely to be close to the true median of the sorted list. This insight is the motivation for the next parallel algorithm we will consider: hyperquicksort.

14.4 HYPERQUICKSORT

14.4.1 Algorithm Description

Hyperquicksort, invented by Wagar [108], begins where our first parallel quicksort algorithm ends, with each process using quicksort to sort its portion of the list. At this point condition 1 of the parallel sortedness requirement has been met, but not condition 2.

To meet the second condition, values still need to be moved from process to process. As in the first parallel quicksort algorithm, we will use a pivot value to divide the numbers into two groups: the lower "half" and the upper "half." *Because the list of elements on each process is sorted, the process responsible for supplying the pivot can use the median of its list as the pivot value.* This value is far more likely to be close to the true median of the entire unsorted list than the value of an arbitrarily chosen list element.

The next three steps of hyperquicksort are the same as the parallel quicksort algorithm we already developed. The process choosing the pivot broadcasts it to the other processes. Each process uses the pivot to divide its elements into a "low list" of values less than or equal to the pivot and a "high list" of values greater than the pivot. Every process in the upper half swaps its low list for a high list provided by a partner process in the lower half.

Now we add an additional step to hyperquicksort. After the swap, each process has a sorted sublist it retained and a sorted sublist it received from a partner. It merges the two lists it is responsible for so that the elements it controls are sorted. It is important that processes end this phase with sorted lists, because when the algorithm recurses, two processes will need to choose the median elements of their lists as pivots.

After $\log p$ such split-and-merge steps, the original hypercube of p processes has been divided into $\log p$ single-process hypercubes, and condition 2 is satisfied. Since the processes repeatedly merged lists to keep their local values sorted throughout the divide-and-swap steps, there is no need for them to call quicksort at the end of the algorithm. Figure 14.3 gives an example of hyperquicksort in action.

Hyperquicksort assumes the number of processes is a power of 2. If we arrange the processes as a hypercube, we can set up the communication pattern of the hyperquicksort algorithm so that all communications are between pairs of adjacent processes (see Figure 14.4). For this reason hyperquicksort was a particularly good fit for first-generation multicomputers, such as the Intel iPSC and the nCUBE/ten, that organized processors as a hypercube.

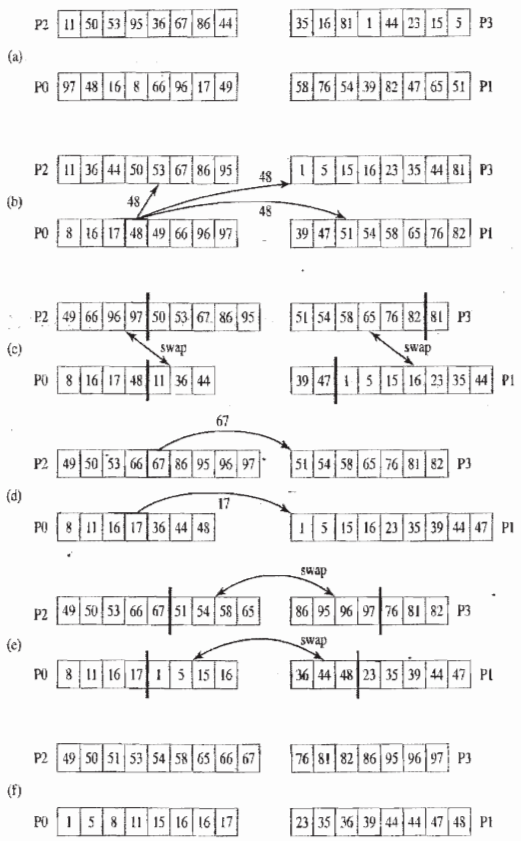


Figure 14.3 Illustration of the hyperquicksort algorithm. In this example 32 elements are being sorted on four processes logically organized as a two-dimensional hypercube. (a) Initially, each process has eight numbers. (b) Each process sorts its own list using quicksort. Process 0 broadcasts its median value, 48, to the other processes. (c) Processes in the lower half of the hypercube send values greater than 48 to processes in the upper half. The processes in the upper half send down values less than or equal to 48. (d) Each process merges the numbers it kept with the numbers it received. Process 0 broadcasts its median value to process 1, and process 2 broadcasts its median value to process 3. (e) Processes swap values across another hypercube dimension. (f) Each process merges the numbers it kept with the numbers it received. At this point the list is sorted.

