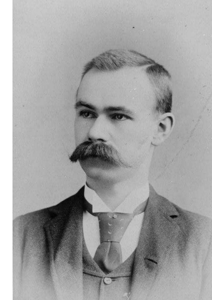
5

Sorting and Selection



Telephone directories are sorted alphabetically by last name. Why? Because a sorted index can be searched quickly. Even in the telephone directory of a huge city, one can find a name in a few seconds. In an unsorted index, nobody would even try to find a name. This chapter teaches you how to turn an unordered collection of elements into an ordered collection, i.e., how to sort *the collection. The sorted collection can then be searched fast. We will get to know several algorithms for sorting; the different algorithms are suited for different situations, for example sorting in main memory or sorting in external memory, and illustrate different algorithmic paradigms. Sorting has many other uses as well. An early example of a massive data-processing task was the statistical evaluation of census data; 1500 people needed seven years to manually process data from the US census in 1880. The engineer Herman Hollerith,*[1] *who participated in this evaluation as a statistician, spent much of the 10 years to the next census developing counting and sorting machines for mechanizing this gigantic endeavor. Although the 1890 census had to evaluate more people and more questions, the basic evaluation was finished in 1891. Hollerith's company continued to play an important role in the development of the information-processing industry; since 1924, it has been known as International Business Machines (IBM). Sorting is important for census statistics because one often wants to form subcollections, for example, all persons between age 20 and 30 and living on a farm. Two applications of sorting solve the problem. First, we sort all persons by age and form the subcollection of persons between 20 and 30 years of age. Then we sort the subcollection by the type of the home (house, apartment complex, farm, ...) and extract the subcollection of persons living on a farm.*

Although we probably all have an intuitive concept of what *sorting* is about, let us give a formal definition. The input is a sequence $s = \langle e_1, \ldots, e_n \rangle$ of n elements. Each element e_i has an associated *key* $k_i = key(e_i)$. The keys come from an ordered

[1] The photograph was taken by C. M. Bell; US Library of Congress' Prints and Photographs Division, ID cph.3c15982.

universe, i.e., there is a *linear order* (also called a *total order*) $\leq$ defined on the keys.[2] For ease of notation, we extend the comparison relation to elements so that $e \leq e'$ if and only if $key(e) \leq key(e')$. Since different elements may have equal keys, the relation $\leq$ on elements is only a linear preorder. The task is to produce a sequence $s' = \langle e'_1, \ldots, e'_n \rangle$ such that s' is a permutation of s and such that $e'_1 \leq e'_2 \leq \cdots \leq e'_n$. Observe that the ordering of elements with equal key is arbitrary.

Although different comparison relations for the same data type may make sense, the most frequent relations are the obvious order for numbers and the *lexicographic order* (see Appendix A) for tuples, strings, and sequences. The lexicographic order for strings comes in different flavors. We may treat corresponding lower-case and upper-case characters as being equivalent, and different rules for treating accented characters are used in different contexts.

Exercise 5.1. Given linear orders $\leq_A$ for A and $\leq_B$ for B, define a linear order on $A \times B$.

Exercise 5.2. Consider the relation R over the complex numbers defined by $x \mathrel{R} y$ if and only if $|x| \leq |y|$. Is it total? Is it transitive? Is it antisymmetric? Is it reflexive? Is it a linear order? Is it a linear preorder?

Exercise 5.3. Define a total order for complex numbers with the property that $x \leq y$ implies $|x| \leq |y|$.

Sorting is a ubiquitous algorithmic tool; it is frequently used as a preprocessing step in more complex algorithms. We shall give some examples.

- *Preprocessing for fast search*. In Sect. 2.7 on binary search, we have already seen that a sorted directory is easier to search, both for humans and for computers. Moreover, a sorted directory supports additional operations, such as finding all elements in a certain range. We shall discuss searching in more detail in Chap. 7. Hashing is a method for searching unordered sets.
- *Grouping*. Often, we want to bring equal elements together to count them, eliminate duplicates, or otherwise process them. Again, hashing is an alternative. But sorting has advantages, since we shall see rather fast, space-efficient, deterministic sorting algorithm that scale to huge data sets.
- *Processing in a sorted order*. Certain algorithms become very simple if the inputs are processed in sorted order. Exercise 5.4 gives an example. Other examples are Kruskal's algorithm presented in Sect. 11.3, and several of the algorithms for the knapsack problem presented in Chap. 12. You may also want to remember sorting when you solve Exercise 8.6 on interval graphs.

In Sect. 5.1, we shall introduce several simple sorting algorithms. They have quadratic complexity, but are still useful for small input sizes. Moreover, we shall

[2] A linear or total order is a reflexive, transitive, total, and antisymmetric relation such as the relation $\leq$ on the real numbers. A reflexive, transitive, and total relation is called a linear preorder or linear quasiorder. An example is the relation $\mathrm{R} \subseteq \mathbb{R} \times \mathbb{R}$ defined by $x \mathrel{R} y$ if and only if $|x| \leq |y|$; see Appendix A for details.

learn some low-level optimizations. Section 5.3 introduces *mergesort*, a simple divide-and-conquer sorting algorithm that runs in time $O(n \log n)$. Section 5.5 establishes that this bound is optimal for all *comparison-based* algorithms, i.e., algorithms that treat elements as black boxes that can only be compared and moved around. The *quicksort* algorithm described in Sect. 5.6 is again based on the divide-and-conquer principle and is perhaps the most frequently used sorting algorithm. Quicksort is also a good example of a randomized algorithm. The idea behind quicksort leads to a simple algorithm for a problem related to sorting. Section 5.8 explains how the kth smallest of n elements can be *selected* in time $O(n)$. Sorting can be made even faster than the lower bound obtained in Sect. 5.5 by exploiting the internal structure of the keys, for example by exploiting the fact that numbers are sequences of digits. This is the content of Sect. 5.10. Section 5.12 generalizes quicksort and mergesort to very good algorithms for sorting inputs that do not fit into internal memory.

Most parallel algorithms in this chapter build on the sequential algorithms. We begin in Sect. 5.2 with an inefficient yet fast and simple algorithm that can be used as a subroutine for sorting very small inputs very quickly. Parallel mergesort (Sect. 5.4) is efficient for inputs of size $\Omega(p \log p)$ and a good candidate for sorting relatively small inputs on a shared-memory machine. Parallel quicksort (Sect. 5.7) can be used in similar circumstances and might be a good choice on distributed-memory machines. There is also an almost in-place variant for shared memory. Selection (Sect. 5.9) can be parallelized even better than sorting. In particular, there is a communication-efficient algorithm that does not need to move the data. The noncomparison-based algorithms in Sect. 5.10 are rather straightforward to parallelize (Sect. 5.11). The external-memory algorithms in Sect. 5.12 are the basis of very efficient parallel algorithms for large inputs. Parallel sample sort (Sect. 5.13) and parallel multiway mergesort (Sect. 5.14) are only efficient for rather large inputs of size $\omega(p^2)$, but the elements need to be moved only once. Since sample sort is a good compromise between simplicity and efficiency, we give two implementations – one for shared memory and the other for distributed memory. Finally, in Sect. 5.15 we outline a sophisticated algorithm that is asymptotically efficient even for inputs of size p. This algorithm is a recursive generalization of sample sort that uses the fast, inefficient algorithm in Sect. 5.2 for sorting the sample.

Exercise 5.4 (a simple scheduling problem). A hotel manager has to process n advance bookings of rooms for the next season. His hotel has k identical rooms. Bookings contain an arrival date and a departure date. He wants to find out whether there are enough rooms in the hotel to satisfy the demand. Design an algorithm that solves this problem in time $O(n \log n)$. Hint: Consider the multiset of all arrivals and departures. Sort the set and process it in sorted order.

Exercise 5.5 ((database) sort join). As in Exercise 4.5, consider two relations $R \subseteq A \times B$ and $Q \subseteq B \times C$ with $A \neq C$ and design an algorithm for computing the natural *join* of R and Q

$$R \bowtie Q := \{(a,b,c) \subseteq A \times B \times C : (a,b) \in R \wedge (b,c) \in Q\}.$$

Show how to obtain running time $\mathrm{O}((|R|+|Q|)\log(|R|+|Q|)+|R\bowtie Q|)$ with a deterministic algorithm.

Exercise 5.6 (sorting with a small set of keys). Design an algorithm that sorts n elements in $\mathrm{O}(k\log k+n)$ expected time if there are only k different keys appearing in the input. Hint: Combine hashing and sorting, and use the fact that k keys can be sorted in time $\mathrm{O}(k\log k)$.

Exercise 5.7 (checking). It is easy to check whether a sorting routine produces a sorted output. It is less easy to check whether the output is also a permutation of the input. But here is a fast and simple Monte Carlo algorithm for integers: (a) Show that $\langle e_1,\ldots,e_n\rangle$ is a permutation of $\langle e'_1,\ldots,e'_n\rangle$ if and only if the polynomial $q(z):=\prod_{i=1}^{n}(z-e_i)-\prod_{i=1}^{n}(z-e'_i)$ is identically 0. Here, z is a variable. (b) For any $\varepsilon>0$, let p be a prime with $p>\max\{n/\varepsilon,e_1,\ldots,e_n,e'_1,\ldots,e'_n\}$. Now the idea is to evaluate the above polynomial $\bmod p$ for a random value $z\in 0..p-1$. Show that if $\langle e_1,\ldots,e_n\rangle$ is *not* a permutation of $\langle e'_1,\ldots,e'_n\rangle$, then the result of the evaluation is 0 with probability at most ε. Hint: A polynomial of degree n that is not identically 0 modulo p has at most n 0's in $0..p-1$ when evaluated modulo p.

Exercise 5.8 (permutation checking by hashing). Consider sequences A and B where A is not a permutation of B. Suppose $h: \mathit{Element}\to 0..U-1$ is a random hash function. Show that $\mathrm{prob}(\sum_{e\in A}h(e)=\sum_{e\in B}h(e))\leq 1/U$. Hint: Focus on one element that occurs a different number of times in A and B.

5.1 Simple Sorters

We shall introduce two simple sorting techniques: *selection sort* and *insertion sort*.

Selection sort repeatedly selects the smallest element from the input sequence, deletes it, and adds it to the end of the output sequence. The output sequence is initially empty. The process continues until the input sequence is exhausted. For example,

$$\langle\rangle,\langle 4,7,1,1\rangle \rightsquigarrow \langle 1\rangle,\langle 4,7,1\rangle \rightsquigarrow \langle 1,1\rangle,\langle 4,7\rangle \rightsquigarrow \langle 1,1,4\rangle,\langle 7\rangle \rightsquigarrow \langle 1,1,4,7\rangle,\langle\rangle.$$

The algorithm can be implemented such that it uses a single array of n elements and works *in-place*, i.e., it needs no additional storage beyond the input array and a constant amount of space for loop counters, etc. The running time is quadratic.

Exercise 5.9 (simple selection sort). Implement selection sort so that it sorts an array with n elements in time $\mathrm{O}(n^2)$ by repeatedly scanning the input sequence. The algorithm should be in-place, i.e., the input sequence and the output sequence should share the same array. Hint: The implementation operates in n phases numbered 1 to n. At the beginning of the ith phase, the first $i-1$ locations of the array contain the $i-1$ smallest elements in sorted order and the remaining $n-i+1$ locations contain the remaining elements in arbitrary order.

```
Procedure insertionSort(a : Array [1..n] of Element)
    for i := 2 to n do
        invariant a[1] ≤ ··· ≤ a[i−1]
        // move a[i] to the right place
        e := a[i]
        if e < a[1] then                                          // new minimum
            for j := i downto 2 do  a[j] := a[j−1]
            a[1] := e
        else                                          // use a[1] as a sentinel
            for j := i downto −∞ while a[j−1] > e do  a[j] := a[j−1]
            a[j] := e
```

Fig. 5.1. Insertion sort

In Sect. 6.6, we shall learn about a more sophisticated implementation where the input sequence is maintained as a *priority queue*. Priority queues support efficient repeated selection of the minimum element. The resulting algorithm runs in time $\mathrm{O}(n \log n)$ and is frequently used. It is efficient, it is deterministic, it works in-place, and the input sequence can be dynamically extended by elements that are larger than all previously selected elements. The last feature is important in discrete-event simulations, where events have to be processed in increasing order of time and processing an event may generate further events in the future.

Selection sort maintains the invariant that the output sequence is sorted by carefully choosing the element to be deleted from the input sequence. *Insertion sort* maintains the same invariant by choosing an arbitrary element of the input sequence but taking care to insert this element in the right place in the output sequence. For example,

$$\langle\rangle,\langle 4,7,1,1\rangle \rightsquigarrow \langle 4\rangle,\langle 7,1,1\rangle \rightsquigarrow \langle 4,7\rangle,\langle 1,1\rangle \rightsquigarrow \langle 1,4,7\rangle,\langle 1\rangle \rightsquigarrow \langle 1,1,4,7\rangle,\langle\rangle.$$

Figure 5.1 gives an in-place array implementation of insertion sort. The implementation is straightforward except for a small trick that allows the inner loop to use only a single comparison. When the element e to be inserted is smaller than all previously inserted elements, it can be inserted at the beginning without further tests. Otherwise, it suffices to scan the sorted part of a from right to left while e is smaller than the current element. This process has to stop, because $a[1] \leq e$.

In the worst case, insertion sort is quite slow. For example, if the input is sorted in decreasing order, each input element is moved all the way to $a[1]$, i.e., in iteration i of the outer loop, i elements have to be moved. Overall, we obtain

$$\sum_{i=2}^{n}(i-1) = -n + \sum_{i=1}^{n} i = \frac{n(n+1)}{2} - n = \frac{n(n-1)}{2} = \Omega\left(n^2\right)$$

movements of elements; see also (A.12).

Nevertheless, insertion sort is useful. It is fast for small inputs (say, $n \leq 10$) and hence can be used as the base case in divide-and-conquer algorithms for sorting.

Furthermore, in some applications the input is already "almost" sorted, and in this situation insertion sort will be fast.

Exercise 5.10 (almost sorted inputs). Prove that insertion sort runs in time $O(n+D)$, where $D = \sum_i |r(e_i) - i|$ and $r(e_i)$ is the *rank* (position) of e_i in the sorted output.

Exercise 5.11 (average-case analysis). Assume that the input to an insertion sort is a permutation of the numbers 1 to n. Show that the average execution time over all possible permutations is $\Omega(n^2)$. Hint: Argue formally that about one-third of the input elements in the right third of the array have to be moved to the left third of the array. Can you improve the argument to show that, on average, $n^2/4 - O(n)$ iterations of the inner loop are needed?

Exercise 5.12 (insertion sort with few comparisons). Modify the inner loops of the array-based insertion sort algorithm in Fig. 5.1 so that it needs only $O(n \log n)$ comparisons between elements. Hint: Use binary search as discussed in Chap. 7. What is the running time of this modification of insertion sort?

Exercise 5.13 (efficient insertion sort?). Use the data structure for sorted sequences described in Chap. 7 to derive a variant of insertion sort that runs in time $O(n \log n)$.

***Exercise 5.14 (formal verification).** Use your favorite verification formalism, for example Hoare calculus, to prove that insertion sort produces a permutation of the input.

5.2 Simple, Fast, and Inefficient Parallel Sorting

In parallel processing, there are also cases where spending a quadratic number of comparisons to sort a small input makes sense. Assume that the PEs are arranged as a quadratic matrix with PE indices written as pairs. Assume furthermore that we have input elements e_i at the diagonal PEs with index (i,i). For simplicity, assume also that all elements are different. In this situation, there is a simple and fast algorithm that sorts in logarithmic time: PE (i,i) first broadcasts its element along row i and column i. This can be done in logarithmic time; see Sect. 13.1. Now, for every pair (i,j) of input elements, there is a dedicated processor that can compare them in constant time. The rank of element i can then be determined by adding the 0–1 value $[e_i \geq e_j]$ along each row. We can already view this mapping of elements to ranks as the output of the sorting algorithm. If desired, we can also use this information to permute the elements. For example, we could send the elements with rank i to PE (i,i).

At the first glance, this sounds like a rather useless algorithm, since its efficiency is $o(1)$. However, there are situations where speed is more important than efficiency, for example for the fast parallel selection algorithm discussed in Sect. 5.9, where we use sorting a sample to obtain a high-quality pivot. Also, note that in that situation, even finding a single random pivot requires a prefix sum and a broadcast, i.e., taking

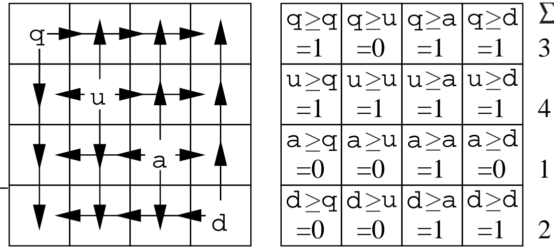


Fig. 5.2. Brute force ranking of four elements on 4×4 PEs

a random pivot is only a constant factor faster than choosing the median of a sample of size $\sqrt{p}$. Figure 5.2 gives an example.

We can obtain wider applicability by generalizing the algorithm to handle larger inputs. Here, we outline an algorithm described in more detail in [23] and care only about computing the rank of each element. Now, the PEs are arranged into an $a \times b$ matrix. Each PE has a (possibly empty) set of input elements. Each PE sorts its elements locally. Then we redistribute the elements such that PE (i, j) has two sequences I and J, where I contains all elements from row i and J contains all elements from column J. This can be done using all-gather operations along the rows and columns; see Sect. 13.5. Additionally, we ensure that the sequences are sorted by replacing the local concatenation operations in the all-gather algorithm by a merge operation. Subsequently, elements from I are ranked with respect to the elements in J, i.e., for each element $x \in I$, we count how many elements $y \in J$ have $y \leq x$. This can be done in linear time by merging I and J. The resulting local rank vectors are then added along the rows. Figure 5.3 gives an example.

Overall, if all rows and columns contain a balanced number of elements, we get a total execution time

$$\mathrm{O}\left(\alpha \log p + \beta \tfrac{n}{\sqrt{p}} + \frac{n}{p} \log \frac{n}{p}\right). \tag{5.1}$$

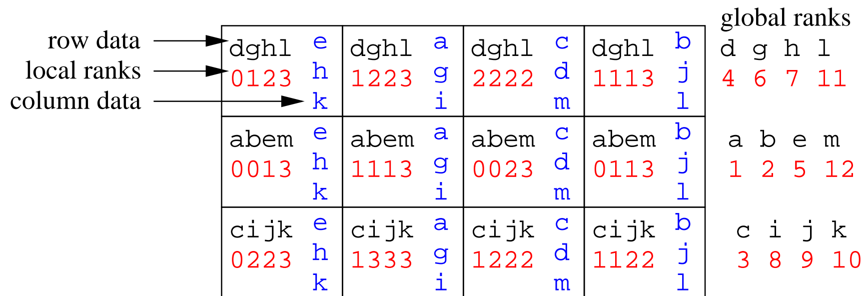


Fig. 5.3. Fast, inefficient ranking of $\langle d, g, h, l, a, b, e, m, c, i, j, k \rangle$ on 3×4 PEs.

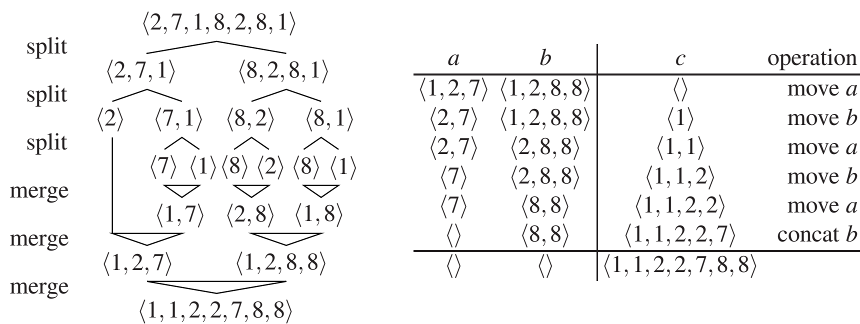
5.3 Mergesort – an $O(n \log n)$ Sorting Algorithm

Mergesort is a straightforward application of the divide-and-conquer principle. The unsorted sequence is split into two parts of about equal size. The parts are sorted recursively, and the sorted parts are merged into a single sorted sequence. This approach is efficient because merging two sorted sequences a and b is quite simple. The globally smallest element is either the first element of a or the first element of b. So, we move the smaller element to the output, find the second smallest element using the same approach, and iterate until all elements have been moved to the output. Figure 5.4 gives pseudocode, and Fig. 5.5 illustrates a sample execution. If the sequences are represented as linked lists (see Sect. 3.2), no allocation and deallocation of list items is needed. Each iteration of the inner loop of *merge* performs one element comparison and moves one element to the output. Each iteration takes constant time. Hence, merging runs in linear time.

```
Function mergeSort(⟨e_1,...,e_n⟩) : Sequence of Element
    if n = 1 then return ⟨e_1⟩
    else return merge(  mergeSort(⟨e_1,...,e_⌊n/2⌋⟩),
                        mergeSort(⟨e_⌊n/2⌋+1,...,e_n⟩))

// merging two sequences represented as lists
Function merge(a, b : Sequence of Element) : Sequence of Element
    c := ⟨⟩
    loop
        invariant a, b, and c are sorted and ∀e ∈ c, e' ∈ a ∪ b : e ≤ e'
        if a.isEmpty          then  c.concat(b); return c
        if b.isEmpty          then  c.concat(a); return c
        if a.first ≤ b.first  then  c.moveToBack(a.PopFront)
        else                        c.moveToBack(b.PopFront)
```

Fig. 5.4. Mergesort



a	b	c	operation
$\langle 1,2,7\rangle$	$\langle 1,2,8,8\rangle$	$\langle\rangle$	move a
$\langle 2,7\rangle$	$\langle 1,2,8,8\rangle$	$\langle 1\rangle$	move b
$\langle 2,7\rangle$	$\langle 2,8,8\rangle$	$\langle 1,1\rangle$	move a
$\langle 7\rangle$	$\langle 2,8,8\rangle$	$\langle 1,1,2\rangle$	move b
$\langle 7\rangle$	$\langle 8,8\rangle$	$\langle 1,1,2,2\rangle$	move a
$\langle\rangle$	$\langle 8,8\rangle$	$\langle 1,1,2,2,7\rangle$	concat b
$\langle\rangle$	$\langle\rangle$	$\langle 1,1,2,2,7,8,8\rangle$	

Fig. 5.5. Execution of *mergeSort*$(\langle 2,7,1,8,2,8,1\rangle)$. The *left* part illustrates the recursion in *mergeSort* and the *right* part illustrates the *merge* in the outermost call.

