A meticulous analysis of mergesort programs*

Jyrki Katajainen**[1] and Jesper Larsson Träff[2]

[1] Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen East, Denmark, Email: jyrki@diku.dk
[2] Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany, Email: traff@mpi-sb.mpg.de

Abstract. The efficiency of *mergesort programs* is analysed under a simple unit-cost model. In our analysis the time performance of the sorting programs includes the costs of key comparisons, element moves and address calculations. The goal is to establish the best possible time-bound relative to the model when sorting n integers. By the well-known information-theoretic argument $n \log_2 n - O(n)$ is a lower bound for the integer-sorting problem in our framework. New implementations for two-way and four-way bottom-up mergesort are given, the *worst-case* complexities of which are shown to be bounded by $5.5n \log_2 n + O(n)$ and $3.25n \log_2 n + O(n)$, respectively. The theoretical findings are backed up with a series of experiments which show the practical relevance of our analysis when implementing library routines for internal-memory computations.

1 Introduction

Given a sequence of n elements, each consisting of a key drawn from a totally ordered universe $\mathcal{U}$ and some associated information, the *sorting problem* is to output the elements in ascending order according to their keys. We examine methods for solving this problem in the internal memory of a computer under the following assumptions:

1. The input is given in an array and the output should also be produced in an array, which may be the original array if the space is tight.
2. The key universe $\mathcal{U}$ is the set of integers.
3. There is no information associated with the elements. Hence, we do not make any distinction between the elements and their keys.
4. Comparisons and moves are the only operations allowed for the elements.
5. Comparisons, moves, and basic arithmetic and logical operations on integers take constant time.
6. An integer can be stored in one memory location and there are $O(n)$ locations available in total.

From assumption 4 it follows that any sorting method must use at least $\Omega(n \log_2 n)$ time (see, e.g., [11, Section 5.3]). However, without this assumption n integers can be sorted in $O(n \log_2 \log_2 n)$ time [1], or even in $O(n)$ time if integers are small (see, e.g., [11, pp. 99–102]).

* Presented at the 3rd Italian Conference on Algorithms and Complexity, Rome, Italy, March 1997.
** Supported partially by the Danish Natural Science Research Council under contract No. 9400952 (project "Computational Algorithmics").

Mergesort is as important in the history of sorting as sorting in the history of computing. A detailed description of bottom-up mergesort, together with a timing analysis, appeared in a report by Goldstine and von Neumann [6] as early as 1948. Today numerous variants of the basic method are known, for instance, top-down mergesort (see, e.g., [17, pp. 165–166]), queue mergesort [7], in-place mergesort (see, e.g., [8]), natural mergesort (see, e.g., [11, pp. 159–163]), as well as other adaptive versions of mergesort (see [5, 14] and the references in these surveys). The development in this paper is based on bottom-up mergesort, or straight mergesort as it was called by Knuth [11, pp. 163–165].

As much as this paper is about sorting, it is about the timing analysis of programs. In contrast to *asymptotical analysis* (or big-oh analysis), the goal in *meticulous analysis* (or little-oh analysis) is to analyse also the constant factors in the running time, especially the constant in the leading term of the function expressing the running time.

To facilitate the meticulous analysis of programs, a model has to be defined that assigns a cost for all primitive operations of the underlying computer. The running time of a program is then simply the sum of the costs of the primitive operations executed on a particular input. Various cost models have been proposed for this purpose. In his early books [10, 11] Knuth used the MIX model where the cost of a MIX assembly language instruction equals the number of memory references that will be made by that instruction, including the reference to the instruction itself. That is, the cost is one or two for most instructions, except that the cost of a multiplication was defined to be ten and that of a division twelve. In his later books [12, 13] Knuth has used a simpler memory-reference model in which the cost of a memory reference is one, whereas the cost of the operations that do not refer to memory is zero.

Knuth analysed many sorting programs in his classic book on sorting and searching [11]. His conclusions were that quicksort is the fastest method for internal sorting but, if a good worst-case behaviour is important, one should use heapsort since it does not require any extra space for its operation.

In this paper we complement Knuth's results by analysing also *multiway mergesort*, which was earlier considered to be good only for external sorting. In the full paper we furthermore analyse a new implementation of the in-place mergesort algorithm developed by Katajainen et al. [8] and the adaptive mergesort variant proposed by van Gelder (as cited in [5]). Our conclusions are different from those of Knuth. In our cost model multiway mergesort is the fastest method for integer sorting, in-place mergesort is the fastest in-place sorting method, and adaptive mergesort is competitive with bottom-up mergesort together with the advantage of being adaptive. An explanation of these results is that the inner loop of mergesort is only slightly more costly than that of quicksort but the outer loop of multiway mergesort is executed less frequently.