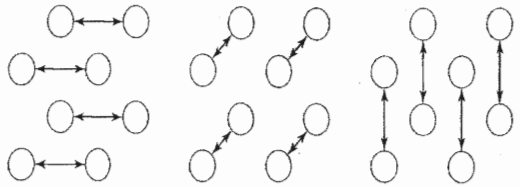


Figure 14.4 Communication pattern of hyperquicksort algorithm. In this example there are eight processes, so the algorithm goes through $\log p = 3$ swap-and-merge steps.

14.4.2 Isoefficiency Analysis

Let's determine the isoefficiency of hyperquicksort. We assume p processors are sorting n elements, where $n \gg p$. At the start of the algorithm each process has no more than $\lceil n/p \rceil$ values. The expected time complexity of the initial quicksort step is $\Theta[(n/p)\log(n/p)]$. Assuming that each process keeps $n/2p$ values and transmits $n/2p$ values in every split-and-merge step, the expected number of comparisons needed to merge the two lists into a single ordered list is n/p. Since the split-and-merge operation is executed for hypercubes of dimension $\log p, (\log p) - 1, \ldots, 1$, the expected number of comparisons performed over the split-and-merge phase of the algorithm is $\Theta(n/p)\log p$, and the expected number of comparisons performed during the entire algorithm is $\Theta[(n/p)(\log n + \log p)]$.

If processes are logically organized as a d-dimensional hypercube, broadcasting the splitter requires communication time $\Theta(d)$. However, since $n \gg p$, the broadcast time will be dwarfed by the time processes spend exchanging list elements. Assuming each process passes half its values each iteration, the time needed to send and receive $n/2p$ sorted values to and from the partner process is $\Theta(n/p)$. There are $\log p$ iterations. Hence the expected communication time for the split-and-merge phase is $\Theta(n \log p/p)$. Since the original quicksort phase requires no interprocess communication, this value is the expected communication complexity of the entire hyperquicksort algorithm.

The sequential time complexity of quicksort is $n \log n$. The communication overhead of hyperquicksort is p times the communication complexity, or $\Theta(n \log p)$. Hence the isoefficiency function for hyperquicksort is

$$n \log n \geq Cn \log p \Rightarrow \log n \geq C \log p \Rightarrow n \geq p^C$$

Memory requirements for this problem are linear; that is, $M(n) = n$. So the scalability function for hyperquicksort is p^{C-1}. The value of C determines the scalability of the parallel system. If $C > 2$, scalability is low.

There is another factor that, when considered, makes the scalability of hyperquicksort even worse. Our analysis has assumed that the median element chosen

by a single process is always the true median, and that every process always sends $n/(2p)$ elements to its partner and receives $n/(2p)$ elements in return each iteration. In reality, the median elements are not the true medians, and the workload among the processes becomes unbalanced. Processes with more elements spend more time communicating and merging. The imbalance tends to increase as the number of processors increases. That's because each process's portion of the complete list is smaller. With a smaller sample, it is less likely that the process's median value will be close to the true median value.

In short, hyperquicksort has two weaknesses that limit its usefulness. First, the expected number of times a key is passed from one process to another is $(\log p)/2$. This communication overhead limits the scalability of the parallel algorithm. We could reduce this overhead if we could find a way to route keys directly to their final destinations. Second, the way in which the splitter values are chosen can lead to workload imbalances among the processes. If we could get samples from all of the processes, we would have a better chance of dividing the list elements evenly among them. These two ideas are incorporated into our third and final algorithm: parallel sorting by regular sampling.

14.5 PARALLEL SORTING BY REGULAR SAMPLING

Parallel sorting by regular sampling (PSRS), developed by Li et al. [74], has three advantages over hyperquicksort. It keeps list sizes more balanced among the processes, it avoids repeated communications of the keys, and it does not require that the number of processes be a power of 2.