Theorem 5.1. *The function merge, applied to sequences of total length n, executes in time* $O(n)$ *and performs at most* $n-1$ *element comparisons.*

For the running time of mergesort, we obtain the following result.

Theorem 5.2. *Mergesort runs in time* $O(n \log n)$ *and performs no more than* $\lceil n \log n \rceil$ *element comparisons.*

Proof. Let $C(n)$ denote the worst-case number of element comparisons performed. We have $C(1) = 0$ and $C(n) \le C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + n - 1$, using Theorem 5.1. The master theorem for recurrence relations (2.5) suggests $C(n) = O(n \log n)$. We next give two proofs that show explicit constants. The first proof shows $C(n) \le 2n \lceil \log n \rceil$, and the second proof shows $C(n) \le n \lceil \log n \rceil$.

For n a power of two, we define $D(1) = 0$ and $D(n) = 2D(n/2) + n$. Then $D(n) = n \log n$ for n a power of two by either the master theorem for recurrence relations or by a simple induction argument.[3] We claim that $C(n) \le D(2^k)$, where k is such that $2^{k-1} < n \le 2^k$. Then $C(n) \le D(2^k) = 2^k k \le 2n \lceil \log n \rceil$. It remains to argue the inequality $C(n) \le D(2^k)$. We use induction on k. For $k = 0$, we have $n = 1$ and $C(1) = 0 = D(1)$, and the claim certainly holds. For $k > 1$, we observe that $\lfloor n/2 \rfloor \le \lceil n/2 \rceil \le 2^{k-1}$, and hence

$$C(n) \le C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + n - 1 \le 2D(2^{k-1}) + 2^k - 1 \le D(2^k).$$

This completes the first proof.

We turn now to the second, refined proof. We prove

$$C(n) \le n \lceil \log n \rceil - 2^{\lceil \log n \rceil} + 1 \le n \log n$$

by induction over n. For $n = 1$, the claim is certainly true. So, assume $n > 1$. Let k be such that $2^{k-1} < \lceil n/2 \rceil \le 2^k$, i.e. $k = \lceil \log \lceil n/2 \rceil \rceil$. Then $C(\lceil n/2 \rceil) \le \lceil n/2 \rceil k - 2^k + 1$ by the induction hypothesis. If $\lfloor n/2 \rfloor > 2^{k-1}$, then k is also equal to $\lceil \log \lfloor n/2 \rfloor \rceil$ and hence $C(\lfloor n/2 \rfloor) \le \lfloor n/2 \rfloor k - 2^k + 1$ by the induction hypothesis. If $\lfloor n/2 \rfloor = 2^{k-1}$ and hence $k - 1 = \lceil \log \lfloor n/2 \rfloor \rceil$, the induction hypothesis yields $C(\lfloor n/2 \rfloor) = \lfloor n/2 \rfloor (k - 1) - 2^{k-1} + 1 = 2^{k-1}(k-1) - 2^{k-1} + 1 = \lfloor n/2 \rfloor k - 2^k + 1$. Thus we have the same bound for $C(\lfloor n/2 \rfloor)$ in both cases, and hence

$$\begin{aligned} C(n) &\le C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + n - 1 \\ &\le \left(\lfloor n/2 \rfloor k - 2^k + 1 \right) + \left(\lceil n/2 \rceil k - 2^k + 1 \right) + n - 1 \\ &= nk + n - 2^{k+1} + 1 = n(k+1) - 2^{k+1} + 1 = n \lceil \log n \rceil - 2^{\lceil \log n \rceil} + 1. \end{aligned}$$

It remains to argue that $nk - 2^k + 1 \le n \log n$ for $k = \lceil \log n \rceil$. If $n = 2^k$, the inequality clearly holds. If $n < 2^k$, we have $nk - 2^k + 1 \le n(k-1) + (n - 2^k + 1) \le n(k-1) \le n \log n$.

The bound for the execution time can be verified using a similar recurrence relation. □

[3] For $n = 1 = 2^0$, we have $D(1) = 0 = n \log n$, and for $n = 2^k$ and $k \ge 1$, we have $D(n) = 2D(n/2) + n = 2(n/2)\log(n/2) + n = n(\log n - 1) + n = n \log n$.

Mergesort is the method of choice for sorting linked lists and is therefore frequently used in functional and logical programming languages that have lists as their primary data structure. In Sect. 5.5, we shall see that mergesort is basically optimal as far as the number of comparisons is concerned; so it is also a good choice if comparisons are expensive. When implemented using arrays, mergesort has the additional advantage that it streams through memory in a sequential way. This makes it efficient in memory hierarchies. Section 5.12 has more on that issue. However, mergesort is not the usual method of choice for an efficient array-based implementation, since it does not work in-place, but needs additional storage space; but see Exercise 5.19.

Exercise 5.15. Explain how to insert k new elements into a sorted list of size n in time $O(k \log k + n)$.

Exercise 5.16. We have discussed *merge* for lists but used abstract sequences for the description of *mergeSort*. Give the details of *mergeSort* for linked lists.

Exercise 5.17. Implement mergesort in a functional programming language.

Exercise 5.18. Give an efficient array-based implementation of mergesort in your favorite imperative programming language. Besides the input array, allocate one auxiliary array of size n at the beginning and then use these two arrays to store all intermediate results. Can you improve the running time by switching to insertion sort for small inputs? If so, what is the optimal switching point in your implementation?

Exercise 5.19. The way we describe *merge*, there are three comparisons for each loop iteration – one element comparison and two termination tests. Develop a variant using sentinels that needs only one termination test. Can you do this task without appending dummy elements to the sequences?

Exercise 5.20. Exercise 3.31 introduced a list-of-blocks representation for sequences. Implement merging and mergesort for this data structure. During merging, reuse emptied input blocks for the output sequence. Compare the space and time efficiency of mergesort for this data structure, for plain linked lists, and for arrays. Pay attention to constant factors.

5.4 Parallel Mergesort

The recursive mergesort from Fig. 5.4 contains obvious task-based parallelism – one simply performs the recursive calls in parallel. However, this algorithm needs time $\Omega(n)$ regardless of the number of processors available, since the final, sequential merge takes that time. In other words, the maximum obtainable speedup is $O(\log n)$ and the corresponding isoefficiency function is exponential in p. This is about as far away from a scalable parallel algorithm as it gets.

In order to obtain a scalable parallel mergesort, we need to parallelize merging. Our approach to merging two sorted sequences a and b in parallel is to split both sequences into p pieces $a_1, \ldots, a_p$ and $b_1, \ldots, b_p$ such that $merge(a,b)$ is the

concatenation of $merge(a_1, b_1)$, ..., $merge(a_p, b_p)$. The p merges are performed in parallel by assigning one PE each. For this to be correct, the elements in a_i and b_i must be no larger than the elements in a_{i+1} and b_{i+1}. Additionally, to achieve good load balance, we want to ensure that $|a_i| + |b_i| \approx (|a| + |b|)/p$ for $i \in 1..p$. All these properties can be achieved by defining the elements in a_i and b_i to be the elements with positions in $(i-1)\lceil(|a|+|b|)/p\rceil + 1..i\lceil(|a|+|b|)/p\rceil$ in the merged sequence. The strategy is now clear. PE i first determines where a_i ends in a and b_i ends in b. It then merges a_i and b_i.

Let $k = i\lceil(|a|+|b|)/p\rceil$. In order to find where a_i and b_i end in a and b, we need to find the smallest k elements in the two sorted arrays. This is a special case of the selection problem discussed in Sect. 5.8, where we can exploit the sortedness of the arrays a and b to accelerate the computation. We now develop a sequential deterministic algorithm *twoSequenceSelect*(a, b, k) that locates the k smallest elements in two sorted arrays a and b in time $O(\log|a| + \log|b|)$. The idea is to maintain subranges $a[\ell_a..r_a]$ and $b[\ell_b..r_b]$ with the following properties:

(a) The elements $a[1..\ell_a - 1]$ and $b[1..\ell_b - 1]$ belong to the k smallest elements.
(b) The k smallest elements are contained in $a[1..r_b]$ and $b[1..r_b]$.

We shall next describe a strategy which allows us to halve one of the ranges $[\ell_a..r_a]$ or $[\ell_b..r_b]$. For simplicity, we assume that the elements are pairwise distinct. Let $m_a = \lfloor(\ell_a + r_a)/2\rfloor$, $\bar{a} = a[m_a]$, $m_b = \lfloor(\ell_b + r_b)/2\rfloor$, and $\bar{b} = b[m_b]$. Assume that $\bar{a} < \bar{b}$, the other case being symmetric. If $k < m_a + m_b$, then the elements in $b[m_b..r_b]$ cannot belong to the k smallest elements and we may set r_b to $m_b - 1$. If $k \geq m_a + m_b$, then all elements in $a[\ell_a..m_a]$ belong to the k smallest elements and we may set ℓ_a to $m_a + 1$. In either case, we have reduced one of the ranges to half its size. This is akin to binary search. We continue until one of the ranges becomes empty, i.e., $r_a = \ell_a - 1$ or $r_b = \ell_b - 1$. We complete the search by setting $r_b = k - r_a$ in the former case and $r_a = k - r_b$ in the latter case.

Since one of the ranges is halved in each iteration, the number of iterations is bounded by $\log|a| + \log|b|$. Table 5.1 gives an example.

Table 5.1. Example calculation for selecting the $k = 4$ smallest elements from the sequences $a = \langle 4, 5, 6, 8\rangle$ and $b = \langle 1, 2, 3, 7\rangle$. In the first line, we have $\bar{a} > \bar{b}$ and $k \geq m_a + m_b$. Therefore, the first two elements of b belong to the k smallest elements and we may increase ℓ_b to 3. Similarly, in the second line, we have $m_a = 2$ and $m_b = 3$, $\bar{a} > \bar{b}$, and $k < m_a + m_b$. Therefore all elements of a except maybe the first do not belong to the k smallest. We may therefore set r_a to 1.

a	ℓ_a	m_a	r_a	$\bar{a}$	b	ℓ_b	m_b	r_b	$\bar{b}$	$k < m_a + m_b$?	$\bar{a} < \bar{b}$?	action
$[4\bar{5}\vert 68]$	1	2	4	5	$[1\bar{2}\vert 37]$	1	2	4	2	no	no	$l_b := 3$
$[4\bar{5}\vert 68]$	1	2	4	5	$12[\bar{3}\vert 7]$	3	3	4	3	yes	no	$r_a := 1$
$[\bar{4}\vert]568$	1	1	1	4	$12[\bar{3}\vert 7]$	3	3	4	3	no	no	$l_b := 4$
$[\bar{4}\vert]568$	1	1	1	4	$123[\bar{7}\vert]$	4	4	4	7	yes	yes	$r_b := 3$
$[\bar{4}\vert]568$	1	1	1	4	$123[\vert]7$	4	3	3		finish		$r_a := 1$
$4\vert 568$			1		$123\vert 7$			3		done		

***Exercise 5.21.** Assume initially that all elements have different keys.

(a) Implement the algorithm *twoSequenceSelect* outlined above. Test it carefully and make sure that you avoid off-by-one errors.
(b) Prove that your algorithm terminates by giving a *loop variant*. Show that at least one range shrinks in every iteration of the loop. Argue as in the analysis of binary search.
(c) Now drop the assumption that all keys are different. Modify your function so that it outputs splitting positions m_a and m_b in a and b such that $m_a + m_b = k$, $a[m_a] \leq b[m_b + 1]$, and $b[m_b] \leq a[m_a + 1]$. Hint: Stop narrowing a range once all its elements are equal. At the end choose the splitters within the ranges such that the above conditions are met.

We can now define a shared-memory parallel binary mergesort algorithm. To keep things simple, we assume that n and p are powers of two. First we build p runs by letting each PE sort a subset of n/p elements. Then we enter the merge loop. In iteration i of the main loop ($i \in 0..\log p - 1$), we merge pairs of sorted sequences of size $2^i \cdot n/p$ using 2^i PEs. The merging proceeds as described above, i.e., both input sequences are split into 2^i parts each and then each processor merges corresponding pieces.

Let us turn to the analysis. Run formation uses a sequential sorting algorithm and takes time $\mathrm{O}((n/p)\log(n/p))$. Each iteration takes time $\mathrm{O}\big(\log(2^i \cdot (n/p))\big) = \mathrm{O}(\log n)$ for splitting (each PE in parallel finds one splitter) and time $\mathrm{O}(n/p)$ for merging pieces of size n/p. Overall, we get a parallel execution time

$$T_{\text{par}} = \mathrm{O}\left(\frac{n}{p}\log\frac{n}{p} + \log p\left(\log n + \frac{n}{p}\right)\right)$$
$$= \mathrm{O}\left(\log^2 n + \frac{n\log n}{p}\right).$$

This algorithm is efficient[4] for $n = \Omega(p\log p)$. The algorithm is a good candidate for an implementation on real-world shared-memory machines since it does sequential merging and sorting in its inner loops and since it can effectively adapt to the memory hierarchy. However, its drawback is that it moves the data logarithmically often. In Sects. 5.13 and 5.14, we shall see algorithms that move the data less frequently.

On the theoretical side, it is worth noting that there is an ingenious but complicated variant of parallel mergesort by Cole [75] which works in time $\mathrm{O}(\log p + (n\log n)/p)$, i.e., it is even more scalable. We shall present a randomized algorithm in Sect. 5.15 that is simpler and also allows logarithmic time.

***Exercise 5.22.** Design a task-based parallel mergesort with work $\mathrm{O}(n\log n)$ and span $\mathrm{O}\big(\log^3 n\big)$. Hint: You may want to use parallel recursion both for independent subproblems and for merging. For the latter, you may want to use the function *twoSequenceSelect* from Exercise 5.20. Be careful with the size of base case inputs. Compare the scalability of this recursive algorithm with the bottom-up parallel mergesort described above.

[4] Note that $\log^2 n \leq (n\log n)/p$ if and only if $p \leq n/\log n$ if $n = \Omega(p\log p)$.

****Exercise 5.23.** Develop a practical distributed-memory parallel mergesort. Can you achieve running time $O(\frac{n}{p}\log n + \log^2 p)$? A major obstacle may be that our shared-memory algorithm assumes that concurrent reading is fast. In particular, naive access to the midpoints of the current search ranges may result in considerable contention.

Exercise 5.24 (parallel sort join). As in Exercises 4.5, 4.23, and 5.5, consider two relations $R \subseteq A \times B$ and $Q \subseteq B \times C$ with $A \neq C$ and design an algorithm for computing the natural join of R and Q

$$R \bowtie Q := \{(a,b,c) \subseteq A \times B \times C : (a,b) \in R \wedge (b,c) \in Q\}.$$

Give a parallel algorithm with run time $O(((|R|+|Q|)\log(|R|+|Q|)+|R \bowtie Q|)/p)$ for sufficiently large inputs. How large must the input be? How can the limitation in Exercise 4.23 be lifted? Hint: You have to ensure that the work of outputting the result is well balanced over the PEs.

5.5 A Lower Bound

Algorithms give upper bounds on the complexity of a problem. By the preceding discussion, we know that we can sort n items in time $O(n\log n)$. Can we do better, and maybe even achieve linear time? A "yes" answer requires a better algorithm and its analysis. How could we potentially argue a "no" answer? We would have to argue that no algorithm, however ingenious, can run in time $o(n\log n)$. Such an argument is called a *lower bound*. So what is the answer? The answer is both "no" and "yes". The answer is "no" if we restrict ourselves to comparison-based algorithms, and the answer is "yes" if we go beyond comparison-based algorithms. We shall discuss noncomparison-based sorting in Sect. 5.10.

What is a comparison-based sorting algorithm? The input is a set $\{e_1,\ldots,e_n\}$ of n elements, and the only way the algorithm can learn about its input is by comparing elements. In particular, it is not allowed to exploit the representation of keys, for example as bit strings. When the algorithm stops, it must return a sorted permutation of the input, i.e., a permutation $\langle e'_1,\ldots,e'_n\rangle$ of the input such that $e'_1 \leq e'_2 \leq \ldots \leq e'_n$. Deterministic comparison-based algorithms can be viewed as trees. They make an initial comparison; for instance, the algorithm asks "$e_i \leq e_j$?", with outcomes yes and no. Since the algorithm cannot learn anything about the input except through comparisons, this first comparison must be the same for all inputs. On the basis of the outcome, the algorithm proceeds to the next comparison. There are only two choices for the second comparison: one is chosen if $e_i \leq e_j$, and the other is chosen if $e_i > e_j$. Proceeding in this way, the possible executions of the sorting algorithm define a tree. The key point is that the comparison made next depends only on the outcome of all preceding comparisons and nothing else. Figure 5.6 shows a sorting tree for three elements.

Formally, a comparison tree for inputs e_1 to e_n is a binary tree whose nodes have labels of the form "$e_i \leq e_j$?". The two outgoing edges correspond to the outcomes $\leq$

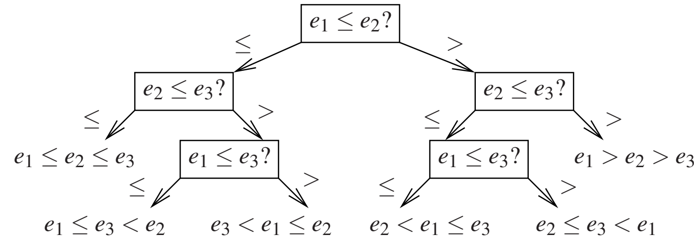


Fig. 5.6. A tree that sorts three elements. We first compare e_1 and e_2. If $e_1 \leq e_2$, we compare e_2 with e_3. If $e_2 \leq e_3$, we have $e_1 \leq e_2 \leq e_3$ and are finished. Otherwise, we compare e_1 with e_3. For either outcome, we are finished. If $e_1 > e_2$, we compare e_2 with e_3. If $e_2 > e_3$, we have $e_1 > e_2 > e_3$ and are finished. Otherwise, we compare e_1 with e_3. For either outcome, we are finished. The worst-case number of comparisons is three. The average number is $(2+3+3+2+3+3)/6 = 8/3$.

and $>$. The computation proceeds in the natural way. We start at the root. Suppose the computation has reached a node labeled $e_i : e_j$. If $e_i \leq e_j$, we follow the edge labeled $\leq$, and if $e_i > e_j$, we follow the edge labeled $>$. The leaves of the comparison tree correspond to the different outcomes of the algorithm.

We next formalize what it means that a comparison tree solves the sorting problem of size n. We restrict ourselves to inputs in which all keys are distinct. When the algorithm terminates, it must have collected sufficient information so that it can tell the ordering of the input. For a permutation π of the integers 1 to n, let ℓ_π be the leaf of the comparison tree reached on input sequences $\{e_1, \ldots, e_n\}$ with $e_{\pi(1)} < e_{\pi(2)} < \ldots < e_{\pi(n)}$. Note that this leaf is welldefined since π fixes the outcome of all comparisons. *A comparison tree solves the sorting problem of size n if, for any two distinct permuations π and σ of $\{1, \ldots, n\}$, the leaves ℓ_π and ℓ_σ are distinct.*

Any comparison tree for sorting n elements must have at least $n!$ leaves. Since a tree of depth T has at most 2^T leaves, we must have

$$2^T \geq n! \quad \text{or} \quad T \geq \log n!.$$

Via Stirling's approximation to the factorial (A.10), we obtain

$$T \geq \log n! \geq \log \left(\frac{n}{e}\right)^n = n \log n - n \log e.$$

Theorem 5.3. *Any comparison-based sorting algorithm needs $n \log n - \mathrm{O}(n)$ comparisons in the worst case.*

We state without proof that this bound also applies to randomized sorting algorithms and to the average-case complexity of sorting, i.e., worst-case instances are not much more difficult than random instances.