The rest of the paper is organized as follows. In Section 2 we introduce a subset of the C programming language, called *pure C*, and an associated cost model. In Section 3 the *worst-case* performance of multiway bottom-up mergesort is analysed under this cost model. The theoretical analysis is backed up by a series of experiments in Section 4. The results are summarized in Section 5 (see Table 2).

2 A C cost model

The model of computation used throughout this paper is a Random Access Machine (RAM) which consists of a *program*, *memory* and a collection of *registers*. A register and a memory location can store an integer. The actual computations are carried out in the registers. In the beginning of the computation the input is stored in memory, and the output should also be produced there.

The machine executes programs written in *pure C* which is a subset of the C language [9]. All the primitive operations of pure C have their counterparts in an assembly language for a present-day RISC processor [15, Appendix A.10]. The execution of each primitive operation is assumed to take one unit of time. The reader is encouraged to compare this assumption with the actual costs of C operations measured by Bentley et al. [3] on a variety of computers.

The data manipulated by pure C programs are *integers* (`int`), *constants*, and *pointers* (`int*`). If a variable is defined to be of type `int`, we assume that the value of this variable is stored in one of the registers, that is, the actual type is **`register int`**. Also pointer variables, whose content indicates a location in memory, are kept in registers, i.e., their type is **`register int*`**.

A pure C program is a sequence of possibly labelled statements. Let `x`, `y`, `z` be not necessarily distinct integer variables, `c` and `d` constants, `p` and `q` pointer variables, and ℓ a label of some statement in the program under execution. The primitive statements of pure C are listed below.

1. *Load statement* "`x = *p;`" loads the integer stored at the memory location pointed to by `p` into register `x`.
2. *Store statement* "`*p = y;`" stores the integer from register `y` at the memory location pointed to by `p`.
3. *Move statement* "`x = y;`" copies the integer from register `y` to register `x`. Also the form "`x = c;`" is possible.
4. *Arithmetic statement* "`x = y` $\oplus$ `z;`" stores the sum, difference, product or quotient of the integers in registers `y` and `z` into register `x` depending on $\oplus \in$ {`+`,`-`,`*`,`/`}. Also the forms "`x = c` $\oplus$ `z;`" and "`x = y` $\oplus$ `d;`" are possible.
5. *Branch statement* "`if (x` $\lhd$ `y) goto` ℓ`;`" branches conditionally to the statement with label ℓ, where $\lhd \in \{<, >, =, \leq, \geq, \neq\}$. Also the forms "`if (c` $\lhd$ `y) goto` ℓ`;`" and "`if (x` $\lhd$ `d) goto` ℓ`;`" are possible.
6. *Jump statement* "`goto` ℓ`;`" branches unconditionally to the statement with label ℓ.
7. *Empty statement* "`;`" does nothing.

Thus pure C statements are simply normal C statements involving at most three addresses. It is easy to translate the C control structures into pure C.

In the basic model the cost of all pure C primitives is assumed to be the same. In reality, computers are more complicated since the actual running time of a program depends on pipelining of the instructions and caching of the data. Therefore, the cost given by the model can only be treated as an approximation of the exact running time. It might be possible to get a more accurate estimate of the running time by assigning a *weight* to every primitive operation, although this still ignores the context in which an operation is being executed.

Knuth has used a simple variant of the weighted pure C model in his recent books [12, 13] when comparing the practical efficiency of different programs. In his *memory-reference model* the cost of every load and store statement is one, whereas the cost of all other primitives involving only registers is zero. The classical goodness measures used in the sorting literature are the number of *key comparisons* and the number of *element moves*. In the following we study only the pure C cost of various mergesort programs, but from these programs the number of memory references, key comparisons and element moves carried out can be readily calculated (cf. Table 2).

In order to make our programs more readable, we will later on use the array syntax "`a[i]`" instead of the pointer syntax "`*(a+i)`". According to our cost model, the cost of the load statement "`x = a[i]`" as well as the store statement "`a[i] = y`" is two. However, if the array is accessed sequentially, it is possible to do a more efficient translation into pure C. The programs to be presented contain very few statements that could be executed in parallel. In our program descriptions independent statements are written on the same line, in order to show where the execution might benefit from parallelism in the hardware (pipelining, instruction parallelism).