14.5.1 Algorithm Description

The PSRS algorithm has four phases (Figure 14.5). Suppose we're sorting n keys on p processes. In phase 1, each process uses the sequential quicksort algorithm to sort its share of the elements (no more than $\lceil n/p \rceil$ elements per process). Each process selects data items at local indices $0, n/p^2, 2n/p^2, \ldots, (p-1)(n/p^2)$ as a regular sample of its locally sorted block.

In the second phase of the algorithm, one process gathers and sorts the local regular samples. It selects $p-1$ pivot values from the sorted list of regular samples. The pivot values are at indices $p + \lfloor p/2 \rfloor - 1, 2p + \lfloor p/2 \rfloor - 1, \ldots, (p-1)p + \lfloor p/2 \rfloor$ in the sorted list of regular samples. At this point each process partitions its sorted sublist into p disjoint pieces, using the pivot values as separators between the pieces.

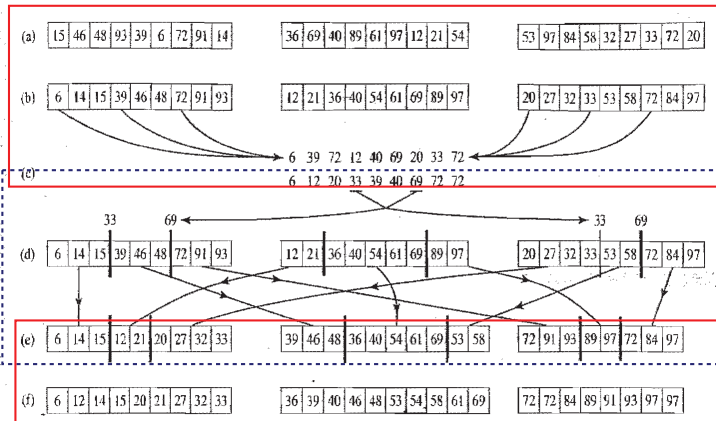
In the third phase of the algorithm each process i keeps its ith partition and sends the jth partition to process j, for all $j \neq i$.

During the fourth phase of the algorithm each process merges its p partitions into a single list. The values on this list are disjoint from the values on the lists of the other processes. At the end of this phase the elements are sorted.

OpenMP

MPI

OpenMP



OpenMP - 1 Proc p threads

MPI - p Processos

OpenMP - 1 Proc p threads

Figure 14.5 This example illustrates how three processes would sort 27 elements using the PSRS algorithm. (a) Original unsorted list of 27 elements is divided among three processes. (b) Each process sorts its share of the list using sequential quicksort. (c) Each process selects regular samples from its sorted sublist. A single process gathers these samples, sorts them, and broadcasts pivot elements from the sorted list of samples to the other processes. (d) Processes use pivot elements computed in step (c) to divide their sorted sublists into three parts. (e) Processes perform an all-to-all communication to migrate the sorted sublist parts to the correct processes. (f) Each process merges its sorted sublists.

Li et al. [74] have proven that the largest number of elements any process may have to merge in phase 4 of the PSRS algorithm is less than $2n/p$; that is, twice its share of the elements. In actuality, experiments have shown that if the elements are selected from a uniform random distribution, the largest partition size is usually no more than a few percent larger than n/p, the average partition size.

14.5.2 Isoefficiency Analysis

Let's determine the isoefficiency of the PSRS algorithm, assuming p processors are sorting n elements, where $n \gg p$.

In phase 1, each process performs quicksort on n/p elements. The time complexity of this step is $\Theta[(n/p)\log(n/p)]$. At the end of phase 1, a single process gathers p regular samples from each of the other $p-1$ processes. Since relatively few values are being passed, message latency is likely to be the dominant term of this step. Hence the communication complexity of the gather is $\Theta(\log p)$.

In phase 2 of the PSRS algorithm one process sorts the p^2 elements of Y. This sort has time complexity $\Theta(p^2 \log p^2) = \Theta(p^2 \log p)$. The sorting process broadcasts $p-1$ pivots to the other processes. Since only $p-1$ values are

being communicated, message latency is most likely the dominant term, and the communication complexity is $\Theta(\log p)$.