Theorem 5.4. *Any comparison-based sorting algorithm for n elements needs $n \log n - \mathrm{O}(n)$ comparisons on average, i.e.,*

$$\frac{\sum_\pi d_\pi}{n!} = n\log n - \mathrm{O}(n),$$

where the sum extends over all $n!$ permutations of the set $\{1,\ldots,n\}$ and d_π is the depth of the leaf ℓ_π.

The *element uniqueness problem* is the task of deciding whether, in a set of n elements, all elements are pairwise distinct.

Theorem 5.5. *Any comparison-based algorithm for the element uniqueness problem of size n requires $\Omega(n\log n)$ comparisons.*

Proof. The algorithm has two outcomes "all elements are distinct" and "there are equal elements" and hence, at first sight, we know only that the corresponding comparison tree has at least two leaves. We shall argue that there are $n!$ leaves for the outcome "all elements are distinct". For a permutation π of $\{1,\ldots,n\}$, let ℓ_π be the leaf reached on input sequences $\langle e_1,\ldots,e_n\rangle$ with $e_{\pi(1)} < e_{\pi(2)} < \ldots < e_{\pi(n)}$. This is one of the leaves for the outcome "all elements are distinct".

Let $i \in 1..n-1$ be arbitrary and consider the computation on an input with $e_{\pi(1)} < e_{\pi(2)} < \ldots < e_{\pi(i)} = e_{\pi(i+1)} < \ldots < e_{\pi(n)}$. This computation has outcome "equal elements" and hence cannot end in the leaf ℓ_π. Since only the outcome of the comparison $e_{\pi(i+1)} : e_{\pi(i)}$ differs for the two inputs (it is $>$ if the elements are distinct and $\leq$ if they are the same), this comparison must have been made on the path from the root to the leaf ℓ_π, and the comparison has established that $e_{\pi(i+1)}$ is larger than $e_{\pi(i)}$. Thus the path to ℓ_π establishes that $e_{\pi(1)} < e_{\pi(2)}$, $e_{\pi(2)} < e_{\pi(3)}$, $\ldots$, $e_{\pi(n-1)} < e_{\pi(n)}$, and hence $\ell_\pi \neq \ell_\sigma$ whenever π and σ are distinct permutations of $\{1,\ldots,n\}$. □

Exercise 5.25. Why does the lower bound for the element uniqueness problem not contradict the fact that we can solve the problem in linear expected time using hashing?

Exercise 5.26. Show that any comparison-based algorithm for determining the smallest of n elements requires $n-1$ comparisons. Show also that any comparison-based algorithm for determining the smallest and second smallest elements of n elements requires at least $n-1+\log n$ comparisons. Give an algorithm with this performance.

Exercise 5.27 (lower bound for average case). With the notation above, let d_π be the depth of the leaf ℓ_π. Argue that $A = (1/n!)\sum_\pi d_\pi$ is the average-case complexity of a comparison-based sorting algorithm. Try to show that $A \geq \log n!$. Hint: Prove first that $\sum_\pi 2^{-d_\pi} \leq 1$. Then consider the minimization problem "minimize $\sum_\pi d_\pi$ subject to $\sum_\pi 2^{-d_\pi} \leq 1$". Argue that the minimum is attained when all d_i's are equal.

Exercise 5.28 (sorting small inputs optimally). Give an algorithm for sorting k elements using at most $\lceil \log k! \rceil$ element comparisons. (a) For $k \in \{2,3,4\}$, use mergesort. (b) For $k = 5$, you are allowed to use seven comparisons. This is difficult. Mergesort does not do the job, as it uses up to eight comparisons. (c) For $k \in \{6,7,8\}$, use the case $k = 5$ as a subroutine.

5.6 Quicksort

Quicksort is a divide-and-conquer algorithm that is, in a certain sense, complementary to the mergesort algorithm of Sect. 5.3. Quicksort does all the difficult work *before* the recursive calls. The idea is to distribute the input elements into two or more sequences so that the corresponding key ranges do not overlap. Then, it suffices to sort the shorter sequences recursively and concatenate the results. To make the duality to mergesort complete, we would like to split the input into two sequences of equal size. Unfortunately, this is a nontrivial task. However, we can come close by picking a random splitter element. The splitter element is usually called the *pivot*. Let p denote the pivot element chosen. Elements are classified into three sequences of elements that are smaller than, equal to, and larger than the pivot. Figure 5.7 gives a high-level realization of this idea, and Fig. 5.8 depicts a sample execution. Quicksort has an expected execution time of $O(n \log n)$, as we shall show in Sect. 5.6.1. In Sect. 5.6.2, we discuss refinements that have made quicksort the most widely used sorting algorithm in practice.

```
Function quickSort(s : Sequence of Element) : Sequence of Element
    if |s| ≤ 1 then return s                                  // base case
    pick p ∈ s uniformly at random                            // pivot key
    a := ⟨e ∈ s : e < p⟩
    b := ⟨e ∈ s : e = p⟩
    c := ⟨e ∈ s : e > p⟩
    return concatenation of quickSort(a), b, and quickSort(c)
```

Fig. 5.7. High-level formulation of quicksort for lists

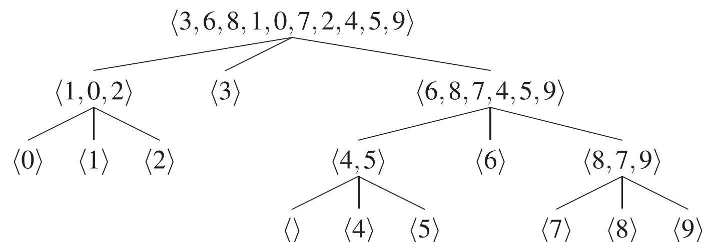


Fig. 5.8. Execution of *quickSort* (Fig. 5.7) on $\langle 3,6,8,1,0,7,2,4,5,9 \rangle$ using the first element of a subsequence as the pivot. The first call of quicksort uses 3 as the pivot and generates the subproblems $\langle 1,0,2 \rangle$, $\langle 3 \rangle$, and $\langle 6,8,7,4,5,9 \rangle$. The recursive call for the third subproblem uses 6 as a pivot and generates the subproblems $\langle 4,5 \rangle$, $\langle 6 \rangle$, and $\langle 8,7,9 \rangle$.

5.6.1 Analysis

To analyze the running time of quicksort for an input sequence $s = \langle e_1, \ldots, e_n \rangle$, we focus on the number of element comparisons performed. We allow *three-way* comparisons here, with possible outcomes "smaller", "equal", and "larger". Other operations contribute only constant factors and small additive terms to the execution time.

Let $C(n)$ denote the worst-case number of comparisons needed for any input sequence of size n and any choice of pivots. The worst-case performance is easily determined. The subsequences a, b, and c in Fig. 5.7 are formed by comparing the pivot with all other elements. This requires $n-1$ comparisons. Let k denote the number of elements smaller than the pivot and let k' denote the number of elements larger than the pivot. We obtain the following recurrence relation: $C(0) = C(1) = 0$ and

$$C(n) \le n - 1 + \max\left\{C(k) + C(k') : 0 \le k \le n-1, 0 \le k' < n-k\right\}.$$

It is easy to verify by induction that

$$C(n) \le \frac{n(n-1)}{2} = \Theta\left(n^2\right).$$

This worst case occurs if all elements are different and we always pick the largest or smallest element as the pivot.

The expected performance is much better. We first give a plausibility argument for an $O(n \log n)$ bound and then show a bound of $2n \ln n$. We concentrate on the case where all elements are different. Other cases are easier because a pivot that occurs several times results in a larger middle sequence b that need not be processed any further. Consider a fixed element e_i, and let X_i denote the total number of times e_i is compared with a pivot element. Then $\sum_i X_i$ is the total number of comparisons. Whenever e_i is compared with a pivot element, it ends up in a smaller subproblem. Therefore, $X_i \le n-1$, and we have another proof of the quadratic upper bound. Let us call a comparison "good" for e_i if e_i moves to a subproblem of at most three-quarters the size. Any e_i can be involved in at most $\log_{4/3} n$ good comparisons. Also, the probability that a pivot which is good for e_i is chosen is at least $1/2$; this holds because a bad pivot must belong to either the smallest or the largest quarter of the elements. So $E[X_i] \le 2 \log_{4/3} n$, and hence $E[\sum_i X_i] = O(n \log n)$. We shall next prove a better bound by a completely different argument.

Theorem 5.6. *The expected number of comparisons performed by quicksort is*

$$\bar{C}(n) \le 2n \ln n \le 1.39 n \log n.$$

Proof. Let $s' = \langle e'_1, \ldots, e'_n \rangle$ denote the elements of the input sequence in sorted order. Every comparison involves a pivot element. If an element is compared with a pivot, the pivot and the element end up in different subsequences. Hence any pair of elements is compared at most once, and we can therefore count comparisons by

looking at the indicator random variables X_{ij}, $i<j$, where $X_{ij}=1$ if e'_i and e'_j are compared and $X_{ij}=0$ otherwise. We obtain

$$\bar{C}(n)=\mathrm{E}\left[\sum_{i=1}^{n}\sum_{j=i+1}^{n}X_{ij}\right]=\sum_{i=1}^{n}\sum_{j=i+1}^{n}\mathrm{E}[X_{ij}]=\sum_{i=1}^{n}\sum_{j=i+1}^{n}\mathrm{prob}(X_{ij}=1).$$

The middle transformation follows from the linearity of expectations (A.3). The last equation uses the definition of the expectation of an indicator random variable $\mathrm{E}[X_{ij}]=\mathrm{prob}(X_{ij}=1)$. Before we can simplify further the expression for $\bar{C}(n)$, we need to determine the probability of X_{ij} being 1.

Lemma 5.7. *For any $i<j$,* $\mathrm{prob}(X_{ij}=1)=\dfrac{2}{j-i+1}$.

Proof. Consider the $j-i+1$-element set $M=\{e'_i,\ldots,e'_j\}$. As long as no pivot from M is selected, e'_i and e'_j are not compared, but all elements from M are passed to the same recursive calls. Eventually, a pivot p from M is selected. Each element in M has the same chance $1/|M|$ of being selected. If $p=e'_i$ or $p=e'_j$, we have $X_{ij}=1$. Otherwise, e'_i and e'_j are passed to different recursive calls, so that they will never be compared. Thus $\mathrm{prob}(X_{ij}=1)=2/|M|=2/(j-i+1)$. □

We can now complete the proof of Theorem 5.6 by a relatively simple calculation:

$$\begin{aligned}\bar{C}(n)&=\sum_{i=1}^{n}\sum_{j=i+1}^{n}\mathrm{prob}(X_{ij}=1)=\sum_{i=1}^{n}\sum_{j=i+1}^{n}\frac{2}{j-i+1}=\sum_{i=1}^{n}\sum_{k=2}^{n-i+1}\frac{2}{k}\\&\le\sum_{i=1}^{n}\sum_{k=2}^{n}\frac{2}{k}=2n\sum_{k=2}^{n}\frac{1}{k}=2n(H_n-1)\le 2n(1+\ln n-1)=2n\ln n.\end{aligned}$$

For the last three steps, recall the properties of the nth harmonic number $H_n:=\sum_{k=1}^{n}1/k\le 1+\ln n$ (A.13). □

Note that the calculations in Sect. 2.11 for left-to-right maxima were very similar, although we had quite a different problem at hand.

5.6.2 *Refinements

We shall now discuss refinements of the basic quicksort algorithm. The resulting algorithm, called *qSort*, works in-place, and is fast and space-efficient. Figure 5.9 shows the pseudocode, and Fig. 5.10 shows a sample execution. The refinements are nontrivial and we need to discuss them carefully.

The function *qSort* operates on an array a. The arguments ℓ and r specify the subarray to be sorted. The outermost call is $qSort(a,1,n)$. If the size of the subproblem is smaller than some constant n_0, we resort to a simple algorithm[5] such as the insertion

[5] Some authors propose leaving small pieces unsorted and cleaning up at the end using a single insertion sort that will be fast, according to Exercise 5.9. Although this nice trick reduces the number of instructions executed, the solution shown is faster on modern machines because the subarray to be sorted will already be in cache.

```
Procedure qSort(a : Array of Element; ℓ, r : ℕ)          // Sort the subarray a[ℓ..r]
  while r − ℓ + 1 > n0 do                                  // Use divide-and-conquer.
    j := pickPivotPos(a, ℓ, r)                             // Pick a pivot element and
    swap(a[ℓ], a[j])                                       // bring it to the first position.
    p := a[ℓ]                                              // p is the pivot now.
    i := ℓ; j := r
    repeat                                                 // a: | ℓ   i→  ←j   r |
      while a[i] < p do i++                                // Skip over elements
      while a[j] > p do j−−                                // already in the correct subarray.
      if i ≤ j then                                        // If partitioning is not yet complete,
        swap(a[i], a[j]); i++; j−−                         // (*) swap misplaced elements and go on.
    until i > j                                            // Partitioning is complete.
    if i < (ℓ + r)/2 then   qSort(a, ℓ, j); ℓ := i         // Recurse on
    else                    qSort(a, i, r); r := j         // smaller subproblem.
  endwhile
  insertionSort(a[ℓ..r])                                   // Faster for small r − ℓ
```

Fig. 5.9. Refined quicksort for arrays

Fig. 5.10. Execution of *qSort* (Fig. 5.9) on $\langle 3,6,8,1,0,7,2,4,5,9 \rangle$ using the first element as the pivot and $n_0 = 1$. The *left-hand side* illustrates the first partitioning step, showing elements in **bold** that have just been swapped. The *right-hand side* shows the result of the recursive partitioning operations.

sort shown in Fig. 5.1. The best choice for n_0 depends on many details of the machine and compiler and needs to be determined experimentally; a value somewhere between 10 and 40 should work fine under a variety of conditions.

The pivot element is chosen by a function *pickPivotPos* that we shall not specify further. The correctness does not depend on the choice of the pivot, but the efficiency does. Possible choices are the first element; a random element; the median ("middle") element of the first, middle, and last elements; and the median of a random sample consisting of k elements, where k is either a small constant, say 3, or a number depending on the problem size, say $\lceil \sqrt{r-\ell+1} \rceil$. The first choice requires the least amount of work, but gives little control over the size of the subproblems; the last choice requires a nontrivial but still sublinear amount of work, but yields balanced subproblems with high probability. After selecting the pivot p, we swap it into the first position of the subarray (= position ℓ of the full array).

The repeat–until loop partitions the subarray into two proper (smaller) subarrays. It maintains two indices i and j. Initially, i is at the left end of the subarray and j is at

the right end; i scans to the right, and j scans to the left. After termination of the loop, we have $i = j+1$ or $i = j+2$, all elements in the subarray $a[\ell..j]$ are no larger than p, all elements in the subarray $a[i..r]$ are no smaller than p, each subarray is a proper subarray, and, if $i = j+2$, $a[j+1]$ is equal to p. So, recursive calls $qSort(a, \ell, j)$ and $qSort(a, i, r)$ will complete the sort. We make these recursive calls in a nonstandard fashion; this is discussed below.

Let us see in more detail how the partitioning loops work. In the first iteration of the repeat loop, i does not advance at all but remains at ℓ, and j moves left to the rightmost element no larger than p. So, j ends at ℓ or at a larger value; generally, the latter is the case. In either case, we have $i \leq j$. We swap $a[i]$ and $a[j]$, increment i, and decrement j. In order to describe the total effect more generally, we distinguish cases.

If p is the unique smallest element of the subarray, j moves all the way to ℓ, the swap has no effect, and $j = \ell - 1$ and $i = \ell + 1$ after the increment and decrement. We have an empty subproblem $a[\ell..\ell-1]$ and a subproblem $a[\ell+1..r]$. Partitioning is complete, and both subproblems are proper subproblems.

If j moves down to $i+1$, we swap, increment i to $\ell+1$, and decrement j to ℓ. Partitioning is complete, and we have the subproblems $a[\ell..\ell]$ and $a[\ell+1..r]$. Both subarrays are proper subarrays.

If j stops at an index larger than $i+1$, we have $\ell < i \leq j < r$ after executing the line marked **(*)** in Fig. 5.9. Also, all elements to the left of i are at most p *(and there is at least one such element)*, and all elements to the right of j are at least p *(and there is at least one such element)*. Since the scan loop for i skips only over elements smaller than p and the scan loop for j skips only over elements larger than p, further iterations of the repeat loop maintain this invariant. Also, all further scan loops are guaranteed to terminate by the italicized claims above and so there is no need for an index-out-of-bounds check in the scan loops. In other words, the scan loops are as concise as possible; they consist of a test and an increment or decrement.

Let us next study how the repeat loop terminates. If we have $i \leq j+2$ after the scan loops, we have $i \leq j$ in the termination test. Hence, we continue the loop. If we have $i = j-1$ after the scan loops, we swap, increment i, and decrement j. So $i = j+1$, and the repeat loop terminates with the proper subproblems $a[\ell..j]$ and $a[i..r]$. The case $i = j$ after the scan loops can occur only if $a[i] = p$. In this case, the swap has no effect. After incrementing i and decrementing j, we have $i = j+2$, resulting in the proper subproblems $a[\ell..j]$ and $a[j+2..r]$, separated by one occurrence of p. Finally, when $i > j$ after the scan loops, then either i goes beyond j in the first scan loop or j goes below i in the second scan loop. By our invariant, i must stop at $j+1$ in the first case, and then j does not move in its scan loop or j must stop at $i-1$ in the second case. In either case, we have $i = j+1$ after the scan loops. The line marked **(*)** is not executed, so we have subproblems $a[\ell..j]$ and $a[i..r]$, and both subproblems are proper.

We have now shown that the partitioning step is correct, terminates, and generates proper subproblems.

Exercise 5.29. Does the algorithm stay correct if the scan loops skip over elements equal to p? Does it stay correct if the algorithm is run only on inputs for which all elements are pairwise distinct?

The refined quicksort handles recursion in a seemingly strange way. Recall that we need to make the recursive calls $qSort(a, \ell, j)$ and $qSort(a, i, r)$. We may make these calls in either order. We exploit this flexibility by making the call for the smaller subproblem first. The call for the larger subproblem would then be the last thing done in *qSort*. This situation is known as *tail recursion* in the programming-language literature. Tail recursion can be eliminated by setting the parameters (ℓ and r) to the right values and jumping to the first line of the procedure. This is precisely what the while-loop does. Why is this manipulation useful? Because it guarantees that the size of the recursion stack stays logarithmically bounded; the precise bound is $\lceil \log(n/n_0) \rceil$. This follows from the fact that in a call for $a[\ell..r]$, we make a single recursive call for a subproblem which has size at most $(r - \ell + 1)/2$.

Exercise 5.30. What is the maximal depth of the recursion stack without the "smaller subproblem first" strategy? Give a worst-case example.

***Exercise 5.31 (sorting strings using multikey quicksort [43]).** Let s be a sequence of n strings. We assume that each string ends in a special character that is different from all "normal" characters. Show that the function $mkqSort(s, 1)$ below sorts a sequence s consisting of *different* strings. What goes wrong if s contains equal strings? Solve this problem. Show that the expected execution time of *mkqSort* is $O(N + n \log n)$ if $N = \sum_{e \in s} |e|$.

Function *mkqSort*(s : *Sequence* **of** *String*, i : $\mathbb{N}$) : *Sequence* **of** *String*
  assert $\forall e, e' \in s : e[1..i-1] = e'[1..i-1]$
  if $|s| \leq 1$ **then return** s // base case
  pick $p \in s$ uniformly at random // pivot character
  return *concatenation of* $mkqSort(\langle e \in s : e[i] < p[i] \rangle, i)$,
    $mkqSort(\langle e \in s : e[i] = p[i] \rangle, i + 1)$, *and*
    $mkqSort(\langle e \in s : e[i] > p[i] \rangle, i)$

Exercise 5.32. Implement several different versions of *qSort* in your favorite programming language. Use and do not use the refinements discussed in this section, and study the effect on running time and space consumption.

***Exercise 5.33 (Strictly inplace quicksort).** Develop a version of quicksort that requires only constant additional memory. Hint: Develop a nonrecursive algorithm where the subproblems are marked by storing their largest element at their first array entry.

5.7 Parallel Quicksort

Analogously to parallel mergesort, there is a trivial parallelization of quicksort that performs only the recursive calls in parallel. We strive for a more scalable solution

that also parallelizes partitioning. In principle, parallel partitioning is also easy: Each PE is assigned an equal share of the array to be partitioned and partitions it. The partitioned pieces have to be reassembled into sequences. Compared with mergesort, parallel partitioning is simpler than parallel merging. However, since the pivots we choose will not split the input perfectly into equal pieces, we face a load-balancing problem: Which processors should work on which recursive subproblem? Overall, we get an interesting kind of parallel algorithm that combines data parallelism with task parallelism. We first explain this in the distributed-memory setting and then outline a shared-memory solution that works almost in-place.

Exercise 5.34. Adapt Algorithm 5.7 to become a task-parallel algorithm with work $O(n \log n)$ and span $O(\log^2 n)$.