3 The worst-case performance of mergesort programs

Let `a` be an array of n elements to be sorted. Further, assume that `b` is another array of the same size. We say that the first element of a subarray of `a` or `b` is its *head* and the last element its *tail*. *Multiway bottom-up mergesort* sorts the elements in passes. Initially, each element of `a` is thought to form a sorted subarray of size one. In each *pass* the subarrays are grouped together such that each group consists of, say, m consecutive subarrays (except the last one which might be smaller), after which the subarrays in every group are m-way merged from `a` to `b`. This way the number of sorted subarrays is reduced from n to $\lceil n/m \rceil$. Then the rôle of `a` and `b` is switched and the same process is repeated until only one subarray remains, containing all the elements in sorted order. The heart of the construction is the merge function, which repeatedly moves the smallest of the heads of (at most) m shrinking subarrays to the output zone in `b`. In the following we present some improvements over textbook implementations, and analyse the performance of these improved implementations.

3.1 Two-way bottom-up mergesort

In two-way bottom-up mergesort, or briefly *two-way mergesort*, two subarrays are merged at a time. Sedgewick [17, pp. 173–174] pointed out that it is advantageous to reverse the order of elements in every second subarray. This way the maximum of the tails of the subarrays will function as a sentinel element saving one pointer test. However, this method will not necessarily retain the order of equal elements, i.e., the resulting sorting method is no longer stable. We present a different optimization which preserves stability and is even more efficient than the reversal method.

In a normal textbook program (see, e.g., [2, p. 63]) a merge of two subarrays is accomplished in a loop where the smallest of the two heads is moved into the output zone, the involved indices are updated accordingly, and at the end of each iteration it is tested whether either of the subarrays is exhausted. However, during one iteration

```
void MERGE(a[], h₁, t₁, h₂, t₂, b[], h₃, t₃) {
        i = h₁; j = h₂; m = h₃;
        u = a[i]; v = a[j];
        if (a[t₁] > a[t₂]) goto test₁;
        goto test₂;
first:  b[m] = u;
        i = i + 1; m = m + 1;
        u = a[i];
test₁:  if (u ≤ v) goto first;
        b[m] = v;
        j = j + 1; m = m + 1;
        v = a[j];
        if (j ≤ t₂) goto test₁;
        for (; i ≤ t₁; i++, m++) b[m] = a[i];
        return;
second: b[m] = v;
        j = j + 1; m = m + 1;
        v = a[j];
test₂:  if (u > v) goto second;
        b[m] = u;
        i = i + 1; m = m + 1;
        u = a[i];
        if (i ≤ t₁) goto test₂;
        for (; j ≤ t₂; j++, m++) b[m] = a[j];
        return; }
```

Fig. 1. An efficient two-way merge.

only the position of one of the heads, not both, is updated. Therefore, one of these tests is superfluous, and can be avoided by more careful programming. Another way of speeding up the program is to check prior to the loop which of the tails is smaller and then write a separate code for the two possible cases. After this it is no longer necessary to test whether the end of the subarray with the larger tail is reached. Moreover, we apply Sedgewick's rule [16, p. 853] that no inner loop should ever end with a jump statement. These ideas are implemented in Fig. 1 as function `MERGE`.

In `MERGE` the value of `a[j]` (`a[i]`) is read before the test whether `j` (`i`) is out of the range or not. However, an element is never used if it is from neither of the subarrays. If required, the reference outside array `a` can be avoided by sorting the first $n-1$ elements and thereafter inserting the last element into its proper location. By binary search this requires only $O(\log_2 n)$ comparisons, after which at most n element moves are to be done.

Depending on the relative order of the tails, there are two symmetric cases. Let us consider the case where the tail of the first subarray is larger than that of the second subarray. In the first inner loop of `MERGE`, a comparison is performed to decide from which subarray an element should be moved to the output zone and, if this element comes from the second subarray, a test is performed at the end to see if that subarray is exhausted or not. Thus, the cost of one iteration is five or six, provided

that pointers are used instead of array cursors. In the second `for`-loop the cost of copying the elements to the output zone is five per element.