In phase 3 of the algorithm, each process uses the pivots to divide its portion of the list into p sections. The processes then perform an all-to-all communication. In the all-to-all communication each process sends and receives $p-1$ messages. Assuming the list sizes are balanced, the total number of elements sent per process is about $(p-1)n/p^2$, which is approximately n/p. Since $n \gg p$, the messages are long, and the time needed to pass a message is dominated by the time needed to transmit its elements, rather than its latency. Hence it makes sense to structure the all-to-all communication so that each process sends and receives $p-1$ messages. That way, every list element is passed only once—directly to the process that needs it. We assume that the processor interconnection network supports p simultaneous message transmissions. In other words, the capacity of the interconnection network increases with the number of processors. (As we saw in Chapter 2, the 4-ary hypertree is an example of an interconnection network for which the bisection width increases linearly with the number of processors.) With this assumption, the overall communication complexity of this step is $\Theta(n/p)$.

In the fourth phase of the algorithm each process merges p sorted sublists. Assuming the list sizes are balanced (which experiments show to be a reasonable assumption), the time required for the merge is $\Theta[(n/p)\log p]$.

The overall computational complexity of the PSRS algorithm is

$$\Theta[(n/p)\log(n/p) + p^2 \log p + (n/p)\log p]$$

Since $n \gg p$, the time needed to sort the regular samples is negligible. The constant of proportionality for the merge step in phase 4 is higher than for the quicksort in phase 1. Hence we need to include the $\Theta[(n/p)\log p]$ term for this phase. Hence the overall computational complexity is

$$\Theta[(n/p)(\log n + \log p)]$$

Assuming the communication capacity of the parallel system increases linearly with p, the overall communication complexity is

$$\Theta(\log p + n/p)$$

Again, since $n \gg p$, the communication time is dominated by the time the processes spend sending sublists to each other, so we can simplify the communication complexity to

$$\Theta(n/p)$$

The parallel overhead of this system is p times the communication complexity, or $\Theta(n)$, plus p times the complexity of the parallel merge step, or $\Theta(n \log p)$. The isoefficiency function for the PSRS algorithm is

$$n \log n \geq Cn \log p \Rightarrow \log n \geq C \log p \Rightarrow n \geq p^C$$

Since $M(n) = n$, the scalability function is

$$p^C/p = p^{C-1}$$

This is the same scalability function we saw for hyperquicksort. However, the PSRS algorithm is likely to achieve higher speedup than hyperquicksort because it keeps the number of keys per processor well balanced.

14.6 SUMMARY

Sorting is an important utility on both serial and parallel computers. In this chapter we have looked at three quicksort-based parallel algorithms suitable for implementation on both multicomputers and multiprocessors.

Our first algorithm introduces the idea of repeatedly halving the lists and exchanging values between pairs of processes until the processes control non-overlapping sublists. Unfortunately, it does not do a good job balancing values among processes.

Hyperquicksort retains the idea of recursively splitting and exchanging sublists. However, by moving the quicksort step from the end of the algorithm to the beginning, it allows a better choice of the pivot value. The design of hyperquicksort was inspired by the architecture of many 1980s multicomputers having a hypercube processor organization. In these systems the time required to send a message was directly proportional to the number of "hops" between the sending and the receiving processors. Hyperquicksort can be implemented so that all messages are between adjacent processors. Hence it optimizes communication time on hypercubes. Because hyperquicksort relies upon a single process to choose the pivot value for the entire cube (or subcube), as the number of processors grows, the quality of the pivot value degrades. As the pivot value strays from the true median, the workloads among the processes become imbalanced, lowering efficiency.

Parallel sorting by regular sampling (PSRS) addresses the load imbalance problem of hyperquicksort by choosing pivot elements from a regular sample of elements held by all the processes. It has the additional advantage that its single all-to-all communication can be implemented so that each element is moved only once (rather than $\log p$ times). This is a good fit for contemporary switch-based clusters, in which the time needed to send a message is about the same for any pair of processors.

14.7 KEY TERMS

external sort
hyperquicksort
internal sort
key
parallel sorting by regular sampling
record
satellite data
sorting problem