5.7.1 Distributed-Memory Quicksort

Figure 5.11 gives high-level pseudocode for distributed-memory parallel quicksort. Figure 5.12 gives an example. In the procedure *parQuickSort*, every PE has a local array s of elements. The PEs cooperate in groups and together sort the union of their arrays. Each group is an interval $i..j$ of PEs. Initially $i = 1$, $j = p$, and each processor has an about equal share of the input, say PEs $1..j$ have $\lceil n/p \rceil$ elements and PEs $j+1..p$ have $\lfloor n/p \rfloor$ elements, where $j = p \cdot (n/p - \lfloor n/p \rfloor)$. The recursion bottoms out when there is a single processor in the group, i.e., $i = j$. The PE completes the sort by calling sequential quicksort for its piece of the input. When further partitioning is needed, the PEs have to agree on a common pivot. The choice of pivot has a significant influence on the load balance and is even more crucial than for sequential quicksort. For now, we shall only explain how to select a random pivot; we shall discuss alternatives at the end of the section. The group $i..j$ of PEs needs to select a random element from the union of their local arrays. This can be implemented

Function *parQuickSort*(s : *Sequence* **of** *Element*, $i, j : \mathbb{N}$) : *Sequence* **of** *Element*
$p' := j - i + 1$ // # of PEs working together
if $i = j$ **then** *quickSort*(s) ; **return** s // sort locally
$v := pickPivot(s, i, j)$
$a := \langle e \in s : e \le v \rangle$; $b := \langle e \in s : e > v \rangle$ // partition
$n_a := \sum_{i \le k \le j} |a|@k$; $n_b := \sum_{i \le k \le j} |b|@k$ // all-reduce in segment $i..j$ *1*
$k' := \frac{n_a}{n_a + n_b} p'$ // fractional number of PEs responsible for a *2*
choose $k \in \{\lfloor k' \rfloor, \lceil k' \rceil\}$ *such that* $\max\left\{\left\lceil \frac{n_a}{k} \right\rceil, \left\lceil \frac{n_b}{p'-k} \right\rceil\right\}$ *is minimized* *3*
send the a's to PEs $i..i+k-1$ *such that no PE receives more than* $\left\lceil \frac{n_a}{k} \right\rceil$ *of them*
send the b's to PEs $i+k..j$ *such that no PE receives more than* $\left\lceil \frac{n_b}{p'-k} \right\rceil$ *of them*
receive data sent to PE i_{proc} *into* s
if $i_{\text{proc}} < i + k$ **then** *parQuickSort*($s, i, i+k-1$) **else** *parQuickSort*($s, i+k, j$)

Fig. 5.11. SPMD pseudocode for parallel quicksort. Each PE has a local array s. The group $i..j$ of PEs work together to sort the union of their local arrays.

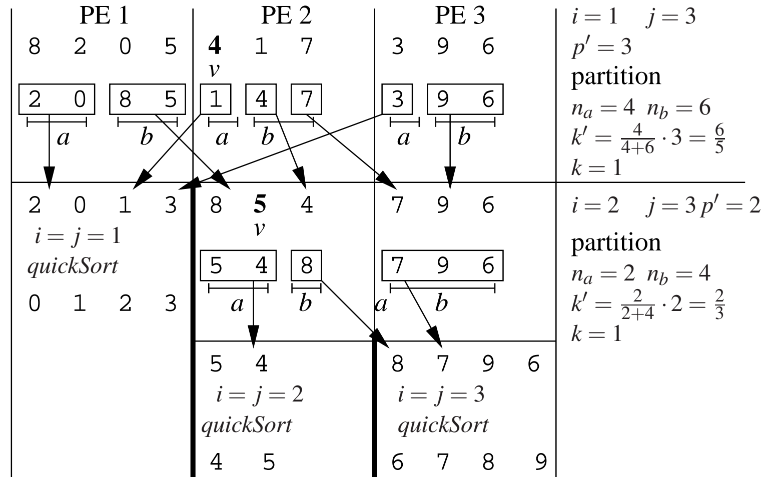


Fig. 5.12. Example of distributed-memory parallel quicksort.

efficiently using prefix sums as follows: We compute the prefix sum over the local values of $|s|$, i.e., PE $\ell \in i..j$ obtains the number $S@\ell := \sum_{i \le k \le \ell} |s|@k$ of elements stored in PEs $i..\ell$. Moreover all PEs in $i..j$ need the total size $S@j$; see Sect. 13.3 for the realization of prefix sums. Now we pick a random number $x \in 1..S@j$. This can be done without communication if we assume that we have a replicated pseudorandom number generator, i.e., a generator that computes the same number on all participating PEs. The PE where $x \in S - |s| + 1..S$ picks $s[x - (S - |s|])$ as the pivot and broadcasts it to all PEs in the group.

In practice, an even simpler algorithm can be used that approximates random sampling if all PEs hold a similar number of elements. We fist pick a random PE index ℓ using a replicated random number generator. Then PE ℓ broadcasts a random element of $s@\ell$. Note that the only nonlocal operation here is a single broadcast; see Sect. 13.1.

Once each PE knows the pivot, local partitioning is easy. Each PE splits its local array into the sequence a of elements no larger than the pivot and the sequence b of elements larger than the pivot. We next need to set up the two subproblems. We split the range of PEs $i..j$ into subranges $i..i+k-1$ and $i+k..j$ such that the left subrange sorts all the a's and the right subrange sorts the b's. A crucial decision is how to choose the number k of PEs dedicated to the a's. We do this so as to minimize load imbalance. The load balance would be perfect if we could split PEs into fractional pieces. This calculation is done in lines 1 and 2. Then line 3 rounds this so that load imbalance is minimized.

Now the data has to be redistributed accordingly. We explain the redistribution for a. For b, this can be done in an analogous fashion. A similar redistribution procedure is explained as a general load-balancing principle in Sect. 14.2. Conceptually, we assign global numbers to the array elements – element $a[x]$ of PE i_{proc} gets a number

$\sum_{\ell<i_{\text{proc}}} |a|@\ell + x$. Note that this can be done using yet another prefix sum calculation. Let $L = \lceil n_a/k \rceil$ denote the maximum number of elements we want to send to one PE. We send the element with global number y to PE $i + \lfloor (y-1)/L \rfloor$. This way, each PE gets at most L elements and the receiving PEs range from $i + \lfloor (1-1)/L \rfloor = i$ to $i + \lfloor (n_a - 1)/L \rfloor \leq i + k - 1$. Since the elements of a have consecutive numbers, they are sent to at most $\lceil |a|/L \rceil + 1$ PEs with consecutive PE numbers. In other words, a is split into up to $\lceil |a|/L \rceil + 1$ pieces of consecutive elements. Each piece can be sent as a single message.

Exercise 5.35. Give detailed pseudocode for a procedure actually doing the message exchange.

We shall not give a detailed analysis of parallel quicksort but restrict ourselves to an outline. The first analysis of the expected performance of sequential quicksort given in Sect. 5.6.1 can be generalized to show that with high probability the depth of the parallel recursion is $\mathrm{O}(\log p)$ – a longer recursion branch would require a sequence of bad pivots that is very unlikely.

Exercise 5.36. Give a formal proof similar to the one of Lemma 5.13.

A single level of recursion takes time

$$\mathrm{O}\left(\max_{k \in 1..p} |s|@k + \log p\right),$$

where the logarithmic term stems from the collective broadcast, reduction, and prefix-sum operations needed to coordinate the PEs. Summing over all levels of recursion, we get a term $\mathrm{O}(\log^2 p)$ for the collective communication operations. If all PEs always had the same number of elements $|s| = \frac{n}{p}$, the remaining work would be $\mathrm{O}(\frac{n}{p} \log p)$ for the recursion and $\mathrm{O}(\frac{n}{p} \log \frac{n}{p})$ for the base-case sequential sorting. Overall, we would get time

$$\mathrm{O}(\frac{n}{p} \log n + \log^2 p).$$

This includes $\mathrm{O}(\log^2 p)$ message startup overhead, $\mathrm{O}(\frac{n}{p} \log p)$ communication volume, and $\mathrm{O}(\frac{n}{p} \log n)$ element comparisons. Hence, we can hope for similar performance as for parallel mergesort if we can bound the load imbalance.

So, let us have a closer look at load balancing. Assuming perfect load balance for the input, the good news is that load imbalance stems only from rounding effects that "should" be small. However, the bad news is that these rounding errors have to be multiplied in each level of recursion. How bad can the rounding errors get? The worst that can happen is that one of the recursive subproblems gets (almost) one PE load's worth of elements more than the other one. Since we always round in an optimal way, we can assume that this additional load is allocated to the *larger* subproblem – of course, it can also happen that the smaller subproblem gets more elements per PE, but only if this results in a smaller imbalance. To make the analysis simple, we shall analyze a modified algorithm that only uses "good" pivots where the smaller

subproblem has size at least $|s|/4$. The worst case is then that we have $k = \log_{4/3} p$ levels of recursion and an imbalance factor bounded by

$$\prod_{i=1}^{k}\left(1+\frac{1}{p\,(3/4)^i}\right) = e^{\sum_{i=1}^{k}\ln\left(1+\frac{1}{p(3/4)^i}\right)} \qquad \text{Estimate A.18}$$

$$\leq e^{\sum_{i=0}^{k}\frac{1}{p(3/4)^i}} = e^{\frac{1}{p}\sum_{i=0}^{k}(4/3)^i} \qquad \text{Equation A.14}$$

$$= e^{\frac{1}{p}\frac{(4/3)^{k+1}-1}{4/3-1}} \leq e^{\frac{1}{p}4\overbrace{(4/3)^k}^{=p}} = e^4 \approx 54.6.$$

The good news is that this is a constant, i.e., our algorithm achieves constant efficiency. The bad news is that e^4 is a rather large constant, and even a more detailed analysis will not get an imbalance factor close to one. However, we can refine the algorithm to get a better load balance. A key observation is that $\prod_{i=1}^{k'}\left(1+1/(p(3/4)^i)\right)$ is close to one if $(4/3)^{k'} = o(p)$. For example, once $j-i \leq \log p$, we could switch to another algorithm with better load balance. For example, we can choose the pivot carefully based on a large sample. Or, we could switch to the sample sort algorithm described in Sect. 5.13. This hybrid algorithm combines the high scalability of pure quicksort with the good load balance of pure sample sort. Another interesting approach is JanusSort [24] that actually splits the PEs fractionally and thus achieves perfect load balance. This is possible by spawning an additional thread on PEs that are fractionally assigned to two subproblems.

5.7.2 *In-Place Shared-Memory Quicksort

A major reason for the popularity of sequential quicksort is its small memory footprint. Besides the space for the input array, it only requires space for the recursion stack. The depth of the recusion stack can be kept logarithmic in the size of the input if the smaller subproblem is always solved first. Is there also a parallel quicksort which is basically in-place? Tsigas and Zhang [316] described such an algorithm whose innermost loop is similar to sequential quicksort. Suppose we want to use p processors to partition an array. We logically split the input array into blocks of size B and keep two global counters ℓ and r, with $\ell \leq r$. The blocks with indices $[\ell+1..r-1]$ are untouched. In the innermost loop, each PE works on two blocks L and R, where the index of L is at most ℓ and the index of R is at least r. As in sequential array-based partitioning (Sect. 5.6.2), the PE scans L from left to right and R from right to left, exchanging small elements of L with large elements of R. When the right end of block L is reached, L is "clean" – all its elements are small. Block L is set aside and the PE chooses the block with index $\ell+1$ as its new block. To this end, the PE increments ℓ atomically and, at the same time, makes sure that $\ell \leq r$. A single CAS instruction suffices provided it can access both counters at the same time.[6] Similarly, a new block from the right is acquired by atomically decrementing

[6] On machines providing only CAS on a single machine word, this can be achieved by making the block size sufficiently large, so that two block counters fit into one machine word.

r. The initial values of ℓ and r are 1 and $\lceil |s|/p \rceil$. Once $\ell = r$, no further blocks remain and the *parallel* partitioning step terminates. It is followed by a cleanup phase. Note that for each PE, there are up to two blocks that are not yet clean. These are cleaned using a sequential algorithm.

It is instructive to analyze the scalability of this partitioning algorithm. First of all, we need $B = \Omega(p)$, since there would otherwise be too much contention for updating the counters ℓ and r. The sequential cleaning step looks at $\Theta(p)$ blocks and hence needs time $\Omega(p^2)$. Apparently, we pay a high price for the in-place property – our noninplace algorithm in Sect. 5.7.1 has a span of only $O(\log^2 p)$.

****Exercise 5.37.** (Research problem) Design a practical in-place parallel sorting algorithm with polylogarithmic span. Hints: One possibility is to improve the Tsigas–Zhang algorithm by using a smaller block size, a relaxed data structure for assigning blocks (see also Sect. 3.7.2), and a parallel cleanup algorithm. Another possibility is to make the algorithm in Sect. 5.7.1 in-place – partition locally and then permute the data such that we obtain a global partition.

5.8 Selection

Selection refers to a class of problems that are easily reduced to sorting but do not require the full power of sorting. Let $s = \langle e_1, \ldots, e_n \rangle$ be a sequence and call its sorted version $s' = \langle e'_1, \ldots, e'_n \rangle$. Selection of the smallest element amounts to determining e'_1, selection of the largest amounts to determining e'_n, and selection of the kth smallest amounts to determining e'_k. Selection of the median[7] refers to determining $e'_{\lceil n/2 \rceil}$. Selection of the median and also of quartiles[8] is a basic problem in statistics. It is easy to determine the smallest element or the smallest and the largest element by a single scan of a sequence in linear time. We now show that the kth smallest element can also be determined in linear time. The simple recursive procedure shown in Fig. 5.13 solves the problem.

This procedure is akin to quicksort and is therefore called *quickselect*. The key insight is that it suffices to follow one of the recursive calls. As before, a pivot is chosen, and the input sequence s is partitioned into subsequences a, b, and c containing the elements smaller than the pivot, equal to the pivot, and larger than the pivot, respectively. If $|a| \geq k$, we recurse on a, and if $k > |a| + |b|$, we recurse on c with a suitably adjusted k. If $|a| < k \leq |a| + |b|$, the task is solved: The pivot has rank k and we return it. Observe that the latter case also covers the situation $|s| = k = 1$, and hence no special base case is needed. Figure 5.14 illustrates the execution of quickselect.

[7] The standard definition of the median of an even number of elements is the average of the two middle elements. Since we do not want to restrict ourselves to the situation where the inputs are numbers, we have chosen a slightly different definition. If the inputs are numbers, the algorithm discussed in this section is easily modified to compute the average of the two middle elements.

[8] The elements with ranks $\lceil \alpha n \rceil$, where $\alpha \in \{1/4, 1/2, 3/4\}$.

As with quicksort, the worst-case execution time of quickselect is quadratic. But the expected execution time is linear and hence a logarithmic factor faster than quicksort.

Theorem 5.8. *Algorithm quickselect runs in expected time* $\mathrm{O}(n)$ *on an input of size n.*

Proof. We give an analysis that is simple and shows a linear expected execution time. It does not give the smallest constant possible. Let $T(n)$ denote the maximum expected execution time of quickselect on any input of size at most n. Then $T(n)$ is a nondecreasing function of n. We call a pivot *good* if neither $|a|$ nor $|c|$ is larger than $2n/3$. Let γ denote the probability that a pivot is good. Then $\gamma \geq 1/3$, since each element in the middle third of the sorted version $s' = \langle e'_1, \ldots, e'_n \rangle$ is good. We now make the conservative assumption that the problem size in the recursive call is reduced only for good pivots and that, even then, it is reduced only by a factor of $2/3$, i.e., reduced to $\lfloor 2n/3 \rfloor$. For bad pivots, the problem size stays at n. Since the work outside the recursive call is linear in n, there is an appropriate constant c such that

$$T(n) \leq cn + \gamma T\left(\left\lfloor \frac{2n}{3} \right\rfloor\right) + (1-\gamma)T(n).$$

Solving for $T(n)$ yields

$$T(n) \leq \frac{cn}{\gamma} + T\left(\left\lfloor \frac{2n}{3} \right\rfloor\right) \leq 3cn + T\left(\left\lfloor \frac{2n}{3} \right\rfloor\right) \leq 3c\left(n + \frac{2n}{3} + \frac{4n}{9} + \ldots\right)$$
$$\leq 3cn \sum_{i\geq 0} \left(\frac{2}{3}\right)^i \leq 3cn \frac{1}{1-2/3} = 9cn.$$

□

```
// Find an element with rank k
Function select(s : Sequence of Element; k : ℕ) : Element
    assert |s| ≥ k
    pick p ∈ s uniformly at random                          // pivot key
    a :=⟨e ∈ s : e < p⟩
    if |a| ≥ k then return select(a, k)                     // [ a ] (k within a)
    b :=⟨e ∈ s : e = p⟩
    if |a| + |b| ≥ k then return p                          // [ a | b = ⟨p,...,p⟩ ] (k within b)
    c :=⟨e ∈ s : e > p⟩
    return select(c, k − |a| − |b|)                         // [ a | b | c ] (k within c)
```

Fig. 5.13. Quickselect

s	k	p	a	b	c
$\langle 3,1,4,5,9,\mathbf{2},6,5,3,5,8\rangle$	6	2	$\langle 1\rangle$	$\langle 2\rangle$	$\langle 3,4,5,9,6,5,3,5,8\rangle$
$\langle 3,4,5,9,\mathbf{6},5,3,5,8\rangle$	4	6	$\langle 3,4,5,5,3,4\rangle$	$\langle 6\rangle$	$\langle 9,8\rangle$
$\langle 3,4,\mathbf{5},5,3,5\rangle$	4	5	$\langle 3,4,3\rangle$	$\langle 5,5,5\rangle$	$\langle\rangle$

Fig. 5.14. The execution of $select(\langle 3,1,4,5,9,2,6,5,3,5,8,6\rangle, 6)$. The middle element (**bold**) of the current sequence s is used as the pivot p.

Exercise 5.38. Modify quickselect so that it returns the k smallest elements.

Exercise 5.39. Give a selection algorithm that permutes an array in such a way that the k smallest elements are in entries $a[1], \ldots, a[k]$. No further ordering is required except that $a[k]$ should have rank k. Adapt the implementation tricks used in the array-based quicksort to obtain a nonrecursive algorithm with fast inner loops.

Exercise 5.40 (streaming selection). A data stream is a sequence of elements presented one by one.

(a) Develop an algorithm that finds the kth smallest element of a sequence that is presented to you one element at a time in an order you cannot control. You have only space $O(k)$ available. This models a situation where voluminous data arrives over a network at a compute node with limited storage capacity.
(b) Refine your algorithm so that it achieves a running time $O(n \log k)$. You may want to read some of Chap. 6 first.
*(c) Refine the algorithm and its analysis further so that your algorithm runs in average-case time $O(n)$ if $k = O(n/\log n)$. Here, "average" means that all orders of the elements in the input sequence are equally likely.

5.9 Parallel Selection

Essentially, our selection algorithm in Fig. 5.13 is already a parallel algorithm. We can perform the partitioning into a, b, and c in parallel using time $O(n/p)$. Determining $|a|$, $|b|$, and $|c|$ can be done using a reduction operation in time $O(\log p)$. Note that all PEs recurse on the same subproblem so that we do not have the load-balancing issues we encountered with parallel quicksort. We get an overall expected parallel execution time of $O(\frac{n}{p} + \log p \log n) = O(\frac{n}{p} + \log^2 p)$. The simplification of the asymptotic complexity can be seen from a simple case distinction. If $n = O\big(p \log^2 p\big)$, then $\log n = O(\log p)$. Otherwise, the term n/p dominates the term $\log n \log p$.

For parallel selection on a distributed-memory machine, an interesting issue is the communication volume involved. One approach is to redistribute the data evenly before a recursive call, using an approach similar to distributed-memory parallel quicksort. We then get an overall communication volume $O(n/p)$ per PE, i.e., essentially all the data is moved.

```
Function parSelect(s : Sequence of Element; k : ℕ) : Element
    v := pickPivot(s)                                          // requires a prefix sum
    a := ⟨e ∈ s : e < v⟩;   b := ⟨e ∈ s : e = v⟩;   c := ⟨e ∈ s : e > v⟩   // partition
    n_a := Σ_i |a|@i;   n_b := Σ_i |b|@i    // reduction
    if n_a ≥ k then return parSelect(a, k)                     // [ a ]
    if n_a + n_b < k then return parSelect(c, k − n_a − n_b)   // [ a | b | c ]
    return v                                                   // [ a | b = ⟨v,...,v⟩ ]
```

Fig. 5.15. SPMD pseudocode for communication-efficient parallel selection

From the point of view of optimizing communication volume, we can do much better by always keeping the data where it is. We get the simple algorithm outlined in Fig. 5.15. However, in the worst case, the elements with ranks near k are all at the same PE. On that PE, the size of s will only start to shrink after $\Omega(\log p)$ levels of recursion. Hence, we get a parallel execution time of $\Omega(\frac{n}{p}\log p + \log p \log n)$, which is not efficient.

5.9.1 *Using Two Pivots

The $\mathrm{O}(\log p \log n)$ term in the running time of the parallel selection algorithm stems from the fact that the recursion depth is $\mathrm{O}(\log n)$ since the expected problem size is reduced by a constant factor in each level of the recursion and that time $\mathrm{O}(\log p)$ time is needed in each level for the reduction operation. We shall now look at an algorithm that manages to shrink the problem size by a factor $f := \Theta(p^{1/3})$ in each level of the recursion and reduces the running time to $\mathrm{O}(n/p + \log p)$. Floyd and Rivest [107] (see also [157, 270]) had the idea of choosing two pivots ℓ and r where, with high probability, ℓ is a slight underestimate of the sought element with rank k and where r is a slight overestimate. Figure 5.16 outlines the algorithm and Fig. 5.17 gives an example.