In each of the $\lceil \log_2 n \rceil$ merging passes there is at most one merge where the second subarray is shorter than the first subarray or where the second subarray is missing altogether. The overall cost caused by these special merges is proportional to $\sum_{d=0}^{\lceil \log_2 n \rceil} 2^d$, which is $O(n)$. In a normal case the subarrays being merged are of the same size. When considering these normal merges only, the extra test is necessary for at most half the elements in each merging pass. Furthermore, the number of normal merges is clearly never more than $n-1$. Therefore, the overall cost caused by the normal merges is bounded by $5.5n \log_2 n + O(n)$. To sum up, the *worst-case* performance of bottom-up mergesort is $5.5n \log_2 n + O(n)$.

3.2 Four-way bottom-up mergesort

Four-way bottom-up mergesort, or simply *four-way mergesort*, merges four, instead of two, subarrays in each merge. This reduces the number of passes from $\lceil \log_2 n \rceil$ to $\lceil \log_4 n \rceil$, where n denotes the size of the input as earlier. In the implementation of four-way merge it is important to find the minimum of the four heads fast. For example, the heads could be kept in a priority queue from which the smallest element is always removed, and after each removal a new element (if any) from the same subarray is inserted into the priority queue. Practical experiments, e.g., those carried out by Katajainen et al. [8], indicate that the best data structure for this purpose is an unordered list, even though with this structure the number of element comparisons will increase from about $n \log_2 n$ to $1.5n \log_2 n$. In the present paper we show that a faster implementation is obtained if a program state is used to remember the relative order of the four heads, instead of using a data structure. The state is changed accordingly when a head is updated.

As in two-way mergesort, in each merging pass there is at most one special merge where the number of merged subarrays is less than four or, if four, the last subarray is shorter than the others. The cost of these special merges is clearly proportional to $\sum_{d=0}^{\lceil \log_4 n \rceil} 4^d$, which is $O(n)$. Therefore, we concentrate on the normal merges where the subarrays are of the same size.

A normal merge goes through *four phases*: in Phase i, $i \in \{0, 1, 2, 3\}$, the end of i subarrays has been reached, that is $4 - i$ of the subarrays being merged still have elements left. Phase 3 reduces to copying in which the cost of the inner loop is five per element. Phase 2 is a two-way merge, the inner loop of which costs at most six per element. Let us now consider how a three-way merge, i.e., Phase 1 can be accomplished efficiently.

In a three-way merge there are *three cases* depending on which of the tails of the subarrays being merged is smallest. All these cases can be handled in a similar fashion. Therefore, we describe here only the case where the tail of the third subarray is smaller than the other two tails. The program fragment taking care of this case is given in Fig. 2 as a block diagram. The blocks with consecutive numbers should be placed physically after each other in the final program.

Let `u`, `v`, and `x` denote the heads of the first, second, and third subarray, respectively. In the program two invariants are maintained:

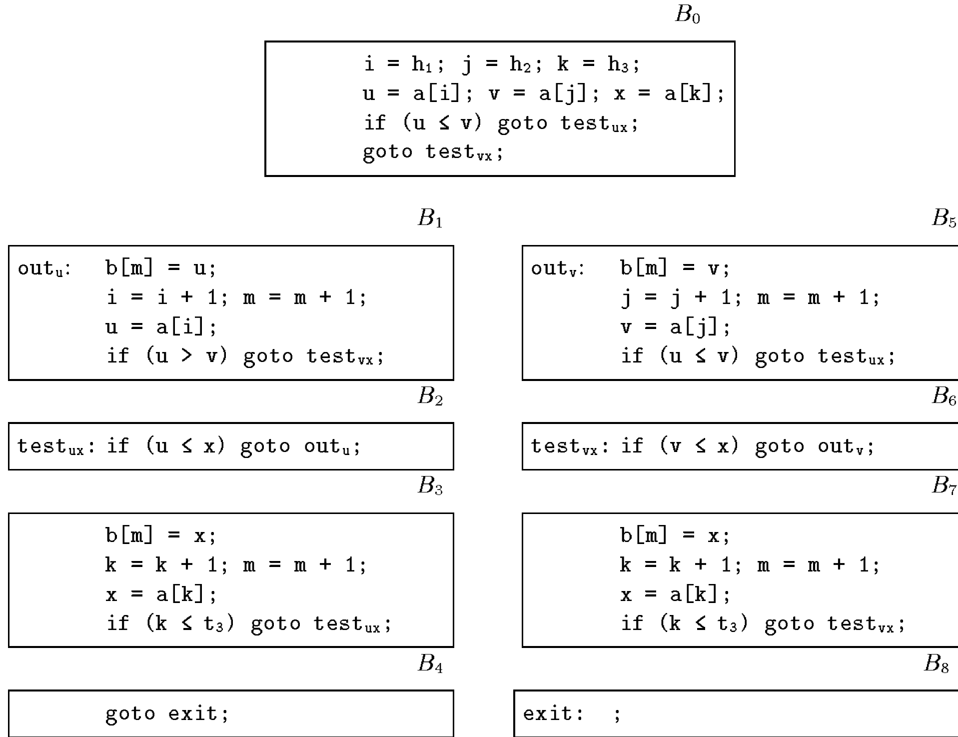

Fig. 2. An efficient three-way merge when the last subarray has the smallest tail.