Function *parSelect2*(s : *Sequence* **of** *Element*; $k : \mathbb{N}$) : *Element*
  if $\sum_i |s@i| < n/p$ **then** // small total remaining input size? (reduction)
    gather all data on a single PE
    solve the problem sequentially there
  else
    $(\ell, r) :=$ *pickPivots*(s) // requires a prefix sum
    $a := \langle e \in s : e < \ell \rangle$; $b := \langle e \in s : \ell \le e \le r \rangle$; $c := \langle e \in s : e > r \rangle$ // partition
    $n_a := \sum_i |a|@i$; $n_b := \sum_i |b|@i$ // reduction
    if $n_a \ge k$ **then return** *parSelect2*(a, k) //
    if $n_a + n_b < k$ **then return** *parSelect2*$(c, k - n_a - n_b)$ //
    return *parSelect2*$(b, k - n_a)$ //

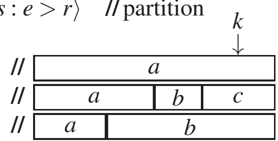


Fig. 5.16. Efficient parallel selection with two splitters

PE 1	PE 2	PE 3	
8 2 0 5	4 1 7 **6**	**3** 9 6	$k = 5$
	r	l	partition
2 0 5 8	1 4 6 7	3 6 9	$n_a = 3 < 5$, $n_b = 5$
a b c	a b c	b c	$n_a + n_b = 9 \ge 5$
5	4 6	3 6	$k = 2$

Fig. 5.17. Selecting the median of $\langle 8,2,0,5,4,1,7,6,3,9,6 \rangle$ using distributed-memory parallel selection with two pivots. The figure shows the first level of recursion using three PEs.

The improved algorithm is similar to the single-pivot algorithm. The crucial difference lies in the selection of the pivots. The idea is to choose a random sample S of the input s and to sort S. Now, $v = S[\lfloor k|S|/|s| \rfloor]$ will be an element with rank close to k. However, we do not know whether v is an underestimate or an overestimate of the element with rank k. We therefore introduce a safety margin Δ and set $\ell = S[\lfloor k|S|/|s| \rfloor - \Delta]$ and $r = S[\lfloor k|S|/|s| \rfloor + \Delta]$. The tricky part is to choose $|S|$ and Δ such that sampling and sorting the sample are fast, and that with high probability $\mathit{rank}(\ell) \leq k \leq \mathit{rank}(r)$ and $\mathit{rank}(r) - \mathit{rank}(\ell)$ is small. With the right choice of the parameters $|S|$ and Δ, the resulting algorithm can be implemented to run in time $O(n/p + \log p)$.

The basic idea is to choose $|S| = \Theta(\sqrt{p})$ so that we can sort the sample in time $O(\log p)$ using the fast, inefficient algorithm in Sect. 5.2. Note that this algorithm assumes that the elements to be sorted are uniformly distributed over the PEs. This may not be true in all levels of the recursion. However, we can achieve this uniform distribution in time $O(n/p + \log p)$ by redistributing the sample.

***Exercise 5.41.** Work out the details of the redistribution algorithm. Can you do it also in time $O(\beta n/p + \alpha \log p)$?

We choose $\Delta = \Theta(p^{1/6})$. Working only with expectations, each sample represents $\Theta(n/\sqrt{p})$ input elements, so that with $\Delta = \Theta(p^{1/6})$, the expected number of elements between ℓ and r is $\Theta(n/\sqrt{p} \cdot p^{1/6}) = \Theta(n/p^{1/3})$.

****Exercise 5.42.** Prove using Chernoff bounds (see Sect. A.3) that for any constant c, with probability at least $1 - p^{-c}$, the following two propositions hold: The number of elements between ℓ and r is $\Theta(n/p^{1/3})$ and the element with rank k is between ℓ and r.

Hence, a constant number of recursion levels suffices to reduce the remaining input size to $O(n/p)$. The remaining small instance can be gathered onto a single PE in time $O(n/p)$ using the algorithm described in Section 13.5. Solving the problem sequentially on this PE then also takes time $O(n/p)$.

The communication volume of the algorithm above can be reduced [157]. Further improvements are possible if the elements on each PE are sorted and when k is not exactly specified. Bulk deletion from parallel priority queues (Sect. 6.4) is a natural generalization of the parallel selection problem.

5.10 Breaking the Lower Bound

The title of this section is, of course, nonsense. A lower bound is an absolute statement. It states that, in a certain model of computation, a certain task cannot be carried out faster than the bound. So a lower bound cannot be broken. But be careful. It cannot be broken within the model of computation used. The lower bound does not exclude the possibility that a faster solution exists in a richer model of computation. In fact, we may even interpret the lower bound as a guideline for getting faster. It tells us that we must enlarge our repertoire of basic operations in order to get faster.

Procedure *KSort*$(s : Sequence$ **of** $Element)$
  $b = \langle\langle\rangle,\ldots,\langle\rangle\rangle : Array\ [0..K-1]$ **of** *Sequence* **of** *Element*
  foreach $e \in s$ **do** $b[key(e)].pushBack(e)$
  $s := concatenation\ of\ b[0],\ldots,b[K-1]$

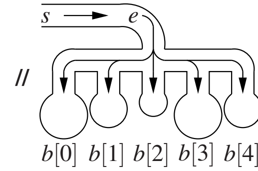


Fig. 5.18. Sorting with keys in the range $0..K-1$

Procedure *LSDRadixSort*$(s : Sequence$ **of** $Element)$
  for $i := 0$ **to** $d-1$ **do**
    redefine $key(x)$ as $(x \operatorname{div} K^i) \bmod K$
    $KSort(s)$
    invariant s is sorted with respect to digits $i..0$

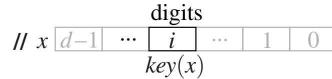


Fig. 5.19. Sorting with keys in $0..K^d-1$ using **l**east **s**ignificant **d**igit (LSD) radix sort

What does this mean in the case of sorting? So far, we have restricted ourselves to comparison-based sorting. The only way to learn about the order of items was by comparing two of them. For structured keys, there are more effective ways to gain information, and this will allow us to break the $\Omega(n\log n)$ lower bound valid for comparison-based sorting. For example, numbers and strings have structure: they are sequences of digits and characters, respectively.

Let us start with a very simple algorithm, *Ksort* (or *bucket sort*), that is fast if the keys are small integers, say in the range $0..K-1$. The algorithm runs in time $O(n+K)$. We use an array $b[0..K-1]$ of *buckets* that are initially empty. We then scan the input and insert an element with key k into bucket $b[k]$. This can be done in constant time per element, for example by using linked lists for the buckets. Finally, we concatenate all the nonempty buckets to obtain a sorted output. Figure 5.18 gives the pseudocode. For example, if the elements are pairs whose first element is a key in the range $0..3$ and

$$s = \langle(3,a),(1,b),(2,c),(3,d),(0,e),(0,f),(3,g),(2,h),(1,i)\rangle,$$

we obtain $b = [\langle(0,e),(0,f)\rangle,\ \langle(1,b),(1,i)\rangle,\ \langle(2,c),(2,h)\rangle,\ \langle(3,a),(3,d),(3,g)\rangle]$ and output $\langle(0,e),(0,f),(1,b),(1,i),(2,c),(2,h),(3,a),(3,d),(3,g)\rangle$. This example illustrates an important property of *Ksort*. It is *stable*, i.e., elements with the same key inherit their relative order from the input sequence. Here, it is crucial that elements are *appended* to their respective bucket.

Comparison-based sorting uses two-way branching. We compare two elements and follow different branches of the program depending on the outcome. In *KSort*, we use K-way branching. We put an element into the bucket selected by its key and hence may proceed in K different ways. The K-way branch is realized by array access and is visualized in Figure 5.18.

KSort can be used as a building block for sorting larger keys. The idea behind *radix sort* is to view integer keys as numbers represented by digits in the range $0..K-1$. Then *KSort* is applied once for each digit. Figure 5.19 gives a radix-sorting

algorithm for keys in the range $0..K^d - 1$ that runs in time $O(d(n+K))$. The elements are first sorted by their least significant digit (*LSD radix sort*), then by the second least significant digit, and so on until the most significant digit is used for sorting. It is not obvious why this works. The correctness rests on the stability of *Ksort*. Since *KSort* is stable, the elements with the same ith digit remain sorted with respect to digits $i-1..0$ during the sorting process with respect to digit i. For example, if $K = 10$, $d = 3$, and

$s = \langle 017, 042, 666, 007, 111, 911, 999\rangle$, we successively obtain
$s = \langle 11\mathbf{1}, 91\mathbf{1}, 04\mathbf{2}, 66\mathbf{6}, 01\mathbf{7}, 00\mathbf{7}, 99\mathbf{9}\rangle$,
$s = \langle 0\mathbf{0}7, 1\mathbf{1}1, 9\mathbf{1}1, 0\mathbf{1}7, 0\mathbf{4}2, 6\mathbf{6}6, 9\mathbf{9}9\rangle$, and
$s = \langle \mathbf{0}07, \mathbf{0}17, \mathbf{0}42, \mathbf{1}11, \mathbf{6}66, \mathbf{9}11, \mathbf{9}99\rangle$.

***Exercise 5.43 (variable length keys).** Assume that input element x is a number with d_x digits.

(a) Extend LSD radix sort to this situation and show how to achieve a running time of $O(d_{\max}(n+K))$, where $d_{\max}$ is the maximum d_x of any input.
(b) Modify the algorithm so that an element x takes part only in the first d_x rounds of radix sort, i.e., only in the rounds corresponding to the last d_x digits. Show that this improves the running time to $O(L + Kd_{\max})$, where $L = \sum_x d_x$ is the total number of digits in the input.
(c) Modify the algorithm further to achieve a running time of $O(L+K)$. Hint: From an input $x = \sum_{0\le\ell<d_x} x_i K^i$ generate the d_x pairs (ℓ, x_ℓ), $0 \le \ell < d_x$, and sort them using radix sort. Use K buckets for the first round of radix sort and $d_{\max}$ buckets for the second round. Observe that the ℓth bucket, $0 \le \ell < d_{\max}$, will contain the multiset of digits that occur as the ℓth least significant digit. Now run LSD radix sort on the original inputs with the following modification: Whenever you concatenate buckets at the end of a call of Ksort, concatenate only the nonempty buckets and do not touch the empty buckets.

***Exercise 5.44 (string sorting).** Modify the algorithm from Exercise 5.42 to sort strings of total length N over an alphabet of size K in time $O(N+K)$.

Radix sort starting with the most significant digit (*MSD radix sort*) is also possible. Here, we apply *KSort* to the most significant digit and then sort each bucket recursively. The only problem is that a bucket may contain much fewer than K elements and then it would be wasteful to sort it further using maybe several rounds of *KSort*. The solution is to switch to another sorting algorithm when the buckets become small. This works particularly well if we can assume that the keys are uniformly distributed. More specifically, let us now assume that the keys are real numbers with $0 \le key(e) < 1$. The algorithm *uniformSort* in Fig. 5.20 scales these keys to integers between 0 and $n-1 = |s|-1$ and groups them into n buckets, where bucket $b[i]$ is responsible for keys in the range $[i/n, (i+1)/n)$. For example, if $s = \langle 0.8, 0.4, 0.7, 0.6, 0.3\rangle$, we obtain five buckets responsible for intervals of size 0.2, and $b = [\langle\rangle, \langle 0.3\rangle, \langle 0.4\rangle, \langle 0.7, 0.6\rangle, \langle 0.8\rangle]$. Only $b[3] = \langle 0.7, 0.6\rangle$ is a nontrivial subproblem. *uniformSort* is very efficient for *random* keys.

Procedure *uniformSort*(s : *Sequence* **of** *Element*)
  $n := |s|$
  $b = \langle\langle\rangle, \ldots, \langle\rangle\rangle$: *Array* $[0..n-1]$ **of** *Sequence* **of** *Element*
  foreach $e \in s$ **do** $b[\lfloor key(e) \cdot n \rfloor].pushBack(e)$
  for $i := 0$ **to** $n-1$ **do** sort $b[i]$ in time $O(|b[i]| \log |b[i]|)$
  $s :=$ *concatenation of* $b[0], \ldots, b[n-1]$

Fig. 5.20. Sorting random keys in the range $[0,1)$

Theorem 5.9. *If the keys are independent uniformly distributed random values in* $[0,1)$*, uniformSort sorts n keys in expected time* $O(n)$ *and worst-case time* $O(n \log n)$*. The linear time bound for the average case holds even if an algorithm with quadratic running time is used for sorting the buckets.*

Proof. We leave the worst-case bound as an exercise and concentrate on the average case. The total execution time T is $O(n)$ for setting up the buckets and concatenating the sorted buckets, plus the time for sorting the buckets. Let T_i denote the time for sorting the ith bucket. We obtain

$$\mathrm{E}[T] = O(n) + \mathrm{E}\left[\sum_{i<n} T_i\right] = O(n) + \sum_{i<n} \mathrm{E}[T_i] = O(n) + n\mathrm{E}[T_0].$$

The second equality follows from the linearity of expectations (A.3), and the third equality uses the fact that all bucket sizes have the same distribution for uniformly distributed inputs. Hence, it remains to show that $\mathrm{E}[T_0] = O(1)$. The analysis is similar to the arguments used to analyze the behavior of hashing in Chap. 4.

Let $B_0 = |b[0]|$. We have $\mathrm{E}[T_0] = O\big(\mathrm{E}[B_0^2]\big)$. The random variable B_0 obeys a binomial distribution (A.8) with n trials and success probability $1/n$, and hence

$$\mathrm{prob}(B_0 = i) = \binom{n}{i} \left(\frac{1}{n}\right)^i \left(1 - \frac{1}{n}\right)^{n-i} \leq \frac{n^i}{i!} \frac{1}{n^i} = \frac{1}{i!} \leq \left(\frac{e}{i}\right)^i,$$

where the last inequality follows from Stirling's approximation to the factorial (A.10). We obtain

$$\begin{aligned}
\mathrm{E}[B_0^2] &= \sum_{i \leq n} i^2 \,\mathrm{prob}(B_0 = i) \leq \sum_{i \leq n} i^2 \left(\frac{e}{i}\right)^i \\
&\leq \sum_{i \leq 5} i^2 \left(\frac{e}{i}\right)^i + e^2 \sum_{i \geq 6} \left(\frac{e}{i}\right)^{i-2} \\
&\leq O(1) + e^2 \sum_{i \geq 6} \left(\frac{1}{2}\right)^{i-2} = O(1),
\end{aligned}$$

and hence $\mathrm{E}[T] = O(n)$ (note that $e/i \leq 1/2$ for $i \geq 6$). □

***Exercise 5.45 (inplace bucket sort).** Develop an sorting algorithm for elements with keys in the range $0..K-1$ that uses the data structure of Exercise 3.31. The space consumption should be $n + O(n/B + KB)$ for n elements, and blocks of size B.

5.11 *Parallel Bucket Sort and Radix Sort

We shall first describe a stable, distributed-memory implementation of bucket sort. Each PE builds a local array of K buckets and distributes its locally present elements to these buckets. Now we need to concatenate local buckets to form global buckets. Note that the bucket sizes can be extremely skewed. For example, 90% of all elements could have key 42. We need to ensure good load balance even for highly skewed inputs – each PE gets at most $L := \lceil n/p \rceil$ elements. We use a similar strategy to that in parallel quicksort and use prefix sums to assign global numbers to all elements. To do this, we compute both the global bucket sizes and a vector-valued prefix sum over the bucket sizes. This can be done in time $O(K + \log p)$; see Sect. 13.3. Let $m_i := \sum_j |b[i]@j|$ denote the global size of bucket i. Then the kth element, $0 \le k < |b[i]@i_{\text{proc}}|$, in bucket i on PE i_{proc} gets a global number $\sum_{j<i} m_j + \sum_{j<i_{\text{proc}}} |b[i]@j| + k$. An element with global number y is assigned to PE $1 + \lfloor (y-1)/L \rfloor$. Assuming that no PE initially has more then L elements, no local bucket will spread over more than two PEs, and hence a PE sends at most $2K$ messages. The average number of received messages is the same. However, in the worst case, there might be a situation where many small buckets are assigned to one PE.

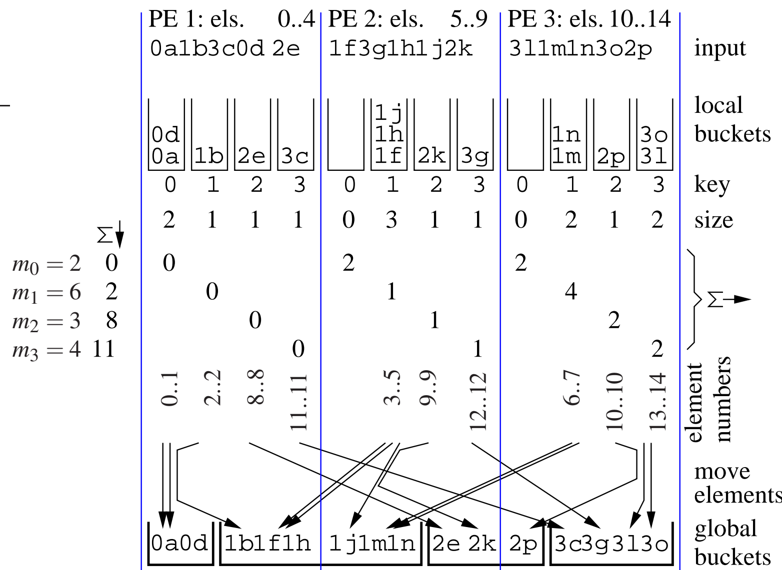


Fig. 5.21. Parallel bucket sort of 15 elements with $K = 4$. The elements have a letter as associated information. For example, $2k$ stands for an element with key 2 and information k. The middle part of the figure shows various sums and prefix sums. m_i is the number of elements in buckets i summed over all processors. The column $\Sigma\downarrow$ shows the prefix sums for the m_i's. The rows $\Sigma\rightarrow$ show the prefix sums of the $|b[i]@j|$. The elements in bucket 1 of processor 2 have global numbers $\sum_{j<1} m_i + \sum_{j<2} |b[1]@j| + k = 2 + 1 + k = 3 + k$, $0 \le k < 3$, i.e., their global numbers span the interval 3..5. PE 2 sends one message to itself; it contains $1j$ and $2k$.

This PE might have to receive $\Theta(pK)$ messages. Therefore, pieces of buckets moved to the same PE should be packed into a single message. This limits the number of sent and received messages to p. Figure 5.21 gives an example. The overall cost of the data exchange is $T_{\text{all}\to\text{all}}(L)$ using an all-to-all communication; see Sect. 13.6.

The above stable distributed bucket sort can be used to implement parallel LSD radix sort. A disadvantage of LSD radix sort is that all elements are moved d times in d-digit radix sort. An alternative is to start radix sort with the *M*ost *S*ignificant *D*igit (*MSD radix sort*). Buckets of size $m_i < L$ can then be assigned to a single PE where they can be sorted locally looking at the remaining digits. Larger buckets still need to be split between several PEs but at least the number of PEs involved decreases. Also, if the PEs are numbered so as to respect locality of communication, then communication between PEs assigned to the same bucket may be faster than global communication.

5.12 *External Sorting

Sometimes the input is so large that it does not fit into internal memory. In this section, we learn how to sort such data sets in the external-memory model introduced in Sect. 2.2. This model distinguishes between a fast internal memory of size M and a large external memory. Data is moved in blocks of size B between the two levels of the memory hierarchy. Scanning data is fast in external memory, and mergesort is based on scanning. We therefore take mergesort as the starting point for external-memory sorting.

Assume that the input is given as an array in external memory. We shall describe a nonrecursive implementation of mergesort for the case where the number of elements n is divisible by B. We load subarrays of size M into internal memory, sort them using our favorite algorithm, for example *qSort*, and write the sorted subarrays back to external memory. We refer to the sorted subarrays as *runs*. The *run formation phase* takes n/B block reads and n/B block writes, i.e., a total of $2n/B$ I/Os. We then merge pairs of runs into larger runs in $\lceil\log(n/M)\rceil$ *merge phases*, ending up with a single sorted run. Figure 5.22 gives an example for $n = 48$ and runs of length 12.

How do we merge two runs? We keep one block from each of the two input runs and one from the output run in internal memory. We call these blocks *buffers*.

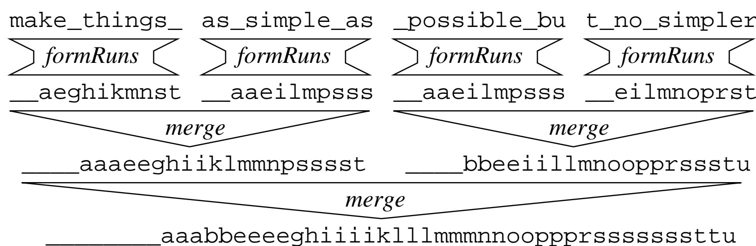


Fig. 5.22. An example of two-way mergesort with initial runs of length 12.

Initially, the input buffers are filled with the first B elements of the input runs, and the output buffer is empty. We compare the leading elements of the input buffers and move the smaller element to the output buffer. If an input buffer becomes empty, we fetch the next block of the corresponding input run; if the output buffer becomes full, we write it to external memory.

Each merge phase reads all current runs and writes new runs of twice the length. Therefore, each phase needs n/B block reads and n/B block writes. Summing over all phases, we obtain $(2n/B)(1 + \lceil \log n/M \rceil)$ I/Os. This technique works provided that $M \geq 3B$.

5.12.1 Multiway Mergesort

In general, internal memory can hold many blocks and not just three. We shall describe how to make full use of the available internal memory during merging. The idea is to merge more than just two runs; this will reduce the number of phases. In *k-way merging*, we merge k sorted sequences into a single output sequence. In each step, we find the input sequence with the smallest first element. This element is removed and appended to the output sequence. External-memory implementation is easy as long as we have enough internal memory for k input buffer blocks, one output buffer block, and a small amount of additional storage.

For each sequence, we need to remember which element we are currently considering. To find the smallest element out of all k sequences, we keep their current elements in a *priority queue*. A priority queue maintains a set of elements supporting the operations of insertion and deletion of the minimum. Chapter 6 explains how priority queues can be implemented so that insertion and deletion take time $O(\log k)$ for k elements. The priority queue tells us, at each step, which sequence contains the smallest element. We delete this element from the priority queue, move it to the output buffer, and insert the next element from the corresponding input buffer into the priority queue. If an input buffer runs dry, we fetch the next block of the corresponding sequence, and if the output buffer becomes full, we write it to the external memory.

How large can we choose k? We need to keep $k+1$ blocks in internal memory and we need a priority queue for k keys. So, we need $(k+1)B + O(k) \leq M$ or $k = O(M/B)$. The number of merging phases is reduced to $\lceil \log_k(n/M) \rceil$, and hence the total number of I/Os becomes

$$2\frac{n}{B}\left(1 + \left\lceil \log_{M/B} \frac{n}{M} \right\rceil\right). \tag{5.2}$$

The difference from binary merging is the much larger base of the logarithm. Interestingly, the above upper bound for the I/O complexity of sorting is also a lower bound [6], i.e., under fairly general assumptions, no external sorting algorithm with fewer I/O operations is possible.

In practice, the number of merge phases will be very small. Observe that a single merge phase suffices as long as $n \leq M^2/B$. We first form M/B runs of length M each and then merge these runs into a single sorted sequence. If internal memory stands

for DRAM and "external memory" stands for hard disks or solid state disks, this bound on n is no real restriction, for all practical system configurations.

Exercise 5.46. Show that a multiway mergesort needs only $O(n \log n)$ element comparisons.

Exercise 5.47 (balanced systems). Study the current market prices of computers, internal memory, and mass storage (currently hard disks and solid state disks). Also, estimate the block size needed to achieve good bandwidth for I/O. Can you find any configuration where multiway mergesort would require more than one merging phase for sorting an input that fills all the disks in the system? If so, what fraction of the cost of that system would you have to spend on additional internal memory to go back to a single merging phase?

5.12.2 Sample Sort

The most popular internal-memory sorting algorithm is not mergesort but quicksort. So it is natural to look for an external-memory sorting algorithm based on quicksort. We shall sketch *sample sort* [110]. It has the same performance guarantees as multiway mergesort (5.2), but only in expectation not in the worst case. On the positive side, sample sort is easier to adapt to parallel disks and parallel processors than merging-based algorithms. Furthermore, similar algorithms can be used for fast external sorting of integer keys along the lines of Sect. 5.10.

Instead of the single pivot element of quicksort, we now use $k-1$ *splitter elements* $s_1, \ldots, s_{k-1}$ to split an input sequence into k output sequences, or *buckets*. Bucket i gets the elements e for which $s_{i-1} \leq e < s_i$. To simplify matters, we define the artificial splitters $s_0 = -\infty$ and $s_k = \infty$ and assume that all elements have different keys. The splitters should be chosen in such a way that the buckets have a size of roughly n/k. The buckets are then sorted recursively. In particular, buckets that fit into the internal memory can subsequently be sorted internally. Note the similarity to the MSD radix sort described in Sect. 5.10.

The main challenge is to find good splitters quickly. Sample sort uses a fast, simple randomized strategy. For some integer a, we randomly choose $(a+1)k-1$ *sample* elements from the input. The sample S is then sorted internally, and we define the splitters as $s_i = S[(a+1)i]$ for $1 \leq i \leq k-1$, i.e., consecutive splitters are separated by a samples, the first splitter is preceded by a samples, and the last splitter is followed by a samples. Taking $a = 0$ results in a small sample set, but the splitting will not be very good. Moving all n elements to the sample will result in perfect splitters, but the sample will be too big. The following analysis shows that setting $a = \Theta(\log k)$ achieves roughly equal bucket sizes at low cost for sampling and sorting the sample.

The most I/O-intensive part of sample sort is the k-way distribution of the input sequence to the buckets. We keep one buffer block for the input sequence and one buffer block for each bucket. These buffers are handled analogously to the buffer blocks in k-way merging. If the splitters are kept in a sorted array, we can find the right bucket for an input element e in time $O(\log k)$ using binary search.

Theorem 5.10. *Sample sort uses*

$$\mathrm{O}\left(\frac{n}{B}\left(1+\left\lceil \log_{M/B}\frac{n}{M}\right\rceil\right)\right)$$

expected I/O steps for sorting n elements. The internal work is $\mathrm{O}(n\log n)$.

We leave a detailed proof to the reader and describe only the key ingredient of the analysis here. We use $k=\Theta(\min(n/M, M/B))$ buckets and a sample of size $\mathrm{O}(k\log k)$. The following lemma shows that with this sample size, it is unlikely that any bucket has a size much larger than the average. We hide the constant factors behind $\mathrm{O}(\cdot)$ notation because our analysis is not very tight in this respect.

Lemma 5.11. *Let* $k\geq 2$ *and* $a+1=12\ln k$. *A sample of size* $(a+1)k-1$ *suffices to ensure with probability at least* $1/2$ *that no bucket receives more than* $4n/k$ *elements.*

Proof. As in our analysis of quicksort (Theorem 5.6), it is useful to study the sorted version $s'=\langle e'_1,\ldots,e'_n\rangle$ of the input. Assume that there is a bucket with at least $4n/k$ elements assigned to it. We estimate the probability of this event.

We split s' into $k/2$ segments of length $2n/k$. The jth segment t_j contains elements $e'_{2jn/k+1}$ to $e'_{2(j+1)n/k}$. If $4n/k$ elements end up in some bucket, there must be some segment t_j such that all its elements end up in the same bucket. This can only happen if fewer than $a+1$ samples are taken from t_j, because otherwise at least one splitter would be chosen from t_j and its elements would not end up in a single bucket. Let us concentrate on a fixed j.

We use a random variable X to denote the number of samples taken from t_j. Recall that we take $(a+1)k-1$ samples. For each sample i, $1\leq i\leq (a+1)k-1$, we define an indicator variable X_i with $X_i=1$ if the ith sample is taken from t_j and $X_i=0$ otherwise. Then $X=\sum_{1\leq i\leq (a+1)k-1}X_i$. Also, the X_i's are independent, and $\mathrm{prob}(X_i=1)=2/k$. Independence allows us to use the Chernoff bound (A.6) to estimate the probability that $X<a+1$. We have

$$\mathrm{E}[X]=((a+1)k-1)\cdot\frac{2}{k}=2(a+1)-\frac{2}{k}\geq\frac{3(a+1)}{2}.$$

Hence $X<a+1$ implies $X<(1-1/3)\mathrm{E}[X]$, and so we can use (A.6) with $\varepsilon=1/3$. Thus

$$\mathrm{prob}(X<a+1)\leq e^{-(1/9)\mathrm{E}[X]/2}\leq e^{-(a+1)/12}=e^{-\ln k}=\frac{1}{k}.$$

The probability that an insufficient number of samples is chosen from a fixed t_j is thus at most $1/k$, and hence the probability that an insufficient number is chosen from some t_j is at most $(k/2)\cdot(1/k)=1/2$. Thus, with probability at least $1/2$, each bucket receives fewer than $4n/k$ elements. □

Exercise 5.48. Work out the details of an external-memory implementation of sample sort. In particular, explain how to implement multiway distribution using $2n/B+k+1$ I/O steps if the internal memory is large enough to store $k+1$ blocks of data and $\mathrm{O}(k)$ additional elements.

Exercise 5.49 (many equal keys). Explain how to generalize multiway distribution so that it still works if some keys occur very often. Hint: There are at least two different solutions. One uses the sample to find out which elements are frequent. Another solution makes all elements unique by interpreting an element e at an input position i as the pair (e,i).

***Exercise 5.50 (more accurate distribution).** A larger sample size improves the quality of the distribution. Prove that a sample of size $\mathrm{O}\big((k/\varepsilon^2)\log(km/\varepsilon)\big)$ guarantees, with probability at least $1-1/m$, that no bucket has more than $(1+\varepsilon)n/k$ elements. Can you get rid of the ε in the logarithmic factor?

5.13 Parallel Sample Sort with Implementations

We learned about sample sort as an external-memory algorithm in Sect. 5.12.2. It is equally useful for parallel sorting [47]. We begin with a high-level description as a distributed-memory algorithm. Then we describe shared memory and MPI implementations in sections 5.13.1 and 5.13.2 respectively.

Figure 5.23 gives high-level pseudocode for distributed-memory sample sort. The number of splitters is now $p-1$: We divide the input into p pieces of about equal size, one for each PE. The procedure *selectSplitters* encapsulates the task of selecting samples in parallel and extracting splitters from them. A pragmatic approximation to a uniform random sample is that each PE chooses a number of samples from its local data proportional to its share in the overall amount of data. A more sophisticated algorithm that actually chooses a uniformly distributed global random sample is described in [273]. This algorithm needs only $\mathrm{O}(\log p)$ communication cost.

Perhaps the simplest way to extract splitters from the sample is to gather (see Sect. 13.5) the local samples at PE 0, sort them there and take the splitters as equidistant elements of the sorted sample as described in Sect. 5.12.2. The splitters are then broadcast to all PEs; see Sect. 13.1. A variant of this idea is to use an all-gather operation and to do the sorting and splitter selection redundantly. This saves us the broadcast but makes other operations slower. If p or n is large, it may be good to use

```
Procedure parSampleSort(s : Sequence of Element)
    b = ⟨⟨⟩,...,⟨⟩⟩ : Array [1..p] of Sequence of Element
    ⟨s_0 = −∞, s_1,...,s_{p−1}, s_p = ∞⟩ := selectSplitters(s)
    foreach e ∈ s do
        determine i such that s_{i−1} ≤ e < s_i
        b[i].pushBack(e)
    send b[i] to PE i for i ∈ 1..p                              // all-to-all
    receive buckets into s
    sort(s)
```

Fig. 5.23. SPMD pseudocode for distributed-memory sample sort

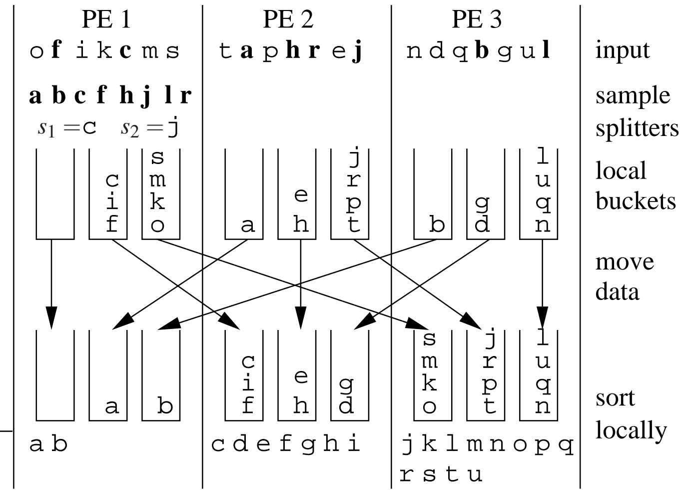


Fig. 5.24. Sample sort of 21 characters on three PEs using oversampling parameter $a = 2$.

a parallel algorithm to sort the sample – perhaps a fast one such as the fast, inefficient algorithm in Sect. 5.2 or the parallel quicksort in Sect. 5.7.1.

In the partitioning phase, we need to locate the right bucket for each element. This can be achieved in time $O(\log p)$ per element by binary search in the sorted array of splitters.[9]

Delivering the buckets to their destination is an all-to-all operation; see Sect. 13.6. The data can be received into the input sequence s, but a size update is needed since the splitters do not perfectly partition the data into equal-sized pieces. Finally, s is sorted locally. Figure 5.24 gives an example.

We now complete the analysis of parallel sample sort.

Theorem 5.12. *Parallel sample sort takes time* $O(\frac{n}{p}\log n + p\log^2 p)$ *assuming sequential sorting of the sample. Using a fast parallel algorithm for sorting the sample, the running time reduces to* $O(\frac{n}{p}\log n + p)$.

Proof. (Outline.) We leave the variant with centralized sorting of the sample as an exercise. By Lemma 5.11, a sample size of $O(p\log p)$ suffices to achieve pieces of size $O(n/p)$. Determining the sample in parallel takes time $O(\log p) = o(p)$ as described above. Sorting the sample using the fast, inefficient algorithm in Sect. 5.2 needs time

$$O\left(\log p + \frac{p\log p}{\sqrt{p}} + \frac{p\log p}{p}\log\frac{p\log p}{\log p}\right) = O(\sqrt{p}\log p) = o(p);$$

see (5.1). Distributing the elements to local buckets takes time $O(\frac{n}{p}\log p)$. Delivering the local buckets to the PEs responsible for them is the time for a nonuniform all-to-

[9] For a more efficient way to find buckets, see [279].

all communication with $h = \mathrm{O}(n/p)$, i.e., $\mathrm{O}(n/p+p)$; see Sect. 13.6.3. Local sorting takes time $\mathrm{O}(\frac{n}{p}\log n)$. Summing all these terms yields the bound. $\square$

We can see that sample sort with parallel sorting of the sample is efficient when $n \gg p^2/\log p$. In order to actually achieve efficiency close to 1, it is important to balance the work between the PEs very well. The following exercise works out how this affects the required sample size.

****Exercise 5.51.** For any constant $\varepsilon > 0$, show that a sample size $\mathrm{O}\big(p\log(n)/\varepsilon^2\big)$ ensures with high probability that no piece is larger than $(1+\varepsilon)n/p$. Hint: Consider a potential piece A that is larger than $(1+\varepsilon)n/p$. Show using Chernoff bounds that it is unlikely that so few samples are taken from A that A is not split into several pieces. Can you show that $\mathrm{O}\big(p\log(p)/\varepsilon^2\big)$ samples are also enough?

5.13.1 Shared-Memory Sample Sort Implementation

We now explain how to implement sample sort on a shared-memory machine using C++11 and the standard library. Compared with the distributed-memory version, matters simplify. In particular, the input and output are just a single global array s. With centralized splitter determination, a single PE can take random samples from s, sort this sample, and copy the splitters to a global splitter array. Hardware caching will make sure that each PE has a local copy of the splitter array in its cache. Instead of invoking an all-to-all collective communication, each PE will copy the local buckets to the appropriate places in s. However, we shall see that a few special measures are needed in order to adapt to NUMA effects; see Sect. 2.4.3.

Listing 5.1 shows our first attempt. Besides the random access iterator[10] s pointing to the beginning of the input array, $n = |s|$, and the number of threads (PEs) p, the (sequential) routine *pSampleSort* is passed a three-dimensional array *buckets*: $\mathit{buckets}[i][j]$ is a vector into which thread i puts all elements from his batch that should go to thread j. Line 6 declares a global *barrier* object, which is later used to synchronize the worker threads. In lines 8–12, the calling thread takes a random sample S using a Mersenne Twister pseudorandom generator of 32-bit numbers with a state size of 19 937 bits and oversampling factor $a = 16\log p$; the declaration of SampleDistribution states that we want to generate integers in $0..n-1$. In lines 13–16, S is then sorted and condensed to contain only the splitters.

The parallel part of the program starts in line 19. Lines 19–45 create p threads running a worker function defined in-place. The worker function is called with the argument i, the second argument in the call $\mathit{thread}(\text{definition of worker function}, i)$, i.e., the local value of *iPE* is i. In the while-loop of the worker function (lines 24–28), thread *iPE* scans its region of the input $s[\mathit{iPE}\cdot n/p..(\mathit{iPE}+1)\cdot n/p-1]$. Each element $e = *\mathit{current}$ is located in the array *splitters* using the function *upper_bound* from the standard library. This function uses binary search to locate the first splitter position

[10] An iterator is an object similar to a pointer that allows a programmer to traverse a data structure.

Listing 5.1. Sample sort n elements in s using p threads

```
template <class Iterator>
void pSampleSort(const Iterator & s, const size_t n, const unsigned p,
    vector<vector<vector<Element> > > & buckets)
{
  mt19937 rndEngine;
  Barrier barrier(p); // for barrier synchronization
  // Choose random samples
  vector<KeyType> S; // random sample of a elements from s
  uniform_int_distribution<size_t> SampleDistribution(0, n−1);
  const int a = (int)(16*log(p)/log(2.0)); // oversampling ratio
  for(size_t i=0; i < (size_t)(a+1)*p − 1; ++i)
    S.push_back((s + SampleDistribution(rndEngine))−>key);
  sort(S.begin(),S.end()); // sort samples sequentially
  for(size_t i=0; i < p−1 ; ++i) // select splitters
    S[i] = S[(a+1)*(i+1)];
  S.resize(p−1);
  vector<size_t> bucketSize(p, 0ULL);
  vector<thread> threads(p);
  for (unsigned i = 0; i < p; ++i) { // go parallel
    threads[i] = thread( [&](const unsigned iPE) // the worker function
    { // distribute elements
      auto current = s + iPE*n/p, end = s + (iPE+1)*n/p;
      auto & myBuckets = buckets[iPE];
      while(current != end) {
        const size_t i = upper_bound(S.begin(),S.end(),
          current−>key) − S.begin(); // binary search
        myBuckets[i].push_back(*current++);
      }
      barrier.wait(iPE, p);
      // now each thread works on bucket "iPE". First compute the total size of bucket iPE:
      size_t myBuckSize = 0;//accumulate into local variable (prevent false sharing)
      for (const auto & b : buckets) myBuckSize += b[iPE].size();
      bucketSize[iPE] = myBuckSize;
      barrier.wait(iPE, p);
      // find the bucket start in s by summing the sizes of the previous buckets (<iPE)
      auto bucketBegin = s;
      for(size_t b = 0; b < iPE ; ++b) bucketBegin += bucketSize[b];
      // copy the bucket 'iPE' from all PEs into s
      auto currOut = bucketBegin;
      for(const auto & b: buckets)
        currOut = copy(b[iPE].cbegin(), b[iPE].cend(), currOut);
      sort(bucketBegin, currOut); // sort the bucket
    } // end of the worker function
    , i);
  }
  for (auto & t : threads) t.join();
}//SPDX−License−Identifier: BSD−3−Clause; Copyright(c) 2018 Intel Corporation
```

larger than e. If no such splitter exists, $splitters.end()$ is returned. Note that the specification of *upper_bound* elegantly avoids the need for explicitly storing a sentinel key ∞. Subtracting $splitters.begin()$ yields the index of the bucket where e should be stored. Each thread uses its own local bucket array $myBuckets = buckets[iPE]$.

Before sorting can continue, a barrier synchronization is necessary (see Sect. 13.4.2), i.e., all worker threads have to wait until all other worker threads have finished the while-loop.

After another barrier synchronization (line 34), each worker thread computes the beginning of the part of the output it is responsible for. This is done by adding the *bucketSize*s of the threads with smaller number (lines 36 and 37). Then, in lines 39–41, the *iPE*-th local bucket from each worker thread is copied to the output array s. Note that bucketBegin is an iterator into the array s and hence all the copying is done into s. Note also that the use of the standard library function *copy* allows the compiler to use highly tuned code here. Finally, in line 42, the *iPE*-th bucket, which now resides in a contiguous piece of the output array s, is sorted by calling the standard library function *sort*.

We measured the speedup of this simple implementation compared with sequential `std::sort` on the machine described in Appendix B. We performed a typical *weak scaling* experiment, i.e., the input size was scaled linearly with the number p of threads used – here, $n = p \cdot 8 \cdot 10^6$. We sort key-value pairs with 8-byte integers each. The lowest line in Fig. 5.25 shows the speedup obtained as a function of the number of threads. The outcome is disappointing. The speedup is never more than seven on a system with 72 physical cores, i.e., it is below 10% of what is suggested by the core count.

Using profiling tools (see Sect. B.8) one can see that for larger thread counts, the CPU cycles are spent mostly in the Linux kernel page fault handler and not on sorting itself. In the Linux kernel used, page fault handling is mostly serial, exposing a serious scalability bottleneck for applications with many page faults. In our implementation, the page faults stem from filling the buckets. The vectors used there are essentially the unbounded arrays in Sect. 3.4. Not only does this imply that data is copied as the arrays grow, but we also suffer from an "improvement" in the operating system that only lazily allocates physical memory in small pages (typically 4 KB) as the allocated virtual memory is actually written to; see `linux.die.net/man/2/set_mempolicy`. This is a clever strategy for sequential execution or a small number of threads but apparently becomes very slow for a large number of threads.