1. prior to the execution of block B_2, `u` ≤ `v`, and
2. prior to the execution of block B_6, `u > v`.

So the outcome of the tests in blocks B_2 and B_6 determines which of the elements `u`, `v` or `x` is to be moved to the output zone. After the test control switches to block B_1, B_3, B_5, or B_7. In each of these blocks five statements are executed before a new test in B_2 or B_6. Therefore, the cost of the inner loop is six per element moved to the output zone.

Let us finally consider Phase 0 where four subarrays are being merged. Now there are *four cases* depending on which of the subarrays has the smallest tail. Due to symmetry, we study only the case where the last subarray is exhausted first. A block diagram for this particular four-way merge is given in Fig. 3. The block numbering indicates again the order of the blocks in the final program.

Let `u`, `v`, `x`, and `y` denote the heads of the four subarrays. In the program fragment of Fig. 3 the following invariants are maintained:

1. prior to B_2, `u` ≤ `v` and `x ≤ y`,
2. prior to B_4 and B_6, `u ≤ v` and `x > y`,
3. prior to B_9, `u > v` and `x ≤ y`, and
4. prior to B_{11} and B_{13}, `u > v` and `x > y`.

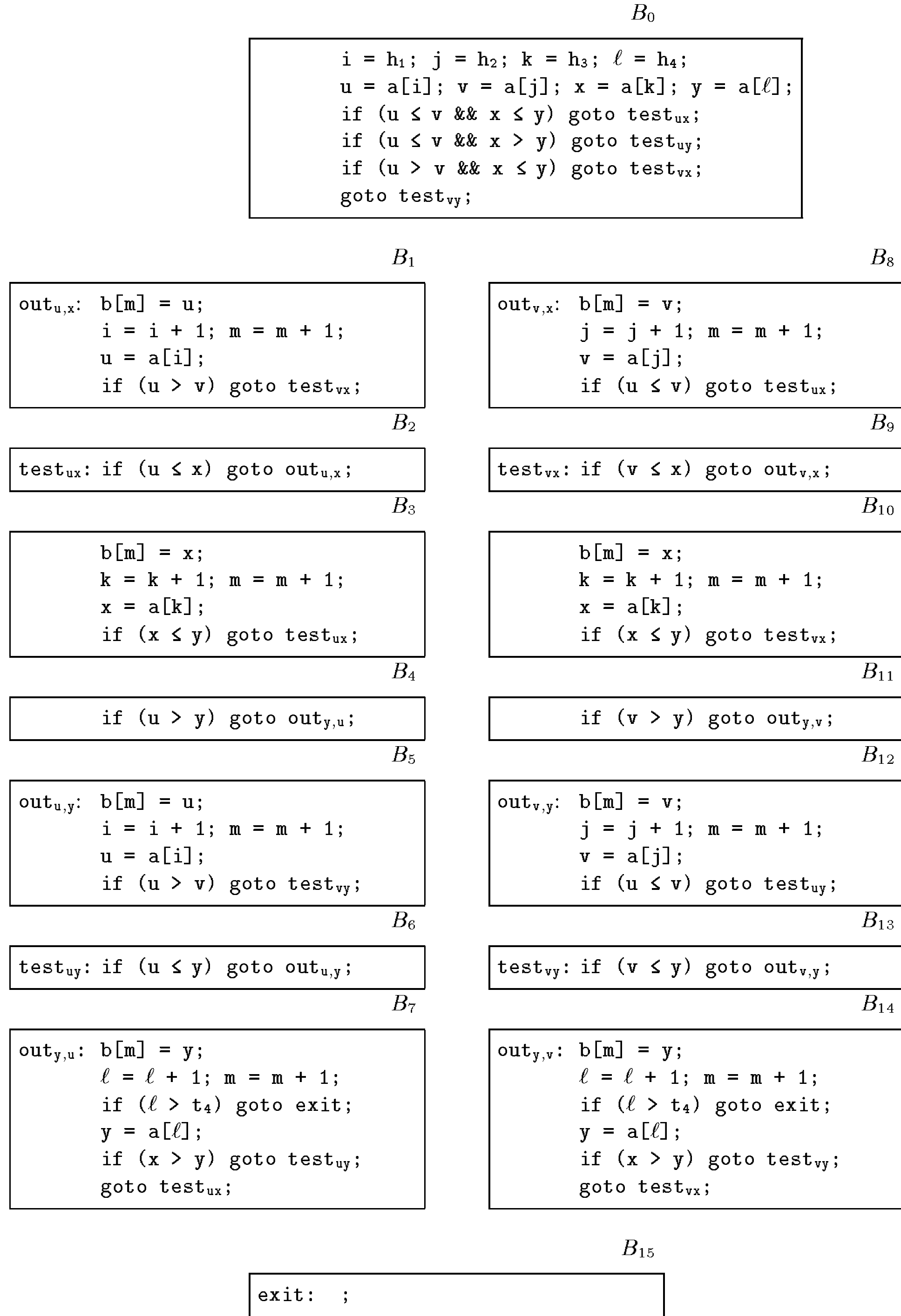

Fig. 3. An efficient four-way merge when the last subarray has the smallest tail.

After a test in B_2, B_4, B_6, B_9, B_{11}, or B_{13} control switches to B_1, B_3, B_5, B_7, B_8, B_{10}, B_{12}, or B_{14}, where an element is moved to the output zone and another test is carried out so that the above invariants hold before the next iteration starts. Each time an element is taken from the last subarray, i.e., when the control is in B_7 or B_{14}, six or seven statements must be executed in addition to the test. If an element is taken from any of the other subarrays, only five additional statements are necessary before starting the next iteration.

To sum up the above calculations, the cost of a normal four-way merge is bounded by $3 \cdot 6 \cdot 4^d + 8 \cdot 4^d + O(1)$ if the size of the merged subarrays is 4^d. In the dth merging pass, $d \in \{0, 1, \ldots, \lceil \log_4 n \rceil\}$, at most $n/4^{d+1}$ normal merges are carried out so their total cost is $6.5n + O(n/4^d)$. Therefore, the cost caused by the normal merges over all passes is $3.25n \log_2 n + O(n)$. Since the cost of special merges is only $O(n)$, the *worst-case* performance of four-way mergesort is $3.25n \log_2 n + O(n)$.

Fig. 2 and 3 describe the four-way mergesort program only in part. Our actual C implementation is about 450 lines of pure C code, corresponding to four variants of Fig. 3, three variants of Fig. 2, and the two-way merge of Fig. 1. One could go a step further and write a program for eight-way mergesort. Back-on-the-envelope calculations show that the worst-case complexity of eight-way mergesort is below $3n \log_2 n + O(n)$. Hence, the performance of eight-way mergesort is — at least in theory — superior to that of four-way mergesort, but the size of the resulting program may invalidate the simple cost model (if the program no longer fits in the program cache). This issue is discussed in the full paper.

4 Experimental results

To test the theoretical predictions we have implemented the mergesort programs developed in the previous sections. For the sake of comparison we have also implemented versions of quicksort and heapsort whose meticulous analyses are given in the full paper. The sorting programs implemented include

1. textbook two-way bottom-up mergesort which avoids unnecessary copying (for the merge routine, see, e.g., [2, p. 63]);
2. efficient two-way mergesort based on the merge function given in Fig. 1; This implementation uses arrays, and according to the simple cost model performs seven or eight instructions in its inner loop.
3. two-way mergesort optimized as above, but using pointers; According to our cost model this version performs five or six instructions in its inner loop.
4. efficient four-way mergesort as developed in Section 3.2, but using pointers;
5. iterative implementation of quicksort taken from the book of Sedgewick [17, pp. 118 and 122];
6. bottom-up heapsort as described by Carlsson [4], but realizing multiplications and divisions by 2 with shifts.