In other words, we face a frequent situation in parallel programming – we have to program around a performance bug of software that we cannot influence. We circumvented the Linux kernel page fault handler by using the Intel `tbb::memory_pool_allocator`, which redirects bucket allocation requests to a pre-assigned internal memory pool. This version performs significantly better; see the line marked "psamplesort-mpool" in Fig. 5.25. However, using all four sockets still does not give any significant speedup compared with two sockets.

An analysis of the profiler and processor performance data (see Sect. B.8) for the improved implementation shows that the number of CPU cycles required per instruction (CPI) significantly increases in the case of threads scattered over all sockets.

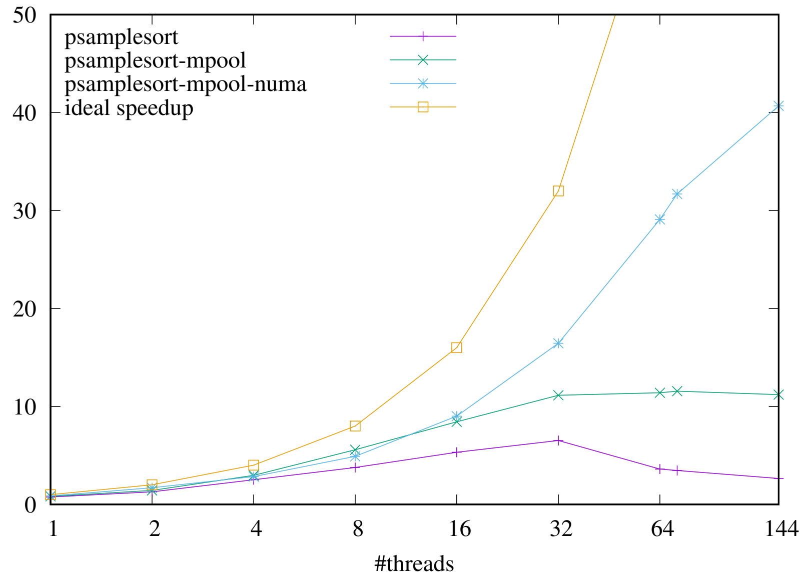


Fig. 5.25. Speedup of different parallel sorting algorithms over `std::sort` sorting $8 \cdot 10^6$ elements per thread.

Increased CPI is particularly prominent for memory access instructions. Since this only happens in the multiple-socket case, NUMA effects seem to be the cause, i.e., memory access latencies go up for accessing memory allocated on a remote socket. This is not surprising, given the first-touch NUMA memory assignment policy in the Linux kernel: If the capacity of the socket local memory permits, an allocation uses the physical memory pages on the socket of the running thread. Using memory from only a single socket can constitute a serious scalability bottleneck, since memory controllers on this socket will be responsible for servicing memory requests of all worker threads in the system. Even if the memory allocation is distributed uniformly over all sockets, the order-of-magnitude higher latency for remote accesses will degrade performance. Most operating systems expose NUMA allocation interfaces, allowing programmers to allocate memory from specific sockets. Unfortunately, these interfaces are not standardized yet and have many differences in semantics (Sect. C.6). An alternative way to control NUMA allocation in Linux is to exploit the first-touch policy. As a first measure, we *pin* worker thread i to physical core i (Sect. C.7); see the function *pinToCore* in the source code. The concrete measures to improve locality are twofold; see function *initialize* in the source code. First, each thread initializes the part of s it later distributes to the buckets. Note that this part is also a good approximation to the part it will finally sort sequentially. Second, each thread reserves a local memory pool for the buckets.

Figure 5.25 shows that the NUMA-optimized implementation exhibits good scaling. As a further experiment (Fig. 5.26), we compare different parallel sorting algorithms using up to 144 threads. An algorithm from the Intel TBB library achieves a speedup that is always below 8 even for large inputs. This algorithm lacks scalability because it uses a very simple parallelization of quicksort that uses sequential partitioning. However, TBB has the advantage to yield some speedup even for small inputs. Likely this is due to the use of efficient light-weight parallel tasks instead of threads. The curve labelled "std parallel mode" refers of an implementation of parallel multiway mergesort, that is available with the parallel version of the STL for `g++` [297]. This algorithm as well as our best sample sort implementation (psamplesort-mpool-numa) achieve speedups around 30 for large inputs. This can be viewed as a success for our sample sort since it is much simpler. The MPI implementation of sample sort discussed in the next section performs even better for large but not too large inputs – achieving speedup of up to 47. We discuss this surprising effect below. Finally, the line labelled ipS^{4}o (inplace super scalar sample sort) refers to a recent inplace variant of sample sort [25]. Somewhat surprisingly, it significantly outperforms all the other algorithms besides saving on memory. With up to 87, the achieved speedup even exceeds the number of cores (72). There are two reasons for this good performance. First, ipS^{4}o performs element comparisons very efficiently and without incurring conditional branch instructions (indeed, ipS^{4}o also outperforms `std::sort` as a sequential algorithm). Second it avoids several sources of overhead involved with noninplace sorting.

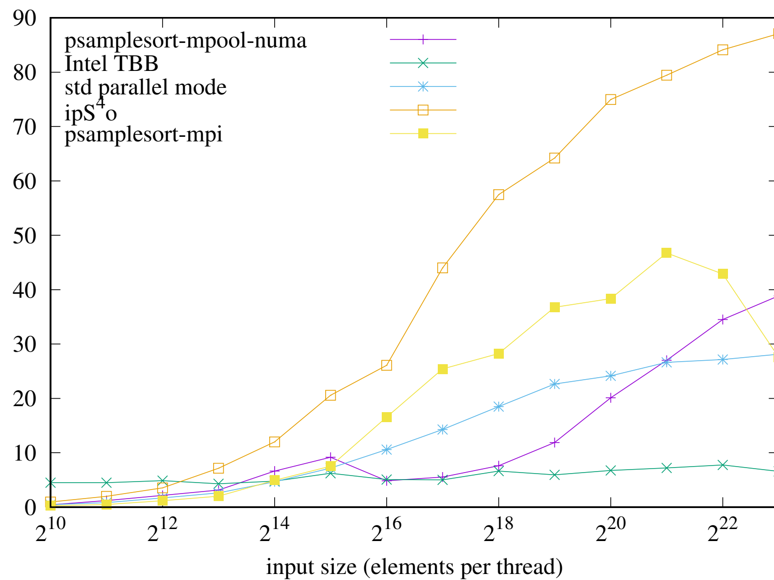


Fig. 5.26. Speedup over `std::sort` as a function of input size with 144 threads.

****Exercise 5.52.** Design and implement a shared-memory parallel sorter that works well over a wide range of values of n and p. A few ideas: Avoid creating threads just for a single sorting call. Switch between different algorithms for different p and n, e.g., the algorithms from Sects. 5.2, 5.4, 5.7, and 5.13. Use profiling tools for tuning. Combine experiments and asymptotic complexity to find the right switching points.

5.13.2 MPI Implementation

Listing 5.2 gives an implementation of parallel sample sort using the message passing interface; see also App. D.[11] The routine *parallelSort* has a template parameter *Element* specifying the element data type from the perspective of C++. Unfortunately, MPI does not know about this and needs to get another specification of this data type, *mpiType*. Each PE samples $1+16\lfloor\log p\rfloor$ of its local elements into the local sample vector *locS* (lines 5–10). These samples are collected in the global sample s using all-gather; see lines 11–13 and also Sect. 13.5). The samples are then sorted using the standard library (line 14). The vector s is reused as a splitter array – the splitter at $s[a\cdot i]$ is moved to $s[i]$ (lines 15–16).

Then the local data is distributed into a vector of bucket vectors *bucket* (lines 17–23). As in the shared-memory implementation, the function *upper_bound* from the standard library is used to find the right bucket. Note that we do not take any special measures with respect to memory management or NUMA effects. Our measurements indicate that MPI and the operating system take care of this quite well.

Now the buckets have to be delivered to the PEs responsible for sorting them. This is done in line 37 using the operation *MPI_Alltoallv*. Doing this requires some preparation (lines 25–36) though. MPI expects senders *and* receivers to specify the length and address of all messages to be delivered. An all-to-all operation with uniform message lengths is used to deliver this information (line 32). These preparations may be a bit cumbersome, but note that they are not a big performance issue. For sorting large data sets, the cost of the preparatory *MPI_Alltoall* is dwarfed by the cost of the subsequent *MPI_Alltoallv*.

Finally, actually sorting the local data is a simple library call (line 39).

Overall, MPI does not get a first prize for extreme elegance, but we end up with a code that has comparable length to the basic shared-memory code and outperforms it significantly.

Figure 5.26 shows the performance of our code when run on a shared-memory system.[12] The MPI code outperforms the shared memory code for large but not too large inputs. This is surprising since a direct shared-memory implementation should usually be faster than a message passing code especially if it goes into a number of complications to handle NUMA effects and to avoid operating system bottlenecks. The point is that MPI does these things implicitly. The kernel bottleneck discussed

[11] We would like to thank Michael Axtmann for providing this implementation and the measurements.

[12] We used GCC 4.8.5 with optimization -o2 and OpenMPI using the Byte Transfer Layer TCP.

Listing 5.2. MPI sample sort

```
template<class Element>
void parallelSort(MPI_Comm comm, vector<Element>& data,
                  MPI_Datatype mpiType, int p, int myRank)
{ random_device rd;
  mt19937 rndEngine(rd());
  uniform_int_distribution<size_t> dataGen(0, data.size() - 1);
  vector<Element> locS; // local sample of elements from input <data>
  const int a = (int)(16*log(p)/log(2.)); // oversampling ratio
  for (size_t i=0; i < (size_t)(a+1); ++i)
    locS.push_back(data[dataGen(rndEngine)]);

  vector<Element> s(locS.size() * p); // global samples
  MPI_Allgather(locS.data(), locS.size(), mpiType,
    s.data(), locS.size(), mpiType, comm);

  sort(s.begin(), s.end()); // sort global sample
  for (size_t i=0; i < p-1; ++i) s[i] = s[(a+1) * (i+1)]; //select splitters
  s.resize(p-1);

  vector<vector<Element>> buckets(p); // partition data
  for(auto& bucket : buckets) bucket.reserve((data.size() / p) * 2);
  for( auto& el : data) {
    const auto bound = upper_bound(s.begin(), s.end(), el);
    buckets[bound - s.begin()].push_back(el);
  }
  data.clear();

  // gather bucket sizes and calculate send/recv information
  vector<int> sCounts, sDispls, rCounts(p), rDispls(p + 1);
  sDispls.push_back(0);
  for (auto& bucket : buckets) {
    data.insert(data.end(), bucket.begin(), bucket.end());
    sCounts.push_back(bucket.size());
    sDispls.push_back(bucket.size() + sDispls.back());
  }
  MPI_Alltoall(sCounts.data(),1,MPI_INT,rCounts.data(),1,MPI_INT,comm);
  // exclusive prefix sum of recv displacements
  rDispls[0] = 0;
  for(int i = 1; i <= p; i++) rDispls[i] = rCounts[i-1]+rDispls[i-1];

  vector<Element> rData(rDispls.back()); // data exchange
  MPI_Alltoallv(data.data(), sCounts.data(), sDispls.data(), mpiType,
    rData.data(), rCounts.data(), rDispls.data(), mpiType, comm);

  sort(rData.begin(), rData.end());
  rData.swap(data);
}
```

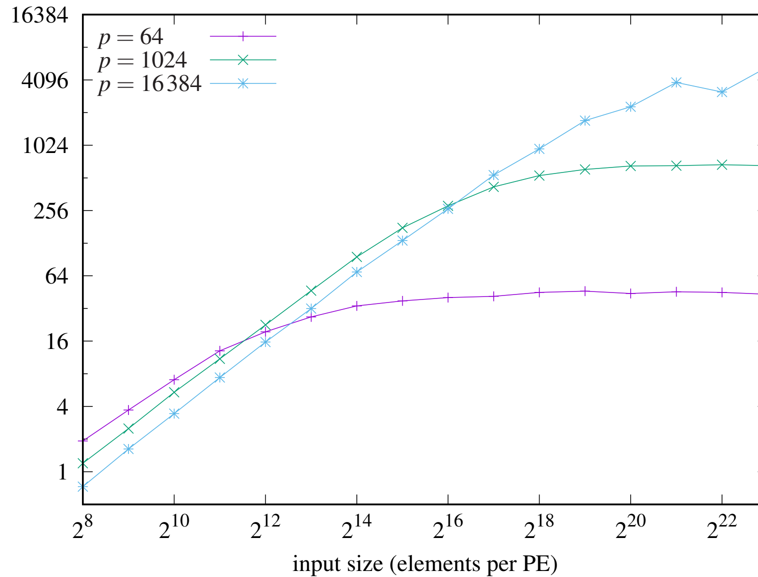


Fig. 5.27. Speedup over `std::sort` as a function of input size on BlueGene/Q. The running times are the median of five trials. The sorted elements were 64-bit random integers.

above does not apply when each PE has an operating system process of its own. Since each process of the MPI code generates its own input data, it automatically allocates the data on the right NUMA-node. Also, MPI pins its processes, i.e., it forces them to be executed on the same core all the time; see Sect. C.7. We have also run the MPI program on up to 16 384 cores of an IBM BlueGene/Q supercomputer.[13] Measuring speedup for large p and large inputs is difficult, since the biggest inputs do not fit into the internal memory of a single node and hence the sequential running time cannot be measured. We overcame this problem by extrapolating the running time of the sequential algorithm. This gives us an optimistic estimate of sequential running time on a hypothetical machine with sufficient memory. Note that using such numbers for computing speedups yields pessimistic estimates for the speedup. Figure 5.27 shows the achievable (extrapolated) speedup as a function of n/p. For $p = 1024$, we achieve a speedup of up to 663 – an efficiency of up to 65%. For such a simple algorithm, this is remarkably efficient. For larger p, the efficiency goes down because we get more and more contention in the interconnection network; see Sect. B.7. Moreover, even for the biggest inputs, with 2^{23} elements per PE, the individual messages in the all-to-all operation have a size of only 256 bytes, so that the startup overheads for

[13] We would like to thank the Gauss Centre for Supercomputing (GCS) for providing computing time through the John von Neumann Institute for Computing (NIC) on the GCS share of the supercomputer JUQUEEN [302] at the Jülich Supercomputing Centre (JSC).

message exchange dominate. Nevertheless, for $p = 16\,384$ we observe a speedup of up to 5391 which is still an efficiency of 33%.

5.14 *Parallel Multiway Mergesort

Multiway mergesort is another external-memory sorting algorithm that is a good candidate for an efficient parallel sorting algorithm. We first describe the shared-memory variant for its elegance and simplicity.

To implement parallel p-way mergesort, we first split the input array s into p equally sized pieces, possibly trying to allocate PEs on the same NUMA node as the RAM storing that piece of data. Each PE then locally sorts the data allocated to it. This takes time $\mathrm{O}(\frac{n}{p}\log\frac{n}{p})$.

For parallel p-way merging, we generalize the splitting idea used in parallel binary (two-way) mergesort (Sect. 5.4). Rather than splitting two sequences into p pieces each, we now split p sequences into p pieces each such that all elements in the first pieces are smaller than all elements in the second pieces, which in turn are smaller than all elements in the third pieces, and so on. We can then obtain the sorted output by sorting the union of the first pieces, sorting the union of the second pieces, and so on. This description assumes that elements are pairwise distinct.

The function *smmSort* in Fig. 5.28 realizes parallel multiway mergesort. After sorting locally, we run p multisequence selections in parallel to find the splitters. The ith processor is responsible for finding the split vector $x@i$ such that the total length of the sequences up to the split elements is equal to $i \cdot n/p$. We then run p incarnations of sequential multiway merging in parallel. The ith incarnation merges the subsequences delineated by $x@(i-1)$ and $x@i$.

We next discuss the search for the splitters. The function *multiSequenceSelect* in Fig. 5.29 describes a sequential algorithm for finding one set of splitting positions. Its input is an array of p sorted sequences and an integer k. It determines, for each sequence, a split index ℓ_i, $1 \le i \le p$, such that $\sum_i \ell_i = k$ and all elements up to any split index are smaller than all elements following a split index. The function maintains two vectors ℓ and r and the invariants $\ell \le r$, $\sum_i \ell_i \le k \le \sum_i r_i$, and

$$\max \cup_i S_i[1..\ell_i] < \min_i \cup_i S_i[\ell_i+1..r_i] \le \max_i \cup_i S_i[\ell_i+1..r_i] < \min \cup_i S_i[r_i+1..|S_i|],$$

Function *smmSort*(s : *Sequence* **of** *Element*) : *Sequence* **of** *Element*
  sort(s); *barrier* // sort locally then synchronize globally
  $x :=$ *multiSequenceSelect*$(\langle s@1,\ldots,s@p\rangle, \left\lceil i_{\mathrm{proc}} \frac{\sum_i |s@i|}{p} \right\rceil)$; *barrier* // find splitters
  return *multiwayMerge*$(\langle\ s@1[x_1@(i_{\mathrm{proc}}-1)+1..x_1],\ldots,$ // assume
    $s@p[x_p@(i_{\mathrm{proc}}-1)+1..x_p]\rangle)$ // $x@0 = \langle 0,\ldots,0\rangle$

Fig. 5.28. SPMD pseudocode for shared-memory multiway mergesort.

```
Function multiSequenceSelect(S : Array of Sequence of Element; k : ℕ) : Array of ℕ
    for i := 1 to |S| do (ℓ_i, r_i) := (0, |S_i|)
    invariant ∀i : ℓ_i..r_i contains the splitting position of S_i
    invariant ∀i, j : ∀a ≤ ℓ_i, b > r_j : S_i[a] ≤ S_j[b]
    while ∃i : ℓ_i < r_i do
        v := pickPivot(S, ℓ, r)
        for i := 1 to |S| do m_i := binarySearch(v, S_i[ℓ_i..r_i])    // S_i[m_i] ≤ v < S_i[m_i + 1]
        if ∑_i m_i ≤ k then ℓ := m else r := m
    return ℓ
```

Fig. 5.29. Multisequence selection. Split the sorted input sequences in S such that the sum of the resulting splitting positions is k and such that all elements up to the splitting positions are no larger than the elements to the right of the splitting positions.

i.e., the elements in the left parts are smaller than the elements in the undecided parts which in turn are smaller than elements in the right parts. Initially, all elements belong to the undecided parts. For simplicity, we assume that all elements are pairwise distinct.

The algorithm works iteratively and continues as long as one of the undecided parts is nonempty. In each iteration, it chooses a random element v from the union of the undecided parts[14] and locates it in all the undecided parts. For each i, we determine m_i such that $S_i[m_i] \leq v < S_i[m_i+1]$ by binary search. This takes time logarithmic in $r_i - \ell_i$. If $\sum_i m_i \leq k$, we set ℓ to m; otherwise we set r to m. In either case, the invariant is maintained.

***Exercise 5.53.** Give detailed pseudocode for a generalization of the function *multiSequenceSelect* that allows keys to appear multiple times. Hint: There is a generic approach that makes keys unique by replacing a key x stored at PE i in position j of the local input array by the triple (x, i_{proc}, j). These triples are ordered lexicographically. You can emulate this ordering without explicitly considering triples in the binary searches. Suppose pivot v has been chosen on PE i at position j of the input. Then, if $i_{\text{proc}} < i$, the binary search should look for the rightmost element with key $\leq v$. At PE i, no search is necessary, and we set $m_i := j$. If $i_{\text{proc}} > i$, the binary search should look for the largest key less than v.

Exercise 5.54. The function *smmSort* in Fig. 5.28 defines the input and output by local arrays. Reformulate your code as a program with explicit parallel loops where the input and output are a single global array.

We turn now to the analysis. Assume *smmSort* is run on p local input sequences of size n/p each. Local sorting takes time $O(\frac{n}{p}\log\frac{n}{p})$, for example using sequential mergesort. Multiway merging takes time $O(\frac{n}{p}\log p)$. Summing this gives time

[14] This can be done by choosing a random number $x \in 1..\sum_i(r_i - \ell_i)$ and by setting v to the element with *global number* x, where the global number of $S_i[y]$ is $y - \ell_i + \sum_{j<i} r_j - \ell_j$ for $y \in \ell_i + 1..r_i$. Pivot v can be found in time $O(|S|)$ by scanning the ranges until the first range i with $\sum_{j \leq i} r_i - \ell_i \geq x$ is found.

$\mathrm{O}(\frac{n}{p}(\log\frac{n}{p}+\log p))=\mathrm{O}(\frac{n}{p}\log n)$, i.e., optimal speedup so far. The barrier synchronizations take time $\mathrm{O}(\log p)$; see Sect. 13.4.2. One iteration of multisequence selection takes time $\mathrm{O}(p\log n)$. From the analysis of quickselect, we know that the expected number of iterations is $\mathrm{O}(\log n)$. However, we have to be careful here. We are running p multisequence selections in parallel and we are only finished when the last of them has finished, i.e., we are interested in the expected maximum execution time of p parallel multisequence selections.