All mergesort programs have the same functionality: they are called with an input array and a work array, and return a pointer to the sorted output, which is either of the two arrays of the call. The running time is measured as the time to execute the entire call to the sorting function. Hence, the time spent in the allocation of

size	mergesort (textbook)	two-way (improved)	two-way (pointers)	four-way (pointers)	quicksort (iterative)	heapsort
100 000						
time (ms)	360	280	240	150	250	440
mems	6 533 608	3 799 996	3 799 996	2 366 613	3 449 301	5 337 697
comps	1 566 804	1 666 803	1 666 803	1 817 456	2 051 474	1 699 474
moves	1 700 000	1 700 000	1 700 000	900 000	1 078 344	1 910 113
200 000						
time (ms)	760	580	510	310	530	1 010
mems	13 866 344	7 999 996	7 999 996	4 733 339	7 268 981	11 274 916
comps	3 333 172	3 533 171	3 533 171	3 829 517	4 381 170	3 598 572
moves	3 600 000	3 600 000	3 600 000	1 800 000	2 250 168	4 019 950
500 000						
time (ms)	1 980	1 510	1 340	840	1 480	3 030
mems	36 716 236	20 999 996	20 999 996	12 833 142	20 536 818	30 138 175
comps	8 858 118	9 358 117	9 358 117	10 102 406	13 082 640	9 646 895
moves	9 500 000	9 500 000	9 500 000	5 000 000	5 857 320	10 700 208
1 000 000						
time (ms)	4 150	3 190	2 830	1 680	3 020	6 850
mems	77 435 094	43 999 996	43 999 996	25 666 483	41 918 488	63 279 048
comps	18 717 547	19 717 546	19 717 546	21 212 307	26 534 802	20 294 732
moves	20 000 000	20 000 000	20 000 000	10 000 000	12 190 528	22 401 825
2 000 000						
time (ms)	8 520	6 510	5 690	3 550	6 550	15 440
mems	162 865 458	91 999 996	91 999 996	55 333 870	88 233 067	132 555 610
comps	39 432 729	41 432 728	41 432 728	44 471 255	5 648 4052	42 589 483
moves	42 000 000	42 000 000	42 000 000	22 000 000	25 363 742	46 802 976
5 000 000						
time (ms)	23 440	18 120	16 100	9 810	16 780	43 460
mems	444 005 872	249 999 996	249 999 996	148 332 746	234 561 702	351 418 010
comps	107 002 936	112 002 935	112 002 935	119 544 746	152 193 888	113 150 419
moves	115 000 000	115 000 000	115 000 000	60 000 000	66 405 096	123 683 773

Table 1. Results for integer sorting. "mems" denotes the number of memory references, "comps" the number of key comparisons, and "moves" the number of element moves.

work space is excluded. Quicksort and heapsort are implemented similarly, in order to make the comparisons as fair as possible.

In Table 1 we give the CPU time spent by the various programs as measured by the system call `clock()`. Times are in milliseconds. We also give the memory reference counts, measured as proposed by Knuth in [13, p. 464-465], the number of key comparisons performed, and the number of element moves. Experiments were carried out on a Sun4 SPARCstation 4 with 85 MHz clock frequency and 64 MBytes of internal memory. We used the GNU C-compiler `gcc` with optimization level `O4`. The running times are averages of several runs with integers drawn randomly from the interval $\{0, 1, \ldots, n-1\}$ by using the C library function `random()`. The sorting programs have all been run with the same input.

The running times follow the analysis according to the simple cost model quite well, whereas counting only memory references is definitely too simple. For instance, the two-way mergesort program based on Fig. 1 makes the same number of memory references as the version doing sequential access with pointers. The simple cost model predicts respective costs $7.5n \log_2 n$ and $5.5n \log_2 n$, which corresponds reasonably well to the measured running times. Quicksort is marginally worse than the pointer version of two-way mergesort, despite its better best-case behaviour. Also in this case, simply counting memory references does not give an accurate prediction of the relative behaviour of the two programs; quicksort is seen to make slightly fewer memory references than mergesort. Four-way mergesort is clearly the best of the sorting programs tested. Compared to two-way mergesort it is about a factor of 1.54 faster, which is not too far from the predicted 5.5/3.25 (the lower-order term is slightly bigger for four-way mergesort compared to that for two-way mergesort).

Compiler optimization gave surprising improvements to all sorting programs, about a factor of 2 to 3. However, the relative quality of the programs was invariant to these optimizations. It is also worth noting that compiler optimizations could *not* improve the textbook mergesort program to perform better than the improved two-way mergesort program based on Fig. 1, even for the latter compiled without optimization.

5 Summary

The theoretical findings of this paper (and the full version) are summarized in Table 2. We plan to carry out a more thorough experimental evaluation of all the sorting algorithms listed there (including the interplay with the compiler).

Acknowledgements

The first author would like to thank Alistair Moffat, Lee Naish, and Tomi Pasanen for their helpful comments.

References

1. A. Andersson, T. Hagerup, S. Nilsson, and R. Raman, Sorting in linear time?, in *Proceedings of the 27th Annual ACM Symposium on the Theory of Computing*, ACM Press, New York, N.Y., 1995, pp. 427–436.
2. S. Baase, *Computer Algorithms: Introduction to Design and Analysis*, 2nd Edition, Addison-Wesley Publishing Company, Reading, Mass., 1988.
3. J. L. Bentley, B. W. Kernighan, and C. J. van Wyk, An elementary C cost model, *UNIX Review* **9** (1991) 38–48.
4. S. Carlsson, Average-case results on Heapsort, *BIT* **27** (1987) 2–17.
5. V. Estivill-Castro and D. Wood, A survey of adaptive sorting algorithms, *ACM Computing Surveys* **24** (1992) 441–476.
6. H. H. Goldstine and J. von Neumann, Planning and coding of problems for an electronic computing instrument, Part II, Volume 2, reprinted in *John von Neumann Collected Works*, Volume V: *Design of Computers, Theory of Automata and Numerical Analysis*, Pergamon Press, Oxford, England, 1963, pp. 152–214.

C program	key comparisons	element moves[1]	pure C primitives[2]	memory references[3]
two-way mergesort				
worst case	$n \log_2 n$	$n \log_2 n$	$5.5n \log_2 n$	$2\ell n \log_2 n$
four-way mergesort				
worst case	$n \log_2 n$	$0.5n \log_2 n$	$3.25n \log_2 n$	$\ell n \log_2 n$
in-place mergesort[5]				
worst case	$n \log_2 n$	$n \log_2 n$	$3.75n \log_2 n$	$2\ell n \log_2 n$
adaptive mergesort[5]				
worst case	$n \log_2 n$	$1.5n \log_2 n$	$8n \log_2 n$	$3\ell n \log_2 n$
randomized quicksort[5]				
best case[4]	$n \log_2 n$	$\Theta(n)$	$3n \log_2 n$	$kn \log_2 n$
standard heapsort[5]				
best case[4]	$n \log_2 n$	$0.5n \log_2 n$	$6n \log_2 n$	$(0.5k + \ell)n \log_2 n$
bottom-up heapsort[5]				
best case	$n \log_2 n$	$n \log_2 n$	$11n \log_2 n$	$(k + 2\ell)n \log_2 n$

[1] A move of an element from one location to another in memory.
[2] When sorting integers.
[3] When the size of a key is k words and that of an element ℓ words.
[4] Assuming that all keys are distinct.
[5] Analysed in the full paper.

Table 2. The behaviour of various sorting programs when sorting n elements, each consisting of a key and some information associated with this key. The low-order terms in the quantities are omitted.

7. M. J. Golin and R. Sedgewick, Queue-mergesort, *Information Processing Letters* **48** (1993) 253–259.
8. J. Katajainen, T. Pasanen, and J. Teuhola, Practical in-place mergesort, *Nordic Journal of Computing* **3** (1996) 27–40.
9. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 2nd Edition, Prentice-Hall, Englewood Cliffs, N.J., 1988.
10. D. E. Knuth, *The Art of Computer Programming*, Volume 1: *Fundamental Algorithms*, Addison-Wesley Publishing Company, Reading, Mass., 1968.
11. D. E. Knuth, *The Art of Computer Programming*, Volume 3: *Sorting and Searching*, Addison-Wesley Publishing Company, Reading, Mass., 1973.
12. D. E. Knuth, *Axioms and Hulls*, Lecture Notes in Computer Science **606**, Springer-Verlag, Berlin/Heidelberg, Germany, 1992.
13. D. E. Knuth, *The Stanford GraphBase: A Platform for Combinatorial Computing*, Addison-Wesley Publishing Company, Reading, Mass., 1993.
14. A. M. Moffat and O. Petersson, An overview of adaptive sorting, *The Australian Computer Journal* **24** (1992) 70–77.
15. D. A. Patterson and J. L. Hennessy, *Computer Organization & Design: The Hardware/Software Interface*, Morgan Kaufmann Publishers, San Francisco, Calif., 1994.
16. R. Sedgewick, Implementing Quicksort programs, *Communications of the ACM* **21** (1978) 847–857. Corrigendum *ibidem* **23** (79) 368.
17. R. Sedgewick, *Algorithms*, 2nd Edition, Addison-Wesley Publishing Company, Reading, Mass., 1988.