Lemma 5.13. *After* $\mathrm{O}(\log n+\log p)$ *expected iterations, all* p *multisequence selections are finished.*

Proof. We first argue as in the proof of Theorem 5.8. With probability at least $1/3$, an iteration is *good* in the sense that it reduces $\sum_i r_i-\ell_i$ by a factor of at least $2/3$. Hence, $k:=\log_{3/2}n$ good iterations suffice to reduce the problem size to 1. We use the Chernoff bound (A.6) to show that it is unlikely that some particular PE will need a large number of iterations to see k good ones. The probability that *any* PE needs a larger number of iterations is at most p times that probability. In order to be able to use a tail bound, we rewrite the definition of the expected values of an integer random variable as $\mathrm{E}[I]:=\sum_{t\geq 0}t\,\mathrm{prob}(I=t)=\sum_{t\geq 0}\mathrm{prob}(I>t)$. So, let I denote the total number of iterations until all PEs have seen k good iterations. Let X^t denote the number of good iterations that a particular PE j has seen after t iterations. X^t can be written as $\sum_{i=1}^{t}X_i^t$, where X_i^t is an indicator random variable with $X_i^t=1$ if and only if iteration i is good for PE j. We have $\mathrm{E}[X^t]=t/3$. We use (A.6) for $\varepsilon=\frac{1}{2}$ and $t\geq 6k$, which yields

$$\mathrm{prob}\left(X^t<\left(1-\frac{1}{2}\right)\mathrm{E}[X^t]\right)\leq e^{-\left(1-\frac{1}{2}\right)^2\mathrm{E}[X^t]/2}=e^{-\mathrm{E}[X^t]/8}=e^{-t/24}.$$

It is now easy to complete the proof. We first observe

$$\mathrm{E}[I]=\sum_{t\geq 0}\mathrm{prob}(I>t)\leq t_0+\sum_{t>t_0}\mathrm{prob}(I>t)\leq t_0+\sum_{t>t_0}p\cdot\mathrm{prob}(X^t<k),$$

where t_0 is any integer. For the first inequality, we used $\mathrm{prob}(I>t)\leq 1$ for all t. For $t_0\geq 6k$, we conclude further

$$\mathrm{E}[I]\leq t_0+p\sum_{t\geq t_0}e^{-t/24}=t_0+p\frac{e^{-t_0/24}}{1-e^{-1/24}},$$

using (A.14). For $t_0\geq 24\ln p+4$, the last expression is bounded by t_0+1. □

Overall we obtain the following result.

Theorem 5.14. *Parallel multiway mergesort takes time* $\mathrm{O}\left(\frac{n}{p}\log n+p\log^2 p\right)$.

Once more, we can replace $\log n$ by $\log p$, since for $n=\Omega\left(p^2\log p\right)$, the term $(n/p)\log n$ dominates the term $p\log^2 p$ and for smaller n, $\log n=\mathrm{O}(\log p)$. Multiway mergesort is efficient for $n=\Omega\left(p^2\log p\right)$.

***Exercise 5.55.** Design a deterministic algorithm for multisequence selection that runs in time $O(p \log^2 n)$ and is a generalization of our algorithm for two-sequence selection in Sect. 5.4. Hint: The smallest or largest midpoint of a range can replace a range endpoint.

****Exercise 5.56.** Varman et al. [321] gave an algorithm for multisequence selection that runs in time $O(p \log n)$. Develop detailed pseudocode that works for arbitrary n and p without requiring much additional memory (e.g., for padded arrays). Can you demonstrate in an implementation that it outperforms our algorithm in practice?

***Exercise 5.57.** Show a lower bound of time $O(p \log \frac{n}{p})$ for multisequence selection in the comparison-based model.

5.14.1 Distributed-Memory Multiway Mergesort

Multiway mergesort is also attractive for a distributed-memory algorithm. Figure 5.30 shows pseudocode and Figure 5.31 gives an example. The algorithm is similar to sample sort; in particular, each element is communicated only once in a single all-to-all communication. The main difference is that the input is sorted locally up front, and this makes it possible to find perfect splitters efficiently. This results in perfect load balance. The main difference with respect to shared-memory multiway mergesort is that multisequence selection has to be implemented differently, since

Function *dmmSort*(s : *Sequence* **of** *Element*) : *Sequence* **of** *Element*
  sort(s)
  $x := dmmmSelect(s, \left\langle \left\lceil i\frac{\sum_i |s@i|}{p} \right\rceil : i \in 1..p \right\rangle)$ // find splitters
  $\langle s_1, \ldots, s_p \rangle := allToAll(\langle s[1..x_1], s[x_1+1..x_2], \ldots, s[x_{p-1}+1..|s|] \rangle)$
  return $multiwayMerge(\langle s_1, \ldots, s_p \rangle)$

Fig. 5.30. SPMD pseudocode for distributed-memory multiway mergesort

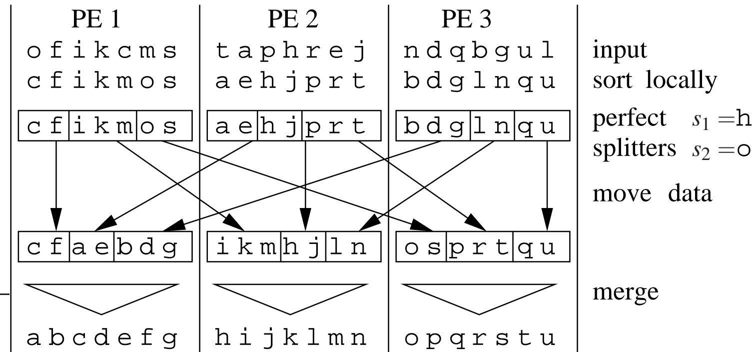


Fig. 5.31. Multiway mergesort of 21 characters on three PEs

```
Function dmmmSelect(s : Sequence of Element; k : Array[1..p] of ℕ) : Array[1..p] of ℕ
    ℓ, r, m, v, σ : Array [1..p] of ℕ
    for i := 1 to p do (ℓ_i, r_i) := (0, |s|)                 // initial search ranges
    while ∃i, j : ℓ_i@j ≠ r_i@j do                            // or-reduction
        v := pickPivotVector(s, ℓ, r)          // reduction, prefix sum, broadcast
        for i := 1 to p do m_i := binarySearch(v_i, s[ℓ_i..r_i])
        σ := Σ_i m@i                                          // vector-valued reduction
        for i := 1 to p do if σ_i ≥ k_i then r_i := m_i else ℓ_i := m_i
    return ℓ
```

Fig. 5.32. SPMD pseudocode for distributed-memory multisequence multiselect with one sequence per PE and p ranks specified by vector k

multiSequenceSelect in Fig. 5.29 makes many remote memory accesses. Our strategy is to perform essentially the same computations but on different PEs. We apply the principle of "owner computes" (see also Sect. 3.1) – each PE is responsible for performing all the computations necessary for its local sequence. By doing this for all p desired ranks at once, we can make the communication very coarse-grained. Figure 5.32 shows pseudocode. The function runs p quickselect algorithms at once, and thus almost all of its local variables are vectors of dimension p. For example, the range $\ell_i..r_i$ now encloses the ith splitting position in s. The search can only terminate if all $p \times p$ ranges have unit size. To find out about that, all PEs have to communicate in an or-reduction collective communication operation; see Sect. 13.2. Picking pivots now also involves communication. This can be done analogously to pivot selection in parallel quicksort (see Sect. 5.7.1) except that the reduction, prefix sum, and broadcast operations involved work component-wise on p-dimensional vectors. Binary searches can be done locally. Adjusting the ranges once more requires a collective communication (reduction) to count the number of elements up to m_i. One iteration of the function *dmmmSelect* takes time $\mathrm{O}(p\log n + p\beta + \alpha\log p)$ for p binary searches and a constant number of reduction/broadcast/prefix operations on vectors of length p. Overall, the selection takes time $\mathrm{O}\left(p\log^2 n\right)$. It is also important to note that the number of startup overheads involved is much smaller – only $\mathrm{O}(\log p \log n)$.

5.15 Parallel Sorting with Logarithmic Latency

With parallel mergesort (Sect. 5.4) and quicksort (Sect. 5.7), we have seen efficient parallel sorting algorithms that sort with span $\mathrm{O}\left(\log^2 n\right)$. On the other hand, in Sect. 5.2 we have seen a fast, inefficient algorithm that sorts in logarithmic time. Here we want to outline a randomized asynchronous CRCW-PRAM algorithm that bridges this gap. For simplicity, we restrict ourselves to the case $n = p$, i.e., there is exactly one element per PE. The basic idea is to use a recursive variant of sample sort.

We use the fast, inefficient algorithm to sort a sample of size $\sqrt{p}$ in logarithmic time. We use an oversampling factor of $\Omega(\log p)$ to obtain an array of $k = \Omega(\sqrt{p}/\log p)$ splitters defining buckets of size $O(n/k)$ with high probability. Now, each PE searches the right bucket for its key using binary search in time $O(\log k)$.

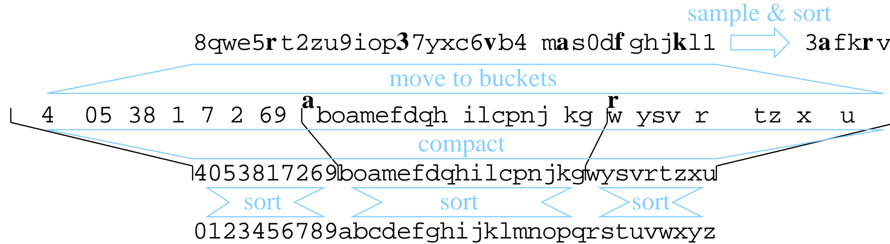


Fig. 5.33. Sorting 36 elements with logarithmic latency. The sample size is 6 and the number of buckets 3.

We then move elements to the right bucket. This is nontrivial. The trick is to allocate a target array for the buckets that is a constant factor larger than necessary with high probability. Each PE repeatedly attempts to copy its element into a random position in its bucket, using a CAS instruction. If this fails, the attempt is repeated until it suceeds. In each iteration, there is a constant success probability. Thus, with high probability, $O(\log p)$ iterations suffice until all PEs have placed their element.

Next, the bucket array is compressed. Denoting an empty entry by a 0 and a full entry by a 1, the position of a full entry in the compressed array is given by a prefix sum over these flags.

Finally, we recurse on the buckets such that again one PE is available for each element. An important technicality is how each PE learns about its bucket. Say that PE i is reponsible for element $A[i]$ where A is the compressed array. Then $A[i]$ is once more located among the splitters s using binary search, say $s[j] \leq A[i] < s[j+1]$. Now, by locating $s[j]$ and $s[j+1]$ in A, PE i can learn the left and right boundaries of its bucket. Figure 5.33 gives an example.

All activities in the first level of recursion run in time $O(\log p)$. Moreover, the number of PEs in a bucket shrinks by a factor $p^{\Omega(1)}$ with high probability. This means that the *logarithm* of the maximum number of PEs having to interact shrinks by a constant factor. Thus, summing over all levels of recursion, we obtain a geometric series summing to $O(\log p)$.

It is an interesting question to what extent this algorithm is practical. In its favor, when $n \gg p$, the algorithm can be generalized in such a way that the number of element comparisons approaches $n \log n$. Moreover, each element is moved only $O(\log\log p)$ times – the number of recursion levels. On the other hand, these element movements are implemented with very expensive random CAS operations, in contrast to the more cache-efficient operations used, for example, by parallel quick-

sort. Our conclusion is that, on currently predominant architectures, finding a sweet spot for the algorithm may be difficult. The algorithm might prove useful for large shared-memory machines that hide the cache miss latency by making extensive use of hardware threads. Such architectures have been built in the past [186, 249] and may reappear in the future.

5.16 Implementation Notes

Comparison-based sorting algorithms are usually available in standard libraries, and so you may not have to implement one yourself. Many libraries use tuned implementations of quicksort.

Canned noncomparison-based sorting routines are less readily available. Figure 5.34 shows an array-based implementation of *Ksort*. It works well for small to medium-sized problems. For large K and n, it suffers from the problem that the distribution of elements to the buckets may cause a cache fault for every element.

To fix this problem, one can use multiphase algorithms similar to MSD radix sort. The number K of output sequences should be chosen in such a way that one block from each bucket is kept in the cache; see also [220]. The distribution degree K can be larger when the subarray to be sorted fits into the cache. We can then switch to a variant of *uniformSort*; see Fig. 5.20.

Another important practical aspect concerns the type of elements to be sorted. Sometimes we have rather large elements that are to be sorted with respect to small keys. For example, you may want to sort an employee database by last name. In this situation, it makes sense to first extract the keys and store them in an array together with pointers to the original elements. Then, only the key–pointer pairs are sorted. If the original elements need to be brought into sorted order, they can be permuted accordingly in linear time using the sorted key–pointer pairs.

```
Procedure KSortArray(a,b : Array [1..n] of Element)
    c = ⟨0,...,0⟩ : Array [0..K−1] of ℕ                  // counters for each bucket
    for i := 1 to n do  c[key(a[i])]++                    // Count bucket sizes

    C := 1
    for k := 0 to K−1 do  (C, c[k]) := (C + c[k], C)      // Store ∑_{i<k} c[k] in c[k].

    for i := 1 to n do                                    // Distribute a[i]
        b[c[key(a[i])]] := a[i]
        c[key(a[i])]++
```

Fig. 5.34. Array-based sorting with keys in the range $0..K-1$. The input is an unsorted array a. The output is b, containing the elements of a in sorted order. We first count the number of inputs for each key. Then we form the partial sums of the counts. Finally, we write each input element to the correct position in the output array

Multiway merging of a small number of sequences (perhaps up to eight) deserves special mention. In this case, the priority queue can be kept in the processor registers [262, 329].

5.16.1 C/C++

Sorting is one of the few algorithms that is part of the C standard library. However, the C sorting routine *qsort* is slower and harder to use than the C++ function *sort*. The main reason is that the comparison function is passed as a function pointer and is called for every element comparison. In contrast, *sort* uses the template mechanism of C++ to figure out at compile time how comparisons are performed so that the code generated for comparisons is often a single machine instruction. The parameters passed to *sort* are an iterator pointing to the start of the sequence to be sorted, and an iterator pointing after the end of the sequence. In our experiments using an Intel Pentium III and GCC 2.95, *sort* on arrays ran faster than our manual implementation of quicksort. One possible reason is that compiler designers may tune their code optimizers until they produce good code for the library version of quicksort. There is an efficient parallel-disk external-memory sorter in STXXL [88], an external-memory implementation of the STL. Efficient parallel sorters (parallel quicksort and parallel multiway mergesort) for multicore machines are available in the GNU standard library [211, 297]. On GPUs, radix sort [282], mergesort [83], and sample sort [198] have been used.

Exercise 5.58. Give a C or C++ implementation of the procedure *qSort* in Fig. 5.9. Use only two parameters: a pointer to the (sub)array to be sorted and its size.

5.16.2 Java

The Java 6 platform provides a method *sort* which implements a stable binary mergesort for *Arrays* and *Collections*. One can use a customizable *Comparator*, but there is also a default implementation for all classes supporting the interface *Comparable*.

The *Arrays* class provides a method *parallelSort*.

5.17 Historical Notes and Further Findings

In later chapters, we shall discuss several generalizations of sorting. Chapter 6 discusses priority queues, a data structure that supports insertions of elements and removal of the smallest element. In particular, inserting n elements followed by repeated deletion of the minimum amounts to sorting. Fast priority queues result in quite good sorting algorithms. A further generalization is the *search trees* introduced in Chap. 7, a data structure for maintaining a sorted list that allows searching, inserting, and removing elements in logarithmic time.

We have seen several simple, elegant, and efficient randomized algorithms in this chapter. An interesting question is whether these algorithms can be replaced

by deterministic ones. Blum et al. [48] described a deterministic median selection algorithm that is similar to the randomized algorithm discussed in Sect. 5.8. This deterministic algorithm makes pivot selection more reliable using recursion: It splits the input set into subsets of five elements, determines the median of each subset, for example by sorting each five-element subset, then determines the median of the $n/5$ medians by calling the algorithm recursively, and finally uses the median of the medians as the splitter. The resulting algorithm has linear worst-case execution time, but the large constant factor makes the algorithm impractical. (We invite the reader to set up a recurrence for the running time and to show that it has a linear solution.)

There are quite practical ways to reduce the expected number of comparisons required by quicksort. Using the median of three random elements yields an algorithm with about $1.188 n \log n$ comparisons. The median of three medians of three-element subsets brings this down to $\approx 1.094 n \log n$ [41]. The number of comparisons can be reduced further by making the number of elements considered for pivot selection dependent on the size of the subproblem. Martínez and Roura [207] showed that for a subproblem of size m, the median of $\Theta(\sqrt{m})$ elements is a good choice for the pivot. With this approach, the total number of comparisons becomes $(1 + \mathrm{o}(1)) n \log n$, i.e., it matches the lower bound of $n \log n - \mathrm{O}(n)$ up to lower-order terms. Interestingly, the above optimizations can be counterproductive with respect to actual running time. Although fewer instructions are executed, it becomes impossible to predict when the inner while-loops of quicksort will be aborted. Since modern, deeply pipelined processors only work efficiently when they can predict the directions of branches taken, the net effect on performance can even be negative [172]. Therefore, in [279], a comparison-based sorting algorithm that avoids conditional branch instructions was developed. This algorithm is also cache-efficient, allows instruction parallelism, and can be made in-place [25]; see also Fig. 5.26. One can also implement quicksort [101] and mergesort [102] in such a way that conditional branches are avoided. An interesting deterministic variant of quicksort is proportion-extend sort [68].

A classical sorting algorithm of some historical interest is *Shell sort* [167, 287], a generalization of insertion sort, that gains efficiency by also comparing nonadjacent elements. It was open for a long time whether some variant of shell sort achieves $\mathrm{O}(n \log n)$ average running time [167, 210]. Only recently was it shown that a randomized version of shell sort does so [132].

There are some interesting techniques for improving external multiway mergesort. The *snow plow* heuristic [184, Sect. 5.4.1] forms runs of expected size $2M$ using a fast memory of size M: Whenever an element is selected from the internal priority queue and written to the output buffer and the next element in the input buffer can extend the current run, we add it to the priority queue. Also, the use of *tournament trees* instead of general priority queues leads to a further improvement of multiway merging [184].

Multiway mergesort and distribution sort can be adapted to D parallel disks by *striping*, i.e., any D consecutive blocks in a run or bucket are evenly distributed over the disks. Using randomization, this idea can be developed into almost optimal algorithms that also overlap I/O and computation [89].

We have seen linear-time algorithms for highly structured inputs. A quite general model, for which the $n \log n$ lower bound does not hold, is the *word model*. In this model, keys are integers that fit into a single memory cell, say 32- or 64-bit keys, and the standard operations on words (bitwise AND, bitwise OR, addition, ...) are available in constant time. In this model, sorting is possible in deterministic time $\mathrm{O}(n \log\log n)$ [16]. With randomization, even $\mathrm{O}\left(n\sqrt{\log\log n}\right)$ is possible [142]. *Flash sort* [241] is a distribution-based algorithm that works almost in-place.

There has been a huge amount of work on parallel sorting. On a CRCW-PRAM, sorting of n integers in the range $1..n$ is possible using logarithmic time and linear work [260]. However, since this algorithm is not stable, it cannot be extended to keys of polynomial size. Allowing a little more time changes the situation, however [32].

Sorting small inputs can be realized in hardware using *sorting networks*, which consist of wires and sorting gates which have two inputs a, b and two outputs $\max(a,b)$, $\min(a,b)$. Batcher's classical result [33] introduces a merging network which merges two n-element sorted sequences using $\mathrm{O}(n \log n)$ gates and a critical path length of $\mathrm{O}(\log n)$. A logarithmic number of merging stages then yields an n-element sorting network with $\mathrm{O}\left(n \log^2 n\right)$ gates and a critical path length $\mathrm{O}\left(\log^2 n\right)$. This algorithm has also been used as the base case of GPU sorting algorithms [83]. Ajtai et al. [11] gave a sorting network with $\mathrm{O}(n \log n)$ gates and a critical path length $\mathrm{O}(\log n)$. Unfortunately, the constant factor involved is prohibitively large. A recent improvement of these factors [133] still remains unpractical.

For practical sorting on large distributed-memory machines, there is a gap between algorithms such as sample sort (Sect. 5.13) or multiway mergesort (Sect. 5.14.1) on the one hand, which communicate their data only once but need $\Omega(p)$ message startups and, on the other hand, polylogarithmic time algorithms such as quicksort (Sect. 5.7) or binary mergesort (Sect. 5.4), which move all the data a logarithmic number of times. This gap can be filled using multilevel generalizations of sample sort and multiway mergesort that move the data k times and need $\mathrm{O}(p^{1/k})$ message startups [23].

For very large data sets, one can combine techniques from external-memory and distributed-memory parallel processing [259].

Exercise 5.59 (Unix spellchecking). Assume you have a dictionary consisting of a sorted sequence of correctly spelled words. To check a text, you convert it to a sequence of words, sort it, scan the text and dictionary simultaneously, and output the words in the text that do not appear in the dictionary. Implement this spellchecker using Unix tools in a small number of lines of code. Can you do this in one line?